BERLIOZ: COMPILING OZ TO JAVA BYTECODE

LE XUAN THANG

NATIONAL UNIVERSITY OF SINGAPORE

2001

BERLIOZ: COMPILING OZ TO JAVA BYTECODE

LE XUAN THANG
(*M. Comp., NUS*)

A DISSERTATION SUBMITTED
FOR THE DEGREE OF MASTER OF COMPUTING
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2001

Abstract

Oz is a multi-paradigm programming language that aims to unify the advantages of object-oriented, concurrent constraint and functional programming. Java, apart from being a programming language itself, is a portable runtime environment whose presence is becoming more prevalent. The possibility of compiling the language Oz to Java bytecode thus seems promising and interesting, both for practical usefulness and theoretical aspects. By doing that, we hope to combine Oz capabilities with the extensive Java runtime support and portability. In this thesis, we show that such a direction is indeed feasible. We present the design and implementation of a compiler and runtime system for Oz based on Java. We concentrate on a base language of Oz, which we call Berli-Oz, and which excludes constraints, functors and support for distributed programming. Oz semantics can be preserved and implemented correctly, and where Java bytecode lacks language facilities equivalent to Oz, it is straightforward to devise a workaround, albeit with a penalty in efficiency. Finally, we highlight possible future improvements, extensions and optimizations.

Acknowledgements

First of all, my thanks go to my supervisor, Dr. Martin Henz, whose many comments, advice and ideas were invaluable throughout the entire project. His extensive knowledge of Oz internals helps me tremendously in understanding how everything works, and his patience and feedback when checking through the many versions of Berlioz and the draft of this paper are especially appreciated.
I also want to thank Leif Kornstaedt and Christian Schülte for their insightful comments and revision of the draft version of this paper.

Contents

List of Figures

Summary

The project Berlioz was started to assess the possibilities of compiling Oz to Java bytecode. The target was an interactive development system where Oz source code can be compiled and executed interactively. The choice of Oz and Java is not totally coincidental. Oz on one hand is a combination of different programming language paradigms, resulting in a powerful and expressive language. Java bytecode on the other hand is a portable medium whose runtime environment (the Java Virtual Machine, or JVM) is readily available on many platforms. A combination of Oz and Java will thus allow Oz language advantages to be available wherever a Java runtime environment exists. Substantial native Java runtime support also exists, reducing efforts needed to develop runtime support for Oz.

In order to focus on the most essential aspects of implementing Oz, we concentrate on a sub-language we call Berli-Oz that contains all built-in data structures except arrays and dictionaries, the object system, concurrency and syntactic support for functional programming. It excludes support for constraint programming (constraint variables, spaces) and distributed programming (pickling, functors, and sites).

To allow dynamically typed values, a common type system is used, in which value types are differentiated by tags. This necessitates certain runtime check for type, and the inefficient wrapping and unwrapping of primitive values. Parameter passing on the other hand is made easier and more convenient, since value types are ubiquitous and there is no need to distinguish between *in* and *out* values. Values in Berlioz are laid out in the Java heap in a graph-like structure. Thread synchronization (suspension and resumption) is approximated using *wait/notify* mechanism inherently supported by all Java objects.

Except for integers and floats, which are implemented using Java's own classes, other value types require certain customized classes. Records make use of an internal hashtable for feature selection. Procedures are compiled into distinct Java objects. This allows the creation of closures in which variables of intermediate scope are stored as object attributes. Pattern matching is implemented as a series of Boolean tests that are carried out in a strictly top-down, left-to-right manner.

Classes and objects in Oz cannot be compiled directly to Java classes and objects, because of first-class messages and multiple inheritance. Instead both are represented using specially constructed Java objects, which contain information required for multiple inheritance, features, attributes and method dispatching. Method dispatching for incoming messages is implemented using pattern matching. Feature selection of Oz objects is syntactically identical to records, while invocation syntax is identical to procedures. This is implemented by making compiled Oz objects inherit the same interface as records and procedures do.

Bytecode generation from the parsed syntax tree is done through delegation between nodes on the tree. The use of a customized code generator that hides away bytecode details helps reduce complexity associated with Java bytecode format and increases code readability.

Due to the intrinsic differences, there is a considerable gap between the source language Oz and target platform the JVM. For the Oz features that do not have a Java equivalent, either some similar feature is used as replacement, or they are approximated programmatically, usually with a loss of efficiency. There is also a performance penalty associated with the JVM itself as a runtime environment. Some of the inefficiencies are inevitable due to the semantics gap between Oz and Java, while the rest may benefit from possible future optimizations. However, Oz semantics is fully preserved, and performance is generally acceptable.

Chapter 1 Introduction

1.1 The Sub-language Berli-Oz

The language Oz has been developed since 1991 at the Universität des Saarlandes (Germany) under the direction of Prof. Gert Smolka, with the first implementation being DFKI Oz 1.0. The latest development of Oz is Oz 3.0, implemented in the programming system Mozart [Moz98]. Since the initial inception, which includes logic variables and first-class procedures, Oz has had some additions over the years, most notably objects, finite-domain constraints and spaces for constraint-based search. In short, Oz is a multi-paradigm programming language designed to combine the advantages of logic, constraint, functional and object-oriented programming, all in a concurrent and distributed setting.

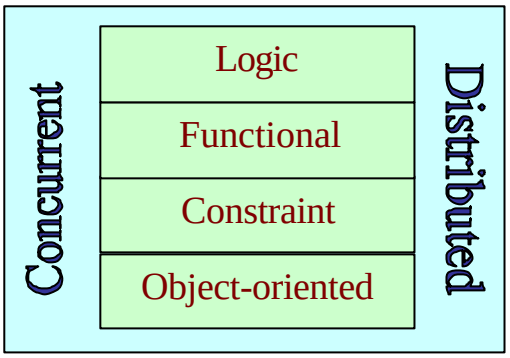


Figure 1-1: Oz features

With such an extensive scope, it is difficult to try to achieve everything in one go. Instead, in this project we shall only focus on a subset of the language Oz named Berli-Oz, which excludes two language extensions: functors and constraints. As such, concepts such as finite-domain and finite-set constraints, search engines, distributed computation and functor will not be covered. In addition, we have also left open the concepts of arrays, dictionaries, futures and by-need futures. Main features of the Berli-Oz are logic variables, concurrency, higher-order procedures and values, and object-oriented features (first-class classes, objects and messages). Though these are briefly described below, for a more extensive treatment of the language Oz and the Oz Programming Model (OPM), see [Smo94] and [Smo95]. The web site for the Mozart system [Moz98] also has a large repertoire of help resources on various Oz topics. Following is a list of Berli-Oz features, each accompanied by a short example:

Transient. Transients are runtime representations of logic variables. (Note that the term transient also includes futures, we however do not cover futures and by-need futures in this text). Threads suspend (synchronize) when accessing an unbound logic variable, and resume when the logic variable is bound to a value. *Binding* is *destructive*, in that the variable is transparently replaced by the value it is bound to. In addition, variables in Oz are *dynamically typed*: the type of each variable is not determined until bound:

```
local
      X Y Z    %% Declare three untyped logic variables
in
      X = Y    %% Bind X and Y
      X = 10   %% Bind X (and Y) to the integer value 10
      {Show Y} %% Display 10
      Z = ab   %% Bind Z to the atom ab
end
```

Constraint store. Transients and values are stored in a special compartment called the constraint store, which is accessible to all threads. The term *constraint store*, however, is used to be consistent with other similar implementations. Since we do not support the constraint extension, this compartment should generally be seen simply as variable, or value, store.

Concurrency. Thread creation is syntactically supported by Oz.

```
local
      X        %% Declare logic variable X
in
      thread   %% Spawn a new thread
            {Wait X} %% Wait until X is bound
            {Show X} %% Display the value of X, which is 10
      end
      {Show X} %% Display X (unbound variable) as '_'
      X = 10   %% Bind X to the value 10
end
```

Literals. These first-class values include atoms (values that include a sequence of characters), names (unique unstructured values, which also include the Boolean values `true` and `false`), integers and floats:

```
local
      X Y Z T     %% Declare four new variables
in
      X = 1       %% Bind X to the integer 1
      Y = 2.2     %% Bind Y to the float 2.2
      Z = X == 1  %% Bind Z to result of (X == 1), which is the name 'true'
      T = 'astring' %% Bind T to the atom 'astring'
      {Show {NewName}} %% Create and display a new name
end
```

Records, lists and tuples. Records in Oz are rational trees consisted of a label and zero or more primitive values known as *features*. Each feature corresponds to a *field*, which can be any Oz value. The number of fields in record is called its *width*. Tuples are records whose features are consecutive integers starting from 1. Lists are either empty lists made up of the atom `nil`, or pairs (tuples whose label is the atom '|' and whose width is 2) with the second field (the one corresponding to feature 2) being another list. Some examples of records, lists and tuples:

```
local
      X Y Z %% Declare three new variables
in
      %% Create a record with features x,y,z and width = 3
      X = rec(x:a y:b z:c)
      %% Create a tuple with width = 3 and field at feature=3 is transient
      Y = at(a b _)
      %% Create a list of three elements
      Z = [a b c]
end
```

Cells. Unlike transient, cells in Oz are like "containers" whose content may change over time. Cells are created in Oz using the built-in procedure `{NewCell CellContent Cell}`. Cell operations include *access* (retrieving content of a cell), *assign* (changing content of a cell) and *exchange* (both cell access and cell exchange are grouped together and carried out in one atomic operation). This is illustrated below:

```
local
```

```
      C = {NewCell 10}  %% Create a new cell with integer value 10
      A                 %% Declare a logic variable A
in
      {Show {Access C}} %% Display the value 10
      {Assign C 1.5}    %% Assign the float value 1.5 to cell C
      {Exchange C A _}  %% Perform an atomic exchange, now A = 1.5
      {Show A}          %% Display the value 1.5
      {Show {Access C}} %% Display content of C: an unbound variable '_'
end
```

Locks. These are special values in Oz that can be used for synchronization. Locks are created in Oz using the built-in procedure `{NewLock Lock}`.

```
local
      L = {NewLock}   %% Declare and initialize a new lock
      C = {NewCell 1} %% Declare and initialize a new cell
in
      %% Create a thread, then obtain lock to perform Exchange on C
      thread lock L in
         {Show {Access C}} %% Display the content of C
         {Assign C 10}     %% Assign 10 to C
      end end
      %% Obtain lock to perform Exchange on C
      lock L in
         {Show {Access C}} %% Display the content of C
         {Assign C 20}     %% Assign 20 to C
      end
end
```

Lexical scoping. The declaration corresponding to a variable occurrence is statically fixed to be the closest surrounding declaration. Oz supports *higher-order procedures*, which means they can be defined in any scope. Due to lexical scoping, procedures are represented at runtime by closures that encapsulate the environment in which they are created. In the example below, procedure P2, created by applying procedure P and P1, binds some variables of intermediate scope with its parameter `N2`.

```
local
      proc {P N P1}
         proc {P1 N1 P2}
            proc {P2 N2}
               N = N1 = N2 = Z
            end
         end
      end
      Z
in
      local A B C P1 P2 in
         %% First, then apply P1 on B to get P2
         %% finally apply P2 on C
         {P A P1}   %% Apply P on A to get P1
         {P1 B P2}  %% Apply P1 on B to get P2
         {P2 C}     %% Finally, apply P2 on C
         Z = 534    %% Now we should have Z = A = B = C
      end
end
```

