UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

Extending Mozart 2 to support multicore processors

Supervisor: Peter Van Roy
Readers: Per Brand (SICS)
Sébastien Doeraene (EPFL)

Thesis submitted for the
Master degree in Computer Science
(120 credits)
Option Artificial Intelligence
by Benoit Daloze

Louvain-la-Neuve

Academic year 2013 – 2014

Contents

Chapter 1

Introduction

Nowadays, most computers have multiple cores. Yet most applications do not take full advantage of them and use only a single core. This might be of not much importance when only two cores are available, but has time goes by, the number of cores rises and the relative disadvantage of using a single core becomes clearer.

We intend to improve the situation for the multi-paradigm Oz programming language in its new implementation, Mozart 2. Mozart 2 [Moz] is a completely redesigned implementation for Oz which brings portability and clarity of the code to a much higher level than its predecessor. It is open source and available at [DJ$^+$13].

Some work has already been done to bring multicore and Oz together. One such essay was done by Konstantin Popov in Mozart 1 but it turned out to be incompatible with the implementation as the implementation assumed a single thread of execution in many places. Another, recent essay, is the language Ozma using concepts from Oz and Scala by Sébastien Doeraene in his master thesis [Doe11].

The goal of this thesis is to extend the Mozart 2 implementation to allow multiple virtual machines to run in parallel. Each virtual machine (from now on abbreviated VM) will have its own kernel thread for its emulator and its own part of memory, granting the full computing power of multicore processors to Oz applications in a single process.

Efficient communication will be provided by VM-level *ports*, detection of termination will be handled by monitoring and memory management will be carefully studied so to be reasonable and controllable.

A clear and well-designed API has been added to use these new features in a clear way and to provide a strong foundation for higher abstractions.

This multicore support has been merged in the official Mozart 2 repository so every user may now use fully his processor cores for Oz programming.

We will start this thesis by a tutorial using the new functionality.

1.1 Motivation

The concurrent programming models of Oz have supported lightweight threads for ages. These threads can be created by thousands and they cleanly integrate the different programming models. For instance they play very well with the declarative model which is very easy to reason about because it is deterministic. Message-passing concurrency allow some nondeterminism but encloses and confines it in a few entities, making the model still quite easy to reason about.

Nowadays, processors are getting more and more cores, or computer are having more and more processors and the model of lightweight threads does not take advantage of this computational power. This is due to the implementation. In current operating systems, a process needs to use multiple operating system threads to actually use multiple cores. This kind of threads is much harder to use, not deterministic and using them properly is so complicated that we tend to avoid them as much as we can.

Some programming languages have the ability to use operating system threads more or less directly. Most of the time, the abstractions do not reduce much the complexity involved in shared-state multi-threaded programming. That is not what we want for Oz, we want to keep the nice properties of the different computation models inside each VM and compose multi-VM programming with message-passing concurrency. We choose message-passing concurrency for multiple VMs as we intend to communicate between VMs with asynchronous messages.

We present in this thesis an addition to the Mozart 2 implementation allowing us to run multiple VMs in parallel with efficient communication between them and proper failure handling while keeping the various properties of message-passing concurrency.

1.2 Contributions

This thesis brings many improvements to Mozart 2. The most important one is the ability to run multiple virtual machines in the same process, taking advantage of multicore processors to achieve parallel computing. Theses virtual machines communicate efficiently through the means of ports on which any immutable value can be sent. Failure handling is done by monitors and there is a guarantee a VM will always be monitored by its parent so failure can not be unnoticed. These virtual machines are conceptually independent from each other, allowing one to crash while keeping the others alive.

Memory management has been redesigned, with the model of Mozart 1 in mind. Now Mozart 2 takes a reasonable amount of memory and do garbage collections as needed to keep low memory usage and high efficiency.

We also added the support for big integers so Oz is no more limited to native integers. The transition between small and big integers is entirely transparent for the user.

Smaller contributions include fixing a couple bugs and memory leaks, improving error reporting in the case of failed unification, fixing the representation of negative numbers, etc.

A few other improvements have been made as well like the restoration of the Mozart 1 Panel (showing real-time memory usage and various other insights about the VM), easier and cleaner build on Mac OS X (thanks to a Homebrew formula, a more flexible configuration and better documentation) and a faster and simpler test runner.

An estimation of the size of our contributions to the code of Mozart 2 can be seen at [Git14]. We became the second developer of Mozart 2 in terms of the number of commits (183) and changed about 4 thousands lines, editing more than 9 thousands lines while developing.

Most of our work is available at [DJ+13]. One can easily get a copy with the Git [TH+05] tool: `$ git clone https://github.com/mozart/mozart2.git`. The rest is available at [Dal14].

1.3 Terminology

We use some specific terminology we would like to clarify from the beginning.

entities We use the term 'entities' when most people would use 'objects' as we want a clear distinction between Oz's objects (which are described by a class) and all the possible entities of the languages. An entity is therefore an instance of a type in Oz like `1`, `3.14`, `hello`, `fun {$ X} X*2 end`, . . .

mutable, stateful The property of certain entities which maintain state and may therefore change over time. Such examples are Oz objects, cells (which can be re-assigned) but also unbound variables as they maintain state at least until they are bound to a given entity. If an unbound variable is bound to a stateless value it actually becomes stateless at binding time. Otherwise it remains stateful as it is bound to a stateful entity.

immutable, stateless The opposite, immutable means the entity will not ever change, like numbers and atoms. A list, tuple or record may qualify as stateless as long as all its elements are bound to stateless entities, like a list of integers.

Part I

Multiple virtual machines

Chapter 2

Tutorial to multi-VM Oz

We want to start this thesis by showing a few use cases for the API we designed. We will therefore take a top-down approach, going into more implementation details as pages are flipped.

We assume the reader has a good knowledge of Oz, otherwise we recommend him to have a look at the Oz online tutorial [HF08], follow the online course “Paradigms of Computer Programming” [VR14] or read the CTM book [VRH04].

We will take as our first example an *embarrassingly parallel* problem, that is a problem which requires little to no coordination between the different tasks. These are often simple to understand and easy to implement.

A good introduction to parallel computing is available in [Bar14]. Many examples are described, yet there does not seem to be one concrete *canonical example* for parallel computing.

2.1 Estimating π with an integral

A well-known *embarrassingly parallel* problem in parallel computing is to estimate π by using a Monte Carlo method, that is randomly generate points inside a square and estimate the ratio of points inside and outside of the inscribed circle. This is unfortunately a rather unrealistic and not representative example. It is very inefficient and depends on strong assumptions on the pseudo-random number generator. We expect that doing the same task twice leads to better results. This is notably wrong if the generator is seeded with the same value and produces therefore the same sequence.

So we take another approach, more efficient and representative of parallel computing problems by computing a definite integral using the rectangle method [Rec].

The reasoning is simple. We start from the fact the tangent of half a right angle (45°) is 1.

$$\tan\left(\frac{\pi}{4}\right) = 1 \iff \frac{\pi}{4} = \arctan(1)$$

Therefore,

$$\pi = 4\arctan(1) = 4\int_0^1 \frac{1}{1+x^2}\,dx$$

Sequential solutions

We will first design a naive sequential solution.

```
functor
import
   System(show:Show)
define
   N=2000000
   ToF=IntToFloat
   Dx=1.0 / {ToF N}

   fun {EstimatePi N}
      fun {Loop I}
         if I == N then
            0.0
         else
            X={ToF I} / {ToF N}
         in
            Dx / (1.0 + X*X) + {Loop I+1}
         end
      end
   in
      4.0 * {Loop 0}
   end

   {Show {EstimatePi N}}
end
```

We use a `functor`, which is similar to a procedure with explicit module dependencies. A functor may also export some values, but we do not do it here. However we import `System.show` as `Show`, which means the `System` module should export a value at `show`.

Arithmetic operators in Oz only succeed if both operand are of the same type, that is integers or floats. So we need to convert integers to floats in a couple places with the builtin function `IntToFloat`, which we aliased to `ToF`.

`EstimatePi` will evaluate `N` times the integral by calling its recursive inner function `Loop`. Then we can simply multiply the sum of them by 4 and we have our approximation of π!

One can execute this code by feeding the part between `define` and the final `end` in the OPI (the Oz Programming Interface, that is the oz-mode of Emacs) after a `declare` or save it to a file `pi.oz`, compile and execute it with `$ ozc -c pi.oz && ozengine pi.ozf`

This works and gives the approximation `3.14159` in two million iterations, which is correct to its fifth (and last shown) decimal place. But it is slow and quite memory hungry, mostly because we use recursion and we are not *tail-recursive*! To get a concrete idea, it takes about 18 seconds on my hardware and uses 1 GB of memory.[1]

Let's fix that by using an accumulator for the sum. We repeat only the body of the `EstimatePi` function for brevity.

[1]These measurements are only there to help the comparison.

```
fun {EstimatePi N}
   fun {Loop I Sum}
      if I == N then
         Sum
      else
         X={ToF I} / {ToF N}
      in
         {Loop I+1 Sum+Dx/(1.0 + X*X)}
      end
   end
in
   4.0 * {Loop 0 0.0}
end
```

Same result and now it takes 14 seconds and 100MB of memory. We can go further by avoiding the expensive division to compute `X` in the loop and simply increment it by `Dx`[2]. By doing so we also remove many conversions to floating-point numbers.

```
fun {EstimatePi N}
   fun {Loop I X Sum}
      if I == N then
         Sum
      else
         {Loop I+1 X+Dx Sum+Dx/(1.0 + X*X)}
      end
   end
in
   4.0 * {Loop 0 0.0 0.0}
end
```

This takes 1.1 seconds and only 4MB of memory!

Parallel solutions

It is now time to really explore parallel computing in Mozart and spawn another VM to do part of the work. This particular problem is really easy to split in smaller computations: we can simply give each VM an equal part of the loop and make it start at the right abscissa.

Splitting the problem in 2 tasks

We will first only split the problem in two tasks to better understand the mechanisms used and then we will generalize to n tasks.

There is a lot to say about Figure 2.1 First, we notice `EstimatePi` now takes an additional argument `From` and `N` has been renamed to `To`. The function will now loop between `From` and `To`, adding `Dx` at each iteration. This means we consider an interval of the integral as it makes more sense to use intervals to divide them. With these two changes it suffices to call `EstimatePi` with $[0, 0.5]$ and $[0.5, 1]$ to split the problem in two equal parts.

[2]One should consider the possible increase in floating-point errors while doing this. In this case the total error is much smaller than the precision of 10^{-5} we want.

```
functor
import
   System(show:Show)
   VM
define
   N=2000000
   ToF=IntToFloat
   Dx=1.0 / {ToF N}

   fun {EstimatePi From To}
      fun {Loop X Sum}
         if X >= To then
            Sum
         else
            {Loop X+Dx Sum+Dx/(1.0 + X*X)}
         end
      end
   in
      4.0 * {Loop From 0.0}
   end

   Master={VM.current}
   {VM.new functor
           import VM
           define
              {Send {VM.getPort Master} {EstimatePi 0.5 1.0}}
           end _}
   PI=thread {EstimatePi 0.0 0.5} end + {VM.getStream}.1

   {Show PI}
   {VM.closeStream}
end
```

Figure 2.1: A parallel version with 2 tasks

We want to execute one of the part in another virtual machine. First we need to `import` the module `VM`. Then we use `{VM.new F}` which creates a new virtual machine identified by the returned identifier (an integer) and running the code of the `functor F`. In this case we ignore the identifier as the VM created has all the needed information at its creation and we need not to interact with it.

The given `functor` then executes half of the work and we need to send this partial result back to the initial VM and add it to the other half. This is possible thanks to *VM ports*, which act like regular ports. One can obtain such a *VM port* by using `{VM.getPort Ident}`. `{VM.current}` is used to retrieve the identifier of the initial VM.

Who says *port* in Oz knows there is a *stream* associated. We therefore retrieve the partial result by reading it from the *VM stream* obtained with `{VM.getStream}`. As it is the first entity sent on the stream we directly access it with `Stream.1`

We wrap the local computation in a lightweight `thread` so both partial results are produced asynchronously and in some cases we could already treat some of them as there are produced. The main thread must only sum the results and it is totally transparent whether a partial result is local or remote.

There is only one thing left to do: close the stream when we stop using it with `{VM.closeStream}`. This is important for proper termination and we will discuss it in details in other chapters.

Notice how passing the computation to the new VM is easy: we simply give a reference to `EstimatePi` which is copied transparently in the new VM. As it is an immutable value (and everything it references also is), there is no way to differentiate it from the one present in the initial VM. First-class procedures are definitely a huge asset in this situation.

We are about twice as fast with this solution, it takes 0.6 seconds and 7 MB of memory!

Generalizing to n tasks

Generalizing to n (`NVMs` in the code) tasks is no very hard but we need to carefully distribute the work between the VMs. In this particular problem, we want to divide the interval of real numbers between 0 and 1 into n equal subintervals. Then VM i, with $i \in [1, n]$ shall use the interval $\left[\frac{i-1}{n}, \frac{i}{n}\right[$.

We would also like to a give a task to the initial VM once it has spawned the others so we need to spawn one less VM and the initial VM does a part of the work instead of just waiting. We were already doing this in the previous example so it should be clear.

The code shown in Figure 2.2 does all of that. We show only the code for the distribution for brevity.

```
Master={VM.current}
NVMs={VM.ncores}
for I in 2..NVMs do
        From={ToF I-1} / {ToF NVMs}
        To={ToF I} / {ToF NVMs}
in
        {VM.new functor
                import VM
                define
                   {Send {VM.getPort Master} {EstimatePi From To}}
                end _}
end
P0=thread {EstimatePi 0.0 1.0/{ToF NVMs}} end
Parts=P0|{List.take {VM.getStream} NVMs-1}
PI={FoldL Parts fun {$ Sum X} Sum+X end 0.0}
```

Figure 2.2: A parallel version with as much tasks as cores there are

It takes about 0.4s with 4 cores. The speedup is not so impressive (3.4) as the task is very short and there is some overhead to create the VMs. If we multiply `N` by 10 to get a longer task then we get a speedup of 3.8 for 4 cores.

This might seem a bit long. But it could be made much shorter using some utility library.

```
fun {Work From#To}
   {EstimatePi From To}
end
Parts={VMUtils.distribute Work VMUtils.floatChunk}
PI={FoldL Parts Number.'+' 0.0}
```

`VMUtils.distribute` assumes the tasks need not to interact, take about the same amount of time and the input can be split in n equal chunks by a function (here `VMUtils.floatChunk`). The definition of `VMUtils` can be found in file `vmutils.oz`.

Conclusion

We have seen many concepts of the VM module in only a few lines. Hopefully there are intuitive and easy to understand. Yet, this example was particularly easy because almost no communication was needed between the created VMs. Notice making the program parallel was relatively easy and did not require too many changes. This is great as it shows our API providing the foundation can be easily used for higher level tasks.

We explored VM creation and communication, it is now time to look at the other concepts of the VM module.

2.2 The VM module

The one concept we have not seen yet is termination and the related monitoring. When a VM calls `VM.new`, it automatically monitors the created VM, which means the creator VM will receive termination information on its VM stream when the created VM terminates. The termination information is a record of the form `terminated(DeadVM reason:Reason)` with `DeadVM` the identifier of the VM which terminated and `Reason` an atom representing the reason of the termination (in the normal case, it is simply `normal`).

These messages are notably useful to know the reason of the termination of a child VM and account for the terminations, for example by counting them to keep track of the number of alive created VMs. A VM may monitor another VM which is not its child by using `{VM.monitor MonitoredVM}`.

At any time, the list of living VMs may be asked with `{VM.list}` which returns the list of identifiers of alive VMs. This can be very useful to debug a program but it should not be used instead of the more expressive and efficient monitors.

