Mozart VM 2.0

Design of the new Virtual Machine of Mozart, an implementation of Oz

Yves Jaradin and Sébastien Doeraene
Université Catholique de Louvain
Roadmap

- Guidelines
- The Store and its memory layout
- Introduction to Oz and C++
- Implementation of the Store in C++
- Dataflow and threads
- Emulator, registers and instruction set
- Builtins
Roadmap (2)

- Some core data types, design and implementation
- Spaces
- Unification, structural equality test and deep pattern matching
- Garbage collection and cloning
- Serialization
- Conclusion
- Questions & answers
Part I

Guidelines
Guidelines

• Future-proof
  – Portable
  – Modular
  – Documented
  – In Oz when possible, in C++ if necessary

• Fast
  – Compiler friendly
  – Processor friendly
Future-proof is new

- Previous VM failed
  - 32 bits assumptions
  - Pervasive design decisions
  - Sparse documentation

- Maintainers failed
  - Compiler assumptions
  - Rotten code
  - Outdated documentation
Portability

• Pure C++11 code
  – Tested on clang 3.1 and gcc 4.7.0
  – No #ifdef's on platform or compiler

• No hypotheses on platform
  – Unavoidable hypotheses are encapsulated
  – Use C++ mechanisms when possible
Modularity

- Declarative coding
  - Don't repeat how
    - Using code generation
    - Using C++ abstraction mechanisms

- At many levels
  - Types
  - Files
  - Inclusion graph

- Extensible
Documented

- Cleaner code
- Presentations
  - In English...
- Wiki
Oz vs. C++

- Oz developers know Oz, not always C++
- Easier to break invariants in C++
- Debugging is easier in Oz
- Oz is more compact and expressive
Fast

- Compiler level
  - Few compilations, many headers
  - Statically known types if possible
  - Standard code => all optimizations enabled
- Processor level
  - No pointer hiding
  - Composition rather than reference
  - Control of indirections
Part II

From a conceptual view of the Store to its memory layout
The store as a graph
The 2-word structure

- Oz is a dynamically typed language
  - The type of values must be stored along with the value
- Entities can change type over time
  - There must exist some way of changing the type of a value, without losing the references to it, and potentially to a representation that needs more memory space
- Some values can be stored in only one memory word
  - We do not want to allocate external memory for them
- But some cannot
  - We can store a pointer to external memory
The 2-word structure (continued)

- Each value is stored as a pair of memory words, one for its type, and one for the untyped value
  - Such a pair is called a node
  - Sometimes the untyped value is a pointer to external memory

<table>
<thead>
<tr>
<th>SmallInt</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unbound</td>
<td>_</td>
</tr>
</tbody>
</table>
The 2-word structure (continued)

- The type field of a node used to be a tag (i.e., an enumeration value) in Mozart 1
- Now it is a pointer to runtime type information

![Diagram](image-url)
The store as a graph of nodes

Tuple

Atom
‘foo’

SmallInt
5

Unbound
_

Atom
‘bar’

Atom
‘foo’
Nodes inside data structures

- Since nodes are so small (only 2 words), it seems quite wasteful to have pointers to nodes everywhere (inside tuples, e.g.).
- We can squash nodes inside data structures, instead of having pointers to nodes
  - Usually it takes less space
  - And it is faster, because there are less pointer dereferencing
The store with squashed nodes

- Nodes lose their identity
- Not a problem for integers, atoms
- But a real issue for big data and unbound variables
References

- To fix the identity issue, we introduce references.
- A reference is a value of special type Reference, whose untyped value is a pointer to a node.
- Operations on the store transparently dereference such References.
- We need not introduce references for types that do not need a strong identity nor external memory (e.g., integers).
  - Such types are qualified as *copyable*.
The store with references

- References solve the identity issue
- Note that references can point inside big data structures
Managing mutability

- Both a Tuple and an Array contain nodes, but a Tuple is immutable while an Array is mutable.
- But references can point inside data structures.
- If the content of the array is changed, the reference is broken!

```
A.2 := 42
```
Stable and unstable nodes

- Fundamental difference: stable and unstable nodes
  - A stable node is part of an immutable data structure
    - Inside a Tuple, a Record, a Contextual Environment, etc.
  - An unstable node is part of a mutable data structure
    - Inside a Cell, an Array, working registers of the emulator, etc.
- Golden rule: no reference to unstable nodes
Changing a node's type (aka the “become” operation)

- 3 reasons for a node to change type:
  - A transient becomes bound
    - (related to Oz semantics)
  - The internal representation should change
    - (e.g., from an optimized one to a full-fledged one, or from a local one to a distributed one)
  - The value must be externalized in a stable node
    - (it was in an unstable node, but now we need a reference to this node – hence it must itself become a reference instead)
- The model supports easily the “become” operation, because all nodes have the same size
A transient becomes bound

A

<table>
<thead>
<tr>
<th>Unbound</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th>Unbound</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
</tr>
</tbody>
</table>

A = B

A

<table>
<thead>
<tr>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th>Unbound</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
</tr>
</tbody>
</table>

A = 5

A

<table>
<thead>
<tr>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th>SmallInt</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
Change of internal representation

A

Unbound

{Wait A}

A

Variable

Representation of a Variable (suspension list)
Externalization in stable node

A

Cell

Tuple

B = @A

B

Unbound

(contents of the tuple)

A

Cell

Reference

B

Tuple

(contents of the tuple)
Part III

Introduction to Oz and C++
Introduction to Oz

- Unification
- Compilation
- Spaces
Unification

• = is commutative in Oz

• Both sides can be quite complex
  – $X=r(r(r(X 1) A) B) C$
  – $Y=r(r(r(r(r(Y D) 2) E) F) G) H$
  – $X=Y$
    → $A=C=F=H=2$ $B=D=E=G=1$

• Can fail
  – $4=5$

• Can block
  – $X=!!Y$ $X=4$

• Can raise an exception
  – $X=${Value.failed ex} $X=4$
fun{MkExp B} 
    fun{Rec E} 
        if E==1 then 
            B 
        else 
            B*{Rec E-1} 
        end 
    end 
end 
in 
Rec 
end
proc {MkExp B Result1}
  local Rec in
  proc {Rec E Result2}
    local IfArbiter1 UnnestApply1 in
      UnnestApply1 = 1
      IfArbiter1 = E == UnnestApply1
      if IfArbiter1 then
        B = Result2
      else
        local UnnestApply2 UnnestApply3 in
          UnnestApply3 = E - 1
          {Rec UnnestApply3 UnnestApply2}
          Result2 = B * UnnestApply2
        end
      end
  end
  Result1 = Rec
end
end
Registers