Functional syntax. Oz supports a rather extensive functional syntax with an expression-oriented language extension. This extension is internally transformed to a core relational language through a process called

unnesting. We have already seen some occurrences of this, for example in `{Show {Access C}}`. In fact, not only procedure application, but most other statements can also appear as expressions (those that resolve to a value):

```
local X P in
      X = local K in thread {P K} end end
      P = proc {$ A B}
          B = try A * 2 end
      end
end
```

Pattern matching. Oz provides extensive support for pattern matching, allowing for convenient decomposition of built-in data structures.

```
local
      X = aa(3 a#b 32)
in
      case X
      of ab(...) then {Show 'this should never happens'}
      [] aa(2:Y#Z ...) then {Show [Y Z]} %% Display [a b]
      [] _ then skip end
      end
end
```

Exception handling. Oz supports standard exception handling using `try...catch...finally` statement. The `try` clauses in Oz are internally transformed into pattern matching constructs. Similar to Java, Oz also has an extension for the `finally` clause, which is guaranteed to be executed regardless of whether there was any exception raised in the `try` block.

```
local
      X = aa(3 a#b 32)
in
      try raise X end %% Simply raise X as an exception
      catch ab(...) then {Show 'this should never happens'}
      [] aa(2:Y#Z ...) then {Show [Y Z]} %% Display [a b]
      [] _ then skip end
      end
end
```

Classes and objects. Both classes and objects in Oz are first-class. Method invocation in Oz is in the form of first-class messages. Oz supports multiple inheritance. We will give examples and discussion to the object system in Chapter 5.

First-class values. Essentially all values in Oz are first-class, including procedures, classes and objects. All exceptions are first-class and any value can be thrown as an exception.

Automatic memory management. In the tradition of symbolic programming languages, Oz implementation must automatically reclaim memory which becomes inaccessible (garbage collection).

1.2 Java

Java is a portable object-oriented programming language with built-in support for concurrency and automatic garbage collection. Java portability, and to some extent, its popularity, comes about primarily because of its bytecode-based design. Java source files are compiled into bytecode in a specific class file format that can be executed on any JVM [Ven99], [LY99]. Furthermore, the bytecode file format and the JVM are largely independent of the language (even though they were obviously designed with the Java language in mind). Thus a compiler that produces bytecode is guaranteed a runtime environment that is platform-independent and widely

available. This is indeed a very attractive option, especially when there is limited resources to perform native porting to different platforms. Also, the extensive Java class library helps ease the job of implementing runtime support: applications have access to concurrency, garbage collection, and graphical interface library, all for free. As a result, there have been many projects that compile programming languages into Java bytecode: Kawa [Both98], the DML compiler [SW00], the MLj project [MLj99], SmalltalkJVM [Sma], J-Eiffel [Kal99], and many more. Other projects that either create or manipulate bytecode also exist, among them are Jasmin [Mey97], BCEL [Dahm99], and BIT [Han97].

1.3 The Java Virtual Machine

The JVM has a code verifier that imposes many restrictions on what Java classes can do. Bytecode that does not pass the verifier (but is otherwise valid) will not be executed, unless the verifier is turned off. For wider deployment in different environments (applications, applets), we aim to generate bytecode that conforms to all verifier criteria. That means there will be less freedom when exploring possibilities with manipulating the JVM internal states. We argue, however, that such restrictions are minimal, and often do not make much difference on eventual design decisions. Overall, code that conforms to the Java verifier criteria can usually be translated into the equivalent Java code in a straightforward manner. There are a few exceptions, most notably of which is probably the `goto` instruction, but even in this case, the effect of a `goto` can generally be simulated using appropriate Java language constructs and statements (e.g. `break` and `continue`). Experiments of compiling some Oz constructs into Java source code show that while compiling to source code has its own difficulties, the required effort is reasonable.

Internally, the JVM is a stack machine with an *operand stack* where instruction arguments are loaded before each instruction is executed, and a *local variable area* with indexing access. Both the operand stack and local variable area are strictly local to each method, however. The verifier essentially enforces that stack frame cannot be manipulated in such a way that cannot be done in a conventional Java program. Another restriction of the JVM, or rather, of the Java class format, is that method size must not be larger than 64K. This may pose a problem when generating the parser and scanner. We will go back to the generated parser in section 6.2.

At bytecode level, the JVM treats integers and Boolean values as identical for most operations. This allows us to freely substitute integer values for branching instructions. This will have a bearing later on in section 6.6 when we discuss the compilation scheme.

The lack of general-purpose machine registers complicates passing parameters to and returning values from procedures. All methods that return more than one value need to pass the *out* values each in a separate wrapper (or value holder) as method parameters. Since *in* and *out* parameters in Oz are implicit, they are defaulted to be *out* and thus a wrapper object is needed for every parameter passing. Compared to the register-based, Oz-oriented virtual machine used in Mozart [Mehl99], the JVM lacks many features. Among them are machine registers, support for built-in data types (structure, transient), built-in tail-optimization, not to mention that the Mozart VM implementation has access to direct memory address (pointers), which is strictly off-limit in Java.

On the other hand, Java and the JVM do have some features similar to Oz and the LVM that substantially reduce our effort in implementation, such as automatic garbage collection, built-in concurrency and exception handling. Reusing Java threads for Oz concurrency, and Java exceptions for Oz exceptions are straightforward. Heap memory allocation in the JVM also suits the graph model used to represent the constraint store well. In particular, the stack model of the JVM (no global registers, all invocations happen on a single operand stack) greatly simplifies our compilation scheme. Because of the simplicity of the operand stack, it is relatively straightforward to arrive with a compilation scheme that satisfies the verifier criteria of consistent operand stack status. We only need to make sure that the compilation of statements and expressions is consistent, something that could easily get much more complicated and error-prone had a virtual machine with multiple general-purpose registers been used as the target.

1.4 Compilation Target

The different options of compiling or interpreting a declarative language to either Java source code or Java bytecode were described in [Both98]. We have chosen the path of generating Java bytecode, because our aim is to be able to create a development environment, where Oz source code can be compiled and executed

interactively, without significant trade-off in speed and responsiveness. In principle, we could have, in addition to directly creating JVM bytecode, generated Java code as well. That direction was in fact explored, and while there are certain complications, overall we find the effort to be manageable. The generated Java code can be seen as a complement, instead of the alternative, of generated bytecode. While bytecode is used for interactive development, Java code allows developers with Java knowledge to understand how their code is translated, allows the exploitation of Java compiler optimizations, and can be used for debugging. There is also a possibility of embedding Java code directly in Oz programs. For simplicity, however, we will focus on Java bytecode as the primary target in this text.

Eventually our aim is to be able to offer a compiler in Java and runtime environment based on the Java Virtual Machine (JVM) that is fully compatible with the language Oz. While this approach entails a performance penalty compared to a runtime environment dedicated to Oz (e.g. the Mozart system [Moz98]), it does have the following advantages:

- Portability and availability: Java runtime environment is becoming more and more prevalent, and with Java's "Write Once, Run Anywhere" concept, Oz programs compiled to bytecode can also enjoy the same availability of runtime environment.
- Lightweight implementation: since the bulk of the runtime support is already born by Java runtime classes, a relatively capable Oz runtime system based on the JVM does not have to be as big as a natively-written Oz runtime. (Although that means we are bound by what the JVM has to offer).
- Potential of Java improvement: Since generated bytecode follows Java's class file format and can be run under any Java runtime environment, it stands to benefit from any performance improvement to the JVM in the future.

1.5 Related Work

Some projects that bear resemblance to Berlioz are Kawa [Both98], the DML compilation project [SW00] and the latest Oz implementation Mozart [Moz98].

Kawa is a compiler for Scheme to Java bytecode. Its interactive model was the one that prompted us to implement a similar system for Oz, and there are similarities in the use of intermediate expressions and class loader. The difference between Oz and Scheme, though, prevents us from going further than that. Tail-call optimization is done through an extension to Continuation Passing Style.

The DML project is a system to compile DML, a dynamically typed extension to ML, to Java bytecode. Tail-call optimization is carried out, persistent value storage is achieved using Java serialization, and distributed programming is supported using Java RMI mechanism. Types are represented using inheritance hierarchy, as opposed to tags in Berlioz.

Mozart is a full-fledged Oz implementation in C++, which contains a virtual machine LVM specially built for Oz. Mozart provides built-in support for: tail-call optimizations; efficient representation of dynamically-typed values; unlimited number of local registers and threads; first-class procedures; and automatic memory management.

Chapter 2 Representing the Constraint Store

2.1 A Common Type System

As mentioned in section 1.3 *in* and *out* parameters in Oz are implicit. In other words, all changes performed on Oz procedure parameters are permanent and must be reflected on the original variables (pass-by-reference). Sometimes it can be very difficult to determine the mode of a parameter. Take the example:

```
proc {P X Y} X = Y end
```

In this case, potentially both X and Y may be affected, and without a thorough, system-wide analysis (which can be impossible should distributed programming is in place), it is difficult to determine between X and Y which one is *in* or *out*. As a result, all parameters are defaulted to be *out*. This necessitates a wrapper class for parameter passing, and a common type system would come in handy for the task. In addition, we have the following considerations:

- There is a need to have destructive binding: each variable when first created is a transient. The type of each variable has to be compatible with that of a transient. (For pointer destructive binding is easy because the object or value stored at a certain memory address can always be replaced. For object reference, however, the transient object that a reference refers to cannot be changed. As a result, to enable destructive binding, that transient object must be prepared to provide a link to the value to which it is bound).
- Type recognition should be delayed until value is used: many operations can take place successfully without explicit type recognition. For example, when binding a transient to a value, the type of the value is irrelevant, as long as it is known to be a determined value (as opposed to reference, or transient). Even when checking for equality, we can usually check simple object reference (pointer) equality, or call a virtual or interface method (for example `Object.equals(Object)`) without explicitly doing type checking (JVM instruction `instanceof`) and typecasting (JVM instruction `checkcast`). Therefore, a common representative usually offers sufficient functionality for many operations.

These factors mandate that either a common type is used, or a common interface is shared by all value-representation classes. Either way, the drawback is that there will be overhead when values are actually used: they need to be unpacked, and then packed when computation is over. This is inevitable for dynamically typed values implemented in an object-reference-only virtual machine like the JVM. It is best if we can avoid boxing and unboxing and use primitive value, for example, Java `int` and `float`. Such a move, however, would render the numeric system inflexible to future expansion, for example to a system with higher precision. The DML project [SW00] chooses to use a common interface. For Berlioz, we are in favor of a common type system, because multiple classes sharing a common interface will result in duplication of code, and moreover, forcing unrelated values to share a common interface may result in not-so-elegant result. For example, the DML project enforces that all values, including non-functional ones, implement a method `apply()`. In contrast, we require that values only implement their own functionalities.