The next example we analyze will also use some synchronization by waiting on the VM stream. This is an easy way to implement a barrier and is very similar to waiting on unbound variables (the only difference being that the VM port binds the variable and not directly the user code).

Bidirectional communication can easily be done with VM ports as well, as the creator VM can just call `VM.current` and save its identifier into a variable to pass it to the created VM. The creator VM knows the created VM identifier thanks to the return value of `VM.new`.

2.3 Finite differences for the heat equation

This is a well known example in parallel computing. The one-dimensional heat equation can be easily found in the notes of the FSAB1104 Numerical methods course by Vincent Legat at Université Catholique de Louvain [Leg13, p.162–172]. It represents the distribution of heat over time. u is the temperature, k the thermal conductivity, c the specific heat capacity and ρ the mass density of the material.

$$\rho c \frac{\partial u}{\partial t}(x,t) = k \frac{\partial^2 u}{\partial^2 x}(x,t)$$

Its discretization is easily found with a finite difference for the space term and an explicit Euler method for the time term. By defining the constant β as $\frac{k\Delta t}{\rho c(\Delta x)^2}$,

$$U_i^{t+1} = U_i^t + \beta \left(U_{i-1}^t - 2U_i^t + U_{i+1}^t \right)$$

As we wish to have a more interesting problem and visualization we generalize to the two-dimensional equation, with cx and cy the possibly different β for each dimension.

$$U_{i,j}^{t+1} = U_{i,j}^t + cy \left(U_{i-1,j}^t - 2U_{i,j}^t + U_{i+1,j}^t \right) + cx \left(U_{i,j-1,j}^t - 2U_{i,j}^t + U_{i,j+1}^t \right)$$

We can therefore compute the value for the next time step of the equation of any point in the given space provided we know the value for the current time step for its four neighbors.

Our sequential program is quite straightforward. The parallel program is more complex as we need to carefully communicate between the different VMs. We compared our parallel implementation with the one in C with MPI from [Bar14]. A visualization of heat propagation and the repartition between VMs can be seen in Figure 2.3. For the practical problem we require that the whole contour stays at temperature 0 and the source to be at the center.

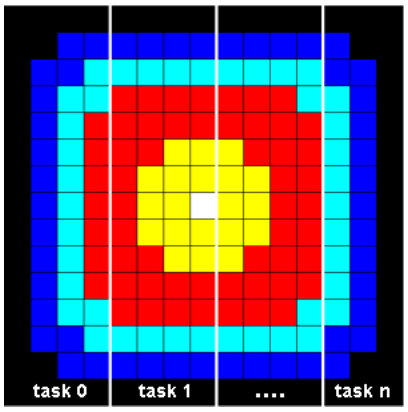


Figure 2.3: A visualization of the heap equation in 2D and the distribution of tasks from [Bar14]

The initial values at $t = 0$ are set to, with w the width, h the height, $i \in [1,h]$ and $j \in [1,w]$:

$$U_{i,j}^0 = (i-1) \times (w-i) \times (j-1) \times (h-i)$$

```
functor
import
   System(show:Show)
define
   W=200 H=200
   CX=0.1 CY=0.1
   STEPS=100

   fun {NewField Init}
      F={MakeTuple field H} in
      for Y in 1..H do F.Y={MakeTuple row W} end
      for Y in 1..H do
         for X in 1..W do
            {Init F.Y.X Y X}
         end
      end
      F
   end

   fun {InitField}
      {NewField proc {$ C Y X}
                   C={IntToFloat (X-1)*(W-X)*(Y-1)*(H-Y)}
                end}
   end

   fun {UpdateField P}
      {NewField
       proc {$ C Y X}
          if Y==1 orelse Y==H orelse X==1 orelse X==W then
             C=0.0
          else
             C=(P.Y.X +
                CX * (P.Y.(X-1) - 2.0 * P.Y.X + P.Y.(X+1)) +
                CY * (P.(Y-1).X - 2.0 * P.Y.X + P.(Y+1).X))
          end
       end}
   end

   fun {Derive F0 Steps}
      fun {Loop F I}
         if I==Steps then F else
            {Loop {UpdateField F} I+1}
         end
      end
   in
      {Loop F0 0}
   end

   F0={InitField}
   FN={Derive F0 STEPS}
   {Show FN.(H div 2+1).(W div 2+1)} % center point
end
```

Figure 2.4: A sequential version for the heat equation

We now discuss our sequential program shown in Figure 2.4. First, a few constants are defined: the width, height, *cx*, *cy* and the numbers of time steps.

We then have three methods to manipulate the two dimensional field. We choose to represent the field as a sequence of rows, a tuple of tuples. `NewField` generates the field and calls the `Init` callback on each cell of the field, along with its X and Y coordinates. Note `{MakeTuple L N}` creates a tuple of label `L` with `N` fields from `1` to `N`, initialized with unbound variables. Giving a value to each of these unbound variables is easy as we can unify them (with the `=` operator) in the callback. `InitField` uses `NewField` and initializes them to the above formula for $t = 0$. As tuple indices start at 1 in Oz we need not to convert our indices, but only to make the result a floating point number.

`UpdateField` also uses the `NewField` abstraction to create the field for the next time step based on the current field. If X or Y denotes a coordinate of the contour, it is set to 0. Otherwise, it is set to the finite difference equation.

We then have our last function `Derive`, which calls `UpdateField` exactly `STEPS` times. Our final block of code initializes a field, calls `Derive` and shows the temperature value for the central point of the space.

People knowing about the *fold left* operation on lists might be tempted to rewrite `Derive` as

```
{FoldL {List.number 1 STEPS 1} fun {$ S _} {UpdateField S} end F0}
```

But we keep it explicit in the main code as it is easier to see the evolution of `Derive` in the parallel program.

We show only the central point to avoid massive output which would be likely unreadable. Programming a visualization is outside the scope of this tutorial, but we can easily give a procedure to show all values if the field size is reasonable (for instance, 20×20).

```
proc {ShowField F}
   for Y in 1..H do
      for X in 1..W do
         {System.printInfo F.Y.X#(if X==W then "\n" else "\t" end)}
      end
   end
end
```

This is nice and concise and it shows how Oz can expressive for such task. The sequential program takes about 8 seconds to complete.

We would like to do better so we now try to make it a parallel program using multiple VMs. As we defined our field as a sequence of *rows*, it makes sense to give each VM a part of the rows. Our distribution is therefore defining chunks separated by horizontal lines, contrary to Figure 2.3. The entire code for the parallel solution is available at [Dal14], file `heat_par.oz`. We show the code by snippets as it is rather long.

The first few lines are identical, except that we import the `VM` module and we define `Master` as the identifier of the initial VM as shown in Figure 2.5.

We start by defining the `functor` which describes what worker VMs shall do in Figure 2.6. We declare four unbound variables, which define in which rows the worker should work in. `From` states the starting row and `To` the last row. `Neighbors` is a list of one or two neighbor workers

```
functor
import
   System(show:Show)
   VM
define
   W=200 H=200
   CX=0.1 CY=0.1
   STEPS=100

   Master={VM.current}
```

Figure 2.5: Lines 1–10: imports and constants

(above and/or below) with which we will communicate. `NNeighbors` is simply the length of the `Neighbors` list as we would like to compute it only once.

We notice thereafter the `NewField` field manipulation method which has been moved inside the functor to be adapted and only consider the rows $[From, To]$. The field is still a tuple of size H to allow direct indexing, but only the values at indices between $From$ and To are filled with rows and their cells initialized.

`InitField` and `UpdateField` can then have the exact same definition as in the sequential version, by using the appropriate `NewField` definition. They are defined between lines 27 and 46 inside the worker functor.

```
   functor F
   import VM
   define
      From To
      Neighbors NNeighbors

      fun {NewField Init}
         F={MakeTuple part H} in
         for Y in From..To do F.Y={MakeTuple row W} end
         for Y in From..To do
            for X in 1..W do
               {Init F.Y.X Y X}
            end
         end
         F
      end
```

Figure 2.6: Lines 11–26: the beginning of the worker functor

We move along with the definition of `Derive` in Figure 2.7. `Derive` itself only has two additional lines compared to the sequential version. The first one sends to each neighbor the row it needs to compute its next region. The time instant is passed as `I`, the row number as `Y` and the row itself as `F.Y`.

Sending rows is done nicely in that single line, but receiving them is much harder. One must realize that some neighbor workers might advance faster than others. Suppose we are computing our rows for $t = 3$ and therefore needs adjacent rows for $t = 2$. Then imagine the fast neighbor above already sends its row for $t = 3$ (this is possible as we just sent it our row for $t = 2$) before the neighbor below could send its row for $t = 2$. We then have on our VM stream [row above $t = 3$, row below $t = 2, \ldots$], which means we must be able to save rows of too recent t for further processing at the next time step. We achieve this by adding them in front of the stream we will process at the next iteration.

```
fun {FindNeighborRows S F Step}
   fun {Loop S Done}
      X|Sr=S
      I#Y#Row=X
   in
      if I==Step then
         F.Y=Row
         if Done+1==NNeighbors then Sr else {Loop Sr Done+1} end
      elseif I>Step then
         X|{Loop Sr Done}
      end
   end
in
   {Loop S 0}
end

fun {Derive F0 Steps S}
   fun {Loop F I S}
      if I==Steps then F else
         for Y#N in Neighbors do {Send {VM.getPort N} I#Y#F.Y} end
         Sr={FindNeighborRows S F I}
      in
         {Loop {UpdateField F} I+1 Sr}
      end
   end
in
   {Loop F0 0 S}
end
```

Figure 2.7: Lines 47–74: `Derive` in the worker functor.

The `FindNeighborRows` function filters the stream `S` to find the corresponding two (or only one if this is the top or bottom worker) adjacent rows at time $t = Step$ for field `F` and returns the stream to be processed for the next time step. This is not so hard as it sounds. We extract the triples from the stream and if we find one with the appropriate time step, we add it in our field and count we found it. If the number of adjacent rows found (`Done`) is equal to the number of neighbors we are done and return `Sr`. If the time step is more recent we recurse and add that message to be processed later at the front of the stream which is returned, while advancing on the stream to find our other adjacent rows. The time step cannot be older than `Step` as we already treated those and the time is only increasing inside a worker.

We are now ready to see the final part of the program orchestrating the workers in Figure 2.8.

The first lines before the `end` describe the initialization logic in the worker. The head of the VM stream is saved in `S`, and the first message is expected to contain the limits of the rows the worker should compute. The worker then sends an `initialized` message to the initial VM to acknowledge the reception of this information. The worker then builds its initial field and wait to receive a second message from the master which must be the atom `go` and acts as a barrier.

We need this synchronization between the master and the workers as we want the first message received by workers to be their configuration (row limits and neighbors). We could relax this by providing a function like `FindNeighborRows` which returns the untreated elements on the stream but we consider it too complex for such case. Note `From` and `To` could easily be passed by capture by defining the worker functor in the for loop which starts VMs but this is not possible for neighbors as the below neighbor VM is not created yet.

Finally the worker starts its computation by calling `Derive` and sends its part of the field to the master. `Record.filterInd` allows to create a record containing only the rows between `From` and `To` as the value other rows is unknown. The now usual `VM.closeStream` terminates the functor execution to tell the VM we are no more interested in events arriving on our stream.

The master execution now really begins by asking the number of cores the processor has, making a tuple to remember VM identifiers, computing the average number of rows a VM shall have and actually create the workers VMs with the worker functor.

We then proceed to send the workers their initialization information and most importantly their neighbor VMs identifiers as we know them now that all workers are spawned.

In the end, we design a `Receive` procedure to assemble the partial worker results into the final field `FN`. Initialization and termination of VMs is also handled in `Receive` so all cases are covered. We let all the workers start once we know they all received their initialization information by sending them the `go` message. Partial results are saved in the master field and termination messages are counted so we know when to stop waiting on workers. A worker which terminates for a reason different than `normal` will raise an exception as we only accept normal termination in `Receive`, so failure will not go unnoticed.

At last we have our results and can display the whole field or just the center point as we choose. Experimentation with 4 cores yields an execution time of 4.5 seconds. The speedup is less impressive than for the first example as many messages are sent between VMs and workers need to wait on other workers rows to continue computation. Yet this is an interesting gain and longer computations tend to have higher gains as the startup phase take relatively less time. As an example, doubling the numbers of time steps takes 16.4 seconds for the sequential solution and only 7.8 seconds for the parallel one.

Conclusion

This example was more substantial and representative of a real parallel program. Interaction between different VMs with VM ports is similar to the interaction between different agents using regular ports.

Parallelization of a program does not always result in improved performance as additional inter-VM communication is not free and a program always has an inherent sequential part, typically the startup and final phases. VMs sometimes need to wait for each other results. All of this makes parallel computing difficult but also challenging and interesting.

```
      S={VM.getStream}
      From#To#Neighbors=S.1
      {Send {VM.getPort Master} initialized}

      NNeighbors={Length Neighbors}
      F0={InitField}
      S.2.1=go % barrier

      FN={Derive F0 STEPS S.2.2}
      MyPart={Record.filterInd FN fun {$ Y R} From =< Y andthen Y =< To end}
      {Send {VM.getPort Master} From#To#MyPart}
      {VM.closeStream}
   end

   N={VM.ncores}
   VMs={MakeTuple vms N}
   Chunk=H div N
   for I in 1..N do VMs.I={VM.new F} end

   % Send neighbors
   for I in 1..N do
      From=(I-1)*Chunk+1
      To=if I == N then H else I*Chunk end
      Below=if I==1 then nil else [From#VMs.(I-1)] end
      Above=if I==N then nil else [To#VMs.(I+1)] end
   in
      {Send {VM.getPort VMs.I} From#To#{Append Below Above}}
   end

   FN={MakeTuple field H}
   proc {Receive S Init Done}
      X|Sr=S in
      case X
      of initialized then
         if Init+1==N then % barrier ends
            for I in 1..N do {Send {VM.getPort VMs.I} go} end
         end
         {Receive Sr Init+1 Done}
      [] From#To#Part then
         for Y in From..To do FN.Y=Part.Y end
         {Receive Sr Init Done}
      [] terminated(_ reason:normal) then
         if Done+1\=N then {Receive Sr Init Done+1} end
      end
   end
   {Receive {VM.getStream} 0 0}

   {Show FN.(H div 2+1).(W div 2+1)} % center point
   {VM.closeStream}
end
```

Figure 2.8: Lines 76–125: Orchestrating the workers

Chapter 3

Design

In this chapter we will discuss all design decisions that lead to this particular solution. There are many of them, some are the foundation of the system, some are little details yet all are important to have a consistent, powerful and intuitive model.

3.1 Concurrency and Parallelism

Lets us first define these two terms. According to [Sun08],

Parallelism
"[. . .] arises when at least two threads are executing simultaneously."

Concurrency
"[. . .] exists when at least two threads are making progress."

So, to achieve parallelism on a single computer we need multiple processors or multiple cores. The way most operating systems propose to achieve parallelism in the same process is to use multiple operating system threads.

Oz has been supporting concurrency with its lightweight threads for ages. But these, at least in the current implementations can only run in a single operating thread and therefore on a single core at a time.

The very first design decision is about what kind of concurrency and parallelism we want. We know we need multiple operating system threads if we want to use multiple core at the same time, that truly execute code in parallel. And we have to choose what kind of mapping between lightweight threads and operating system threads we want. We definitely want the ability to use all cores and have a parallel computation, with the least possible overhead for synchronization and communication.