- **X**
  - General purpose
  - Arguments passing

- **Y**
  - Saving X regs across calls

- **G & K**
  - Contextual environment
  - $K \rightarrow$ compile time constant
  - $G \rightarrow$ definition time constant
proc {MkExp B Result1}
  local Rec in
  proc {Rec E Result2}
    local IfArbiter1 UnnestApply1 in
    UnnestApply1 = 1
    IfArbiter1 = E == UnnestApply1
    if IfArbiter1 then
      B = Result2
    else
      local UnnestApply2 UnnestApply3 in
      UnnestApply3 = E - 1
      {Rec UnnestApply3 UnnestApply2}
      Result2 = B * UnnestApply2
    end
  end
  end
  Result1 = Rec
end
end

Move k(0), x(2)
InlineEq x(0), x(2), x(3)
CondBranch x(3)
Unify x(1), g(0)
InlineMinus1 x(0), x(2)
Move x(1), y(0)
Move x(2), x(0)
CreateVarMove y(1), x(1)
Call g(1), i(2)
Move g(0), x(0)
Move y(1), x(1)
CallBuiltin3 k(1), x(0), x(1), x(2)
Unify x(2), y(0)
Abstraction

2 (arity)

CodeArea

CreateAbstraction i(2), k(0), i(2), x(2)
ArrayInitElement x(2), i(0), x(0)
ArrayInitElement x(2), i(1), x(2)
Unify x(2), x(1)
Return

CodeArea
Move k(0), x(2)
InlineEq x(0), x(2), x(3)
CondBranch x(3), $t, $f, $e

$t:
  Unify x(1), g(0)
  Return

$f:
  InlineMinus1 x(0), x(2)
  AllocateY i(2)
  Move x(1), y(0)
  Move x(2), x(0)
  CreateVarMove y(1), x(1)
  Call g(1), i(2)
  Move g(0), x(0)
  Move y(1), x(1)
  CallBuiltin3 k(1), x(0), x(1), x(2)
  Unify x(2), y(0)
  DeAllocateY
  Return

$e:
  ...

SmallInt

1

BuiltIn

'*'
Space

- Isolated store modifications
- Own threads
- Hierarchical
- Clonable
- Mergeable
- Can have a choice point
- Has a root variable
- Used by combinators and CP
Purely logic Append

proc {Append Xs Ys Zs}
  choice
    Xs=nil Ys=Zs
  [] X Xr Zr in
    Xs=X|Xr Zs=X|Zr
    Append Xr Ys Zr
  end
end
end
Solution search

fun {DFE S}
    case {Space.ask S}
        of failed then nil
        [] succeeded then [S]
        [] alternatives(2) then C={Space.clone S} in
            {Space.commit S 1}
        case {DFE S}
            of nil then {Space.commit S 2} {DFE C}
            [] [T] then [T]
        end
    end
end

Shamelessly lifted from Ch. Schulte's thesis
Using it

local S W in
    S={Space.new}
    {Space.inject S proc{$ R}
        A#B#C=R in
        A=1|2|_
        B=3|4|_
        C=[ _ _ _ _ 5]
        {Append A B C}
    end}
    {Space.merge {DFE S}.1 W}
    {Browse W}
end
Smarter approach

proc {Append Xs Ys Zs}
  dis
    Xs=nil Ys=Zs
  [] X Xr Zr in
    Xs=X|Xr Zs=X|Zr then
      {Append Xr Ys Zr}
  end
end end
Introduction to C++11

- More than a mix between C and Java
- A functional language interpreted by compilers
- Types as data, templates as programs
Simple template

template<class T>
struct List {
  T* elem;
  List* next;
};
Template (partial) specialization

template<>
struct List<int> {  
    int elem;
    List* next;
};

template<class T>
struct List<T*> {  
    T* elem;
    List* next;
};
A simple meta-function

template <class T>
struct UnPointer {
    typedef T type;
};

template <class T>
struct UnPointer<T*> {
    typedef T type;
};
Meta-function usage

template<class T>
struct List {
    typename UnPointer<T>::type* elem;
    List* next;
};
List types

struct Nil {}

template<class T, class U>
struct Cons {
    typedef T head;
    typedef U tail;
};

typedef Cons<char, Cons<int, Nil> > myList;

myList::head foo(myList::tail::head x);
template<class L> struct Struct;

template<class H>
struct Struct<Cons<H, Nil> >:: {
H elem;
};

template<class H, class T>
struct Struct<Cons<H, T> >:: {
H elem;
Struct<T> others;
};
Templated methods

template<class H, class T, class U>
struct Getter;
template<class H, class T>
struct Struct<Cons<H, T> >{
    H elem;
    Struct<T> others;
    template <class U>
    U get() {return Getter<H, T, U>::get(this);}
};
Templated methods (2)

template<class H, class T, class U>
struct Getter {
    static U get(Struct<Cons<H, T> >* t) {
        return t->others.get<U>();
    }
};
template<class H, class T>
struct Getter<H, T, H> {
    static H get(Struct<Cons<H, T> >*t) {
        return t->elem;
    }
};
Variadic templates

template<class... Ts> Struct {};
template<class H, class... Tr> Struct<H,Tr...> { 
    H elem;
    Struct<Tr...> others;
};

template<class... Ts>
Struct<Ts...> mkStruct(Ts... args);
Values as template parameters

template<int X, class T> FooImpl;
template<class T> Foo {
    typedef FooImpl<sizeof(T), T> type;
};

template<int* x, int (MyClass::*p) (float)> X;
X<&someGlobalInt, MyClass::*round> x;
Part IV

C++ implementation of the Store aka the Object Model
Node, StableNode and UnstableNode