2.2 The Graph Model

In representing the constraint store, the Mozart system uses a graph model [Mehl99]. In Berlioz, we also model the constraint store to closely resemble a graph. That is, leaves on a tree can be trees themselves, and cyclic trees are allowed. For example take the following Oz code:

```
local X Y Z M in
      X = M %% Both X and M are still transient
      Y = a
      Z = b(1:X 2:Y c:_)
      Z.c = d(Z) %% Define a cycle
end
```

The equivalent graph looks like:

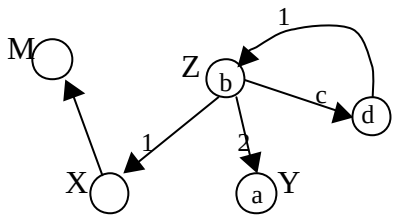


Figure 2-1: Graph model

A variable in Oz can be bound to another variable in a process called *unification*. This is a complex graph-rewriting process aiming at making two variables equivalent to each other [Mehl99]. Briefly, two variables are equivalent when either one of the following conditions is satisfied:

- They are equal (with regards to the appropriate equality test).
- They are both records, with the same arity, and each of their fields is pairwise equivalent.

A logic variable is initially represented by a transient in the constraint store. Adding information to that variable through unification can turn it into a reference or determined node. In fact, binding a variable to a determined value is but one of the possible scenarios of unification. Unification can also happen between two unbound logic variables, in which case one of the variable becomes a reference to the other. The transformation from a transient to a reference occurs transparently, such that after binding, the two variables can be seen as literally unified (into one). Note that the reference chain may grow, which means all operations on reference nodes need to first perform path collapsing. If both variables are bound, unification checks if their values are equal, and fails if they are not. If both variables are bound to records with the same arity, unification is performed pairwise for each of their fields. An example of what happens during unification is given below:

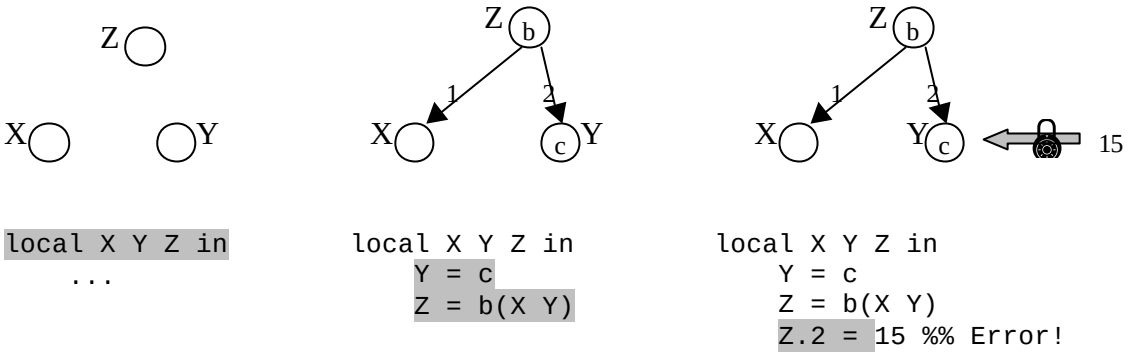


Figure 2-2: Unification

We will present the unification algorithm used in Berlioz in section 2.4, after discussing `TaggedNode`, the actual Java class used to represent each node on the graph.

2.3 The TaggedNode

The graph model in [Mehl99] represents each node of the graph as a tagged pointer consisted of two components: the tag (4 bits) and the pointer (28 bits). Because both the components can fit into one memory word, this allows for fast access to the node, and fast recognition of node types (transient, reference or

determined). There is a restriction, however, of 16 (2^4) basic tag types. Any data type other than the original types must be grouped under the EXT tag, which incurs a runtime overhead.

In Java, such a direct manipulation of pointers is not allowed. Instead, the closest resemblance of a pointer - the object reference - is closely guarded and cannot be changed. In order to incorporate a tag for each class, the tag must become a separate field inside each object. By having an entire integer field dedicated to the tag, there is little restriction on the number of tags. Compared to Mozart, however, this has the obvious disadvantage of another layer of indirection: every node access or type checking involves extra instructions. In fact, for numeric values (integers and floats), the overhead will be two operations for boxing and unboxing, one from `TaggedNode` to the integer (or float) object, and one from the object to the actual numeric value. In addition, there will be a space overhead, as each numeric value requires two Java objects. This overhead is significant even when compared to the DML implementation where primitive `int` and `float` with one layer of wrapper are used. We assert, however, that this is inevitable if we are to achieve a numeric system with infinite precision in the future.

A `TaggedNode` is thus consisted of a tag and a generic Java object:

```
public class TaggedNode
{
     protected int         tag;     // The tag of this node
     protected Object      obj;     // The actual value
     public boolean void getTag() {
      return tag;
     }
}
```

All objects of type `TaggedNode` are created directly on the Java heap. In Java only their references need to be kept track of and worked on by programs. We discuss the placement of these references and how it is maintained in section 4.1.

Exceptions in Oz are also first-class values, which is `TaggedNode` in Berlioz. Because Java allows only subclasses of `Throwable` to be thrown as exception, a natural workaround would be to provide a wrapper class `OzException` that subclasses `Throwable` and contains a `TaggedNode`. However, we can go one step further and specify that `TaggedNode` inherit from `Throwable` directly. This way not only we ensure that all values in Berlioz can be thrown as exception, there is no need to make any explicit conversion from an exception to/from an Oz value. The drawback of inheriting `Throwable` directly, however, is that there will be a runtime penalty [SW00]. To implement this scheme, all we have to do is to insert an `extends` clause in the class definition, as follows (the rest of the class remains as is):

```
public class TaggedNode extends Throwable
{
     //...
}
```

The tags that are currently supported are:

- REF: reference node.
- TRANS: transient.
- INT: integers (covered in section 2.5)
- FLOAT: floating-point numbers (covered in section 2.5)
- ATOM: Oz atoms (covered in section 2.7)
- NAME: Oz names (covered in section 2.7)
- PROC: first-class procedures (covered in section 4.3)
- RECORD: records, lists and tuples (covered in section 2.6)
- LOCK: lock objects (covered in section 2.10)

- CELL: normal cells (covered in section 2.9)
- CLASS: Oz classes (covered in Chapter 5)
- OBJECT: Oz objects (covered in Chapter 5)
- JCLASS: Java class objects (covered in section 5.7)
- JOBJECT: Actual Java objects (covered in section 5.7)

We do not support the THREAD tag (for first-class threads), since invocation on the thread object in Java is not allowed (see section 3.1 on threads for more). Among these tag types, only TRANS and REF are special types that frequently need recognition (during unification, for example). All remaining types are determined values that usually are not referred to unless a type-specific operation is being executed (for example, when performing lock). Because the number of special tag types is minimal, checking for a transient or reference tag can be done using a simple integer equality check:

```
public class TaggedNode extends Throwable
{
     //...
     public boolean isTransient() {
      return tag == TRANS;
     }
     public boolean isReference() {
      return tag == REF;
     }
}
```

To ease node creation at runtime, there is a `NodeFactory` class that constructs and returns nodes of different tags. Some frequently used literal values are statically created to speed up node construction and help reusability, for example the Oz names `true` and `false`, or the integers from –1 to 4.

2.4 Unification

Unification in Berlioz uses an algorithm similar to that described in [Mehl99], which is a variation of rational tree unification algorithm for cyclic trees. The details of rational tree unification algorithms can be found in [Col82] and [HS84]. Some slight differences compared to what is mentioned in [Mehl99]:

- In case unification for any two nodes fails, an exception is raised immediately in Berlioz, whereas in [Mehl99] only a **fail** flag is set in termination status, but the unification process continues until there is no more nodes to unify. We choose to terminate the process early since there is no need to continue once the result is already determined (i.e. failure).
- We optimize for the case where no trees (records) are involved. That means the creation of the stack for visited nodes and nodes to be unified are delayed until both of the nodes are determined to be records.

Pseudo code for the unification algorithm used in Berlioz is presented below:

```
public static void unify(TaggedNode node1, TaggedNode node2)
{
   Stack todo = null;
   Set explored = null;
   do {
      // If todo is not null, this is NOT first iteration
      if (todo != null) {
         while (!todo.isEmpty()) {
            (node1, node2) = todo.pop();
            if (explored.contains(node1, node2)) continue;
            if (todo.isEmpty()) return; // No more nodes
         }
      }
```

```
      if (node1 == node2) continue;
      if (node1.isTransient()) {
         if (node2.isDetermined()) node1.bind(node2);
         else if (node1 != node2) node1.setReference(node2);
      } else if (node2.isTransient()) {
         node 2.bind(node1);
      } else { // Here we're sure both nodes are determined
         if (node1.equals(node2)) {
            if (node1.tag == RECORD) {
               if (todo == null) {
                  todo = new Stack();
                  explored = new Set();
               }
               explored.add(node1, node2);
               for (Iterator ite = getArity(node1); ite.hasNext(); ) {
                  Object feat = ite.next();
                  todo.add(node1.select(feat), node2.select(feat));
               }
            }
         } else {
            throw NodeFactory.createAtom(
               "Binding different logic variables:" + node1 + "&" + node2);
         }
      }
   } while (todo != null && !todo.isEmpty());
}
```

2.5 Integer and Float

For these numeric types, currently we are using internal Java wrapper data types (`java.lang.Integer` and `java.lang.Float`). The advantage is that this is simple and easy to implement. The downside is that Java data types have certain restriction on precision. In particular, Oz integers have infinite precision, which is not achievable with pure Java integer. However, because operations that required access to the numeric value are rare and localized, effort required to change the runtime type of integers and floats in the future is insignificant. (The nodes are only narrowed down to numeric values in negation, binary arithmetic and pattern matching operation).

2.6 Records, Lists and Tuples

The three types: record, tuple and list share the same tag RECORD, but their underlying implementation is different. For generic record type, we use a subclass of `SortedMap` in Java to provide both sorted features (this is needed for arity comparison) and hash access. Because elements in `SortedMap` must be of type `Comparable`, we define a total order on features as INT < ATOM < NAME, where natural order is used when comparing elements of the same type (for names, their internal representation is transformed into strings before comparison). Tuple is implemented using simple Java array, in which array indices directly correspond to the features. Note that this simple type is only used for actual tuple whose width is already fixed. Lists known at compile time are slightly optimized by using a Java array whose elements are list elements. For general cases, lists are naïvely implemented using proper pairs, which are tuple of two fields. Another possible optimization for lists and tuple would have been to give them each a separate tag in order to speed up recognition and pattern matching. We choose not to adopt that, however, since type recognition can also be done efficiently using a common virtual method implemented differently for each type. Moreover, the use of different tags may complicate equality checking and unification unnecessarily.