One might wonder why we do not try to "simply" run the different lightweight threads in different operating system threads so we have all the advantages. This is a very interesting but also a very complex approach. As you may know, interacting with any mutable entity hold by some thread with another operating system thread is unsafe and requires either locks or atomic operations. This means that every time a lightweight thread needs some mutable resource which is not exclusively managed by its own operating system thread, synchronization is needed. Some language implementations work around this fact by having a Global Interpreter Lock [GIL], a huge mutex protecting all VMs structures, at the expense of having almost no parallelization!

The declarative model in Oz embraces the functional programming model and provides accordingly some interesting immutable data structures. If we limited ourselves to using these immutable entities, we could safely use them from different operating system threads. Then, either the garbage collector would have to be adapted to properly detect references from the different operating system threads or we would need to make a copy of the entities in each thread. Making copies would multiply the memory used for each shared entity and detecting references from multiple threads would have a considerable impact on the efficiency of garbage collection.

More importantly stateless entities are few and important entities of the declarative model such as read-only or dataflow variables have an internal state until they are bound to a stateless entity[1]. The state change from not-yet-bound variable to "bound to some determinate entity" must be protected and there is no really good solution. Either we have a single lock for the binding operation and we substantially reduce parallelism or we have a lock for each shared read-only or dataflow variable which implies a serious impact on memory usage.

Now, Mozart 1 already used a trick to take advantage of multiple cores on a single machine. This is possible thanks to its distribution model which also works locally, by spawning multiple processes. And the local version was even optimized to use shared memory between the different processes for communications. However, going through shared memory is still a lot slower than reading and writing to the memory of the same process and making different processes leaves no place to optimize anything which could be shared between the two virtual machines, unless we pay the expense and (huge) complexity of using more shared memory.

We took a somewhat similar approach in Mozart 2 by having separate virtual machines, but running in the same process, each in its own operating system thread. And we made some optimizations so we do not need to duplicate some resources which can be shared. We can therefore reuse some nice principles of distributed programming present in Mozart 1.

We also introduce VM ports and streams, which is a very efficient way to communicate between VMs and behave like regular Oz ports and streams, except only immutable values can be sent on the port and therefore received on the stream.

Finally we provide primitives for handling VM failure such as explicit monitoring and communication of termination information to the parent VM.

3.2 Independence

Ideally, the different VMs are as independent as possible. It would a serious error if one VM inadvertently changed the result of another VM. Also, the least sharing there is, the least contention and the least complexity. Yet, sometimes it is worth to share some part of the system because it might make for significant memory savings.

Each virtual machine has its own heap, and almost everything allocated goes there or temporarily on its thread stack. The VMs we design share almost nothing and their heaps are entirely disjoint. This makes everything a lot easier and ensures scalability as we have very few data dependencies between VMs.

We discuss independence further in the corresponding implementation section 7.1.

[1]which will never happen if bound to a stateful entity as they can only be bound to a single entity for their entire lifetime.

3.3 Communication

As we stated, we would like to have efficient inter-VM communication. This means we must be able either to share or copy entities between VMs. Sharing entities in memory would mean to solve the whole synchronization problem all over again.

Sharing entities by having them *sited*, that is staying in one VM and the other only gets a proxy would require some distribution support we do not have. It is far from trivial to properly manage the lifetime of such proxies and sited entities, but it would provide support for communicating stateful entities. This possibility should be added once the distributed subsytem is implemented in Mozart 2.

Therefore we choose to copy entities and restrict ourselves to stateless entities. Copying stateful entities is a bad idea as changes would not be replicated in other VMs. We use the serialization using the Pickle module as it is already available and has the expected behavior to raise an exception for stateful entities. We discuss the performance implications of using Pickle instead of a lower-level copy in later chapters.

3.4 Termination

Termination in a single-VM world is easy. We can just call `exit()` and we are done.

Termination of multiple VMs is a much more complex problem. We need first to detect when we should terminate a VM. And then the VM should cleanly terminate and release the entirety of its allocated memory, as failing to do so would be a very serious leak and might prevent spanning other VMs.

Detecting termination

Mozart 1 was not very good at detecting termination as it actually never terminated a functor execution and was blocking on a `select()` call. Mozart 2, however, detects when nothing can change the state of the computation and exits the emulation loop. We would like to keep this nice property which notably makes the manual `{Application.exit 0}` at the end of the computation unnecessary.

There are a handful reasons why a VM might terminate. We classify them in five *reasons*.

normal There is nothing left to do or all threads are forever blocked and the computation may not progress anymore.

exception An exception was raised an was uncaught.

kill The VM decides to terminate itself via `{Application.exit ExitCode}` or another VM has asked for this VM to terminate (`{VM.kill Other}`).

outOfMemory The VM runs out of memory and needs to be terminated.

abnormal There is a bug in the VM implementation and we could detect it, yet we had to terminate the current VM.

The ***reasons*** above are also the ones we would like to report to the monitoring VMs.

You might wonder if we should not split ***normal*** in two reasons. It seems wrong that the VM terminates *normally* if every thread is forever blocked. Yet, in Oz we often have threads blocked on unbound variables or lazy computations waiting to be performed and this is perfectly fine for the rest of the computation. For instance, when we have an infinite lazy stream, it is even needed that the producer thread waits to compute the rest and just terminates when the consumer thread has processed the part it wanted from the stream.

Termination procedure

We now discuss how we practically terminate a VM and release its resources. Fortunately, Mozart 2 has already some support to cleanly exit its different levels of execution loops. When we receive the order to stop, we break of all those loops and reach the termination procedure. The termination procedure notably includes stopping the timers, closing the VM port, notifying the monitors and finally deleting the VM and releasing all its resources.

Exit status

In the end, when all VMs have terminated we are left with an interesting problem: with which exit status (or exit code) should we exit the process? There is only one process exit status (from now on 'process status') for many VMs so we need to carefully choose one VM status or combine them. A VM status is either successful if its termination *reason* is ***normal*** or a failure otherwise. We explore three different strategies.

One of them is to state the last living VM status is the process status. But then if the initial VM dies abnormally with failure (for example by an uncaught Oz exception), and a created VM exits later with success, should we consider the whole run to be a success? It certainly would not work well for tests.

Another strategy would be to only consider a VM status eligible for the process status if it is a failure status and not a success. If all VM terminate with a success status we would of course exit the process with a successful status. But it means we would consider the abnormal termination of any VM to be a failure for the entire application run. If we want to support lightweight agents as in Erlang, where failure can be expected and handled, this is definitely not a viable alternative.

A third option would be to specify the initial VM status is always the process status. This is a very simple rule but it may sound arbitrary. It implicitly means the initial VM is monitoring the other VMs[2] and should choose a failure status if something bad happens. If the failure of a sub-VM is expected, the initial VM can just take the appropriate actions and exits successfully once the work is done. If each VM does exactly the same job, it makes not much sense to choose the status of the initial VM and in that case it might be better to choose one of the failing VM statuses if there is any (the second strategy). But this is very rare, as some VM usually assembles the final result, and that VM is often chosen to be the initial one and monitors the other VMs.

This last option is quite a good compromise for many situations, so that is the one we chose. It is also the exact behavior we would like for the exit status of a process performing tests. The initial VM might still implements its own strategy by exiting with the wanted status and monitoring the other VMs.

[2]which is actually the case in our design and conception of monitoring, as a VM always monitor its children and the initial VM therefore monitors all VMs by transitivity.

3.5 API considerations

We would like our API to provide the necessary foundation for all kinds of usage of multiple VMs, yet be as simple and minimal as possible.

We choose to add these procedures to a module – a `functor` with a given name localized at some path or URL – named `VM`.

We depend on `Pickle` being loaded when we send a message on a VM port because we need its capability to serialize these messages. A simple way to achieve that is to `import Pickle` in the `VM` functor and `{Wait Pickle}` before sending a message. This is our only external module dependency.

Entities: VM identifiers and VM ports

One way to think about an API is what kind of entities it manipulate. Oz is a language with relatively few yet very useful entities and we would like to add entities only if we really need them.

We need some way to identify a VM. For instance, if we want to monitor or send a message to another VM, we need a way to tell which VM we want to interact with. This will be the return value of `{VM.current}` and `{VM.new}`.

It needs to be unique per process and we want the ability to transfer an identifier to another VM so VMs can know each other. This means the identifier must a be a value which may be serialized. It would also be nice for debugging purpose to have the ability to print these identifiers and easily recognize which corresponds to which VM.

With these ideas in mind, we see two alternatives: simple integers and global names. Names are unique by definition and they represent well an identifier with this uniqueness property. However, they are hard to distinguish and their value is chosen arbitrarily. Integers on the other hand are easy to distinguish and we can create an order such that a bigger identifier means the VM has been created later. But they do not hide their identity at all. As we think VMs in the same process should be cooperative, and so the mutual secret is not so important, we find integers the best alternative. Ultimately the programmer needs not to know how is represented such an identifier (which he only observes when printing them) as the API never requires the programmer to create an identifier but provides procedures to retrieve identifiers.

So we managed to introduce no new entity for VM identifiers, but what about VM ports? We would like to be as close as possible to a `Port` in Oz, but we cannot use regular ports because they have different semantics. Also, while regular ports contain some state about the position in the stream, VM ports do not. It is the responsibility of each VM to keep track of where we are in the VM stream.

So we introduce internally another entity which implements `Send`, answers true to `IsPort` and whose `Value.type` (the way to query the type of an entity in Oz) is `port`. There is only one property of the Port interface we do not implement, `Port.sendRecv`, which is a special case of `Send` but cannot always be implemented by `Send`. We will discuss about `Port.sendRecv` more in details later. We must keep in mind `Send` can only send immutable values on VM ports until there is some kind of distribution subsystem.

The way to get a VM Port is to use `{VM.getPort VMIdent}`. We can also do the reverse operation with `{VM.identForPort VMPort}`. As we have this bijective relation between the two entities we might wonder whether we could not merge them into one using the internal VM port representation. Although tempting and rather easy to implement we would like to cleanly separate what is a VM identifier which represents a virtual machine from a VM port on which messages can be sent to some VM. It would also not make sense to have the type of a VM identifier to be `port` or that we *monitor* or *kill* a port.

Dead or alive?

We would like our API to depend the least possible on the status of a VM, that is either running or terminated. We could also consider the status *not-yet-created* but we reject immediately this one as there is no possibility to get an identifier of a future VM except by guessing it, which is something we want to discourage so much that we define it as an error and reject any such fake identifier.

We have a great news, if someone use VM identifiers properly (does not try to guess them), no procedure of the VM module will throw an error because a VM terminated. But of course, the semantics might be affected. We discuss the impact of this on each concerned procedure.

Let us start with `{VM.getPort VMIdent}`. If the concerned VM is running, we return a VM Port (which behaves transparently as a regular `Port`). If the VM is terminated, we should as well return a VM Port as otherwise the timing of the call would affect the result, which means calling `VM.getPort` earlier or later with the same identifier would change the result.

The real choice is in `{Send VMPort Value}`. `Send` is asynchronous in Oz, so we want to keep this property. What should we do if the VM described by `VMPort` is terminated? Should we wait indefinitely or let the message be ignored when the receiver has terminated? We can take the special case of a `Send` executed while the receiving VM is running, but the receiving VM is terminated before the message is handled. We can not interact with the sending VM as the message has already been sent and there is no way for a terminated VM to run again and process the message. As we want asynchronicity, there is only one solution which is to let messages sent to a terminated VM be ignored. If the sending VM expects an answer it will of course never get it but this case should be detected and handled with monitors.

`{VM.monitor VMIdent}` makes the current VM monitor the VM identified by `VMIdent`, that is the calling VM will receive a notification of the termination of the other VM on its stream, provided the calling VM actually lives long enough to see that condition. We already have a problem here, how do we ensure a termination never go unnoticed? In particular, if a VM crash very soon after its creation, we might not even have the time to call `VM.monitor` after `VM.new`. Erlang solves this by proposing a built-in function `spawn_monitor` [Eri]. We solve this by stating the parent VM always monitors its (direct) children, so the VM calling `VM.new` creates but also always monitor a new VM. This leaves no possibility to miss a VM termination. We still need to consider what happens when we try to monitor an already terminated VM. It follows logically that we should know about its termination immediately and receive the termination information on our stream.

Finally, what should `{VM.kill VMIdent}` do on a terminated VM? There is nothing we can do and there is nothing to do, so it simply becomes an operation without effect as its effect has already happened (the VM has terminated).

Chapter 4

Memory Management

We present memory management as a whole chapter as there is a lot to say. There are some considerations specific for a multi-VM implementation but many of them are also valid for any kind of VM with dynamic memory management and garbage collection.

Oz is a high-level dynamic language and does not require the developer to care explicitly about memory management. Which means there must be some garbage collector if we want a reasonable memory usage after some time.

We would like to clarify a couple terms for discussing memory management.

A **space** is contiguous area of memory used by the emulator. We never talk about Oz's `Space` in this thesis so there should be no confusion.

The **heap** or more precisely the heap of the emulator contains everything it uses, notably including all the Oz entities. The underlying heap of the C++ program is always referenced as the 'C++ heap' to make the distinction clear. The heap is the active or *primary* space the VM uses. The other space which is not currently used by the VM is known as the 'backup space' or the *second* space.

4.1 Garbage Collection

Mozart 2 implements an exact copying two-space garbage collector using an algorithm close to Cheney's [Che70]. That is a lot in one sentence, so we will explain the terms one by one. It is *exact* because it knows exactly what should be kept alive. The *roots* are known and there is no guessing nor any kind of *stack scanning*. It just computes the transitive closure of all root references and that is all there is to keep. It is a *two-space* garbage collector because it uses two *heaps* of the same size for storing its entities. It is a *copying* garbage collector because it collects by copying entities from one space to another. Once the collection is done, the old space can be *reclaimed* and the entities which have not been copied are forgotten. A very useful reference for all these *terms* is available in [Glo].

Cheney's algorithm includes a nice trick when using a copying garbage collector to solve the cyclic reference problem by making the already copied entities point to the copy in the new space, so further references will just follow the pointer, be updated to that pointer and do not copy the entity again.

This garbage collector (now abbreviated GC) has many nice properties, its main drawback being it uses twice the needed heap space, at least during garbage collection.

4.2 Existing memory management

The memory management implemented originally in Mozart 2 is very simple. Each space is managed by a memory manager. Each memory manager lazily allocates its space, a memory block of 768 MB (configurable) and never releases it. This mean both spaces are available at all times once the first GC happened. The GC is triggered when 95% of the heap get used. The implementation took advantage of that always available second space, to use it in a couple places and not only for garbage collection.

The main problem is doing a GC only when 95% of 768 MB is used is rather late and expensive. Most programs do not need such a large active size and scattering entities across hundreds of megabytes does not seem to be the most cache-friendly strategy. It was done this way as it is very easy to implement and worked relatively well for small programs.

Most current operating system do not immediately give memory a program asks for but only when it accesses it. Which means, for small programs Mozart never did a GC and did not consume much memory as the total memory used was small. This is however more problematic for bigger programs as they use a lot more memory than necessary and do a GC too rarely.

4.3 Improved memory management

We discuss how we implemented a better memory management. We followed the memory management present in Mozart 1 while considering Mozart 2 particularities.

Any serious memory management will grow or shrink its heap so the memory usage roughly corresponds to the memory needed to run the program. We call *active size* the number of bytes needed at any moment for the program to run, counting the memory usage of all referenced entities. But of course we do know that value only just after a GC, as then the usage of the heap corresponds to the active size. At any other moment our usage of the heap is bigger than the active size and it is our job to choose an appropriate heap size for the active size.

Mozart 1 computes the heap size from 3 factors: the active size, the property `gc.free` which dictates how much free space there must be after a GC and the property `gc.tolerance`, how much the emulator is allowed to use above the free space for better memory allocations [DKH+]. Both properties are expressed as percentages of heap size. The default for `gc.free` is 75% and for `gc.tolerance` 20%. According to that documentation, the threshold in Mozart 1 is computed using $threshold = active \times \frac{100}{100-free}$ (which is $active \times 4$ with the given `gc.free` default) and the heap size using $heap\,size = threshold \times \frac{100}{100-tolerance}$.