- As seen previously, the store is made of nodes
  - A node has a type and a value
  - There are stable and unstable nodes
- The type field is a pointer to the RTTI of the type
- The value is a so-called MemWord (memory word)
- Stable and unstable nodes are typed separately

```cpp
class Node {
    const Type* type;
    MemWord value;
};

class StableNode {
    Node node;
};

class UnstableNode {
    Node node;
};
```
Runtime type information

- Runtime information about types are stored in instances of the class Type.
  - Name and UUID
  - Properties: isCopyable, isTransient, isFeature
  - Behavior with respect to unification, equality test and pattern matching
  - Behavior with respect to garbage collection and space cloning
  - For features, how to compare them
  - Pretty-printer for values of the type

- More information about these to come
MemWord

• A MemWord is a generic type that can store “things” while making sure it fits in one memory word

• Basically it's a union of all the types in the world
  – Values of selected types smaller than a memory word (among which pointers) are stored as is
  – For bigger types, a pointer to the value is stored, and external memory is allocated
MemWord (continued)

- It exposes a unified interface, that can be used independently of whether external storage is necessary or not

```cpp
union MemWord {
    // [...] Implementation
    //      details

    template<class Q>
    void alloc(VM vm);

    template<class Q>
    Q& get();
};
```

- To store a value of type Q:
  - Call `alloc<Q>()` once
  - Use `get<Q>()` to access the value in read-write mode
Accessing nodes: RichNode

- StableNode & UnstableNode can be refs
  - Need dereferencing
  - Two types to consider
  - Good for writing
- RichNode contains stable or unstable node ptr
  - Dereferenced on construction
  - Castable on implementation types
  - Simpler for reading
A trivial data type: Cell

- Probably the most trivial data type in Mozart is the Cell (at least while spaces are ignored)
- We define it as a template specialization of Implementation<T>, where T is Cell, in a sort of C++ DSL (domain specific language)
- The class Cell itself is a subclass of Type, and has a singleton instance which is the RTTI of the Cell data type
template <>
class Implementation<Cell>: public WithHome {
public:
    typedef SelfType<Cell>::Self Self;
public:
    Implementation(VM vm, RichNode initial)
        : WithHome(vm) {
        _value.init(vm, initial);
    }

    inline
    Implementation(VM vm, GR gr, Self from);

public:

    void printReprToStream(Self self, VM vm,
                            std::ostream& out, int depth) {
        out << "<Cell: " << repr(vm, _value, depth) << ">";
    }

private:
    UnstableNode _value;
public:
   // CellLike interface

   OpResult isCell(Self self, VM vm, bool& result) {
      result = true;
      return OpResult::proceed();
   }

   inline
   OpResult exchange(RichNode self, VM vm, 
                     RichNode newValue, 
                     UnstableNode& oldValue);

   inline
   OpResult access(RichNode self, VM vm, 
                   UnstableNode& result);

   inline
   OpResult assign(RichNode self, VM vm, 
                   RichNode newValue);
};
Implementation<Cell>::Implementation(
    VM vm, GR gr, Self from)
    : WithHome(vm, gr, from->home()) {
        gr->copyUnstableNode(_value, from->_value);
    }

OpResult Implementation<Cell>::access(
    RichNode self, VM vm, UnstableNode& result) {
    result.copy(vm, _value);
    return OpResult::proceed();
}

OpResult Implementation<Cell>::assign(
    RichNode self, VM vm, RichNode newValue) {
    if (!isHomedInCurrentSpace(vm))
        return raise(vm, u"globalState", "cell");

    _value.copy(vm, newValue);
    return OpResult::proceed();
}
OpResult Implementation<Cell>::exchange(
    RichNode self, VM vm, RichNode newValue,
    UnstableNode& oldValue) {

    if (!isHomedInCurrentSpace(vm))
        return raise(vm, u"globalState", "cell");

    oldValue = std::move(_value);
    _value.copy(vm, newValue);

    return OpResult::proceed();
}
A trivial data type: Cell

- The class Implementation<Cell> defines entirely the properties, behavior, memory layout and operations of the Cell data type.
- In this case, all of those are the defaults:
  - It has no UUID
  - It is non-copyable, non-transient and cannot be used as a feature
  - It has token equality
  - The MemWord of a Cell node is an Implementation<Cell>*

Proper access: Interfaces

- Abstracts the implementation
  - Allows optimized implementation
  - Allows complex behaviors (objects)
  - Allows extension
- Defined as specialization of template Interface
  - Gives default behavior
  - Lists known implementations
- Can be constructed from stable, unstable or rich nodes
class CellLike;

template<>
struct Interface<CellLike>: ImplementedBy<Cell> {
    OpResult isCell(RichNode self, VM vm, bool& result) {
        result = false;
        return OpResult::proceed();
    }

    OpResult exchange(RichNode self, VM vm,
                      RichNode newValue,
                      UnstableNode& oldValue) {
        return raiseTypeError(vm, u"Cell", self);
    }

    OpResult access(RichNode self, VM vm,
                    UnstableNode& result) {
        return raiseTypeError(vm, u"Cell", self);
    }

    OpResult assign(RichNode self, VM vm,
                    RichNode newValue) {
        return raiseTypeError(vm, u"Cell", self);
    }
};
Marker classes

- Allow declarative coding
- Interpreted by code generator
- Both for implementations and interfaces
Implementation markers

- **Copyable**
  - The MemWord can be copied.
  - The value has no node-based identity
  - Efficient for small integers, etc.

- **Transient**
  - Undefined methods suspend if possible

- **StoredAs<T>**
  - Custom storage in the MemWord
Implementation markers (2)

- **StoredWithArrayOf<T>**
  - The representation is followed by an array of T
  - Changes constructors to take array size
- **WithValueBehavior**
  - Specify non-token value behavior wrt unification & cloning
- **WithStructuralBehavior**
- **WithVariableBehavior<C>**
Implementation markers (3)

- **WithHome**
  - Local to a space

- **BasedOn, NoAutoGCollect, NoAutoSCClone**
  - Escape mechanism for some very internal types
Interface markers

- **ImplementedBy<...>**
  - Lists known implementations
  - Optimized codepath

- **NoAutoWait**
  - Transients without implementation have default behavior
Part V

Dataflow and threads
Dataflow

- Most operations can suspend
- Two operations:
  - Suspension (wait)
  - Waking up (notify all)
Suspension

