The Mozart Constraint Subsystem
System Presentation

G. A. Gutiérrez¹ A. F. Barco² M. A. Villanueva³ A. Cardona² D. Montenegro² S. Miller² J. F. Díaz³ C. Rueda² P. Van Roy¹

ICTEAM, Catholic University of Louvain, Belgium¹
Avispa, Pontificia Universidad Javeriana - Cali, Colombia²
Avispa, Universidad del Valle, Colombia³

Sep 16, 2013
Introduction
Context

New Mozart virtual machine

New implementation of the Mozart Programming System: Easier and transparent integration of different subsystems (e.g. Constraint and Distribution subsystem)

Our Goal

To contribute with the new Mozart implementation by integrating a state of the art constraint environment like Gecode.
Important Aspects to Consider

Several aspects of the implementation need to be taken into account

For Oz users

- Implementation of branching heuristics in Oz that will be used by the Gecode search engines
- Definition of search engines directly in Oz
- Support for programming paradigms like logic programming

For new Mozart implementation

- Garbage collection of constraint variables
- Interaction between threads and the constraint subsystem
At Low Level

Mozart Space

X 0
Y 1
S 0

Gecode Space

intvar 0#10 10#30
0 1

setvar {1}...{1,2,3,4}

boolvar
0

proc {CSP R}
  X :: 0#10  Y :: 10#30
  S = {FS.dom {1} {1,2,3,4}}
  in
  R = [X Y S]
  % Some constraints
  % Distribution
  end

  {Space.new CSP Sp}
Specific Objective

Efficient abstraction in Mozart that enables the interaction between constraint variables and

- Corresponding Gecode variables
- Other variables and language constructs

Integration

Every mozart space is associated with a corresponding gecode space. The idea is then that the gecode space encapsulates all the data and functionality of the constraint programming related operations.
Constraint Variable Declaration
Variable Support

We have to consider the interaction between the Mozart virtual machine, the Mozart space in which the variable is declared and its corresponding Gecode space where the variable is represented.
Steps

The following actions are performed when declaring a constraint decision variable

- A new variable in the Mozart space is created
- A new variable in the Gecode space is introduced
- An association between the two is maintained
Important Remarks

- The actions described above are independent of the variable type.
- Each Gecode space has a vector for each constraint variable type (IntVar, SetVar and BoolVar).
- One way association is enough to declare variables and to post propagators.
Constraint Posting and Propagation


The Mozart Constraint Subsystem
Posting propagators is achieved through a procedure call.

**Example**

Consider the simple example of posting constraint $X + Y = 5$. This constraint is posted by calling the procedure

```
{FD.linear post([1 1] [X Y] '=' 5)}.
```
Steps

1. For \([1, 1]\) and 5 create type compatible Gecode counterparts (IntArgs and int)

2. For \([X, Y]\), access the corresponding Gecode implementations and create an argument array of the corresponding type (IntVarArgs)

3. For \(=:\) create the correspondent Gecode relation (IRT_EQ)

4. Post the constraint in the gecode space:
   \[
   \text{linear(gspace,iva,ia,irt,c)};
   \]
Lazy vs. Eager propagation

Difference between the way constraint propagation is triggered

- Propagation in Gecode is executed \textit{on-demand}: Posting a propagator does not (necessarily) execute it
- Propagation in Mozart is executed eagerly: Posting a propagator executes it for the first time

Conclusion

The new Mozart implementation adopts the \textit{on-demand} propagation approach
Space Stability

Declarative Concurrency Support
- In Mozart, propagators and threads can synchronize on constraint variables
- Determining a fixed-point depends on both propagation and thread execution

Propagation thread
Space stability notion is changed to take into account VM threads and propagators: Whenever constraint variables exist there exists also an extra artificial thread, that runs propagation, suspending on an space variable called StatusVar
The Mozart Constraint Subsystem

Branching
Current decisions

- Current Mozart allows the use of built-in and user-defined branching strategies
- Initially, we will provide support for branchers written in Oz

Gecode branchers

Making Gecode branchers available from Mozart poses several design challenges (e.g. interaction between different kinds of branchers)
Example

```proc{CSP Root}
X := 1#10
Y := 2#100 in
% some constraints on X and Y here
{FD.builtinBrancher X}
{FD.ozBrancher Y}
end
```

Suppose that `{FD.builtinBrancher X}` in Gecode and that `{FD.ozBrancher Y}` in Oz. During the solving process the second brancher may become active and its execution will thus run Oz code from Gecode. How to do that in language and implementation-safe ways?
Optimizations

Batch recomputation support: Maintain an ordered collection of branchers, contrary to the Mozart old design!