In Mozart 2 we choose to only grow or shrink the heap by a factor of 2, that is either we double it or we reduce it to its half. We also introduced `gc.min` and `gc.max` which are the bounds of the heap size. In the case the heap size cannot be doubled because it would be bigger than `gc.max`, then the heap size is set to `gc.max`. The same goes for `gc.min`. We use `gc.tolerance` as the minimal percentage (by default 10%) we always keep free so we may still allocate a few things during the GC (notably some special entities lists). We compute our *wished heap size* as the above $heap\,size$ and shrink or grow our actual heap to fit it. This means we take some liberty compared to the model of Mozart 1 as our tolerance might be larger of almost 50% in the worse case (the wished size is just above the actual size so we double it).

As a picture is worth a thousand words, we draw the state of the heap between GCs and just before GC as Figure 4.1. The allocated part is what is used of the heap. The free part is what is unused.

Just before the GC, you might notice on the picture *allocated* to be greater or equal the *threshold*. In general, we do not just reach exactly the threshold but we might go over it if we asked a large allocation. We know then we should GC as quick as possible but we must first terminate the current instruction. In the worse case, if the allocation asked is bigger than *actual heap size* - *threshold*, we have no choice but to do an external allocation to temporarily hold that demand. But as soon as the instruction ends, we perform a GC and free that external allocation. This worse case should happen quite rarely as that allocation is at least of 10% (when $actual = wished$) to 100% (when $actual \approx 2 \times wished$) of the wished heap size.

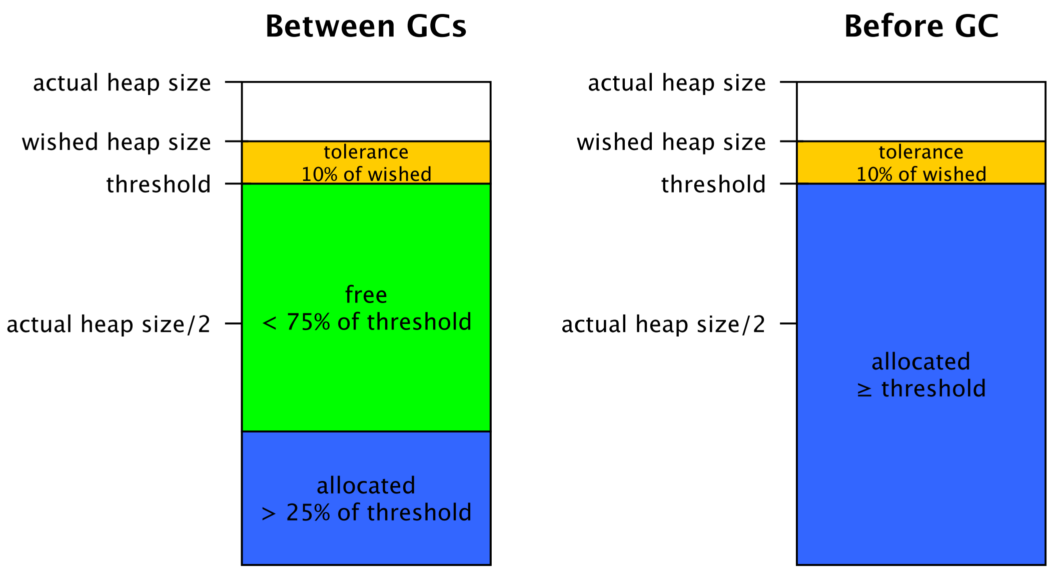


Figure 4.1: Evolution of the heap in Mozart 2

We now show the case where, after the GC, the wished heap size is lower than the half and we shrink the heap in Figure 4.2 along with the opposite, when the wished heap size is above the old heap size and we need to grow the heap in Figure 4.3. An interesting question is how do we resize the heap? Well, resizing means we need a space of a different size and that space should be filled with all the current allocations. It turns out we have quite a good copier at our hand: our copying GC! It indeeds copies and properly updates all references. It also forgets dead entities but there are none as we did not execute any instruction between the first GC (to remove garbage and compute the active size) and the second GC (to copy to an appropriately-sized heap). This mean the drawings of the 'Before GC', 'After GC', 'Shrink' and 'Grow' all represent distinct memory areas (spaces). We only need at most 2 spaces at any time for a VM though, the 'Before GC' space is released once the first GC is done and the 'After GC' space is released once it has been copied to the new heap, either 'Shrink' or 'Grow'.

This double GC might not be the most efficient way but it is certainly very clean, reliable and it does not happen so often. Most programs start by doing many allocations (the heap grows), then reach a stable active size and keep it for a while (few adjustments of the heap size) and finally they terminate (the heap shrinks).

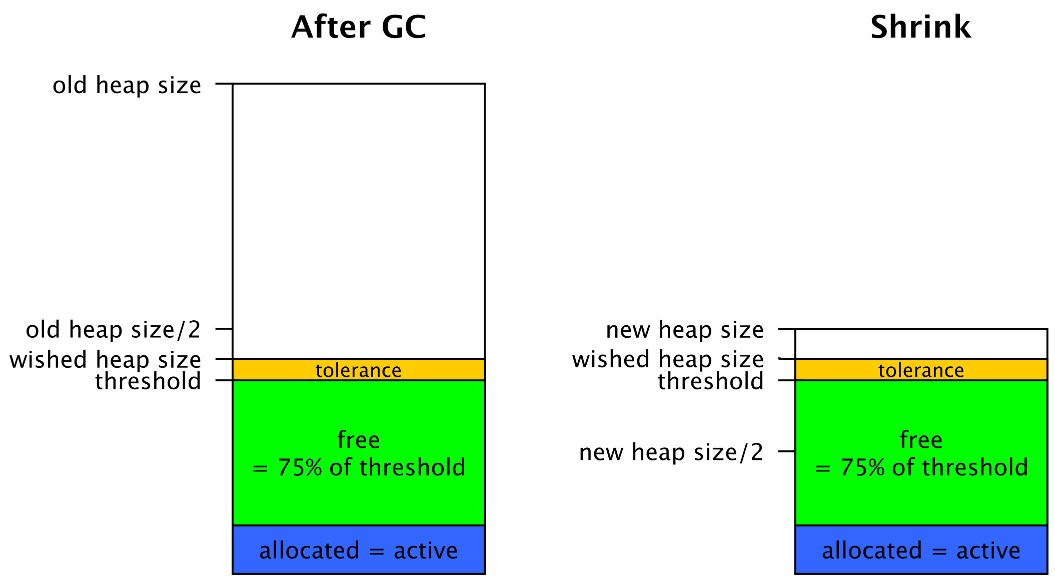


Figure 4.2: Shrinking the heap

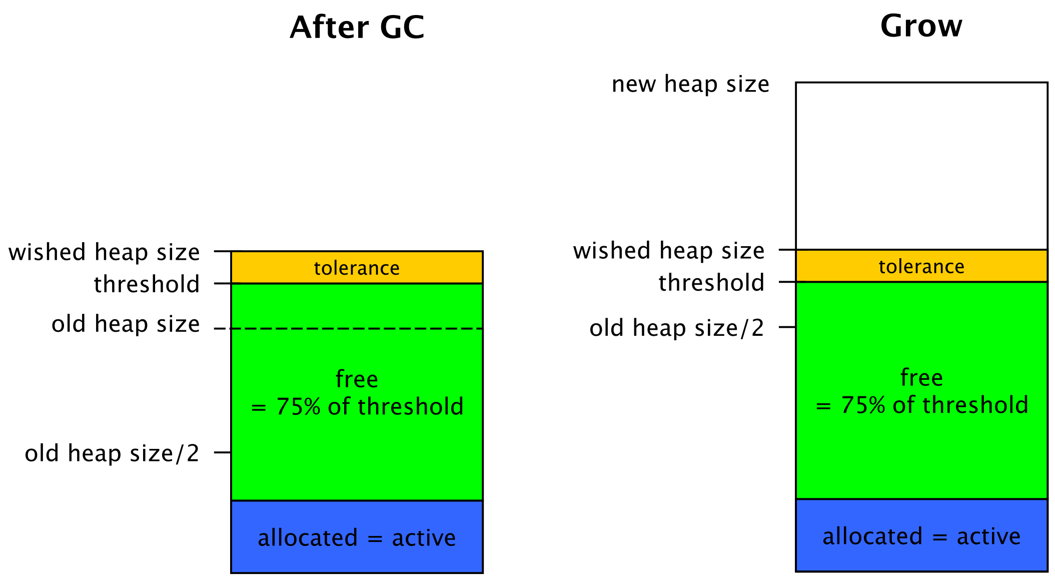
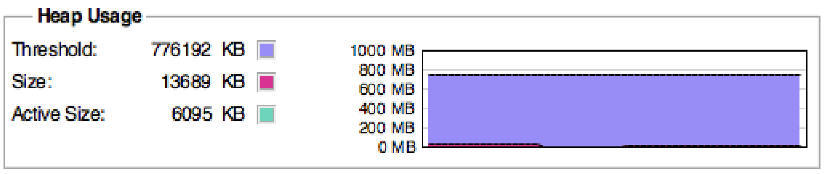


Figure 4.3: Growing the heap

4.4 Visualizing heap usage in real-time

A good way to visualize the memory management and the effect of the garbage collection in Mozart is to use the `Panel`. That is why we added the `Panel` back in Mozart 2.

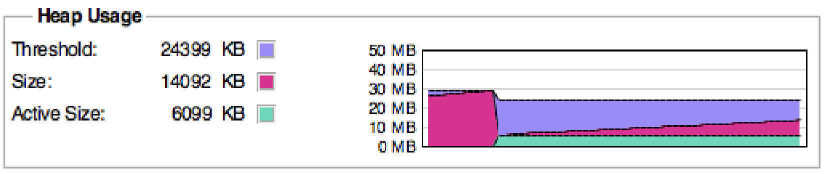
As we saw, the memory management before this thesis was quite simple: a space of 768 MB (configurable) was allocated, and GC would happen only when 95% of it got used. We therefore see the threshold close to 95% of 768 MB.


Heap Usage

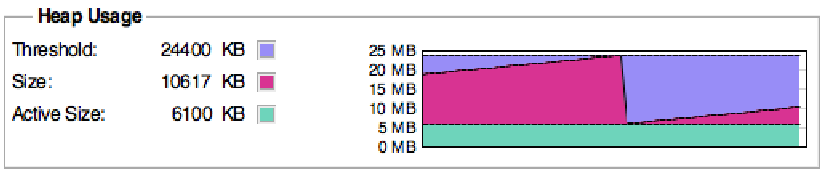
Threshold:	776192	KB	🟪
Size:	13689	KB	🟥
Active Size:	6095	KB	🟩

With the new memory management, the chart is much more interesting.

Before the first GC, the active size is set to 0 as there is no "last active size". Soon, the first GC happens as the memory used reaches the threshold, at which time the active size is computed. The threshold is then set to $\frac{100}{100-gc.free}$ × `active`, which is simply 4 × `active` with the default `gc.free` value of 75%.


Heap Usage

Threshold:	24399	KB	🟪
Size:	14092	KB	🟥
Active Size:	6099	KB	🟩


Heap Usage

Threshold:	24400	KB	🟪
Size:	10617	KB	🟥
Active Size:	6100	KB	🟩

One can open the panel and look at its 'Memory' tab to see this visualization. The Panel can be shown by using 'Open Panel' in the OPI or by importing `Panel` and calling `Panel.open` as in Figure 4.4. This last way is also used to start the Panel within multiple VMs in the same process, each in a separate window.

```
functor
import
   Panel
define
   {Panel.open}
   % rest of the code
end
```

Figure 4.4: Starting the Oz Panel in a functor

4.5 Finding all live references

We would like to expand on our statement in the first section by explaining what is meant by the live references to be 'the transitive closure of all root references'. A copying GC also has to properly updates all references in the entities of the new space, which is something needing great care. Many language implementations do not use a precise GC with moving semantics – we copy but also want the copy to only refer to new entities – as this is something rather hard to achieve.

To find all live references, we compute what is called the reachable set. And we start at the VM roots. The VMs roots are:

- the builtin modules, that is all modules defined in C++. These should not be used in user applications but they are used to implement the core and system modules such as `Int`, `List`, `Application`, ...
- the property registry, containing various properties as key-value pairs such as `gc.free`.
- the pending alarms which may wake the VM up at some future time.
- a list of entities which need to be kept but are not referenced by anything. For instance, some IO entities like TCP connections and results of read/write operations which must stay valid even if nothing in Oz reference them anymore.
- some roots from the environment, which include the VM streams in a multi-VM environment.
- and the biggest source of references: all the lightweight threads.

Each of these defines what it references so we can compute the transitive closure of references. A tuple references all its values, a record all its features and values, a small integer nothing, a lightweight thread mostly its stack frames and a stack frame references its variables and its procedure. All entities include a special constructor for the GC which allow them to be built with the new entities locations. The copy is done in Last In First Out order, copying the leafs (which reference nothing) first so other entities have access to the already-built copies of their referenced entities.

4.6 Extending to multiple VMs

Until now we only discussed memory management in the context of a single VM. What should we do when we run multiple VMs? Since VMs are independent and do not share entities we can have a very simple solution: do nothing! If each VM runs in its own thread and handles its own spaces and no VM knows a space of another then they each work in their own different parts of the C++ heap and there is no possible collision or even race condition for the spaces. The spaces can grow or shrink as wanted as they are always distinct memory areas.

The other solution would be to share entities between VMs or threads. This would bring huge complexity as accessing (reading or writing) any mutable entity would require locking or atomic operations. This would be useful in the model we described at the beginning of chapter 3 when we talked about running lightweight threads in different operating system threads, but we rejected that model for its inherent complexity.

However we need 2 spaces per VM with this model, at least if we allow concurrent GCs. This seems a waste of space, considering we could do much better. The second space is only used during GCs and while having concurrent GCs is nice, it is also rather rare given a program should spend most of its time in execution of the instructions and not in the GC. So we propose a time-memory tradeoff in the direction of much less memory for potentially a small increase of time.

4.7 Single backup space

At every instant, except during the GC, we would like that only one space per VM to be used. We could follow the initial space management technique of Mozart 2 – two spaces kept at all times – but it would be considered rather expensive with many VMs.

We propose instead to use a single second space for all VMs. This implies a couple things.

First, the memory usage would be greatly reduced, by a factor tending to 2 as the number of VM increases. Formally, if *heaps* is the sum of the sizes of the primary spaces (the heaps) used by each VM and *max* is the largest of those spaces (obviously $max < heaps$ with multiple VMs), the total memory usage would be at most, at all time, $heaps + max$. Without a single GC backup space it would be $2 \times heaps$.

Second, only a single GC can be done at one time, as there is only one backup space. We could increase that to a small number of backup spaces instead of just a single one if it revealed to be a problem. But as GCs are quite fast and programs spend most of their time not in GC but in instructions there is currently no need to increase it given the low contention and waiting times.

Finally, this scheme will likely require more size changes of the backup space – that is allocations and deallocations – as every time a VM asks to GC, the size of the backup space might need to be changed to match the size wanted by the VM. We should not allow to use a bigger space as they are swapped after the GC and we do not want to increase memory usage uselessly so this is a strong requirement and the backup space must be of the exact size asked by the VM. The size of the backup space must be adjusted to the one wanted by the VM as soon as the VM is granted access to it.

This is actually rather easy to implement once we removed any permanent reference to backup space. The backup space is then considered an external resource which the VM must ask to be granted access and that access is protected by a simple mutex. The fact it is considered a temporary external resource tends to promote sparse usage for short durations in the code.