- Only variables have suspension lists
- Wake up the suspension list:
  - When bound
  - On becoming needed
- Non-var suspension => immediate wake-up
Wake up

• Thread
  – Insert into the scheduler if not present

• Variable
  – Binds to unit
  – Useful for waitOr

• Unit
  – Do nothing
Emulated threads

- Contains X registers
- Created with a call on the stack
- Private stack
  - Optimized to avoid dereferencing on unwind
  - Need special treatment for GC
Part VI

Emulator, registers and instruction set
The emulator

- Mozart uses a semi-compiled, semi-interpreted model with byte code
- The emulator executes some byte code in some given context
- It is designed to support:
  - Operations on the Store presented earlier
  - Lightweight multithreading
- Each lightweight thread has its own context
Context of the emulator

- The emulator works in a given context, of which mutable parts are thread-local
  - A pointer to the next byte code instruction
  - 4 sets of registers: X, Y, G and K registers – each register being an Oz value, i.e., a node
  - A stack of frames, with return information and exception handlers
 Registers

• The 4 sets of registers serve different purposes
• X registers are unstable, temporary storage for local computations
  – They are all invalidated upon a procedure call
• Y registers are unstable, temporary storage allocated in the stack frame
  – They are preserved by procedure calls
• G registers are stable values in the contextual environment of the currently running abstraction
• K registers are stable values that are literal constants in the code (or any other compile-time constant)
Register allocation and deallocation

- X registers are owned by the thread. Code areas advertise how many X registers they need. Upon entry in a code area, new X registers are allocated if necessary. They are freed when the thread dies.

- Y registers are owned by the stack frame. They are explicitly allocated and deallocated by instructions in the byte code (see later)

- G and K registers are allocated along with their owning Abstraction and CodeArea, respectively. Like any other stable node, they are never freed, but are subject to garbage collection
Byte code and instruction set

- Mozart byte code is an array of unsigned integers (currently unsigned short's)

- An instruction has an opcode, followed by a variable number of arguments
  - Given an opcode, the number and types of arguments is known (except for one exception)

- Arguments can be:
  - An immediate constant integer (e.g., width for tuple creation, offset for jump)
  - An X, Y, G or K register index
Instruction format

• In the following slides, we present the instruction set using the following notation:

\[ \text{OpFoo}(\text{bar: Int, a: XReg, b: KReg}) \]

• This means that the instruction Foo takes 3 arguments:
  – bar is an immediate integer, stored as is in the byte code
  – a is the index of an X register
  – b is the index of a K register

• Hence, its total size is 4 elements of byte code
Instruction format (continued)

- For clarity, we introduce the argument type Offset (in addition to Int, XReg, YReg, GReg and KReg)
- An Offset is an immediate integer which designates a label in the byte code
- It is calculated in number of elements of the byte code array, and starts counting at the end of the current instruction
Skip

- OpSkip()
- Trivial instruction with no argument, does nothing
Moves (aka register copy)

- \textbf{OpMoveXX}(source: \texttt{XReg}, dest: \texttt{XReg})
- \textbf{OpMoveXY}(source: \texttt{XReg}, dest: \texttt{YReg})
- \textbf{OpMoveYX}(source: \texttt{YReg}, dest: \texttt{XReg})
- \textbf{OpMoveYY}(source: \texttt{YReg}, dest: \texttt{YReg})
- \textbf{OpMoveGX}(source: \texttt{GReg}, dest: \texttt{XReg})
- \textbf{OpMoveGY}(source: \texttt{GReg}, dest: \texttt{YReg})
- \textbf{OpMoveKX}(source: \texttt{KReg}, dest: \texttt{XReg})
- \textbf{OpMoveKY}(source: \texttt{KReg}, dest: \texttt{YReg})

- Copy (in the sense of \texttt{StableNode::copy()} and \texttt{UnstableNode::copy()}) the specified source register into the specified dest register.

- Note that there are no moves to \texttt{G} nor \texttt{K} registers, since these registers are stable.
Double moves

- OpMoveMoveXYXY(src1: XReg, dest1: YReg, src2: XReg, dest2: YReg)
- OpMoveMoveYXYX(src1: YReg, dest1: XReg, src2: YReg, dest2: XReg)
- OpMoveMoveYXXY(src1: YReg, dest1: XReg, src2: XReg, dest2: YReg)
- OpMoveMoveXYYX(src1: XReg, dest1: YReg, src2: YReg, dest2: XReg)

- Aggregate of two moves: copy src1 into dest1 and src2 into dest2

- Their existence comes from past experience that two consecutive moves are often produced by the codegen. A simple peephole optimization can produce these instructions instead.
Y register allocation

- OpAllocateY(count: Int)
- Allocates count Y registers in the current stack frame
- Precondition: the current stack frame must not have Y registers yet
- Precondition: count > 0
- Note: they must be deallocated with OpDeallocateY (see hereafter) before any use of OpReturn or OpTailCall (see later)
Y register deallocation

- OpDeallocateY()
- Deallocates the Y registers in the current stack frame
- Precondition: the current stack frame must have allocated Y registers
Oz variable creation

- OpCreateVarX(dest: XReg)
  OpCreateVarY(dest: YReg)

- Creates a new Unbound variable in the specified dest register.

- OpCreateVarMoveX(dest1: XReg, dest2: XReg)
  OpCreateVarMoveY(dest1: YReg, dest2: XReg)

- Same as above + copy into the second register, but more efficient
  - Again, usually the result of peephole optimization
Exception handling

- **OpSetupExceptionHandler(handler: Offset)**
- Push an exception handler on the stack
- From the next instruction on, if an exception is triggered, control will continue at the label specified by handler
  - X registers are all invalidated, and X(0) contains the exception that was thrown
  - The handler is removed from the stack
- The exception handler must be popped off the stack explicitly with **OpPopExceptionHandler**
Exception handling (continued)

- OpPopExceptionHandler()
- Removes an exception handler from the top of the stack
- Precondition: the top of the stack must be an exception handler
Call builtins

- OpCallBuiltin0(builtin: KReg)
- OpCallBuiltin1(builtin: KReg, arg1: XReg)
- \ldots
- OpCallBuiltin5(builtin: KReg, arg1: XReg, \ldots, arg5: XReg)
- OpCallBuiltinN(builtin: KReg, N: Int, arg1: XReg, \ldots, argN: XReg)