Because of different types involved, we provide a common interface for record to hide away the underlying implementation. This interface encompassed all operations that must be supported by all records, such as feature selection, adjoining (record extension, getting the label, etc.). Later on in Chapter 5 when the object system is discussed, we shall see that feature selection are not particular to records, they are also available on classes,

objects, and, with an extension for low-level access to Java (see section 5.7), even Java objects. Therefore, we will deliberately separate the interface used for feature selection from that for the generic record. Our definition of the two interfaces looks as follows:

```
public interface IFeatureContainer
{
     public TaggedNode lookupFeature(TaggedNode feature);
     public boolean hasFeature(TaggedNode feature);
}

public interface IRecord extends IFeatureContainer
{
     public void adjoinAt(TaggedNode feature, TaggedNode newField);
     public TaggedNode getLabel();
     public TaggedNode getWidth();
}
```

By using a generic interface, all operations on records can proceed without knowing the actual underlying type, whether it is a proper record, list or tuple. We can also freely replace any record data type with another, provided the latter also implements this interface.

2.7 Names and Atoms

Names and atoms are records whose width is 0. As such, a separate class that implements the interface `IRecord` is used to house names and atoms. Most of the methods in this class are trivial. The only method that is of interest is `getLabel()`, which returns the name or atom itself. Because atoms in Oz are unique, one hashtable for atoms is maintained inside the `NodeFactory`. When an atom is requested, the factory first checks if it is already created, in that case the node is returned, otherwise it is created and added to the hashtable. Note that the use of a hashtable in this case means all created atoms exist persistently (at least until the system shuts down). With the recent introduction of the Java reference package `java.lang.ref`, a future enhancement will be to make use of the `WeakReference` or `SoftReference` class, to make sure that atoms are kept in memory only when needed, and that they are garbage collected properly when not in use.

2.8 Procedures

Not having discussed the internals of first-class procedures, we shall restrict ourselves to the premise that procedures become objects in the JVM, which are created at procedure definition and stored in the `obj` field of the `TaggedNode` like any other Oz value. This object can then be retrieved and cast to appropriate type before being used during procedure application. In section 4.3, we will go back to this topic and discuss how lexical scoping and first-class procedures can be realized.

2.9 Cells

Cells in Oz are “memory slots” that can have their content changed, similar to variables in other languages like Java or C++. Since the `TaggedNode` class already offers a ‘wrapper’ around a generic Java object, implementing cells is a simple matter of manipulating ‘wrapped’ object, i.e. the `obj` field of each `TaggedNode`. Accessing the cell means retrieving the value of the `obj` field, cell assignment means to assign `obj` to a new value, and exchange means a combination of those two actions, while atomicity is achieved by synchronizing on the cell object itself (using the equivalent of the Java keyword `synchronized`).

2.10 Lock

Locks in Oz are special values that can be synchronized on during execution of some of statements. We implement this in Berlioz by wrapping the relevant block of code in the equivalent of a Java `synchronized` block, with the synchronized object being the lock object itself. There is a runtime check on the object being locked to make sure it is indeed of type lock (i.e. the tag is LOCK).

2.11 Classes and Objects

Classes and objects in Oz will be discussed in Chapter 5. For the time being, it suffices to say that objects and classes also become first-class values in Berlioz, which can be assigned to the `obj` field in `TaggedNode`, with the tag being either OBJECT or CLASS.

2.12 Summary

All values in Berlioz are wrapped inside the general-purpose class `TaggedNode`. The actual types of the values are differentiated using a *tag*. Recognizing a tag is usually delayed until absolutely necessary. The actual value is stored inside `obj` field of each object of type `TaggedNode`. All nodes are laid out in a graph-like topology, which are created directly on the Java heap. Integers and floats are implemented using the Java classes `java.lang.Integer` and `java.lang.Float`, respectively. Records, tuples, lists, names and atoms all implement the same interface that allows feature selection and label retrieval, but with different underlying implementation. Procedures, classes and objects are implemented using distinct Java classes. Their instantiated object is stored in the `obj` field just like any other first-class value.

Chapter 3 Concurrency

3.1 Threads in Oz

On the use of threads, we have a choice: either to reuse Java threads, or to develop our own thread mechanism. We found the latter a rather overwhelming task, since in that case native library has to be built in many platforms (when Java is available). Also, efforts need to be spent to make sure the customized library can fit in well with existing Java libraries (which would be nearly impossible considering the enormous amount of standard Java libraries today). A customized native library would also require special privileges to be executed, since making native calls is a restricted right in Java security model. The choice therefore comes easily to reusing Java thread, which is both simple and convenient.

The use of Java thread, however, presents us with its own disadvantages. First, threads in Java are not cheap, and certainly we cannot boast of the statistics achieved by native threads in Mozart system, (upward to 100,000). Second, thread allocation in Java is not fair, which means we cannot guarantee fair slice of processor time to created threads. Lastly, we have to depend on what Java provides for thread manipulation, which is rather poor: suspension, resumption and termination of thread are all deprecated, as they are deemed to be deadlock-prone. It is this poor support that prompts us to look for another alternative to achieve thread suspension and synchronization, which will be discussed in the next section. In addition, we also do not allow direct invocation on thread objects to prevent possible misbehavior.

3.2 Thread Suspension

Threads in Oz suspend when accessing the content of a transient. The Mozart system achieves this by explicitly storing the thread in a `suspendList` inside the transient node itself, and explicitly wakes up all threads in this list when the transient is bound [Mehl99]. Because built-in Java threads are used in Berlioz, we are limited by Java restrictions. In contrast to Mozart, explicit manipulation of threads is discouraged in Java, since they potentially can lead to deadlock situations. As such, we have to settle on another way to achieve thread suspension, which was previously taken by the DML project [SW00]: to use *wait/notify* mechanism inherently supported by every Java object. Whenever a thread wants to wait until a transient is bound, it calls `Object.wait()` on the transient. This way there is no need to explicitly keep a list of suspended threads, as they will all be waiting on the same object. When the transient is bound, it simply invokes `Object.notifyAll()` on itself, which will wake up all suspended threads. Typical code for a transient is as follows (exception handling is omitted for clarity[1]):

```
public class TaggedNode
{
     public Object getValue() {
      if (this.isTransient()){
            synchronized (this) {
            this.wait();
          }
      }
      if (this.isReference())
            return this.deref().getValue();
      return obj;
     }

     public void bind(Object obj, int tag) {
      // we ignore checking if this is already bound
      this.tag = tag;
```

[1] The call to `Object.wait()` throws `InterruptedException`, which must be caught or re-thrown. We choose to omit its handling, however, since its omission does not affect much program execution flow.

```
        this.obj = obj;
        synchronized (this) {
              this.notifyAll();
        }
      }

      public TaggedNode deref() {
        TaggedNode node = this;
        while (node.isReference()) node = node.obj;
        return node;
      }
}
```

3.3 Variable Equality

A problem arises when we want to establish transient equality. Two transients can be equal even when they are not yet bound, namely after they have been unified. If we have an equality expression, for example (X == Y), our objective obviously will be to know the result as soon as possible. If the result is not yet determined and both X and Y are still unbound, the execution thread needs to suspend until new information arrives at *either* of X or Y. However, we should not wait until X and Y is bound in order to wake up, because that may be too late. During that time X and Y (or some other variables that have been previously unified to X or Y) may have been unified without being bound, in which case the test can return true.

In this case, it is necessary that the execution thread does not call `Object.wait()` on either of X and Y, but on a common object called *synchronized object*, which is shared and stored by both X and Y[1]. When something new happens at X or Y by result of either unification or binding, the action wakes up the suspended thread by calling `Object.notify()` on the synchronized object. We call this kind of thread suspension *soft waiting*, because the thread is woken up not just when the transient is bound, but whenever new information arrives. We reuse the name `waitOr` from Mozart to call the actual static method.

Because there can be many threads waiting for new information this way, the synchronized objects (created from `waitOr` calls) must be kept in a suspension list. Each time `waitOr` is called, a new *synchronized object* is created and added to the suspension list of both variables involved. At the time of thread wakening, the list will be traversed, `Object.notify()` is called on all of its members, after that the list is reset (to be empty). (Because two `notify()` calls can potentially be made for each member of the list – one from each transient, often one will be unnecessary. The redundancy is harmless, however). Sample code for equality checking is as follows:

```
public class TaggedNode
{
      public boolean equals(TaggedNode node) {
        TaggedNode t1 = this.deref();
        TaggedNode t2 = node.deref();
        while (t1.isTransient() && t2.isTransient()) {
              if (t1 == t2) return true;
              TaggedNode.waitOr(t1, t2);  // Wait for new information
              if (t1.isReference()) t1 = t1.deref();
              if (t2.isReference()) t2 = t2.deref();
        }
        if (t1 == t2) return true;
        return (t1.tag == t2.tag &&
              t1.getValue() == t2.getValue());
      }
```

[1] Here the point is that this synchronized object must be accessible by both X and Y, so that we can be notified whenever something new happens, regardless of whether it occurs at X or Y. We could have taken either X or Y as that object, and pass it to the other party. However, a new object probably helps make things clearer.

```
}
```

3.4 Summary

Java thread is used directly at runtime. Thread synchronization is implemented using Java's wait/notify mechanism. The object on which `wait` is to invoked can be either the transient itself, or an artificially created object. These objects need to be kept in a `suspendList` that is released and `notify`-ed when new information arrives, either as a result of binding, or when a transient becomes a reference node.

Chapter 4 The Compilation Environment

4.1 Accessing the Constraint Store

In Berlioz, a new variable can be introduced explicitly through the use of `local ... end` construct, procedure parameters, and pattern matching variables (see section 4.5 on pattern matching for detail), or implicitly (for globally defined variables such as `Show` or `NewName`)[1]. In Java, as mentioned in section 2.3, all variables are created on the heap, and in order to access them at runtime, there must be a way to obtain their object reference. For strictly local variables, we can directly reuse the local variable stack of the JVM. For each newly introduced variable, we reserve a slot on this stack to store its object reference. This slot will then be reclaimed when the variable goes out of scope. Procedure parameters (which are compiled to Java method parameters) are treated similarly since the JVM also places them on the local variable stack[2]. The exception in this case is that since their slots are handled by the code generator (discussed in section 6.5), we do not need to reserve and reclaim them. For globally defined variables, their references are kept in static fields inside a Java class accessible at runtime.