4.8 Analysis of memory usage

We want to discuss here how much memory a Mozart 2 process may take and how we can control it. This might be very interesting for tuning performance and keeping the total usage reasonable.

Memory is used in three ways.

- The most important usage is for the VM heaps which contain every Oz entity. At most one heap can be replicated during garbage collection as we have a single backup space so the total usage is, using the above variables, $heaps + max$. Exceptionally, just before and during the GC a VM might use some external memory if a very large allocation was made. But it is immediately freed once the GC is done.
- Some structures belong to no VM are allocated separately.
  - The VM environment containing the list of VMs and the VMs themselves.
  - Serialized messages between VMs which need to be allocated in a neutral space so the receiver can read and copy the message when it wants and does not need to synchronize with the sender.
- Some structures which should not follow the general GC cycle.
  - The list of protected nodes the GC must not collect. They are mostly IO-related structures which need to live for an indeterminate amount of time because of asynchronous IO.
  - Builtins procedures, which live for the whole life of the VM. They could probably even be shared between VMs if they are truly immutable.
  - Big integer internal representations, as explained in section 6.12, page 53.

It is expected that most usage comes from the VM heaps and the rest to be marginal. The number of built-ins is constant, the number of external IO structures depends on the number of Oz structures using them, the number of big integer internal representations is bound by the number of the Oz BigInt entities and the number of VMs should be known by the programmer. Only the number of VM messages is not bound, but the programmer decides how to send messages and the messages reside for very few time in external memory, only the time for the receiver VM to finish its instruction and unpickle the message.

Survey of allocations

No memory leaks are expected in the current version of Mozart 2. A few leaks where found as we analyzed the code but they were simple typographical errors. We fixed them with a few keystrokes and the advices of Sébastien Doeraene to make sure what was intended. Some compact string and threads were allocated in the wrong heap, making them memory leaks and builtins procedures did not release correctly their allocations.

Controlling the heap sizes

We give control to the programmer of each VM minimal and maximal heap size via two properties: `gc.min` and `gc.max`. These properties are inherited when a VM is created, the child uses the properties of the parent.

The properties of the initial VM can also be set via the command line arguments `--min-memory` and `--max-memory` *in megabytes* as in

```
$ ozengine --min-memory 32 --max-memory 512 myfunctor.ozf.
```

The properties `gc.min` and `gc.max` can be set like any other Oz property: via `Property.put`. Their are expressed in number of bytes (this might change but it is compatible with Mozart 1's `gc.min`). We show the way in Figure 4.5.

```
functor
import
   Property
define
   MegaBytes=1024*1024
   {Property.put 'gc.max' 512*MegaBytes}
   ...
end
```

Figure 4.5: Setting the property `gc.max`

The `gc.max` property is considered immediately: a GC is performed if usage is above the new threshold.

On the other hand, `gc.min` is only considered on the next GC. We believe this is right as if the current heap size is smaller than the new `gc.min`, we have no interest of performing a GC immediately as it would use more memory sooner. If the current heap size is bigger and one is concerned about too high memory usage he should set `gc.max` which would provoke the GC if too much memory is used. Also it is likely that the current size is better adapted to the active size than the given `gc.min` so this could provoke an additional GC which may slow the program while not changing the heap size. Therefore, if one wishes to set the minimal heap size, we recommend setting it for the whole process as a command line argument as it is immediately considered or set `gc.max` as well.

If one is concerned about setting `gc.max` in a new VM as early as possible, there are a few solutions. If the parent should have a similar memory usage, it is likely wise to set it in the parent so it will be inherited. Otherwise, the best is to set it early in the code executed by the new VM. If one has direct control on the functor executed by the VM, one can just set the property after `define`. If not, one can compose functors by wrapping the functor to be executed in a functor setting the property then executing the sub-functor as in Figure 4.6.

Default heap size limits

The default maximal heap size is currently 768MB and the minimal heap size 32MB. This minimal heap size is more than strictly necessary but it avoids too much early GCs while being quite a low memory footprint on current hardware.

Controlling the process memory usage

With all this knowledge we can finally expect that a given Mozart 2 emulator at any time uses no more than this number of bytes for its spaces, annotating the value of `gc.max` for some VM vm as $gc.max_{vm}$:

$$\text{maximal total memory used by spaces} = \sum_{vm}(gc.max_{vm}) + \max_{vm}(gc.max_{vm})$$

```
FunctorToExecute=...

% Other={VM.new FunctorToExecute} % must be changed to:

functor SetMemory
import
   Property
   Module
define
   MegaBytes=1024*1024
   {Property.put 'gc.max' 512*MegaBytes}
   % apply (execute) FunctorToExecute
   {Module.apply [FunctorToExecute] _}
end
Other={VM.new SetMemory}
```

Figure 4.6: Composing functors to set `gc.max` early

The first term is simply the sum of the (maximal) heap sizes and the second term is the largest size of the second space, used when a GC happens for the VM with the largest heap.

This computation may of course change if some `gc.max` is updated while running the program. One should consider the maximal value of these `gc.max` if they change over time for an upper bound on memory usage by spaces.

The total memory usage of the emulator process should never significantly and for a long duration exceed this upper bound as we expect other allocations to be marginal to the 'maximal total memory used by spaces'.

In the case one does not modify the `gc.max` property at run-time, we can expect with n, an upper bound on the number of virtual machines executing at any instant in the emulator, that the memory usage be not significantly larger than $(n + 1) \times$ *startup maximal heap size* bytes. That last variable being the maximal heap size given as a command-line argument or the default value, in bytes.

Concretely, for a single VM with the default max heap size, this is 1.5 GB. If we set the max heap size with the command-line argument to 256 MB, we have 512 MB for 1 VM, 768 MB for 2 VMs, 1 GB for 3 VMs and 1.25 GB for 4 VMs.

Chapter 5

Reference of the VM module

The API for creating new virtual machines is available under the Oz module `VM`.

A virtual machine is fully independent from others and runs in its own operating system thread, which grants the full computing power of multicore processors to Oz applications in a single process.

Virtual machines communicate by each providing a so-called `VMPort`, which is very similar to a standard Oz `Port`, but may only send `Pickle`-able values, that is essentially immutable values. As the distribution subsystem is not implemented in Mozart 2, there is no easy way to share mutable entities.

Virtual machines are identified by a unique integer for a given OS process. These are not recyclable and will only refer to a specific virtual machine for the entire life of the process. As they are mere integers, they are the representation of choice of a virtual machine to transfer on a `VMPort`.

One must never try to predict virtual machine identifiers. Giving an integer to any of the procedures below which is not the identifier of a virtual machine will raise an error `vm(invalidVMIdent)`. Other exceptions related to VM management also have the form `vm(Error)`.

The API has been carefully designed so no procedure fails depending on the fact a VM has been terminated or not.

The way to control memory usage is detailed in section 4.8, page 36.

We classify the API in three categories: VM management, communication and failure handling.

5.1 VM Management

{VM.ncores ?NCores}

Returns the number of hardware threads (cores) of the executing hardware. To reach maximal efficiency we should have about the same number of VMs.

If it can not be detected, a system error `vm(cannotDetectNumberOfCores)` will be raised. This should never happen on common platforms.

{VM.current ?VMIdent}

Returns the identifier of the current VM.

{VM.new App ?NewVMIdent}

Starts a new Oz virtual machine in the same process. `App` can be an `atom`, in which case it is the application URL (the initial functor). Or it can be a `functor` in which case it is applied as the application functor. Returns the identifier of the newly-created VM.

If `App` is an atom, it must be a correct path to a compiled functor. If it is a `functor` defined in the code, it must not reference any mutable entity as it will be serialized by `Pickle`.

{VM.list ?VMs}

Returns a list of the running virtual machine identifiers, in the order they were started. Virtual machines which have terminated are not reported in this list. It should only be used for debugging purposes, as the list returned might becomes very quickly outdated.

5.2 VM Communication

{VM.getPort VMIdent ?VMPort}

Returns the `VMPort` for the given virtual machine identifier. This can be given as a port to `Send`.

Even if the concerned virtual machine has terminated or closed its port, this succeeds and returns a valid `VMPort`.

{Send VMPort Value}

Sends `Value` to `VMPort` asynchronously. `{Pickle.pack Value}` will be called if the port is still opened, which raises an error if `Value` can not be serialized by `Pickle`.

If the target virtual machine has terminated or closed its port, the message will be ignored.

{VM.getStream ?VMStream}

Used to get the VM stream of the current VM.

The first time it is called, it is guaranteed to return the beginning of the stream, so no message is missed. Semantically, it is as if the first call was done atomically at the virtual machine creation.

On further calls, only new messages will appeared in the returned stream (it returns the *tail* of the stream). This is done to prevent memory leaks as it would mean we need to keep a reference to the beginning of the stream. If the stream has been closed with `{VM.closeStream}`, `nil` will naturally be returned.

The virtual machine will not terminate by itself once `{VM.getStream}` has been called, until closing the stream with `{VM.closeStream}`. We can not know when another virtual machine might send us some value on our stream (this is similar to handling external IO events), so the VM will wait for further messages on its stream.

{VM.closeStream}

Tells the current virtual machine we are no more interested in the VM stream. This binds the tail of the stream to `nil`, and ignores any further messages sent on the `VMPort`. Virtual machines using `Send` will not wait, but their message will be ignored. Calling `VM.closeStream` once it has already been called for a given VM will do nothing.

{VM.identForPort VMPort ?VMIdent}

Returns the virtual machine identifier for the given `VMPort`.

5.3 VM Failure handling

{VM.kill VMIdent}

Kills the specified virtual machine, by making it stop after terminating its current instruction. One VM can terminate itself by calling `VM.kill` on its own VM identifier if it wishes to stop its computations.

The VM to be killed will be terminated before its next call.

If the virtual machine has already terminated, it does nothing.

{VM.monitor VMIdent}

Make the current VM notified when the VM identified by `VMIdent` terminates. Note a VM creating another (using `VM.new`) always monitors the newly-created VM.

The notification is done by sending a message `terminated(DeadVMIdent reason:Reason)` to the monitoring VM stream. `DeadVMIdent` is the VM identifier of the now terminated VM and reason is one of the following:

normal The VM terminated normally by having all its threads finished or blocked forever.

exception The VM terminated because of an uncaught exception.

outOfMemory The VM terminated because there was no memory left. This can be adjusted by

kill The VM terminated either because it decided to via `Application.exit` or some VM asked for it to terminate with `VM.kill`.

abnormal The VM terminated because of an implementation bug.

unknown The VM was already terminated when `VM.monitor` was called.

If the virtual machine identified by `VMIdent` has already terminated, the termination message is sent immediately and the reason is `unknown` (because it disappeared with the termination of the VM). The parent VM will get the true reason and might communicate it to other VMs if needed.

One VM cannot monitor itself, `vm(cannotMonitorItself)` is raised if such attempt is done.

Part II

Implementation

Chapter 6

A tour of the Mozart 2 implementation

Mozart 2 [Moz] is a completely redesigned implementation for Oz 3.

It has been written by Sébastien Doeraene and Yves Jaradin in 2011 and enjoys a very clean and readable code. The language used for the emulator is C++11, and the implementation is using in general recent technologies.

A lot of effort has been made to define a maximum of the language in Oz itself. For instance, the compiler is written in Oz, a good part of the serialization (`Pickle`) is also written in Oz, . . .

As this master thesis is about "Extending Mozart 2", we will present its codebase so the necessary references will make sense and the reader can get a much improved idea of what it means for the implementation.

For the interested, the code is available at [DJ$^+$13] and we will be mostly looking at files in `vm/vm/main`. One can read the Section 6.10 for a better understanding of the organization of the files.

6.1 Top-level view

We will start our journey through the code base by having a global view of the system.

As every C or C++ program, it starts in the `main()` function.

The `main()` function parse the command-line arguments, sets up an asynchronous IO thread and finally calls `VirtualMachine::run()`.

A `VirtualMachine` has a `VirtualMachineEnvironment` (which is shared in the case of multiple VMs), a `ThreadPool` (a queue of scheduled lightweight `Thread`s), two `MemoryManager` (each handling its memory space), a `GarbageCollector`, a `PropertyRegistry` and various other structures global to the `VirtualMachine`.

Finally, `Thread::run()` is the emulation loop, doing one instruction at a time.

6.2 PropertyRegistry

The `PropertyRegistry` is a mapping of atoms to Oz values. This mapping is also accessible in the C++ code. In fact a variable (like a integer) can be bound in C++ and reflect a specific property. This makes it an excellent place for configuration but also to exchange entities between Oz and C++.

6.3 Builders and matchers

The builders: going from C++ to Oz

In many language implementations, the conversion of values between the host and the target language is painful.

This is not true in Mozart 2 in which the mapping is clear, concise and precise.

First, every `DataType` has a `build` method, which makes the construction very clear by being explicit. For instance `SmallInt::build(vm, 42)`.

Second, a DSL has been developed to ease these conversions, and the type checking of C++ is smartly used to choose the right conversion.
The DSL is composed of the methods `build<T>(VM vm, T value)` under the `mozart` namespace.

Every builder returns an `UnstableNode`.

Here is a listing of the mapping between primitive C++ and Oz types:

`int`	⇔	`Int`
`double`	⇔	`Float`
`bool`	⇔	`Bool`
`const char*`	⇔	`Atom`
`unit_t`	⇔	`Unit`

Table 6.1: Mapping between primitive types of C++ and Oz

Then we have some builders specialized for aggregate types:

`buildNil(vm)`	⇒	`nil`
`buildCons(vm, head, tail)`	⇒	`Head|Tail`
`buildList(vm, elements...)`	⇒	`List`
`buildTuple(vm, label, fields...)`	⇒	`Tuple`
`buildSharp(vm, fields...)`	⇒	`Tuple` with label `'#'`
`buildArity(vm, label, features...)`	⇒	a `Record`'s arity
`buildRecord(vm, arity, fields...)`	⇒	`Record`

The matchers: going from Oz to C++

The matchers of Mozart 2 allow to achieve pattern matching on Oz values in C++ code!

We will use the example documented in `matchdsl.hh`.

```
proc {DoSomething Value}
   case Value
   of X#42#Y then {Show X} {Show Y}
   [] Z andthen {IsInt Z} then {Show Value}
   end
end
```

This can be translated in C++ to:

```
void doSomething(VM vm, RichNode value) {
  using namespace mozart::patternmatching;

  RichNode X, Y;
  nativeint intValue;

  if (matchesSharp(vm, value, capture(X), 42, capture(Y))) {
    show(X);
    show(Y);
  } else if (matches(vm, value, capture(intValue))) {
    show(intValue);
  } else {
    return raiseTypeError(vm, "int or 42-pair", value);
  }
}
```

We notice a few things here.

First, we have a set of `matches(vm, value, pattern)` methods which return whether they matched the given pattern.

The pattern is any of the following:

- a `wildcard()`, which matches any value.
- a simple C++ value, mapping to an Oz value by looking at the table 6.1 in the previous section, which matches if the values are equal.
- a variable `capture()` which will save the value in the given variable if the matches succeed (taking the variable type in account).

6.4 Builtins and ozCalls

Builtins: calling C++ code from Oz

Calling the `Builtins` from Oz is done the same way as in Mozart 1, via *boot* modules. A very simple `Module` is the `ModAtom` module in Figure 6.1.

We see the builtins are modeled as C++ classes inheriting `Builtin<T>`.

```
class ModAtom: public Module {
public:
  ModAtom(): Module("Atom") {}

  class Is: public Builtin<Is> {
  public:
    Is(): Builtin("is") {}

    static void call(VM vm, In value, Out result) {
      result = build(vm, AtomLike(value).isAtom(vm));
    }
  };
};
```

Figure 6.1: The Atom module

Builtins are serialized as a simple pair of atoms: (module name, builtin name).