- Calls the specified builtin with the specified arguments. The number of arguments given must match the arity of the builtin.
- We'll talk about builtins later, but basically it is like an Abstraction, but written in C++.
Inlined builtins

- `OpInlineSomeBuiltIn(arg1: XReg, …, argN: XReg)`
- Calls the builtin “SomeBuitlin” with the given arguments. The number of arguments given must match the arity of the builtin.
- Available for some very frequent builtins (e.g., those that back an operator, like '.')
- The code for these opcodes is automatically generated
  - In Mozart 1.4.0, they were written by hand (usually with inlined assembly) → violation of the DRY principle
Procedure call

- **OpCallX**(target: XReg, arity: Int)
  **OpCallG**(target: GReg, arity: Int)
- **OpTailCallX**(target: XReg, arity: Int)
  **OpTailCallG**(target: GReg, arity: Int)

- Calls the procedure target (abstraction, builtin, object, etc.) with arity actual parameters.

- The calling convention of Mozart is that parameters are given through X registers from 0 to (arity-1).

- Non tail-call variant push the current state on top of the stack. After that, all X registers are invalid.

- Tail-call variants are functionally equivalent to OpCall + OpReturn, only they do not consume stack space.
Return of procedure call

- OpReturn()
- Pops a stack frame off the stack and installs it. This effectively returns from the current invocation of a procedure.

- Precondition: the current stack frame has no allocated Y registers
- Precondition: the top of the stack is a regular stack frame (not an exception handler)
Branches

• OpBranch(distance: Offset)
  • Jumps to the label specified by distance

• OpCondBranch(test: Xreg, falseDistance: Offset, trueDistance: Offset, errorDistance: Offset)
  • Jumps to the label specified by:
    – falseDistance if test == false
    – trueDistance if test == true
    – errorDistance otherwise
Deep pattern matching

- OpPatternMatch(value: XReg, patterns: KReg)
- patterns must have the following form:
  \((\text{pat}_1 \# \text{off}_1) \# (\text{pat}_2 \# \text{off}_2) \# \ldots \# (\text{pat}_N \# \text{off}_N)\)
- Applies the deep pattern matching test to value against pat1 to patN, successively
- If none succeeds, does nothing
- Otherwise, jumps to the label specified by the offset corresponding to first pattern that matched
- Captures are stored directly in X registers
- X registers mentioned in any pattern are invalidated
- More on deep pattern matching later
Shallow switch (not implemented yet)

- **OpSwitch**(value: XReg, labels: KReg)
- labels must have the following form: `switch(lab1:off1 lab2:off2 … labN:offN)`
- Extracts the “label-or-self” of the given value
  - For records: the label of the record
  - For other values: the value itself
- Lookup the label-or-self in the features of labels
- If it is not present, does nothing
- Otherwise, jumps to the corresponding label
- (jump forward {CondSelect labels label-or-self 0})
Unification

- OpUnifyXX(lhs: XReg, rhs: XReg)
- OpUnifyXY(lhs: XReg, rhs: YReg)
- OpUnifyXG(lhs: XReg, rhs: GReg)
- OpUnifyXK(lhs: XReg, rhs: KReg)

- Unifies the two given operands
Initialization of “array” elements

- OpArrayInitElementX(target: XReg, index: Int, value: XReg)
- OpArrayInitElementY(target: XReg, index: Int, value: YReg)
- OpArrayInitElementG(target: XReg, index: Int, value: GReg)
- OpArrayInitElementK(target: XReg, index: Int, value: KReg)

- Initializes the element of the target “array” at the specified index with the given value
- Used to initialize the elements of tuples, records and abstractions created with the instructions to follow
Creation of Abstractions


- Creates an Abstraction with the given arity, code area, number of G registers, and stores the result in dest

- The G registers must be initialized afterwards with **OpArrayInitElement** instructions
Creation of records

- **OpCreateTupleK**(label: KReg, width: Int, dest: XReg)
  - Creates a Tuple with the given label and width, and stores the result in dest

- **OpCreateRecordK**(schema: KReg, width: Int, dest: XReg)
  - Creates a Record with the given schema and width, and stores the result in dest
    - The schema is a value which contains both the label and the arity
  - Precondition: the width must match that of the schema

- In both cases, fields must be initialized afterwards with **OpArrayInitElement** instructions
Creation of a Cons pair

- \texttt{OpCreateConsXX(head: XReg, tail: XReg, dest: XReg)}
- Creates a Cons pair with the specified head and tail, and stores the result in \texttt{dest}
Part VII

Builtins
Builtins

- Organized by boot modules
- In and out parameters from the signature
  - Heavy template meta-programming
- Can be called as regular procedures
  - Lazily generated bytecode
- Can be made available as VM instructions with a marker class (InlineAs<opcode>)
class ModCell: public Module {
    public:
        ModCell(): Module("Cell") {} 
    class New: public Builtin<New> {
        public:
            New(): Builtin("new") {} 
            OpResult operator()(VM vm, In initial, 
                                Out result) {
                result = Cell::build(vm, initial);
                return OpResult::proceed();
            }
    }
};
...
class ExchangeFun: public Builtin<ExchangeFun> {
    public:
    ExchangeFun(): Builtin("exchangeFun") {} 

    OpResult operator()(VM vm, In cell, In newValue, Out oldValue) {
        return CellLike(cell).exchange(vm, newValue, oldValue);
    }
};

...
Part VIII

Some core data types and their design and implementation
SmallInt, aka nativeint

- In Oz, integers have arbitrary precision. Because using BigIntegers all over the place would be a huge performance bottleneck, integers are implemented by two data types: SmallInt and BigInt (the later not being implemented yet)
- A SmallInt is used to represent any signed integers that can be stored in one memory word, by definition.
- The C++ side type nativeint is defined as an alias for intptr_t, which is exactly that.
Properties of SmallInt

- A SmallInt can, by design, be stored in a memory word, hence SmallInt is StoredAs<nativeint>
- From Oz semantics: it is not Transient, and is compared by Value
- Since it is StoredAs and requires no identity, it is also Copyable
- From Oz semantics: it is a Feature, and hence has a UUID
- Its UUID is even special to ensure that integers always appear first in arities
Interfaces implemented by SmallInt