Old stability assumption

Having only one active brancher is assumed for the stability check of Mozart spaces

Oz branches rely on
- They are specified by means of the choice construction
- There is only one brancher that is active at any given time
Blocking Semantics

Drawback: `choice` statement waits until the space in which it is executed becomes stable. The blocking semantics of `choice` does not allow other statements following it to be executed.

```plaintext
proc {CSP Root}
    X =: 1#10
    Y =: 2#10

in
    C = {FD.reflect.min X}
    choice X =: C [] X != C end
    D = {FD.reflect.med Y}
    choice Y <=: D [] Y >: D end
end
```


The Mozart Constraint Subsystem
Searching
Difficulties

Gecode engines will not be available in the first version of the integration

- Gecode search engines work on raw Gecode spaces (not aware of Mozart VM)
- Search engine creates new computation spaces: Garbage collection of unused spaces becomes a non-trivial problem
- Only those spaces representing solutions will be accessible from Mozart
- Investigating how these new spaces are to be associated to Mozart spaces
Oz defined engines: Use the Space module operations (garbage collection will not be an issue)

**Space.ask:** Runs propagation of the Gecode space until reaching a fixed point (Gecode Space::status)

**Space.clone:** Creates a copy of a computation space. In this case, copies of both the Mozart space and the Gecode space are created

**Space.commit:** Decides to use one of the alternatives present in the space
A fourth operation of the space module is

\[ V = \{ \text{Space.merge } S \}. \]

- The operation merges the variables and constraints of \( S \) with the space the operation is called from.
- This operation is not supported: There is no merge equivalent counterpart in Gecode.
- For solving CSPs in which information can only be derived by propagation (no threads) we provide an approximation called \( \text{Space.dataMerge} \)
Merging Data

Internally

Let us call $S_g$ the Gecode space associated to $S$, $T$ the space of $V$ and $T_g$ its corresponding Gecode space. Merging the data of $S$ into $T$ amounts to consider the constraint variables of $S_g$ and to add equality constraints with the corresponding variables of $T_g$

- Notice that there is no merging of propagators.
- The equality constraints represent their impact in the data of both spaces.
- The case of threads generating new information needs to be studied in more detail (forthcoming release).
Example

```haskell
proc {Most Root}
  S E N D M O T Y
in
Root = sol(s:S e:E n:N d:D m:M o:O t:T y:Y)
  {FD.dom 0#9 Root}
{FD.distinct post(Root)}
{FD.linear post([1] [S] '\\=:' 0)}
{FD.linear post([1] [M] '\\=:' 0)}
{FD.linear post([1000 100 10 1 1000 100 10 1 ~10000 ~1000 ~100 ~10 ~1] [S E N D M O S T M O N E Y] '=', 0)}
{FD.distribute Root size_max val_min}
end
proc {Obj O N}
  {FD.linear post([10000 1000 100 10 1 ~10000 ~1000 ~100 ~10 ~1] [N.m N.o N.n N.e N.y O.m O.o O.n O.e O.y] '>' 0)}
end
{Show {SearchBest Most Obj}}
```
Conclusions
Future work

- Support for sets, booleans and floats domains
- Space primitives waitStable and choose
- Support of the Gecode Interactive Search Tool (GIST)
- Support for solving optimization CSPs with optimization functions written in Mozart.
- Batch recomputation support
- Implementation of space merge
Acknowledgments

We would like to give special thanks to:

- Gustavo Gomez
- Carlos A. Martínez
- Julián Camargo
- Sébastien Doeraene

We also thank the anonymous reviewers for their comments and suggestions.