To find them back, modules are registered at VM initialization with `VirtualMachine::registerBuiltinModule()`.

Ozcalls: calling Oz code from C++

This a great advancement in Mozart 2: Oz code can now calls C++ code!

Two types of calls are available: the lightweight `asyncOzCall()` which calls the given *callable* in a new thread and `ozCall()` which calls Oz code synchronously, waiting for the call to return.

The second one is quite complex to implement as we need to keep state while waiting the result of the call, yet letting other threads run.

Output arguments need to be wrapped in `ozcalls::out()`, as other arguments are expected to be as input.

We will use this feature for sending values on a VM port in the next chapter.

6.5 Pickle

The `Pickle` module allows to serialize immutable values. These values are therefore conceptually separate from the VM and it is a tool of choice to transfer entities between VMs.

The `Pickle` module in Mozart 2 is in large part written in Oz. In particular, most of code to choose what to save on the stream is done by that Oz code in `Pickle.oz`. There is some low-level `Serializer` support for what cannot be done without support from the VM.

The format of Pickle in Mozart 2 is quite different from the format of Mozart 1, so be careful to not to mix them.

6.6 Atoms

The atoms in Mozart 2 are stored in a so-called atom table. This `AtomTable` is implemented using a crit-bit tree [Ber04], a prefix tree supporting fast lookup, insertion, deletion and a couple other operations. We only use the first two operations as the way we handle removal is by replacing the tree by a new empty tree at each garbage collection.

This means atoms are garbage collected, which is a nice improvement over Mozart 1! There are still some language implementations out there who do not garbage collect their atoms as they consider it too expensive or too complex to implement (they do not support atoms to move in memory). But it is then a perfect opportunity for leaks and potentially DoS attacks if any atom is constructed dynamically from an external source and there is not limit to that creation. The programmer must then care about each string to atom conversion to ensure only a known number of atoms can be created that way. Such language implementations include reference implementations of Erlang and Ruby.

Atoms are obtained by calling `vm.getAtom(string)`, which finds or inserts the given string in the atom table.

6.7 The store: a collection of nodes

The store in Mozart 2 is really the core of the implementation. Each entity is saved in a `Node` (using the graph terminology because entities can be linked to each other circularly) and there are different kind of Nodes to express the different needs of the entities and their storage. Essentially, the garbage collector only cares about nodes, and the mechanism to copy them is implemented in a class appropriately named `GraphReplicator`.

So a `Node` is always exactly two memory words: one for the type, a pointer to a `TypeInfo` instance and one for the value.

If the values fits a single memory word (this is the case for booleans, small integers, double-precision floating point numbers on 64-bit platforms, atoms and a few others) then that is all there is for the node. If not, the "value" is a pointer to some external memory which contains the actual in-memory representation of the entity.

As a node is at least two memory words, a nice trick is used by the `GraphReplicator` to store the list of nodes to copy inside the destination nodes themselves, so no additional storage is necessary for that list (which is very useful for the GC as we might be short on memory and avoids fragmentation if the list was saved separately in the new space). This works as before the graph replication, the destination nodes are not yet the copy of the source entity, and their two memory words can be used to store a pointer to the source entity and a pointer to the next node in the list.

`Node` is an abstract class and has two subclasses: `StableNode`, which is a node guaranteed to never change and `UnstableNode` which may change over time. Nodes have a strong identity and may not be copied. But `UnstableNode`s may be *moved*, that is transit from one `UnstableNode` to another, leaving the original in a valid but empty state.

Nodes can also be of type `Reference::type()` so there are a reference to another node. This provides the flexibility needed for the variable bindings allowed in Oz.

To traverse these references, the `RichNode` class is provided. A `RichNode` is just a pointer to some `Node` which transparently deferences it if it is a reference and allows to use the actual node by checking its type and casting it to a representation of its real type so specific methods of the type may be used.

For example, if we take a simplified definition of `SmallInt::add`, we notice immediately a particular usage of nodes.

```
UnstableNode SmallInt::add(VM vm, RichNode right) {
  int a = value();
  int b = getArgument<int>(vm, right);
  return SmallInt::build(vm, a + b);
}
```

Input arguments to functions are `RichNode`s as we need to access them, and the output is an `UnstableNode`, to represent the fact the caller must handle the storage and may move the contents to another node.

We notice something very similar in Builtin definitions, input arguments are of type `In`, which is just an alias of `RichNode`, and output arguments of type `Out`, which is just an `UnstableNode`.

6.8 Memory management

The memory management is implemented in mainly two classes in Mozart 2: in `VirtualMachine` and in the two `MemoryManager`. The part in `VirtualMachine` is to decide when to do the garbage collections and how the space sizes should evolve.

Each `MemoryManager` has a single *base block* of a given size, which is used as a memory pool. That is, most allocations are simply done by incrementing a pointer which describes the next available location in the block, and when the block is getting full, it is the job of the garbage collection to make room. The release of memory is done by discarding a pool entirely at once.

Another allocation scheme is present in the `MemoryManager` with the `malloc()` and `free()` functions, which are used for *managed* allocations. This is mainly used for dynamic buffers and internal structures of the VM.

`malloc()` use the *base block* initially but prefers to reuse a block of the *free list* if one is available. `free()` adds the block in the *free list* for further usage by another allocation. The *free list* is composed of power-of-two block sizes, ranging from 2 memory words (the size of the smallest Oz entity in memory) to 128 memory words (1024 bytes on 64-bit). Bigger blocks are allocated and freed using the regular `std::malloc()` and `std::free()` functions, as they are rather rare.

Mozart 2 use two memory managers when doing the garbage collection, as it uses a *copying* garbage collection algorithm. The second memory manager was used for relatively important allocations, yet ones that we have the guarantee they will be entirely released at the end of the current instruction. This happened for serialization and space cloning. Both of these operations need to create a potentially large structure to handle cyclic Oz structures correctly but can discard it once the operation is done. As we will see this usage of the second space has been removed with the ability to reduce memory usage when running multiple VMs.

6.9 Control flow: waiting on a variable and exceptions

The implementation of lightweight and *pausable* (which may be stopped at some safe point and resumed from there) threads is often a very challenging problem in virtual machines.

Most operating systems support threads but these are *heavy* threads which use lots of memory and other resources, as well as taking significant time to be created as they are not far from full processes.

The common way to implement lightweight and pausable threads is to use the low-level C functions `setjmp(3)` and `longjmp(3)`. These are as powerful as `GOTO` between functions. They are used for brutal jump of the program counter such as unwinding the stack (which is needed to implement exceptions) or even to implement continuations. But there is a considerable problem by using them in C++. They do not do any bookkeeping of structures held by the current functions nor its callers, destructors are not called and it may very well results in leaking resources. Normal exceptions in C++ care to call all needed destructors by properly leaving every scope but there are much much slower.

So Mozart 2 takes an interesting approach in using these calls to implement the wait on an unbound variable (which requires to *pause* the current thread until the variable is bound). Oz exceptions are also implemented by this technique and require to unwind the stack and continue execution once the exception handler is found.

This means the code leading to a `{Wait X}` operation or to raising an Oz exception at the C++ level must be *idempotent*, that is have the same result if run once or multiple times. We obviously would like that no important side effect happens more than once. This seems a rather strong requirement but in practice not so much. First, many operations are naturally idempotent, because they are *pure* and do not interact with any external entity (such as `A + B`). These operations may ask for some memory for some intermediate nodes, and then have to *wait* or *raise* an Oz exception, but this is not an issue as we have a pool allocation scheme and so when the GC happens, it will not copy these useless nodes. Finally, a function named `protectNonIdempotentStep` can be used to protect non-idempotent parts of the code as a last resort.

We saw how preemption of a thread due to waiting on a variable or raising an Oz exception is implemented. But preemption of threads in general is quite different. Before each call, a thread checks if it should be preempted. This first checks a flag which is either set by a timer which fires every millisecond (normal preemption) or by the environment so the thread emulation loop is left to handle some external event. If that is not case, then it checks if garbage collection should be performed.

6.10 Organization

It is about time we discuss the organization of the virtual machine implementation.

The first and very clear distinction is between the `VirtualMachine` itself and the `VirtualMachineEnvironment`.

All the code for the `VirtualMachine` must have absolutely no dependency so it can be easily ported to many architectures, providing C++11 code can be compiled for the given architecture. All that C++ code is contained in `vm/vm/main` in the repository.

On the other hand an environment may have some dependencies, such as the current environment for the `ozemulator` binary – the `BoostEnvironment` – depends in the C++ library Boost. Environments inherit from `VirtualMachineEnvironment`, which provide sensible defaults so a simple environment can be implemented with few code. This is the example of the `TestEnvironment` used for testing the VM in C++ which requires only 40 lines of code! This makes the environment the ideal extension point for anything which has an external dependency.

The main VM code simply replaces its existing behavior be using the environment for that operation and the `VirtualMachineEnvironment` simply replicates the default behavior. Files of the `BoostEnvironment` are located in `vm/boost/main`. The definition of the `main()` function is in `boosthost/emulator/emulator.cc`.

There are three kind of files. The *headers* files which have a basename ending in `-decl.hh` are the usual C++ headers declaring most classes. The *implementations* files, the other `.hh` files. And finally the *big-algorithms* files, ending in `.cc`. This choice to move away from the traditional two kinds of headers/implementations files is to provide better inlining and run-time performance at the cost of a slower compilation. The fact almost all *headers* files are easily included in *implementation* files makes it easy to use the different classes public methods and keeps a clean file separation for the forward-declared classes.

6.11 Building the system

Mozart 2 has a few dependencies as listed in the README [DJ+13]. There is in fact a README per platform as some things change significantly between the different platforms.

Here are the dependencies:

Git and SVN are used to fetch the source of Mozart 2 and some dependencies built from source.

CMake is the chosen build system.

Boost The C++ library is used exclusively in the `BoostEnvironment`, which is one of the possible concrete environment for a Mozart 2 `VirtualMachine`. It provides some functionality which did not make it yet into standard C++ or is rather specialized. Many standard functionalities in C++11 are drop-in replacements for Boost's equivalent. The main libraries used at the moment are asynchronous IO, time conversion, UUID generation, option parsing and multiprecision (big integers).

LLVM/Clang is used in the generator to parse correctly C++ files and generate appropriate code for the different classes inheriting `DataType` and the interfaces. It notably generates the different `TypedRichNode<T>` classes. In many cases it needs to be built from source to preserve the Clang AST headers which are not present in many distributions of LLVM/-Clang.

Java is used to build and execute the bootcompiler (written in Scala, necessary as Oz's compiler is written in Oz). The Scala support is downloaded automatically, one only needs a working JVM and Java compiler.

GCC or Clang to compile the VM. GCC should be preferred except on Mac OS.

Tcl/Tk for the graphical part of Oz.

Emacs for the Oz Programming Interface.

GTest Google Test, a test framework used for testing the C++ part of the VM. The build system handles the automatic build of gtest if not given in the configuration.

GMP The GNU Multi-Precision library is an optional dependency. Boost's multiprecision may use it if available which may significantly increase big integers performance.

Build

The README should describe the exact procedure to follow to build Mozart 2.

However, be aware that the built LLVM/Clang has typically some trouble to find the system C++11 headers so one must often help it by setting `MOZART_GENERATOR_FLAGS` or `CMAKE_CXX_FLAGS`.

6.12 Big integers

As the final section of this chapter we consider the addition of big integers to Mozart 2 to summarize a few concepts we just saw with a concrete addition to the code base.

Mozart 2 before this thesis had no support for big integers (that is, integers which can not be represented as a single memory word). We added support for `BigInt` as an exercise to familiarize ourselves with the Mozart 2 codebase.

As we want to keep any big dependency outside the core code of the VM, it also means we need to interact with the Boost environment, which provides the necessary *multiprecision* library.

The first obvious step is to add a `DataType` named `BigInt`, which has value behavior and is no different in any way from a `SmallInt` in Oz. We then need to implement the `Comparable` and `Numeric` interfaces and the backend to handle these operations. The final step is to fully integrate `BigInt` in the codebase by adapting the necessary conversions from or to strings, floats, etc and use the environment to create big integers. The basic `VirtualMachineEnvironment` raises overflow exceptions if one tries to create a `BigInt` has it has no support to do so.

Memory Management

Properly handling the memory used by `BigInt` is not trivial.

The `BigInt` C++ class has a member named ‘value’ which is a `std::shared_ptr<BigIntImplem>`. A shared pointer is a smart pointer which automatically does reference counting. `BigIntImplem` is an abstract class (or rather an interface) defining the different methods a BigInt implementation should provide. `BoostBigInt` implements this interface and its only member is a `mp_int`, the concrete representation of the big integer. But of course, the memory used for the big integer is not all hold in that field, as the field has a fixed size but big integers do not. So a `mp_int` is probably just some metadata and a pointer to some external memory. In some optimized cases it might be allocated on the stack but in general it is allocated on the normal C++ heap.

Deallocation works fine, as the destructor of a `BigInt` is called after a GC if the `BigInt` is no more reachable, which calls the `std::shared_ptr<T>` destructor, which itself calls `BoostBigInt`’s destructor which is responsible to free the external memory. The shared pointer now frees the `BoostBigInt` and we are done.

Ideally a big integer would also be saved in the VM heap as other entities and their values. But this is a dangerous thing to do considering the few guarantees we have about memory usage of the big integer library and how we move memory. That library is either Boost multiprecision’s own big integer library or the GNU Multi-Precision library.

The GC copies entities from one space to another and expects, once it is done, that absolutely nothing uses memory from the old space. We do not have this guarantee if, for instance, the big integer library chooses to share some data between two big integers or expects for some data it allocated to have a dynamic storage duration spawning across GCs. In these cases the copy of the big integer done by the GC might not be entirely contained in the new space and still reference some memory in the old space which would be fatal.

Boost multiprecision's own big integer library allows to pass a `std::allocator` [MK] and GMP allows to set the memory functions it uses [GMP]. We could think to define the allocation function as `VirtualMachine::malloc` which uses the VM heap, deallocation to be `VirtualMachine::free` and reallocation to be a combination of these two with some copying. Yet, nothing guarantees we can expect any of these library to stop using memory in the old space – that is, to have called `free()` on every allocation done in the old space – once the copy of big integers is done.

Chapter 7

Adding multicore support

7.1 No global state

The first and the most important change when implementing multiple virtual machines in the same operating system process with multiple threads is to remove any form of global state. It is unsafe and undefined behavior to manipulate anything with different operating system threads which is not synchronized and can potentially change[1].

Hopefully, Mozart 2 has been designed with multi-VMs in mind, and almost every state is attached to a particular VM. This is achieved in most cases by explicitly adding an explicit `VM` parameter to every function.

However, there was some global state left in `BoostBasedVM` and of course in the `main()` function. The first step was to remove this global state and replace it with references to a particular VM. Along this change, `BoostBasedVM` became two classes: a `BoostVM`, extending `VirtualMachine` and a common `BoostEnvironment` for what could be global (either immutable or synchronized).

Process considerations

We must be careful to take care of replacing anything that could kill the entire process which is a deadly form of global state change, as we want only the concerned VM to terminate even in exceptional situations such as out-of-memory. So any call to `exit()`, any signal representing an exception sent to the process, any exception that could be uncaught must be handled to only terminate the current VM.

Generalizing this idea, most changes should only affect a given VM and therefore only the corresponding operating system thread.

This include the current working directory which is defined per process in POSIX. But Mozart 2 does not provide a way to change it as it is a rather deprecated operation in a concurrent environment so this is already solved.