- ValueEquatable
- IntegerValue
- Comparable
- Numeric
Cons pair

- A Cons pair is an optimized representation for a record with label '|' and features 1 and 2 (i.e., of the form '|'(1:H 2:T), aka H|T)
- It simply contains two nodes, one for the head and one for the tail
- Since its content is immutable, those two nodes are stable
- The arity is implied, and can reconstructed out of thin air if needed
- Finally, a Cons pair has structural equality
Properties of Cons

- Because it contains exactly two nodes, it is not StoredAs nor StoredWithArrayOf
- Hence, it is not Copyable
- From Oz semantics, it is not Transient nor Feature, and has Structural behavior
Interfaces implemented by Cons

• StructuralEquatable
• Dottable (with features 1 and 2 only)
• RecordLike
  – Its label is always '|'
  – Its arity list is always [1 2]
Memory layout of Cons

```
Cons
    SmallInt
        15
        Cons
    SmallInt
        11
        Cons
    SmallInt
        1989
        Atom
        'nil'
```
“List” with non-optimized records
foo(a:15 b:foo(a:11 b:foo(a:1989 b:bar)))
Atoms

- Atoms have the interesting property that they look like strings, and yet are required to be highly efficient, for equality test and lookup as feature.

- So, we want them to be StoredAs and Copyable by design.

- For that purpose, they are stored as a pointer, such that all equal atoms have the same pointer.

- This requires a means to find or create the unique pointer that represents a given atom string.

- Atoms are literals, which means that they can be used as labels of records.
Atom table: interface

• The atom table is a VM-global data structure that offers one operation:
  
  AtomImpl* get(size_t size, const nchar* data)

• This operation finds the VM-unique AtomImpl whose contents match the given size and data, and returns its address

• If no such AtomImpl existed, one is created and its address is returned (of course, this new AtomImpl enters the pool of candidates for the next invocation)
Atom table: implementation

- We basically need a set data type
- The atom table is implemented as a crit-bit tree
  - It is basically a binary tree, with branching when two entries of the set have the same bit-pattern up to that point but not included
- Crit-bit trees have the following properties of interest:
  - \(O(n)\) lookup and insertion, where \(n\) is the length of the string (not the number of items in the set)
  - Small memory footprint (3 words per item in the set, in addition to the actual value)
Properties of Atom

- An Atom can, by design, be stored in a memory word, hence Atom is StoredAs<AtomImpl*>
- From Oz semantics: it is not Transient, and is compared by Value
- Since it is StoredAs and requires no identity, it is also Copyable
- From Oz semantics: it is a Feature, and hence has a UUID
- (Note the incredible similarity to the properties of SmallInt! So Atom is really as efficient as a small integer)
Interfaces implemented by Atom

- ValueEquatable
- Comparable (with other atoms)
- AtomLike
- And, like any other literal:
  - Literal
  - Dottable, but with no valid feature
  - RecordLike, with itself as label and an arity list which is nil
Names (OptName and GlobalName)

• In Oz, a Name is unforgeable, globally unique value that can be used as label or feature (and has no specific operation)

• The unforgeable part means that it should have Token equality

• If it were not for the globally unique and feature requirements, names would be the most trivial Token-behaved data type one can imagine

• Let us pretend that it is true, for a while
  – These relaxed requirements define the OptName
Properties of OptName

- There is no data in an OptName (its only meaningful property is its identity), hence it is trivially StoredAs<“nothing”>
  - We'll see later (with spaces) that this is not true, as it has a home space and hence is StoredAs<SpaceRef>
- From Oz semantics, it has Token equality, is not Transient nor a Feature (remember: we're in the relaxed version)
- Because of Token equality, it is not Copyable
From OptName to GlobalName

• By default, when creating a name (e.g., with \{NewName\}), an OptName is created, because it is highly optimized

• When one attempts to use an OptName as a feature, or to serialize it, an OptName must “deoptimize” itself to acquire a global identity
  – A global identity brings 1) total order and 2) serializability
  – We encode a global identity as a UUID

• In these circumstances, an OptName “becomes” a GlobalName
Properties of GlobalName

- A GlobalName contains a UUID (and a home space), which is too big to be StoredAs.
- From Oz semantics, it has Token equality, is not Transient, and is a Feature.
- It is not Copyable.
- As features, GlobalNames are totally ordered according to their UUIDs.
Interfaces implemented by OptName and GlobalName

- **NameLike**
- And, like any other literal:
  - **Literal**
  - Dottable, but with no valid feature
  - RecordLike, with itself as label and an arity list which is nil
- Additionally, OptName implements the interface PotentialFeature, which allows it to become a GlobalName when one attempts to use it as a feature
Classes an Objects

- In Mozart 1, classes and objects were implemented with a rather pervasive Oz object model that had implications in many places
  - Most notably, there was an explicit self register which needed to be maintained by several agents (and which, btw, caused subtle memory leaks)
- In Mozart 2, we use a totally different approach:
  - Object is a normal data type, which receives no special care from global mechanisms such as the emulator
  - Class is even an ADT written in pure Oz
Object data type

- Rag-bag of a data type which looks like a number of other data types
  - Has features – looks like records
  - Has attributes – looks like an array (but indices are features, not integers)
  - Is callable – looks like an abstraction
  - May contain a lock (property 'locking' of the class)
- Created from a “model”: the class
- The callable interface forwards to the class, but features and attributes are stored in the object
What do we store in an Object?

- A reference to its class
- A reference to a Record which holds its features
- A reference to an Arity for its attributes
- An array of its attributes
- The lock is *not* stored
  - Actually the Object type does not know it has a locking behavior
  - It is completely taken care of by the class, which stores the lock as a feature of the object
Properties of Object

- From Oz semantics: it has Token behavior, is not a Transient nor a Feature
- Hence, it is not Copyable
- It is StoredWithArrayOf<UnstableNode>, for its attributes
Operations available on objects

- Get the class of the object
- The Dottable interface methods (dot, hasFeature and condGet) forward to the record of its features
- attrGet, attrPut and attrExchange to access the attributes
- For the Callable interface, it exposes something that looks like this:

  proc {$ M}
    {ObjApply M Obj {ObjGetClass Obj}}
  end

With Obj and ObjApply being in G registers, and ObjGetClass being an inlined builtin.
Interfaces implemented by Object

- Callable
- Dottable
- ObjectLike
  - getClass, attrGet, attrPut and attrExchange
- ChunkLike
  - Yes, IsChunk Obj is true
Where is the rest?