With the above arrangement, however, we still have problem with variables of intermediate scope. Certain constructs (thread, procedure and class) require to be compiled into distinct Java classes[3]. In these cases, because the local variables are out-of-scope (with regards to Java), they are no longer accessible inside the created classes. Hence, it is necessary to pass the object references of these variables and keep them in internal Java class fields. Thus in summary, variables in an Oz program can be accessed from three different sources:

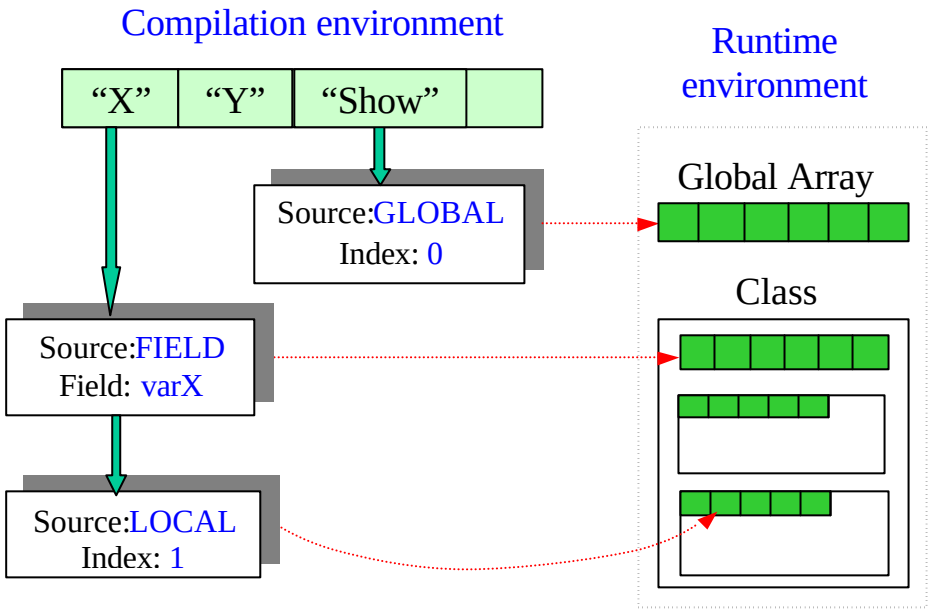
- Globally bound variables such that `System` and `NewName` are stored in a statically created array.
- Local variables such as those introduced by the `local ... end` statement are stored in the local variable area of the JVM. These are allocated as needed, and reclaimed when the variables go out of scope.
- Procedure parameters are already stored in the local variable area by the JVM. Their indices are used to generate bytecode, but no allocation is necessary.
- Variables of intermediate scope (for threads, procedures and classes) are stored as internal Java class fields.

In order to generate bytecode for accessing the variables, in many cases we need to know runtime information as to whether they are. This includes the source (where their references are kept), and the *position* of these variables. (The term position is used here in a rather broad sense, which means either the index in local variable stack, the name of a class field or the index of a static array). In these cases, we have to simulate what happens at runtime to know position information, so that correct bytecode can be generated. In addition, this simulator must be able to handle lexical scoping, i.e. the suppression of previous variables of the same name when a new variable comes into scope. In Berlioz these are achieved using a compile time simulator called *compilation environment*, which is described in more details in the following section.

4.2 Compilation Environment

This simulated environment is implemented in the form of a hashtable that hashes the name of each variable to a linked list, of which each element is a structure containing position information. For variables on local variable stack, this is simply the index. For class field, information is the name of the field. For external static array, both the field name and array index are stored (or only the index if the field name is already known). Each time a new variable comes into scope, its name is hashed into the hashtable to see if a variable of the same name already exists. (By the term *name*, we mean the literal string as appears in Oz source, for example `X, Y, System`, etc.). If it does, position information for the new variable is appended to the linked list. When that variable goes out of scope, its position information is simply removed from the list. The process is illustrated as follows:

[1] Strictly speaking, the implicit introduction of these variables is only for programmers' convenience during development. When functor is implemented, any functor that makes use of these predefined variables will have to declare and load them explicitly.
[2] The parameter indices start from 0 for static methods and 1 for non-static methods.
[3] More details on the compilation of these constructs will be discussed later.




```
proc {P P1}
   X = 1 %% X is a variable of intermediate scope
   proc {P1 X}
      X = 2 %% This X is a local variable
   end
end
```

Figure 4-1: Compilation Environment

During compilation, whenever there is a need to access a variable, we retrieve position information for the nearest occurrence of that variable from the hashtable by hashing the variable name. Assuming there is one linked list entry found, the top-most structure will contain the position we need, based on which appropriate bytecode can be generated. If no entry was found, it means the variable has not been declared, and a compilation exception is raised.

4.3 First-class Procedures

Similar to some other languages (Scheme, ML), first-class procedures in Oz with lexical scoping also have to provide access to variables of intermediate scope. Typically, this is implemented by creating a closure, which encompasses the procedure code and the environment at the time of procedure creation. In Java, the closure concept can be achieved using a Java object, which contains reference to the surrounding environment. In fact, the inner class concept introduced to Java since JDK1.1 can be seen as an object-oriented form of closure. Each inner class, apart from its own method definitions, also contains a reference to the surrounding class. Since inner classes are eventually unnested into top-level classes anyway, we can achieve the same effect for closures without actually compiling to inner classes (which will slightly complicates compilation). In Berlioz, first-class procedures are compiled into Java classes with a method designated for procedure application (we will refer to this as the method `apply()`). Variables of intermediate scope are explicitly passed in the constructor of the procedure class, and stored inside the constructed object as instance attributes. The procedure parameters are passed right before procedure invocation (i.e. when the method `apply()` is called). For procedures that take more than one parameter, we choose the simple way of retaining the number of parameters as is. This way, the parameters can be simply retrieved and placed on operand stack one by one immediately before `apply()` is

invoked. This means the arity (number of parameters) of `apply()` method may vary, depending on the actual number parameters as appear in Oz source.

There are two problems with varying number of parameters, however. The first involves method signature for method invocation. Java requires both the class name and method name to be known before invocation. As the name of procedure class may not be known at the time of compiling the procedure application, it is not possible to insert bytecode that makes a virtual call at this point. Instead, this is circumvented by calling an interface method in Java, provided the signature of that method is known, and an interface containing that method signature exists. To prepare for all possible cases, however, we need to create an interface that defines *all* the possible `apply()` methods, or create a large number of interfaces just to accommodate these methods signature. Both of these are, of course, quite inefficient. The second problem is when a procedure has too many parameters. The JVM only allows a maximum of 255 parameters per method [LY99], so any procedures with more than 255 parameters cannot afford to pass them as simple method parameters. One way to overcome these problems is by mandating that all procedures have 1-arity, in which case all parameters must be grouped in a tuple, or an array before being passed. That would save us from having to create and implement classes with `apply()` methods of different arities with the side effect of some slowness (due to indirection) and the extra memory requirement involved. In Berlioz, we choose to have a hybrid approach to allow for gracefully degrading of performance, which involves a "threshold" point of parameter count. If a procedure parameter count is larger than the threshold, the parameters will be passed in an array; otherwise they are passed individually. For efficiency, interfaces of common arities (between 0 and 4) are pre-created, while interfaces for arities larger than 4 but smaller than the threshold are created by the compiler only when needed.

In addition to `apply()` method, Berlioz procedures also implement a `getArity()` method as required by Oz core modules [HK00]. As with `apply()` method, `getArity()` is also housed in a special interface whose name is known at compile time. Noting that `getArity()` is only implemented by procedures, we chose to name the interface as simply `Procedure`. This also comes in handy when implementing the Oz core procedure `IsProcedure`: we only need to insert bytecode to check if the object in question is an instance of the `Procedure` interface.

As an example that summarizes above discussions, consider the following Oz code segment:

```
proc {P X}
      X = A * 2
end
```

In Berlioz, this will be compiled into bytecode that is equivalent to:

```
class Proc implements Apply1, Procedure {
      TaggedNode A;
      public Proc(TaggedNode _A) {
            this.A = _A;
      }

      public void apply(TaggedNode X) {
            // procedure code goes here ...
      }

      public int getArity() {
            return 1;
      }
}
unify(P, new Proc(A));
```

Note that for all procedures, `apply()` method is always `void`, i.e. it does not return any value. This is because procedure application after being transformed to relational core syntax (see section 6.3 for details) does not return any value.

4.4 Threads

Perhaps not so surprisingly, thread construction and execution are very similar to those of procedures. When a thread is instantiated, all variables of intermediate scope are explicitly passed in the constructor and stored in instance fields of the thread object. The equivalent of `apply()` in procedures is `run()` for threads, which is executed when `Thread.start()` is invoked. All we need to do is to fill in the body for `run()`, instantiate the thread and invoke `Thread.start()`, the rest is left to the JVM to finish off.

4.5 Pattern Matching

Pattern matching in Oz tries to determine if a given expression (literal, variable, tuple, list or record) is of a certain pattern, which can be a literal, a (named or unnamed) variable, or a structural combination of variables and/or literals. Below we use the term *matching expression* to refer to the expression used for pattern matching. For different patterns, the conditions required for matching are as follows:

- A literal pattern matches only when the variable is bound to the given literal value.
- An unnamed variable pattern (represented by the underscore) matches all variables.
- A named variable pattern always matches if (a) it appears only once in the pattern, and (b) the variable is newly introduced, not one escaped with an exclamation mark '!'. In this case there is a side effect: the variable is installed into local variable space for the execution of the corresponding `case` branch.
- If a named variable pattern is introduced more than once in the pattern, e.g. `X|X`, or if its introduction was suppressed, e.g. `!X`, equality check must be done. Only the first appearance of a non-escaped variable is installed into local variable space as side effect. For example, only the first X is installed for pattern `X|X`, while no new variable is installed for `!X`.
- A structural pattern matches when the matching expression is of the same structural type (record, list or tuple) with the same arity, label, and all elements (label, features and fields) of the matching expression match pairwise with the corresponding elements of the pattern. In case the matching succeeds, the side effect is the total side effect of all the label, features and fields of that pattern. Note that variables at positions of label or features are implicitly escaped and therefore must be matched using equality check. In other words, the variables at these positions cannot be introduced ones.

In Oz, pattern matching is carried out in a strictly sequential manner. Execution thread may suspend when matching a certain pattern if information required to determine the outcome of matching is not sufficient. Patterns are tested one by one from the top down, until the first pattern that matches, at which point control is passed to the branch corresponding to that pattern. If no pattern matches, control is passed to the `else` branch if such a branch exists; otherwise an exception is raised.

When a named non-escaped variable appears in a pattern, it must be installed[1] into local variable space so that code belonging to that pattern branch can have access to it. (Note: this does not apply to record label and features, which are implicitly escaped). They must be uninstalled (i.e. marked as invalid), however, at the end of pattern matching if tests for that pattern fails, so that local variable space can be restored and ready for the next pattern. Problem arises when we consider the case of partially matched pattern. Matching has failed, but some variables may have been installed. Should we have kept track of those installed, so that we know which ones must now be marked as invalid? Also, what if a non-escaped variable appears more than once in a pattern? How can we ensure that at first appearance, it should only be copied to its allocated slot (in local variable stack), but at subsequent appearances, it should be tested for equality? With an all-by-delegation compilation scheme (see section 6.6 for more details on the compilation process), a number of housekeeping tasks need to be performed. In particular, sub-patterns must somehow know of their roles in the 'parent' patterns to compile correctly, while 'parent' patterns must ensure they 'clean up' after their sub-patterns properly.