File descriptors are another example which we solve differently. Instead of exposing directly file descriptors (integers) we have a wrapper around it which makes them mutual secrets between the VMs as they are not serializable. The operating system already handles concurrent operations on a file with different descriptors so we need not to protect these operations.

[1]We consider here that atomic operations are always implicitly synchronized.

For the Tk widget toolkit – notably used to show entities in the Browser –, which the emulator uses as an external process we chose to create one per VM. The code already handled this as it creates the process via a pipe on UNIX or creates the process and communicates via a socket on Windows. However, on UNIX, a `SIGPIPE` signal was sent when the pipe was closed on the receiving end which by default killed the entire process. We fixed this by ignoring `SIGPIPE` and rely instead on the system call error `EPIPE`.

The process environment variables are a counterexample. We want to share these environment variables between the VMs as there are appropriate procedures to interact with them (`OS.putEnv`, `OS.getEnv`). The calls to the underlying C functions had to be serialized as they are not thread-safe.

7.2 Single IO thread

All virtual machines share a single IO thread, which performs all IO operations. This thread calls `boost::asio::io_service::run()`, which runs asynchronous IO tasks given to it.

For consistency, the initial process thread is doing the asynchronous IO. Each spawned thread hosts a VM and initializes it by itself.

7.3 VM ports and streams

The first choice we made was to allow sending only immutable values on a VM Port. As we have no distribution subsystem to handle mutable entities, there is no clear way how to send a mutable entity.

A good way to check for immutable values and at the same time ensure there is no reference to the sending VM in the received value is to use the `Pickle` module.

The simplest method to implement VM-level ports would be to use the existing TCP connections and pickle/unpickle on the stream.

Of course, we can do much better and we do the pickling to a separate memory block (yet in the same process of course), which the sender allocates and the receivers frees once the data has been received and unpickled.

One could think a direct copy of in-memory structures would be more efficient, but if done directly between the source and target VMs it would require synchronization. If done via a third neutral entity, no synchronization would be needed but we have to copy the subgraph to a “fake” VM, which does not make sense and is way more complicated than pickling.

Implementation of Send

Serialization is done by the `Pickle` module, defined in Oz and `Send` on a `VMPort` is done in C++ as `Send` is polymorphic and ultimately calls `VMPort::send()`. So C++ code needs to call Oz code, which is a perfect example for `ozCall()`!

```
UnstableNode result;
ozcalls::ozCall(vm, "mozart::boostenv::BoostVM::sendOnVMPort",
                picklePack, value, ozcalls::out(result));
```

`ozCall()` needs a key to remember its state, which is here the fully qualified function name. `value` is the message and `picklePack` is `Pickle.pack` which is found in the `PropertyRegistry` (the `Pickle` module registers it itself when loaded).

An alternative would be to use `ReflectiveEntity` but they ultimately also use `ozCall()` and introduce some overhead.

7.4 Memory management

For the implementation of the single backup space we discussed in section 4.7, the first thing to consider is the complex usage of the second space, which had to be removed, so we can limit the number of second spaces used at a time and depend on them only for garbage collection. The VM simply uses the primary space for such allocations as they should not exceed the primary space capacity as we keep some margin between the threshold and the actual capacity. If they exceeded the primary space capacity they will simply be allocated in temporary external allocations released immediately after the operation.

7.5 Synchronization

If we want our system to perform well and virtual machines to almost always run in parallel, we need to care to have as few synchronization and locks in the implementation of multiple VMs.

During the design phase, we cared to avoid any unneeded synchronization between the VMs as it might be costly and it usually reveals a bad pattern (acting on other's behalf without their consent).

Also, it is well known that adding locks greatly increases the complexity of a system.

There was only one place before this thesis where inter-thread synchronization was used, to coordinate the asynchronous IO thread and the main VM thread. That mechanism uses a queue of callbacks, so the IO thread just enqueues a callback when the IO result is ready, then the VM is preempted and takes care to call the callback before returning to the main emulation loop. To allow preemption from other threads, a few atomic flags are defined in `VirtualMachine`, which are checked regularly (at every call in the current implementation).

This is a very useful abstraction and we were even able to reuse it without modifications to add VM Ports.

A couple structures need to be protected when adding multiple VMs.

VM list

The first and obvious is the list of VMs. We need to keep such a list for various reasons. One is to know the list of alive VMs as it may be a very useful information. Another is to find a VM from its identifier, as most procedures in the `VM` module take an identifier and operate on the corresponding VM. So we need at least some kind of mapping between VM identifiers and `VMs` and a way to iterate the `VMs`. Obviously, we need to synchronize usage of that structure, as at anytime a VM be created or terminated and therefore be added or removed from the structure.

Second memory manager

The second mutex protects and thus control the access to the second memory manager, so only one VM uses it at a time, for its garbage collection. We might have some contention there, and it would be interesting to study whether having a pool of second memory managers would be worth the extra memory used.

Monitor lists

The third synchronized structure is the list of monitors of a VM. We protect it by a mutex so other VMs might add themselves in the list. This should never be a bottleneck as monitoring operations are rather rare and the mutex only synchronize the monitoring VM during a very short operation (the addition of a single VM identifier in a list).

Port closed

Finally, we also have an atomic flag for querying wether a VM port is opened or closed, so we do not bother to do anything when sending a message on a closed VM port.

7.6 Proper termination

When living in a multi-VM world it would be nice if terminated VMs would at some point completely disappear from memory, and all related structures be deallocated. It means that no one should ever access anything related to that VM later and some thread has to initiate the destruction of that VM and actually perform it.

Part III

Evaluation

Chapter 8

Evaluation

8.1 Tests

All features have been carefully tested by adding a test file to the original Mozart test suite. The file is located in `platform-test/base/vm.oz`. The test suite was not integrated yet to Mozart 2 and the original runner not entirely compatible. We made some small changes to make the runner compatible but also designed a runner ourselves which allows to get feedback much more quickly (it compiles and executes the test directly, instead of going through three separate command line invocations including two compilations). We also added support to show the operands when a unification fails to help understanding the cause.

There is a test case per procedure of the `VM` module and some additional tests to test them together. Each test case takes care to test for erroneous and exceptional cases. For instance, the three procedures taking a VM identifier as input test for invalid identifiers and identifiers of dead VMs. All terminations reasons are tested: *outOfMemory*, *exception*, *kill* and *normal*.

It is worth noting Oz naturally provide a nice equality assertion with unification (`=`, which works on all entities, structures included) and gives an easy way to fail a test with `fail`. As an example, we show a part of getPort's tests.We differentiate the unifications used for assignation from assertions by using a convention of using surrounding spaces for assertions.

```
getPort(proc {$}
            P={VM.getPort {VM.current}}
            S={VM.getStream}
         in
            {Send P hello}
            S.1 = hello

            try
               {VM.getPort 12345 _}
               fail
            catch error(vm(invalidVMIdent) ...) then
               skip
            end
         end)
```

8.2 Performance

We would like to evaluate the performance of our work.

Mozart 2 only had a built-in wall-time clock with 1 millisecond precision. Mozart 1 only has a wall-time clock with 1 second precision. Therefore we added a precise monotonic clock with nanosecond precision to Mozart 2 (the actual resolution is in a few dozens of nanoseconds). A monotonic clock has the big advantage that it is not affected by system time adjustments. We can of course still measure the total time to run a process with `time(1)` but it is not very accurate and we can not measure parts of the program.

Measurements were made on a server with a quad-core processor Intel Core i5-2300, each core with a clock speed of 2.80GHz. Several measurements were made for each benchmark to ensure stability and low error in the measurements. We took care of not measuring GC time by setting a sufficiently high minimal memory with `--min-memory` for the whole benchmark.

To achieve some statistical significance we take 30 consecutive measurements for each benchmark and report the mean and the maximal deviation (abbreviated *max dev.* in tables) from the mean. This means, with the simplest prediction interval that we have a probability of $\frac{N-1}{N+1} = 93.5\%$ for the next measurement to fall in the range of the measurements seen [Pre].

Most benchmarks are very fast to complete, so we also repeat the operation. The time we report is the time it takes for n iterations of the operation, divided by n, so that corresponds to a single operation given the iterations have no significant effect on each other. This allows us to analyze peak performance with a warmed-up system. As we have a precise clock we also do the benchmark with a single iteration to check the consistency of our results.

8.3 Spawning a VM

One thing we definitely want to measure is the time to spin up a new VM in the same process. And we would like to contrast that to spawning a new process, which is what was done in Mozart 1 to use multiple cores.

We do multiple benchmarks. The code is in `benchNewVM.oz`.

spawn is measuring the time to a create a new VM for the initial VM. It does not take account the start-up of that other VM as another thread does that. Only the construction of the C++ `VirtualMachine` instance is done by the calling thread. All the initialization process is done in the new thread.

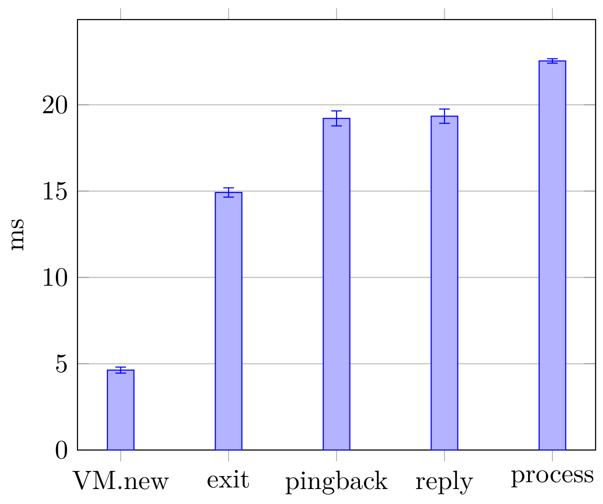
exit measures the time to create a VM and to wait until its termination, that VM doing nothing. As a parent VM always monitor its child VM, we simply wait for the corresponding termination message.

pingback creates a VM which will send back a simple integer on the initial VM port and waits for that message to appear on the stream of the initial VM.

reply measures the time to create a VM, send on its port the atom `hello`, to which the other VM will reply with the atom `world` once it received the `hello` and to wait to receive `world` on the initial VM stream.

process spawns a new Mozart 2 emulator and waits for its termination.

Looking at Figure 8.1, we see it takes less than 5 milliseconds to create a new VM and less than 20 to interact with it and terminate it. This is rather satisfying! The jump from 5 to 15



benchmark	time (ms)	max dev.
VM.new	4.63	0.174
exit	14.92	0.270
pingback	19.21	0.435
reply	19.34	0.414
process	22.54	0.129

Figure 8.1: Results for spawning new VMs

between **VM.new** and **exit** is expected as the VM thread really starts its own VM by loading the Base functor. The termination process itself should be pretty quick but we do wait for the termination message to arrive in the initial VM. **pingback** and **reply** take 4 to 5 ms more than **exit**, which is most likely due to loading the `VM` module in the created VM. Finally, spawning a new **process** takes about 22.5 ms, which is slower than **exit**.

8.4 Sending messages on VM ports

We would like to know what is the cost of using VM ports and what usage is more efficient. We also wonder what is the performance of regular ports.

We analyze mainly 4 cases of port usage. As sending and receiving messages is fast, we send and receive N messages. N for regular ports is 10^6 and N for VM ports is 10^4. The chosen message is simply `unit` to prevent too high memory usage. *The times reported are the average for a single message.*

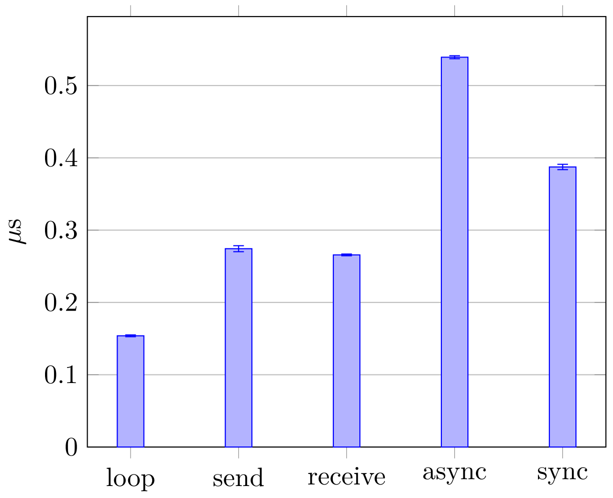
send Send N messages.

receive 'Receive' the messages by looping N times on the stream.

async Send N messages, and then after read them on the stream.

sync Send 1 message on the port, receive it and repeat that N times.

The performance of regular ports is reported in Figure 8.2. As N is very large we report the time for an iteration of an empty (`skip`) loop from 1 to N. We notice regular ports are very efficient, it takes less than one microsecond to send and receive a simple message! This is expected when you know sending a message on a regular `Port` is just binding the tail of the stream to `Msg|NewTail`. It might be surprising that sending all messages and then receive them (**async**) is slower than sending and receiving them one by one (**sync**). But this is actually caused by **sync** doing a single loop and **async** doing two! Subtracting the time for an empty loop from **async** time gets us very close to **sync** time $(0.539 - 0.154 = 0.385)$.

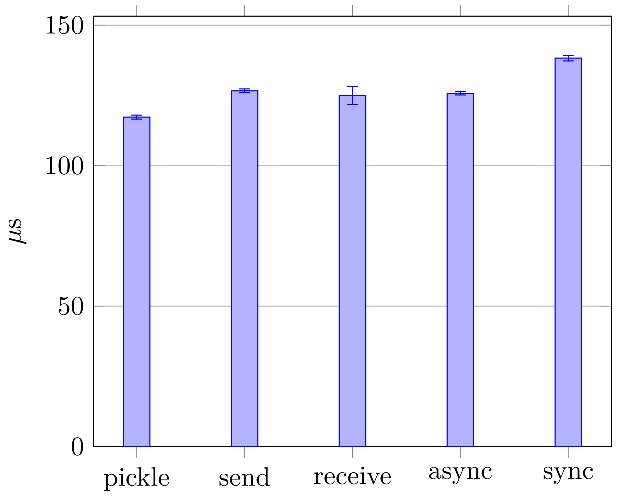


benchmark	time (μs)	max dev.
loop	0.154	$1.27 \cdot 10^{-3}$
send	0.274	$4.15 \cdot 10^{-3}$
receive	0.266	$1.22 \cdot 10^{-3}$
async	0.539	$2.10 \cdot 10^{-3}$
sync	0.387	$3.68 \cdot 10^{-3}$

Figure 8.2: Results for sending messages on regular ports

Now we look at VM ports with Figure 8.3. We replaced the loop benchmark which took an insignificant time by **pickle** which measures the time to `Pickle.pack` the message. We analyze Pickle further in the next section. We notice the vast majority of the time to send a message on a VM Port is due to pickling! We are about two orders of magnitude slower than regular ports.

We need to define precisely what the benchmarks do on VM ports as now they might involve multiple VMs. **send** sends messages on the VM Port of a VM which does nothing with them. **receive** measures the time in the created VM to receive all messages while the initial VM send all the messages. **async** measures the duration from starting to send all messages by the initial VM to receiving the last message in the created VM. And finally, **sync** is the time for the initial VM to send 1 message on its own port and receive it on its own stream, repeated N times.



benchmark	time (μs)	max dev.
pickle	117.28	0.71
send	126.63	0.68
receive	124.92	3.19
async	125.73	0.56
sync	138.27	1.00

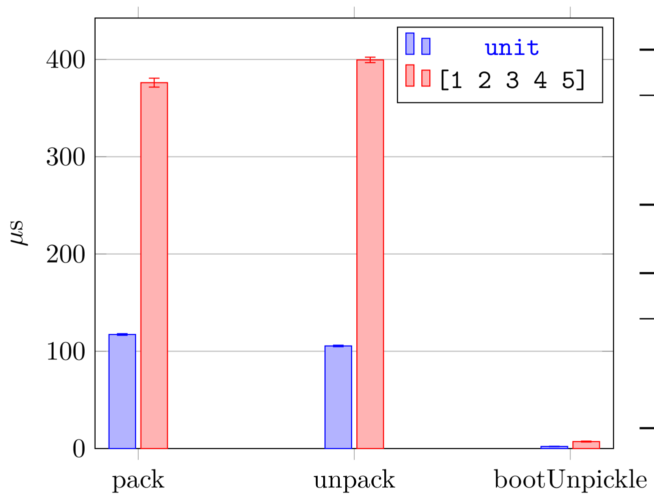
Figure 8.3: Results for sending messages on VM ports

There are some interesting results here. First, **receive** takes about the same time than **send**.