- That's all there is to objects in the VM
- The rest is in
  - The base environment (the Class ADT)
  - The compiler (to desugar the class notation)
- These two components must of course be coherent
Part IX

Spaces
Spaces

- Form a tree
- Installed in the store by script
- Uninstalled from the store by trail
- Stability and admissibility as in the old VM
- Can be extended for CP with Gecode
Cloning values

- WithHome marker class
  - Stores the home space
  - Implements the cloning/copying decision
- Non-token values are cloned
- Top-level variables and token are not
Speculative binding

• Binding a transient t to a node n
• By a thread not in t's home (necessarily in a subspace)
• Backed-up on the trail of the thread's space
• Re-unified on space installation
Part X

Unification, structural equality test and deep pattern matching
Why unification?

- In Oz, unification of two variables A and B is a very important operation
  - It is the front-end to variable binding
  - It provides the basis for logic programming (with spaces)
- It is rather pervasive, as it needs information about the “behavior” of each data type with respect to structural equality
Challenges in unification

- Unification of A and B when either one is a variable must be fast, because unification is used for shallow variable binding, which are very common operations.
- Structures may have cycles, and unification must be able to unify cycles (rational tree unification).
- Different data types “behave” differently with respect to unification and structural equality.
  - Simple intuition: variables can be bound to make unification happen, whereas non-variables can only be equal or not.
- Unification can be suspended in some cases, and must be resumed afterwards in the state it left.
- It is highly related to spaces (speculative bindings).
Different behaviors of data types

- Variable behavior: variables may be bound to the corresponding value to satisfy unification
  - Issue: when there are two variables, what should happen?
- Token behavior: they are equal (and hence unifiable) iff they have the same identity
- Structural behavior: they are equal only if they have the same shallow structure; but they can contain other nodes, which must be unified pair-wise
- Value behavior: degenerated case of structural behavior when there is no inner node
Different behaviors of data types (continued)

- In C++, data types are annotated with marker classes that specify their behavior
- By default, data types assume Token behavior
- WithStructuralBehavior
-WithValueBehavior
- WithVariableBehavior<unsigned char priority>
  - When unifying two variables, the one with the highest priority is bound to the other
  - If priorities are equal, the choice in non-deterministic
Binding priorities

- Failed values have the lowest priority, because any `Var = Failed` must bind the variable to the failed value (and not raise an exception), and any `!!Var = Failed` must wait.
- Read-only values have the second-to-lowest priority, because any `Var = !!U` must bind the variable to the read-only (and not wait).
- In theory, Unbounds and Variables may have the same priority:
  - But the bind() method is more efficient in Unbound (no wakeups to do), so Unbound has the highest priority.
Algorithm: the big picture

• Helper data structure: a stack of pairs of nodes that are to be unified
  – Initially it contains an only pair with the two entities that need to be unified
  – Afterwards, it is used for entities contained in values with structural behavior

• Top-level loop:
  – Pop a pair off the stack
  – Process that pair (this can succeed and push zero to many pairs on the stack; or it can fail or raise an exception)
  – Loop until the stack the stack is empty, or there was a failure
Algorithm: process a pair (A, B)

- If A and B have the same identity, succeed
- If A or B has Variable behavior, bind it to the other (use priority when both are variables)
  - This can suspend (if it is a read-only) or raise an exception (if it is failed value)
- Otherwise, both are determined. If they do not have exactly the same type, then fail
  - Indeed, if one is a Token, the only way it could have succeeded is by the first bullet
  - If both have Structural/Value, they must have the same shallow structure to succeed, and by construction all values with the same shallow structure have same type
Process a pair (A, B) (continued)

- If A and B have Token behavior, then fail (again, the only success case was already tested)
- If they have Value behavior, compare them for equality with the data type-provided test
  - If they are equal, succeed, else fail
- If they have Structural behavior, compare them for shallow structure equality with the data type-provided test (this can push pairs on the stack)
  - If shallow structures do not match, then fail
  - Otherwise, make a backup of node A, then make it “become” a Reference to B (handles cycles), and proceed
Algorithm: some subtleties

- When binding a variable in a subspace, the binding can be speculative
  - This is handled in the data-type provided bind() method, and hence the unification does not need to care about that
- The first test (for identity equality) is very important (not only for Token behavior)
  - Together with the temporary “become” on structural values, it handles cycle detection
  - It prevents attempts to bind a variable to itself
- If a binding suspends, the unification must continue to the extent possible; all suspensions are kept in a list and must be applied afterwards
Post-processing

- On failure, exception or suspension, all temporary bindings must be undone, by restoring the backups that were taken.
- This is not necessary on success, and it is even desirable to keep them, as that speeds up subsequent unifications and equality tests.
- However, it is necessary if the current space is not the top-level space, for those bindings bypass speculative bindings.
Post-processing (continued)

• On suspension, one or more nodes need to be waited upon
  – Construct a control variable that waits upon all of them, and suspend on this control variable

• This leaves us with a list of pairs of nodes still to bind
  – Construct two tuples with lhs's and rhs's
  – Remember that we want to unify these two tuples only, rather than the original nodes
Equality test, aka entailment check

- The test \( A == B \) must check for structural equality, in the same way that unification must apply structural equality.

- The algorithm is very close to unification, with only one difference:
  - When either \( A \) or \( B \) has Variable behavior (or both), the algorithm must suspend, instead of binding the variable.

- New role for the identity test: a variable is equal to itself, even when it is not bound.

- On success, no need to undo temporary bindings either.