It turns out that all these housekeeping jobs are unnecessary simply by using the logical *compilation environment* (discussed in section 4.2). At the time of pattern matching compilation, all new variables are simply declared and reserved a slot in local area stack. This is only a logical process, so no bytecode has been

[1] Here installation means to initialize the corresponding slot in local variable stack with a valid object reference, by first performing an object storing instruction (the Java bytecode `astore`).

generated, and actual runtime stack has not been affected. When a non-escaped variable appears in the pattern, the logical compilation information is read. If the slot is still empty, an object-storing bytecode is generated, otherwise, bytecode necessary for reading the object and performing equality check is generated. The logical variable slot information is retained when compiling the accompanying branch, and then simply discarded. This way there is no need to keep track of and invalidate any actual runtime slot if pattern matching has failed. The fresh variables from subsequent patterns will simply be written over the previously allocated slots, without knowing whether these slots contain reference to 'live' objects or just garbage.

4.6 Summary

The variables at runtime can come from three different sources: global static fields, local variable stack and class fields. Information about the source of variables is obtained by using the compilation environment, which simulates the runtime environment. First-class procedures are implemented by Java class with internal holders for variables of intermediate scope. These variables are passed during procedure creation and used during procedure application. Procedures implement a special `Apply` interface that defines an `apply()` method, which allows procedure application to be invoked without knowing procedure class names. Threads are implemented likewise, however the `apply()` method is replaced by `run()`. The compilation environment helps detect the positions of both procedure parameters and variables of intermediate scope to allow correct bytecode generation. Pattern matching is carried out as a sequential series of boolean tests, in which the installation of (fresh) pattern variables is facilitated by using the same compilation environment that maintains other variables.

Chapter 5 The Object System

5.1 Objects in Oz

The object system in Oz is a language extension that seeks to integrate object-oriented features in a concurrent, functional setting with lexical scoping. Most important characteristics of objects in Oz are:

- Attributes: objects in Oz can have named attributes, whose value can change over time. Attributes values can be accessed, assigned and exchanged, much similar to cells. If the declaration of an attribute contains an initial value, that value becomes the default initial value for all objects of a class. Otherwise, each new object gets a fresh unbound variable for the given attribute. This is illustrated below:

```
class Cls
      attr a1:10 a2
      meth show1 {Show @a1} end
      meth show2 {Show @a2} end
      meth inca1  a1 <- @a1 + 1 end
end
Obj1 = {New C show1} %% 10 is displayed
{Obj1 inca1}         %% Increase attribute a1 of Obj1
{Obj1 show1}         %% 11 is displayed
Obj2 = {New C show1} %% Create a new object, 10 is displayed, not 11
{Obj2 show2}         %% _ is displayed (since value of a1 is unbound)
```

- Features: these are also named entities of Oz objects, with the difference that they can be assigned only once. In many ways, object features are similar to record features, especially considering that object and class feature selection has the same syntax as record feature selection. A feature is shared among all objects of a class if the feature declaration contains an initial value, and in this case, it is accessible from both the class and its objects. If an initial value does not exist, however, the feature is specific to each individual object, and feature selection is only available for the object. This is illustrated below:

```
class Cls
      feat f1:10 f2
      meth init skip end
end
Obj = {New Cls init}
{Show Obj.f1} %% Successful, f1 is a feature of Obj
{Show Cls.f1} %% Successful, f1 is a feature with initial value
{Show Obj.f2} %% Successful, f2 is a feature of Obj
{Show Cls.f2} %% Unsuccessful, f2 is not a feature with initial value
```

- Oz supports the expression `self`, which acts like a regular Oz object: it can accept, dispatch messages and perform feature selection. In the following example, the method `invoke` simply constructs another message and invokes it on `self`. If the message happens to be `show(F)`, self will perform feature selection with the feature `F` and display the result.

```
class Cls
      feat f1:10 f2:20
      meth init skip end
      meth invoke(M N) {self M(N)} end
      meth show(F) {Show self.F} end
end
Obj = {New Cls init}
```

```
{Obj invoke(show f1)} %% show(f1) is invoked, which display 10
{Obj invoke(show f2)} %% show(f2) is invoked, which display 20
{Obj invoke(get f1)}  %% Error, method get(f1) not found
{Obj invoke(show f0)} %% Error, feature f0 not found
```

- Features, attributes and method names are all first-class. In fact, their values are not necessarily determined until runtime. Similarly messages for method invocation are also first-class and may only be determined at runtime. In the example below, a message is computed at runtime before being passed in an object invocation.

```
class Cls
      feat f1:10 f2:20
      meth init skip end
      meth show(X) {Show self.X} end
end
fun {F A} show(A) end
Obj = {New Cls init}
%% Invoke on Obj with some messages computed at runtime
{Obj {F f1}} %% Message show(f1) is sent, displays 10
{Obj {F f2}} %% Message show(f2) is sent, displays 20
{Obj {F ff}} %% Message show(ff) is sent, error: feature ff not found
```

- Objects and classes are all first-class. They can be bound to any variable, just like other values.

```
%% Here Cls is bound to an Oz class
class Cls
      feat f:20
      meth init skip end
      meth show {Show self.f} end
end
B = Cls            %% Bind  to Obj
Obj = {New B init} %% Create an object of type B
A = Obj            %% Bind A to Obj
{A show}           %% Invoke on A, 20 is displayed
```

- Oz allows multiple inheritance.

```
class A
      meth a() {Show 'inside A'} end
end
class B
      meth b() {Show 'inside B'} end
end
class Cls from A B
      meth init() skip end
end
Obj = {New Cls init}
{Obj a} %% Successful, 'inside A' is displayed
{Obj b} %% Successful, 'inside B' is displayed
```

- Oz supports full compositionality for classes: they can appear anywhere in source code. In the following example, a class definition is created with each application of procedure P. This class then takes a parameter of procedure P (which is actually a variable of surrounding scope) as a method name. Note how the method name is undetermined at compile time and resolved at runtime.

```
proc {P A C}
```

```
      class C
         meth A {Show A} end
      end
end
thread MName = m end %% A thread that binds MName to the atom 'm'
{P MName M} %% Bind M to a class with method MName
            %% This suspends until MName is bound, which is after
            %% the above thread terminates
{P n N} %% Bind N to a class with method 'n'
{New M m} %% Display 'm'
{New N n} %% Display 'n'
{New M n} %% Error, method 'n' not found in object of type M
```

An extra note that can be seen from the above examples is that objects are instantiated using the object creation procedure `New`, taking in the class variable and an initial message, as follows:
`{New Class InitialMessage ?CreatedObject}`.

5.2 Basic Implementation

Since Java itself is an object-oriented language, it may be tempting to reuse its internal object and class mechanism for Oz. In fact, doing that would bring about substantial benefits, both in terms of performance and development effort. Ideally we would be able to make a direct mapping between Java classes and Oz classes, replace the call to `{New ...}` in Oz with the Java operator `new`, and rely solely on Java's method invocation for Oz method invocation. Unfortunately, that direction is not feasible primarily for the following reasons:

- Oz classes allow multiple inheritance, while Java only allows single inheritance. In section 5.4 below, we shall see how multiple inheritance can be achieved using internal Java class fields.
- Features, attributes and message names in Oz are all first-class, and only determined at runtime. On the contrary, Java class definition requires all class fields and method names to be known at compile time.
- Oz methods are invoked using first-class messages. Because method names in Java are static and must be known at compile time, there is no way messages determined at runtime in Oz could be dispatched using simple Java method invocation. In section 5.5 we shall come back to this topic and see how runtime message dispatching is approximated in Java.

As a result, we shall be using Java classes to explicitly store necessary inheritance information for Oz class. The generated (Java) class is also the place where code for all methods resides. Access to features and attributes, on the other hand, are not provided from the class, but from the instantiated objects. To facilitate runtime message dispatching, all methods in each Oz class are grouped together into a single `applyMessage()` method that takes in the incoming message and performs appropriate message distribution and processing. To enable late binding, `applyMessage()` must be prepared to take in the reference of the current object, which is much similar to the implicit piggybacking of the *this* pointer inside all non-static methods in both C++ and Java. Note that even though the JVM does make available the *this* pointer in every non-static method (it is located at index 0 in local variable stack), that pointer in `applyMessage()` is not usable for us. The reason is because it is actually the runtime representation of the Oz class, not the Oz object we expect.

A typical signature for the `applyMessage()` method thus looks as follows:

```
public void applyMessage(OzObject thisPtr, TaggedNode message)
{
      // Method dispatching code goes here ...
}
```

Since generated bytecode for each Oz class is associated with and contained in the class, not the instantiated objects (of that class), object invocation of the form `{Obj mesg}` simply delegates the incoming message to its owner class:

```
public class OzObject
{
 public void processMessage(TaggedNode message) {
      owner.applyMessage(this, message);
 }
}
```

For class invocation of the form `ClassVar,mesg`, things are much similar, except that now `applyMessage()` is invoked not on the owner class, but on the particular class that is bound to `ClassVar`. In Oz, class invocation can only appear inside a method definition, which means in compiled code, it must appear inside an `applyMessage()` method itself. The *this* pointer in this case will be the *this* object that is the first parameter of the surrounding `applyMessage()`. A typical compiled Oz class with a method that makes the call `ClassVar,mesg` will then look like:

```
public class ConstructedClass extends OzClass
{
 public void applyMessage(OzObject thisPtr, TaggedNode message)
 {
      // ...
      (ClassVar.getObject()).applyMessage(thisPtr, mesg);
      // ...
 }
}
```

One interesting detail when comparing Oz with other object-oriented languages (C++, Java) is that in Oz, a class invocation of the form `ClassVar,mesg` is not what it may seem. It is neither a static call as in Java, nor is `ClassVar` a super class of the invoked class as it is in C++. Instead, `ClassVar` can be any class, and the call is still a virtual call. It is the *this* pointer which determines whether access to features and attributes is successful. Consequentially, there are pieces of code in Oz that get executed successfully, but may look like error in other languages. Take the example:

```
class A
      attr a:10
      meth show {Show @b} end
end
class B
      attr b:15
      meth init
            {Show @b} %% Successful, 15 is displayed
            A , show  %% Successful, 15 is displayed
      end
end
```

At first glance, class A seems not compilable, since the method `show` accesses a seemingly non-existing attribute `b`. However, this is allowed in Oz (and compilable in Berlioz), because features and attributes access rely solely on the *this* pointer, not on the surrounding class. When the method `init` is invoked on an object of class B, it in turns invokes method `show` on class A and pass itself as the *this* pointer to A. Being an object of type B, the *this* pointer does contain an attribute named `b`, which means the statement `{Show @b}` can be executed successfully.

Finally, because objects simply delegate their work to the owner class most of the time, the difference in behavior between different Oz objects becomes almost negligible. If we consider the fact that in Oz all object initialization work (attribute and feature initialization) are done at runtime, all Oz objects are essentially identical (albeit with different owner classes). Indeed, this is the way objects are implemented in Berlioz. Each Oz object becomes instance of the same Java class (hereby referred to as `OzObject` class). At runtime each

receives and stores a reference to their owner class. After extracting and storing necessary features and attributes locally, the object simply waits for incoming messages to send to the owner. The equivalence of the Oz procedure `{New ...}` in Java will be as follows:

```
public class NewProcedure implements Apply3
{
 public void apply(TaggedNode clazz, TaggedNode msg, TaggedNode result)
 {
      OzObject newobj = new OzObject((OzClass)clazz.getObject());
      newobj.apply(msg);
      unify(newobj, result);
 }
}
```

5.3 Seamless Syntax Integration

Syntactically, most constructs related to the Oz object system are distinct enough for straightforward compilation. There are two syntactic features, however, that require attention: object invocation and feature selection for both objects and classes.

