Mozart System Limitations for Version 1.3.0
(draft)

Peter Van Roy and Seif Haridi

April 28, 2004

"In der Beschränkung zeigt sich erst der Meister."
"*Within limitations the Master is first revealed.*"
– Johann Wolfgang von Goethe (1749–1832)

This article explains the differences between the ideal computer defined in *Concepts, Techniques, and Models of Computer Programming* and the Mozart system version 1.3.0.[1] For most practical purposes, the Mozart system implements all the computation models in the book exactly, respecting the semantics of Chapter 13 and the syntax of Appendix C. There remain a few differences between the language of the book and the Mozart implementation. These differences apply to Mozart on all platforms, Unix, Windows, and Mac OS X, unless indicated otherwise.

Improvement with respect to version 1.2.5

The main limitation that was removed since version 1.2.5 is the memory leakage due to unreachable identifiers in stack frames and closures. In version 1.3.0, garbage collection removes all unreachable identifiers from stack frames and closures. In version 1.2.5, the only way to remove these identifiers was to rewrite the program so that it does a last call whenever an identifier had to be removed (taking advantage of last call optimization). In version 1.3.0, programs do not have to be rewritten.

1 Graphical user interface

The standard GUI interface for Mozart is `tk`, which is called through the `Tk` and `QTk` modules. The `tk` process may stop working (no longer accept drawing commands from the Mozart process) if the Mozart process sends drawing commands at a rate that is too high. A typical example is opening the `Oz Panel` and then creating a large number of threads quickly, like in the Fibonacci example of Chapter 3. The `Oz Panel` updates its display whenever a new thread is created, which may cause the problem to appear.

2 Memory properties and limitations

Memory allocation is automatic: it is allocated when needed for data and recovered when the data is no longer needed. Most variable sizes, such as the length of lists, the number of

[1] For more information on the book see `http://www.info.ucl.ac.be/people/PVR/book.html`.

record fields, the size of arrays and dictionaries, the number of characters in an atom, the number of threads, and the precision of integers, have no upper limit except that imposed by available memory.

A Mozart process can grow and shrink. When the system is first started, it is several megabytes in size. During execution, it requests memory from the operating system according to need. If memory use decreases, then all unused memory is eventually returned to the operating system.

The following subsections list the few limitations that remain regarding memory usage.

2.1 Memory use

2.1.1 Maximum memory size

Mozart has been ported to 32-bit and 64-bit architectures. On 32-bit architectures, a program can have a maximum of 2048 MB of active data each time a garbage collection starts. The *active memory size* is the total size of all data structures that are needed by the executing program. In practice, the actual limit is somewhat smaller than this; it depends on how much garbage the program generates. To be precise, if the amount of garbage is g MB when a garbage collection starts then the maximum active memory is $2048 - g/2$ MB. This limit is due to the automatic memory management.

The distributed programming abilities give a crude but effective way to go beyond this limit. First use the `Remote` module to create new sites (i.e., operating system processes), as explained in Chapter 11. Then partition the application over these sites. Parts of the application that communicate heavily with each other should be put on the same site. If the partitioning is done carefully, this will have almost no effect on performance. Furthermore, because of network transparency, it will require almost no change to the application program.

2.1.2 Garbage collection and reaction time

Mozart implements a dual-space copying garbage collector. The execution time of this algorithm is proportional to the active memory size, not to the total memory size. While the algorithm executes, no other execution can take place in the process. If the active memory size is large, this can cause a long interruption in service.

There are two ways to guarantee that a program can react quickly to external events. The first is to make sure that no garbage collections occur during critical parts of the program. With care, this can be ensured by manually invoking garbage collection at non-critical times, using the zero-argument procedure `System.gcDo`.

The second technique is to use the `Remote` module together with Mozart's support for transparent distributed programming. With `Remote`, create a new site that only has a small amount of active data. This will guarantee that garbage collection on the new site is fast, so the new site can react quickly. The original site can be slow, as long as it delegates quick reactions to the new site. Mozart's distributed garbage collection algorithm has no effect on reaction time since it does not monopolize execution on a site.

2.1.3 Open distributed programming

The memory area used to store program instructions is garbage collected. This means that distributed programs can transfer procedure values, objects, classes, and functors between sites indefinitely without resulting in any memory leaks. This is particularly important for

open programs, such as client/server applications in which new clients can come and go indefinitely.

2.1.4 Compiler memory use

The compiler is part of the interactive run-time system. It may use large amounts of memory when compiling large programs or programs that access large data structures in the run-time environment. This memory use is temporary. After the compiler finishes, the memory is reclaimed.

2.1.5 Loaded modules are permanent

Modules are loaded on demand. If they are not needed, then they are not loaded and they will not take up any execution time or memory space. But once loaded and installed, a module remains in the system permanently.

2.2 Memory leaks

A memory leak occurs when there is a long-lived reference to a growing data structure that is no longer needed by the program. There are a three unusual situations that sometimes result in memory leaks: record arities and atoms, interactive variables, and flow problems. We explain these situations and show how to avoid the leaks. Of these three situations only the first, record arities and atoms, is due to a system limitation. We plan to remove this leak in a future release.

2.2.1 Record arities and atoms

All memory areas of the virtual machine are garbage collected *except* for the atom table and the table of record arities. We recommend to avoid as much as possible dynamically creating atoms (with `StringToAtom` or `VirtualString.toAtom`) or records with new arities (for example, with `Record.filter`). Tuples do not have entries in the arity table, so they are not subject to this restriction.

This does not affect most programs, since records (instances of record arities) are garbage collected. But programs that create new record arities indefinitely, such that the number of no-longer-needed record arities increases indefinitely, will have a memory leak. Programs that interface with databases, such that new atoms are created indefinitely, will also have memory leaks. In the latter case, we recommend the use of strings instead of atoms.