- \( \not= \), aka disentailment check, is derived trivially from \( == \).
Deep pattern matching

• In deep pattern matching, the two operands are not symmetric: a value A is matched against a pattern B

• Very close to equality test, except when:
  – The pattern side is a capture/wildcard: always succeed and capture the variable
  – The pattern side is a conjunction of patterns pat1 to patN: add to the todo list the pairs (value, pat1) to (value, patN)

• One additional subtlety: never do temporary bindings (they would be wrong, and they are not needed since there is no cycle in a pattern anyway)
Part XI

Graph replicators
Garbage collection and Space cloning
Garbage collection (GC)

- Double space copy algorithm for garbage collection
  - When one space is full, allocate a new one of the same size
  - Identify roots of GC
  - From roots, traverse the graph of accessible nodes, and copy them to the new space
  - Graph traversal implies cycle detection
  - Afterwards, free the entire original space
Space cloning (SC)

- Very similar algorithm for space cloning
  - There is one root which is the space to be cloned
  - From this root, traverse the graph of accessible nodes
  - For structural/value nodes, dive into the structure and clone its components
  - For nodes that are homed inside the space to be copied, replicate it as homed in the new space
  - For nodes that are homed outside of the space and nodes without home (which are conceptually homed in the top-level space, just make a reference
Graph replicator (GR)

• A graph replicator is an abstraction over GC and SC, that captures the graph traversal and copy algorithm template, with cycle detection

• It offers a means to replicate the following items:
  – Space* (aka SpaceRef)
  – Runnable*
  – StableNode and UnstableNode
  – StableNode* (aka StableRef)

• Note that UnstableNode* is nonsense in our model, so it is not offered
GR basic algorithm

- Helper structures: for each type T from the previous slide:
  - A todo list of pairs (T* dest, T* source) meaning that *source must be replicated at address dest (memory for dest having been allocated beforehand)

- External initialization: populate the todo lists with roots of GR

- Basic algorithm: while at least one todo list is non-empty:
  - Pop a pair off one non-empty todo list
  - Process it with instanciation-dependent code (this can push new items in the todo lists)
GR algorithm with cycle detection

- After processing a pair, we mark the source as having already been GRed to the dest
  - For SpaceRef and Runnable*, there are dedicated fields in classes Space and Runnable
  - For StableNode, we make it “become” of type GRedToStable with the corresponding dest
    - GRedToStable has an only field StableNode* dest and is StoredAs<StableNode*>  
  - For UnstableNode, we make it “become” of type GRedToUnstable with the corresponding dest
    - GRedToUnstable has an only field UnstableNode* dest and is StoredAs<UnstableNode*>  
  - (StableRef is handled differently, see later)
Process a node in GR

- When processing a node in GR, first dereference it (transparent thanks to RichNode)
- Call the gCollect() (for GC) or sClone() (for SC) of the node's type RTTI
  - Parameters: source and destination
- Let the type handle its replication itself
- GRedToStable and GRedToUnstable simply copy() their stored destination into the new destination
Process a StableRef

• When processing a StableRef, we have to replicate the pointed node, and return the address of the GRed side of the pointed node

• Avoid allocating new StableNodes when possible
  – If the pointed node is a GRed type, get a stable reference of its destination (might allocate a new StableNode if it is a non-Reference UnstableNode)
  – If not, then allocate a new StableNode dest, return its address, and put (dest, source) in the todo list for StableNodes
Choosing the right todo list

• The order in which we try the todo lists matters
  – Some orderings could allocate unnecessary new StableNodes
• First, test for spaces, runnables and StableNodes: they never need to allocate new StableNodes, and they might add new todo items
• Second, test for UnstableNodes: they need to allocate a new StableNode only if it is a reference to a GRedToUnstable whose dest is not copyable; and they might add new todo items
• Finally, test for StableRefs: they always need to allocate a new StableNode (except if the source is a GRedToUnstable whose dest is a Reference, or if it is a GRedToStable)
GC instantiation

• Initialize with GC roots:
  – Reference to the top-level space
  – Runnable threads
  – Pending alarms
  – Roots given by the environment (e.g., nodes used as feedback from I/O operations)

• Replicate everything unconditionally (obviously, the old ones will die)
SC instantiation

• Only one root: the space to be cloned
• But trickier
  – Spaces, runnables and nodes marked as replicated must be backed up, and restored afterwards, as they will be reused
  – Things homed in a parent space of the space to be cloned must not be cloned (but copied instead)
Part XII

Sorialization
Sorialization

- Two-phases
  - In-memory structure, abstract node graph
  - Bytes, serializing nodes and references to nodes
- Extensible
  - Node types identified by UUIDs
- Portable
  - Based on Google protocol buffers
Node definition

message Procedure {
  option (kind) = "bf2e1b5b-37a2-4f7c-b4c1-85c7161c5b0e";
  required uint64  arity      = 1;
  required NodeRef code      = 2;
  repeated NodeRef captures = 3;
}
Part XIII

Conclusion and future work
Next steps to implement Oz 3

- Port the compiler of Mozart 1.4.0 so that it produces byte code for Mozart 2
  - Master's thesis for 2012-2013: write a new, cleaner compiler for Mozart 2 (from scratch, in Oz)
- Distribution subsystem
  - On top of a reflective architecture
  - Implement the distribution protocols in Oz, for Oz
- Constraint subsystem
  - Integration with the CPS library Gecode
  - Work under progress by the AVISPA group in Columbia
- Bindings to external libraries (e.g., Tk)
Towards a release

• Beta version expected for early September
  – Usable by third year students
• Release expected for December
  – Christmas present :-)

A platform for future research

• Conflict-free Replicated Data Types (CRDT)
  – Thanks to this new Mozart 2, we will be able to implement CRDT protocols in Oz, using the same reflective architecture

• Other?
Backup slides

- More on C++
template<int X, class T> FooImpl;

template<class T> Foo {
    typedef FooImpl<sizeof(T), T> type;
};

template<int* x, int (MyClass::::*p) (float)> X;

X<&someGlobalInt, MyClass::round> x;
Dependant parameters

template<class T, T* t> X;
...
int a;
X<int, &a> s;

template<class C, class T, void (C::*m)(T)> Y{
    static void do(C obj, T param) {
        obj->*m(param);
    }
};
template <class Bar> 
int foo(Bar&& x){
    return other(std::forward<Bar>(x));
}

...

foo(12+456);
foo(x);
Lambda functions

auto f = [](int a, int b) -> int { return a+b; };

int x=4;
auto plus4 = [=](int a) -> int { return a+x; };
auto plusX = [&](int a) -> int { return a+x; };

std::function<int (int, int)> f2=f;