The first case, object invocation with first-class messages, has the same syntax as procedure invocation. At source code level, an object invocation looks exactly the same as a procedure invocation with one parameter. To avoid the need to make the (unnecessary) runtime distinction between procedures and objects, we can make Oz objects implement the same interface that procedures do. Because Oz objects are invoked with one message, they must have an `apply()` method that takes one parameter. Typical Oz objects thus look in Java as follows:

```
public class OzObject
{
 OzClass owner;
 public OzObject(OzClass ownerClass) {
      this.owner = ownerClass;
 }
 public void apply(TaggedNode message) {
      owner.applyMessage(this, message);
 }
}
```

The second case, feature selection for both objects and classes, has the same syntax as record feature selection. Similar to the first case above, objects, classes, and records feature selection are totally indistinguishable at source code level. We could implement runtime check for value type, or better still, make objects and classes implement the same interface that records do. This leads to the following definition of objects and classes (note that `OzClass` is declared *abstract*, as it must be subclassed before it can be used).

```
public class OzObject implements Apply1, IFeatureContainer
{
 OzClass owner;
 public OzObject(OzClass ownerClass) {
      this.owner = ownerClass;
 }
 public void apply(TaggedNode message) {
      owner.applyMessage(this, message);
 }
 public TaggedNode lookupFeature(TaggedNode feature) {
      // Feature lookup code goes here ...
 }
 public boolean hasFeature(TaggedNode feature) {
      // Feature existence check code goes here ...
```

```
 }
}

public abstract class OzClass implements IFeatureContainer
{
 public TaggedNode lookupFeature(TaggedNode feature) {
      // Feature lookup code goes here ...
 }
 public boolean hasFeature(TaggedNode feature) {
      // Feature existence check code goes here ...
 }
 // This method must be implemented by subclasses
 public abstract void applyMessage(OzObject thisPtr, TaggedNode message);
}
```

5.4 Inheritance

As mentioned in section 5.2, one important reason that prevents Oz classes from a simple and direct mapping to Java classes is that Oz allows multiple inheritance. To this end, we have no choice but approximate the effect by storing the reference to super classes in each Oz class. To avoid duplicated features, attributes and methods names in classes at the same inheritance level[1], these super classes must be checked during class construction. Each class contains a hashtable of all features and attributes, which hashes the feature and attribute names to their initial value if exists, or to *null* if there is no initial value. Each class also contains a method hashtable that hashes the method name to the appropriate class. When a class is constructed, these hashtables must be built, after that the super classes are traversed to check for duplicates. For reusability, this piece of code is kept in an abstract class (`OzClass.java`) that must be inherited by all compiled Java classes. Pseudo code for this process is as follows:

For each features of this class
    Check for duplicate feature
    Mark as class feature or object feature
    Store feature and initial value (if any) in feature hashtable
For each attributes of this class
    Check for duplicate attribute
    Mark as class attribute or object attribute
    Store attribute and initial value (if any) in attribute hashtable
For each method name of this class
    Check for duplicate method name
    Store method name in method table
If super classes is not empty
    Build inheritance graph
    For each set of classes in graph with same inheritance distance
        Check for duplicate of features, attributes or method names

For objects, each time a new object is created the feature and attributes hashtables are traversed to initialize those entries without an initial value. The localized version of these hashtables (with all *null* entries properly initialized with fresh variables) is copied over to the object, ready to be accessed (using `@`, `<-` for attributes, and feature selection for features). Pseudo code for each object construction would then look as follows:

For each features of owner class
    If feature is object feature
        Create transient as value
    Store feature and value in feature hashtable
For each attributes of owner class
    If attribute has no initial value

[1] For more information on inheritance graph, inheritance distance and closeness, see [Henz97] and [Ste90].

Create transient as initial value
Store attribute and value in attribute hashtable

5.5 Methods

We have known that all methods as appear in Oz source code will be compiled into one single `apply` method with one parameter. How would we distinguish the appropriate message and invoke corresponding bytecode? Since the format and behavior of method patterns are almost the same as normal patterns, it is convenient to reuse pattern matching in this case. Because pattern matching takes the entire message, including method name and features into account, while method name in Oz are already unique, this may look slightly inefficient. However, we argue that this is actually not so. If method name matches, pattern matching for features simply extract the corresponding fields to initialize introduced variables (which are often called "method parameters" in other object-oriented languages), so that must be done sooner or later anyway. Inefficiency instead occurs when a message, statically known at compile time, is created before invoking a method, only to be broken up immediately at method entrance (with the use of pattern matching). There is one optimization for this case mentioned in [Henz97], that is, to delay message creation, hoping that the message itself is not used but only its fields, in which case message creation can be eliminated altogether. Another type of optimization is to abandon pattern matching altogether and instead relying on the unique method names to dispatch incoming messages to appropriate methods. This may require creating an artificial Java method name for each Oz method whose name is not statically known. As method memory addresses are not available in Java, when a message arrives, Java reflection mechanism must be used for dispatching. Because of the complexity involved, we decided not to perform these optimizations at this stage and instead settle for the more generic scheme. As example, given the piece of code below:

```
meth getBoxVolume(length:L width:W height:H)
...
end
meth getSphereVolume(radius:R)
...
end
meth getConeVolume(radius:R height:H)
...
end
```

The compiled code will then be equivalent to:

```
proc {ApplyMessage Message}
 case Message of getBoxVolume(length:L width:W height:H)
      ...
 [] getSphereVolume(radius:R)
      ...
 [] getConeVolume(radius:R height:H)
      ...
 end
end
```

The only scenario that pattern matching is not equipped to deal with is when default values are used in method pattern. In this case we simply ignore the corresponding feature during pattern matching, and then check for its existence later inside method body (or rather, `case` branch body). Take the example:

```
meth getBoxVolume(length:L width:W height:H <= DefaultHeight)
      ...
end
```

The compiled code is equivalent to:

```
proc {ApplyMessage Message}
```

```
// Note: we make sure arity is 3, but ignore the last feature
case Message of getBoxVolume(length:L width:W ...)
 local H in
      if {HasFeature Message height} then
       H = Message.height
      else H = DefaultHeight
      end
      ...
 end
end
end
```

By reusing code written for pattern matching, we can compile methods with relatively little effort. For each Oz class, the set of defined methods, and subsequently, the corresponding bytecode, are different. That means the body of `applyMessage()` for each compiled Java class is also different, and a new Java class must be created for each Oz class that appears in source code.

5.6 Other Issues

To implement attribute access, assign and exchange, we simply reuse the CELL concept. In fact, each attribute is an actual cell, just like any other cell created with the procedure `{NewCell ...}`.

For feature selection, the incoming feature is hashed against the hashtables (in either class or object). If found, the corresponding value is returned, otherwise an exception is raised.

The expression `self` must be available inside each Oz method. In addition, access to `self` is implicitly made for each attribute-related operation. We have mentioned in section 5.2 that a reference to the current object is implicitly [1] passed in every call to the method `applyMessage()` of the owner class in order to facilitate late binding. That very object reference is, indeed, `self`. Each time `self` is needed, all we have to do is to simply insert the bytecode to load that object reference from the parameter area (which is also the local variable stack).

5.7 Extending Oz Objects to Java Objects

Sometimes it may be convenient to have access to pure Java classes and objects by directly creating Java objects, invoking Java methods, etc. from the context of Oz source code. This can be useful when, for example, there is a need to access an existing Java library. Potentially, Oz data can also be passed to Java methods, making the boundary between Oz and Java less distinguishable. To this end, Berlioz allows Java methods (virtual and static) to be invoked in a very similar manner to Oz procedures. Static and non-static field values can also be loaded in a manner similar to feature selection. In many ways, making a virtual call on a Java object, or a static call on a Java class, is much similar to invoking Oz objects. In fact, this is practically the same at source code level. The only difference appears at creation time. While Oz objects are created using the `New` procedure, Java objects are created using a special `MakeJObject` procedure, which takes in a message as the Java class name and constructor parameters. Creating a Java class, on the other hand, involves another dedicated procedure called `MakeJClass`, which takes in a single atom as class name. As example, consider the following piece of code:

```
local AStr String System in
      %% Create the Oz value for the Java class String
      String = {MakeJClass 'java.lang.String'}

      %% Make a static call on the String class, then print the value
      {Show {String valueOf(100)}}

      %% Create a String object
```

[1] Here we say implicit with regards to original Oz source code. In compiled Java bytecode (which is hidden from Oz programmers), the passing of the current object is of course quite visible.

```
        AStr = {MakeJObject 'java.lang.String'('Sample string')}

        %% Create the Oz value for the Java class System
        System = {MakeJClass 'java.lang.System'}

        %% Print out Str1 to console using a virtual call
        {System.out println(Str1)}
end
```

Notice how we manage to retain Oz-like syntax for procedure invocation and feature selection, while still achieve a concise representation comparable to Java (and which should look familiar to Java programmers). Part of this is thanks to the syntax integration scheme we mentioned in section 5.3. We only need to create a new class that implements the `Apply1` (for application with 1-arity) and `IFeatureContainer` interfaces and the requirement of syntax integration is satisfied. Inside the implemented methods (`apply` and `getFeature`), Java reflection mechanism is used to make runtime method invocation and field retrieval. The choice of Java reflection in this case is inevitable, since the messages (method names) and objects are only determined at runtime, though the cost is considerable since Java reflection operations are often very expensive. For the runtime representation, note that we are using the same class to represent both Java classes and Java objects. This is because with Java reflection, operations on static and non-static methods and fields are almost identical. The only difference is that for static cases, the targeted Java object is *null*, which we can conveniently set inside `MakeJObject` and `MakeJClass`. For data passed from Oz context to Java, generally we retain the values as are. The exceptions: Oz names `true` and `false` are converted to Java boolean values of the same name, and Oz atoms are converted to Java strings.

5.8 Summary

Oz classes and objects are represented by Java objects. Oz classes are represented by instances of distinct Java classes, in which inheritance information, attributes and features are explicitly stored as class fields. All Oz objects on the other hand are represented at runtime by instances of the same Java class. All Oz methods are compiled into a single `apply()` method, in which the actual method dispatching is performed through pattern matching. Syntax integration of feature selection (to be the same as that for records), and object invocation (to be the same as that for procedures) is achieved by making compiled Oz objects inherit the same interface as records and procedures do. A simple extension exists to allow Oz communicate directly to the Java sub-system by allowing Java object creation and invocation of both virtual and static Java methods.