2.2.2 Interactive variables

Using interactive variables (created by **`declare`**) in program fragments may give the illusion of a memory leak. The same fragment in a standalone program will not necessarily leak. Interactive variables are pointed to by the global environment. They are always reachable unless there is another **`declare`** with the same identifier. If an interactive variable is bound to a growing data structure, then this data structure will not be reclaimed. For example, the following fragment declares `S`, `P`, and `Loop` as interactive variables, and therefore has a memory leak:

```
declare
S P={NewPort S}
thread {ForAll S proc {$ X} skip end} end
```

```
proc {Loop} {Send P foo} {Loop} end
{Loop}
```

Because the port stream `S` is always reachable, this fragment will quickly exhaust virtual memory. The following simple modification does not have a memory leak:

```
local S P Loop in
   P={NewPort S}
   thread {ForAll S proc {$ X} skip end} end
   proc {Loop} {Send P foo} {Loop} end
   {Loop}
end
```

Here, `S` is part of a local environment, which disappears because of last call optimization when `Loop` is called.

2.2.3 Flow problems

A memory leak can happen in a producer/consumer pattern if on average the producer produces elements faster than the consumer can consume them. If this goes on indefinitely, then the unconsumed elements will pile up, causing a memory leak. The solution is to limit the rate at which the producer produces elements, either by putting a buffer between the producer and consumer or by using demand-driven concurrency, where the consumer asks for elements by need.

3 Numeric limitations

There are a few limitations due to finite internal representation of numbers:

- Procedures can have no more than 4096 arguments.
- Integers of 28-bit or less precision (small integers) are stored in registers. Floating-point numbers and integers needing more than 28-bit precision (big integers) are stored on the heap. This means that calculations are significantly faster and memory-efficient when done with small integers rather than with floating-point numbers or big integers.
- Floating-point numbers are 64-bit precision, usually in the IEEE floating point standard.
- The range of a finite domain variable is from 0 to $2^{27} - 2$, i.e., from 0 to 134217726.
- The range of a finite set domain variable is from {} (the empty set) to $\{0, ..., 2^{27} - 2\}$.

4 Distribution limitations and modifications

The distribution subsystem of Mozart differs from the ideal model explained in Chapter 11. We explain these differences in terms of limitations and modifications. A *limitation* is an operation that is specified but is not possible or has lower performance in the current release. A *modification* is an operation that is specified but behaves differently in the current release.

The next major Mozart release, version 1.4.0, which is due late 2004 or early 2005, is planned to have a complete rewrite of the distribution support. All distribution protocols will then be part of a system component called the DSS (Distribution Subsystem). This will remove many of the limitations listed in this section.

4.1 Performance limitations

This class of limitations affects only performance, not the semantics. They can safely be ignored if performance is not critical.

- The code of functions, procedures, classes, and functors (but not objects) is always inserted in messages that are passed between sites, even if the code is already present at the destination. In future releases, the code will be copied across the network only if it is not present at its destination. In both current and future releases, this limitation does not cause a memory leak since at most a single copy of the code can exist per site.
- The distributed garbage collection algorithm reclaims all unused entities except for cycles of stateful entities on at least two different owner sites (a cross-site cycle). For example, if two sites each own an object that references the other, then they will never be reclaimed. It is up to the programmer to break the cycle by updating one of the objects to no longer reference the other.
- If a site crashes that has references to entities created on other sites, then these entities are not garbage-collected. Future releases will incorporate a lease-based or similar technique to recover these entities. In a *lease-based* collector, remote references are leased for a fixed time. The remote site must send a message asking for a lease renewal before the time is up. If the owner receives no renewal messages in time, it is allowed to conclude that there are no remote references.

4.2 Functionality limitations

This class of limitations affects *what* operations are available to the programmer. They document where the full language specification is not implemented.

- Objects created with `NewStat` or `NewAsync` should not use **`self`**. The simple fix is to add an attribute `selfattr` and initialize it with the object reference. That is, replace the following initialization:

  ```
  meth init
     skip
  end
  ```

  by this one:

  ```
  meth init(S)
     selfattr:=S
  end
  ```

  and create the object as follows (with `NewStat` or `NewAsync`, whichever is desired):

  ```
  S={NewStat MyClass init(S)}
  ```

  In each method that needs **`self`**, use `@selfattr` instead.
- On Windows, the `Remote` module has limited functionality. Only a single option is possible for fork, namely `sh`. Future releases will add more options.

- The `Connection` module can establish a connection across a firewall if the site that initiates the connection is allowed to communicate with the other site. This partially solves the firewall problem of Mozart 1.1.0. In a future release, this problem will be completely solved by making the connection protocol a user-defined Oz procedure.
- Threads, dictionaries, arrays, and spaces are sited, even though they are in Base modules. In future releases, it is likely that dictionaries and arrays will be made unsited. Threads and spaces will be made stationary entities that can be called remotely (like ports).
- When a reference to a constrained variable (finite domain, finite set, or free record) is passed to another site, then this reference is converted to a *read-only variable* on that site. The read-only variable will be bound when the constrained variable becomes determined.
- If an exception is raised or a handler or watcher is invoked for an *object*, then the `Entity` argument is undefined. For handlers and watchers, this limitation can be bypassed by giving the handler and watcher procedures an external reference to the object.
- If an exception is raised or a handler is invoked for an *object*, then the attempted object operation cannot be retried. This limitation can be bypassed by programming the object so that it is known where and in which method the error was detected.

4.3 Modification

There is currently only one modification:

- A handler installed on a variable will *retry* the operation (i.e., bind or wait) after it returns. That is, the handler is inserted before the operation instead of replacing the operation.