This is because we start receiving at the same time than sending and sending is much slower. Had we started to 'read' messages once they all had been sent, we would be in the regular port **receive** case. Unpickling could take some time though, but it would be done in parallel by the receiving VM and in our case we used the boot unpickler, which is very fast (about 2 μs/message). **async** is very close to receive but it measures the time from sending the first message to receiving the last, thus only counting the additional time to send the first message. When we compare this to regular port we see there is no significant increase in processing the stream, because it is done in parallel. Finally, **sync** involves no parallelism and the VM keeps interrupting itself to receive the message it sent, which is why it is slightly slower. In light of this we look at pickling and unpickling as a big part of sending message is serialization.

8.5 Pickle

We analyze pickling and unpickling of two entities: the simple `unit` we used before and the list of integers from 1 to 5. **pack** measures `Pickle.pack` and **unpack** measures `Pickle.unpack`, both being mostly written in Oz. **bootUnpickle** is using the boot unpickler, written in C++, used for loading the base modules, including Pickle itself. It is normally not accessible to Oz code but we made it for the benchmark.



Pickling `unit`

benchmark	time (μs)	max dev.
pack	117.24	0.83
unpack	105.47	0.76
bootUnpickle	2.13	0.18

Pickling the list `[1 2 3 4 5]`

benchmark	time (μs)	max dev.
pack	376.14	4.58
unpack	399.54	2.78
bootUnpickle	7.16	0.48

Figure 8.4: Results for pickling `unit` and the list from 1 to 5

If we take a look at Figure 8.4, the first thing we notice is the boot unpickler is so much more efficient than the regular one. There does not exist yet a more efficient pickler for Mozart 2 but it would be very interesting to analyze how it performs as it could likely decrease the time to send the simplest message on a VM port by 100μs. Bigger messages would have an even bigger gain as we see pickling is slower as the input is more complex. This is of course expected as there is more to handle and produce.

About pickling and unpickling

Note pickling and unpickling is not always a conversion between two formats but rather between three if we consider the boot unpickler or VM ports. `Pickle.pack` and `Pickle.unpack` respectively output and take as input a `VirtualByteString`, which is an Oz concept for binary virtual strings, that is tuples (with label `#`) of entities which can be simply dumped to raw bytes, the third format, represented as a `CompactByteString` in Oz. The conversion from `VirtualByteString` to a raw sequence of bytes is done by the C++ version of `VirtualByteString.toCompactByteString`.

A `VirtualByteString` might still be composed of a lot of entities which depend on the sending VM so we actually need this raw bytes representation for sending messages between VMs and whenever we want to go outside the sending VM, such as saving a pickled value on disk. The boot unpickler takes a (`std::istream`) as input, which may be constructed from a `std::string`. The raw bytes sequence is represented in C++ as a `std::vector<unsigned char>`, so we also need a copy it to a `std::string` (but it should be quite fast compared to other operations). So, when sending messages on a VM port, we additionally need to convert the result of `Pickle.pack` to raw bytes. When unpickling, we only need a conversion between two similar C++ representations as we use the boot unpickler. But both of these operations are much faster than the actual pickling done by `Pickle.pack`.

Chapter 9

Future work

The foundation for multi-VM programming in Mozart 2 has been laid out. But, as always, there is still plenty opportunities to improve it. We describe here some relevant improvements that could be made.

9.1 Transfer on VM ports

As one tries to use multiple VMs to parallelize some non trivial task, the efficiency of communication comes very often into play. There are multiple way to communicate between VMs and each will have different performance characteristics.

Pickle

Serialization with the `Pickle` module is what is used in our implementation. It is simple because it already exists, does the stateless check we need to ensure proper behavior until proper distribution of entities is available and it is reliable as it is the format used to save functors on disk.

Yet, in the current implementation is rather slow as we just saw, especially when compared to Mozart 1. A good part of this might be due to the fact `Pickle` is mostly defined in Oz in Mozart 2. One significant way to improve the efficiency of communication between VMs would surely be to improve the efficiency of pickling and unpickling. Reducing the time to pickle a simple `unit` to 10μs [1] in our benchmark would imply the message to be sent in $send - old\ pickling + new\ pickling = 126.63 - 117.28 + 10 = 19.35\mu$s, a gain of 84.7%!

Copying a part of the entity graph

An alternative would be to consider doing the copy of the entities at the C++ level. If we want to copy directly between the sending and the receiving VMs, we need to synchronize them. The cost of synchronization might be quite high, yet it would be interesting to try it.

A probably better solution would be to copy from the sending VM to a neutral zone dedicated for the particular message, and then from there the receiving VM would copy it to its own space. This is very much alike to Pickle, except doing it at the C++ entities graph level, which requires much less packing, parsing and other conversions of representation.

[1] It should be doable, Mozart 1 takes $1.06 \pm 0.02\mu$s to pickle `unit` on slower hardware.

Mozart 2 already has a class named `GraphReplicator` for replicating a part or the whole graph of entities. The garbage collector uses it as well as the space cloner. The garbage collector in particular copies the entities from one space to another. We would need something similar, but where we can specify the starting node of the subgraph and ensure there is no reference to the original VM in the copy. There are very few entities who keeps a reference to a VM directly so that should be relatively easy.

Another concern would be to allocate these memory blocks for messages and use the right size. If we want to avoid locking, then we need to have one memory block per message, which might be very expensive in the number of allocations. If we accept some contention, then we could use one block per pair of sending and receiving VMs or one block per VM. If we can compute the exact size needed for the copy, the duration to hold the lock might be very small.

9.2 Port.sendRecv

Although this is still about inter-VM communication, `Port.sendRecv` is about expressivity.

`{Port.sendRecv Port Msg ?Ack}` has been added to the `Port` module due to the rules of computation spaces forbidding a subspace to bind a variable of the parent space. We are in a similar situation as we cannot transfer an unbound variable on a VM Port.

The usage looks like this in the sender.

```
local
   P={NewPort Stream}
   Reply
in
   {Port.sendRecv P SomeMsg Reply}
   {DoSthWith Reply}
end
```

The receiver will receive the pair and replies by binding `Ack`.

```
for Msg in Stream do
   case Msg
   of Input#Ack then Ack={Compute Input}
   ...
   end
end
```

Notice that for regular ports not involving subspaces, `Port.sendRecv` can be defined very simply.

```
proc {SendRecv P Msg ?Ack}
   {Send P Msg#Ack}
end
```

It actually is a very useful form of bidirectional communication as it is very expressive. We need no more to filter the VM stream to find the reply, we can just wait on the variable to be bound!

And it would not be so hard to implement. One would likely start by creating a 'foreign' `ReadOnlyVariable` data type, which would behave as a standard `ReadOnlyVariable`. Except that, when bound, it would send its pickled value to the other VM, which would bind its local `ReadOnlyVariable` to the unpickled value.

`Port.sendRecv` would then return a standard read-only variable in the calling VM and insert it, in a VM-wide weak (to not leak) dictionary under a generated identifier, to find it back when wanting to bind it. It would then pickle the `Msg` and send along it the generated identifier. The receiving VM would unpickle the `Msg`, create a `ForeignReadOnlyVariable` with the given identifier and deliver that to the VM stream as `Msg#Ack`. When `Ack` would be bound, its implementation would tell the other VM to bind its read-only variable. Finally, one should care about binding between variables so the behavior of telling the first VM is kept even if the foreign read-only variable is bound to another dataflow variable.

9.3 Contention

As we saw in the implementation chapter, there are relatively few critical regions. However, the only global structure of the process, the list of VMs, is susceptible to contention. Currently, a simple mutex is protecting the list. All operations need protected access, but most operations do not modify the list. Only creating and terminating a VM modify the list.

Therefore it seems it would be a good idea to use a read-write lock, so the lock can be shared amongst readers. Concretely, the C++14 standard adds a `std::shared_timed_mutex` which could be used for this. Readers could use a `std::shared_lock` and writers use a `std::unique_lock`.

Another option would be to define the list as a lock-free data structure, if it turns out to be feasible and actually faster. If we were using a singly-linked list and we did not try to remove elements it would be relatively simple to implement as there is a well known algorithm to insert at the head of the list using compare-and-swap [Har01]. But adding the ability to delete elements properly makes the problem a lot more complex.

9.4 Module references

Currently, when trying to pass a module such as `VM` or `OS` directly to a newly created VM (not importing them in the functor) it results in a failure as they reference mutable entities. This might be surprising for `VM` when you know most procedures are simply built-in procedures defined in C++ and these built-in procedures are very easy to pickle, they are serialized as simple pairs containing the module name and the built-in name (e.g.: `VM.current` is serialized as (BootVM, current)). But what happens when we use `VM.getPort` such as in figure 9.1 is the whole `VM` module is serialized, because Oz sees it as a reference to some entity 'VM', and using the value at feature `current`. And that fails, because `VM` imports `Pickle` (for serializing inter-VM messages), which imports `OS` which contains some mutable entities. And even if we never used such dependency they are all loaded when trying to pickle the `VM` module as, Pickle, when seeing any entity referencing read-only variables (futures) tries to load and evaluate them as it cannot transfer such unbound variables.

This is perfectly reasonable for many entities and functors in general, but a module is localized and should be easy to load in the other VM from the same URL. Or the compiler might be able to detect `Module.Feature` and transfer only that value instead of the whole module.

There is a workaround however: use a local reference of the module, by importing it in each virtual machine using it. In this particular case we could move `SendToMaster` inside the functor given to `VM.new` and we would not need to pass `VM` as an argument. `VM` would be a reference to the VM module imported in the created virtual machine and not in the initial virtual machine. In general though, if the procedure using the module needs to be used by many virtual machines, we need to pass the reference to each module as additional arguments, which is shown in the figure and far from ideal as callers now also need to pass the modules.

One might wonder if dynamic linking with Module.link could not come to the rescue. A procedure would then import all its module dependencies using Module.link. It might be slower but it could partly solve the problem. The module manager also ensures the module is loaded only once. Currently it is still problematic as Module itself cannot be serialized so we would need to pass Module as an argument, which is only advantageous if we have many dependencies (one arguments instead of many).

Note that distributed Oz also has this problem for sharing modules. We believe the right solution would be to special case the pickling of modules so they are loaded from their location (URL) and not dumped as any other entity. With distribution it might more complex as the module might not be accessible by that URL. But for multi-VM in the same process, the assumption that the module still exists on disk or at some external location when the sub-VM tries to import it seems reasonable.

```
functor
import
   System(show:Show)
   VM
define
   Master={VM.current}

   proc {SendToMaster VM Msg} % VM is needed here
      {Send {VM.getPort Master} Msg}
   end

   {VM.new functor
           import VM % necessary to pass it to SendToMaster
           define
              {Show hello}
              {SendToMaster VM hi}
           end _}

   {Show {VM.getStream}.1}
   {VM.closeStream}
end
```

Figure 9.1: Module references

Chapter 10

Conclusion

This thesis started with the audacious project of making the Oz programming language implementation supports multicore processors. We believe we achieved that goal successfully and laid out the proper foundations. We took the time to design a well-thought solution to the multicore support, to evaluate it and to discuss the possible further directions, as any work can ever be improved. We also made various improvement to Mozart 2 and learned a handful of ideas along the way.

Many people tried making multi-VM programming language implementations and many failed because the existing implementation did too much assumptions. We managed to succeed in great part thanks to the clean design of Mozart 2 envisioning multi-VMs support. We acknowledge highly the work of Sébastien Doeraene, Yves Jaradin and others contributors of Mozart 2 for providing such a nice implementation.

We would also like to thank all contributors to Oz for a very singular and well-though programming language, which keeps surprising us by its unique techniques. May it live long and inspire many ideas to language designers!

Bibliography

[Bar14] Blaise Barney. *Introduction to Parallel Computing.* Lawrence Livermore National Laboratory, 2014. `https://computing.llnl.gov/tutorials/parallel_comp/`. 9, 15

[Ber04] D. J. Bernstein. Crit-bit trees. `http://cr.yp.to/critbit.html`, 2004. 49

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11), November 1970. `http://people.cs.umass.edu/~emery/classes/cmpsci691s-fall2004/papers/p677-cheney.pdf`. 29

[Dal14] Benoit Daloze. Code outside the mozart 2 repository. `http://eregon.me/thesis`, 2014. 4, 17

[DJ$^+$13] Sébastien Doeraene, Yves Jaradin, et al. Mozart 2 repository. `https://github.com/mozart/mozart2/`, 2013. 3, 4, 45, 52

[DKH$^+$] Denys Duchier, Leif Kornstaedt, Martin Homik, Tobias Müller, Christian Schulte, and Peter Van Roy. *Garbage Collection Properties.* `http://mozart.github.io/mozart-v1/doc-1.4.0/system/node65.html`. 30

[Doe11] Sébastien Doeraene. Ozma: Extending scala with oz concurrency. Master's thesis, Université Catholique de Louvain, 2011. 3

[Eri] Ericsson AB. *Erlang Run-Time System Application Reference Manual.* `http://www.erlang.org/doc/man/erlang.html`. 28

[GIL] Global interpreter lock. `http://en.wikipedia.org/wiki/Global_Interpreter_Lock`. 23

[Git14] GitHub. Contributions to mozart/mozart2. `https://github.com/mozart/mozart2/graphs/contributors`, 2014. 4

[Glo] Ravenbrook Limited. *Memory Management Reference – Glossary.* `http://www.memorymanagement.org/glossary/index.html`. 29

[GMP] GMP. *Custom Allocation.* `https://gmplib.org/manual/Custom-Allocation.html`. 54

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, 2001. 69

[HF08] Seif Haridi and Nils Franzén. Tutorial of oz. `http://mozart.github.io/mozart-v1/doc-1.4.0/tutorial/index.html`, 2008. 9

[Leg13] Vincent Legat. Syllabus of numerical methods. `http://www.uclouvain.be/32190.html`, 2013. 15

[MK] John Maddock and Christopher Kormanyos. *Multiprecision cpp_int back-end*. Boost. `http://www.boost.org/doc/libs/1_55_0/libs/multiprecision/doc/html/boost_multiprecision/tut/ints/cpp_int.html`. 54

[Moz] Mozart 2 website. `http://www.mozart-oz.org`. Mozart Consortium. 3, 45

[Pre] Prediction interval. `http://en.wikipedia.org/wiki/Prediction_interval#Non-parametric_methods`. 62

[Rec] Rectangle method. `http://en.wikipedia.org/wiki/Rectangle_method`. 9

[Sun08] Sun Microsystems. *Multithreaded Programming Guide*, 2008. `http://docs.oracle.com/cd/E19253-01/816-5137/816-5137.pdf`. 23

[TH$^+$05] Linus Torvalds, Junio Hamano, et al. Git, a free and open source distributed version control system. `http://git-scm.com/`, 2005. 4

[VR14] Peter Van Roy. Paradigms of computer programming. `https://www.edx.org/course/louvainx/louvainx-louv1-01x-paradigms-computer-1203`, 2014. 9

[VRH04] Peter Van-Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT press, 2004. 9