Chapter 6 Implementation

6.1 Overview

The compilation process involves parsing proper Oz statements into an abstract syntax tree, which is consisted of Oz constructs (represented by Java objects) in a hierarchy of parent-children relationship. These constructs correspond roughly to actual Oz constructs as appear in the Oz grammar (see Appendix A) and are generally of three types: expressions, statements and patterns[1]. Certain rules from the context-free grammar govern whether a construct of a particular type can appear in a particular position in source code. (For example, an expression cannot appear in a statement position, otherwise an exception is raised). From the abstract syntax tree, bytecode is generated by asking the top-most construct (which is usually just a holder for a list of Oz statements) to compile itself, passing an instance of the code generator (this will be discussed in section 6.5) and the current compilation environment (see section 4.2). The top-level construct compiles itself by simply asking all its children to compile themselves, passing each of them the same copy of code generator and compilation environment. This process is repeated from the top down till the lowest (i.e. innermost in source code nesting) constructs[2]. After chunks of bytecode have been generated, they can be immediately loaded (using a subclass of `java.lang.ClassLoader`) for execution, producing results interactively. Alternatively, bytecode can also be written out to disks in the form of Java class files, (which conforms to the Java class file format as defined in [LY99]). The process is illustrated as follows:

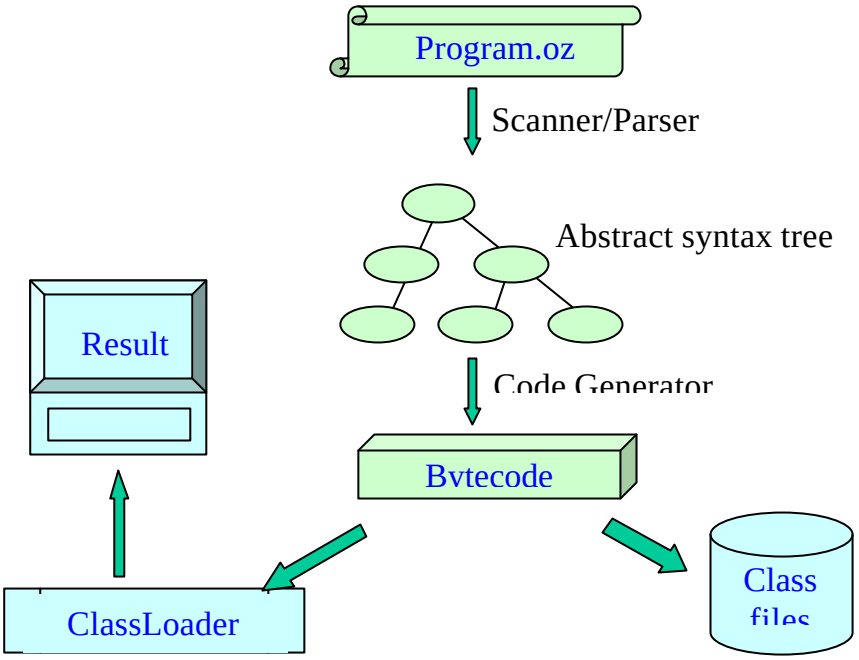


Figure 6-1: Berlioz architecture

[1] This is true: due to their definitions in the grammar, even patterns in Oz form some kind of hierarchy and can generally be considered part of the syntax tree, even though patterns do not directly appear on the tree like statements and expressions do. Patterns are components of, and therefore, hidden inside, statements and expressions that use them.

[2] In programming terms, this is made possible by mandating that each subclass of `OzConstruct` must implement a method of the signature (we will return to this method in section 6.6):

```
public void compile(JCompiler codeGenerator, CompileEnv env, ...);
```

6.2 Scanner and Parser

As scanner generator, we use JFlex version 1.2.2 [Kle99]. The parser was generated using Java CUP version 0.10j [Hud97]. The context-free grammar was largely based on the official Oz grammar as appear in [HK00]. Some adaptations to the original grammar were made to reduce the size of parser, details are presented in the next section. The complete adapted grammar is presented in appendix A. The changes, however, necessitate other modifications to eliminate shift-reduce conflicts, and eventually some grammar definitions become broader than they actually are. This is not a problem because we can always catch those special cases and report an error message. In fact, we can even afford a more descriptive error message, since the error is precisely known, and context information is readily available.

6.3 Transformation to Base Language

Typically the generated parser can be very large even with a relatively modest grammar. To reduce the size of original grammar, we have combined a number of similar constructs between expression and statements, for example the `local...end, thread...end, if...end` constructs. The key to make this combination possible is due to the nature of Oz language constructs. Before discussing that, let us define what expressions and statements are. The Oz Notation [HK00] distinguishes between expressions, language constructs that can be evaluated to a value, and statements, constructs that do not evaluate to any value. An expression position is therefore defined as a position where a value is required to continue execution, while a statement position is one where there must be no value left after execution. For example, consider this piece of code:

```
local X Y in
      65    %% This is an error: expression at statement position
      X = 4
      {P Y} %% Assumes P has been defined elsewhere
end
```

Here both `X = 4` and `{P Y}` are at statement position, and their execution does not leave any value behind. The expressions `X, Y, P` and `Y` on the other hand are at expression position, which means they must be compiled such that there is some value returned at runtime. In case the requirements associated with a position cannot be fulfilled when compiling a construct, a compilation exception is raised, as is the case with the expression `65` above. It turns out, however, that in some cases we can violate this rule, specifically, we allow statements to be at expression position. Below we shall discuss how this is actually handled.

Though expressions and statements in the Oz grammar appear distinct, a number of expressions are always transformed into statements internally. In fact, it can be seen that all of the Oz statements can appear at expression positions, and when they do, they must be *unnested* to become proper statements before being compiled. This process, which is termed *unnesting*, is also the process that transforms the functional Oz syntax into the internal relational form. Granted, we could have made a clear distinction between expressions and statements as defined in the Oz notation [HK00]. However, that would have resulted in many duplicated definitions that are eventually parsed to the same construct (note that unnesting is a necessity, not an option). Instead, we exploit the dual nature of statements in Oz (represented at compile time by the Java class `OzStatement`) by combining the similar definitions of statements/expressions into one. Depending on the position of the construct, appropriate actions are taken. If it is at a statement position, no action is necessary. If it is at an expression position, the `OzStatement` must be prepared to take in one extra variable called the *return value*. We enforce this by specifying an abstract method in `OzStatement` to be implemented by all subclasses, as follows:

```
public class OzStatement {
      public abstract void setReturnVar(Langel variable);
}
```

We call this step *channeling*, since the variable seems to be channeled inside, or absorbed by, the `OzStatement` and "disappear". (Note that we make a distinction between *channeling* and *unnesting*. Unnesting is the entire process of transforming the Oz functional syntax, while channeling is but one step performed in that process). This step is done repeatedly inwards until the innermost expression, which must be

either a proper expression (one that is not a dual nature statement, and is represented by the Java class `OzExpression`) or an assignment. In case that is a proper expression, a new assignment is created. Otherwise, if that was already an assignment, a new assignment is created with one side being the "channeled" variable. There are two general cases when unnesting takes place:

- Case 1: A variable is assigned to an `OzStatement`. In this case the variable is simply channeled inside the `OzStatement`.
- Case 2: An `OzStatement` is at an expression position, which does not fall into the case 1. For example, when an `OzStatement` is a procedure parameter, or is raised as an exception. In this case a new artificial variable is created and channeled inside the `OzStatement`. This variable then replaces the `OzStatement` at the original expression position, which means it may again be channeled inside another `OzStatement` according to case 1.

For example:

```
local X in
      proc {$ A}
            A = X
      end = thread local M in M end end
end
```

Becomes (according to case 2):

```
local X ArtificialVar1 in
      proc {ArtificialVar1 A} A = X end
      ArtificialVar1 = thread local M in M end end
end
```

And finally (according to case 1):

```
local X ArtificialVar1 in
      proc {ArtificialVar1 A} A = X end
      thread local M in ArtificialVar1 = M end end
end
```

If even after all code has been parsed and transformed and yet some `OzExpression` is still at statement position, we have an error ("expression at statement position"), and the compiler simply raises an exception. The opposite case – `OzStatement` at expression position – seldom happens, because if they exist, we will have already transformed all of them. The only case where we have a "statement at expression position" is when a variable is channeled towards a `skip` statement. For example with the piece of code below:

```
local X in
   X = local Y in Y = 1 skip end %%Error! Statement at expression position
end
```

6.4 The Intermediate Language

With the exception of the two language extensions (constraints and functors), there are altogether 17 constructs of the intermediate language, divided into two groups `OzExpression` and `OzStatement`. We have briefly mentioned in the previous section that they roughly correspond to expressions and statements, respectively. From the definitions of statements and expressions in the previous section, it follows that the difference in compilation between `OzStatement` and `OzExpression` is that bytecode generated from an `OzStatement` does not change the operand stack when executed. Meanwhile bytecode generated by an `OzExpression` leaves a value on the operand stack. The actual type of this value may vary; details of which will be discussed in section 6.6 below.

The 17 constructs are briefly summarized in the following table. The fourth column (construct attributes) indicates the name and type of fields inside each Java class representing the constructs. The attributes are very similar to the actual rules in the core Oz grammar. Note that these constructs are obtained after source code in regular Oz syntax has already been unnested.

Construct	**Type**	**Oz syntax**	**Construct attributes**
BlockExp	OzStatement	`local...end`	body: OzStatement
ThreadExp	OzStatement	`thread...end`	body: OzStatement

IfExp	OzStatement	`if...end`
		`if...else`
		`...end`
			expr: OzExpression,
			if_branch: OzStatement,
			else_branch: OzStatement

CaseExp	OzStatement	`case...end`	expr: OzExpression, branches: OzStatement[] (+ OzPattern[])
AssignExp	OzStatement	assignment (unification)	left: OzExpression, right: OzExpression
ClassExp	OzStatement	class creation	class: OzExpression, methods: OzStatement (+ OzPattern[])
ClassBinaryExp	OzStatement	class attribute assign & exchange (`<-`), class invocation (`,`)	left: OzExpression, right: OzExpression
ProcExp	OzStatement	`proc...end` `fun...end`	proc: OzExpression, body: OzStatement
ApplyExp	OzStatement	procedure application	proc: OzExpression, params: OzExpression[]
TryExp	OzStatement	`try...end`	try_content: OzStatement catch_clauses: OzStatement[] (+ OzPattern[]) finally_clause: OzStatement
RaiseExp	OzStatement	`raise...end`	exception: OzExpression
LockExp	OzStatement	`lock...end`	lock: OzExpression, body: OzStatement
VoidExp	OzStatement	`skip`	none
SingletonExp	OzExpression	literal, variable, _	expr: OzExpression
UnaryExp	OzExpression	`@, ~, !!`	expr: OzExpression
BinaryExp	OzExpression	`+, -, *, /, ^, div, …`	left: OzExpression, right: OzExpression
RecordExp	OzExpression	record construction	label: OzExpression, features: OzExpression[], fields: OzExpression[]

Table 6-1: Classification of constructs of the intermediate language

An object diagram of these constructs is presented in the next page.

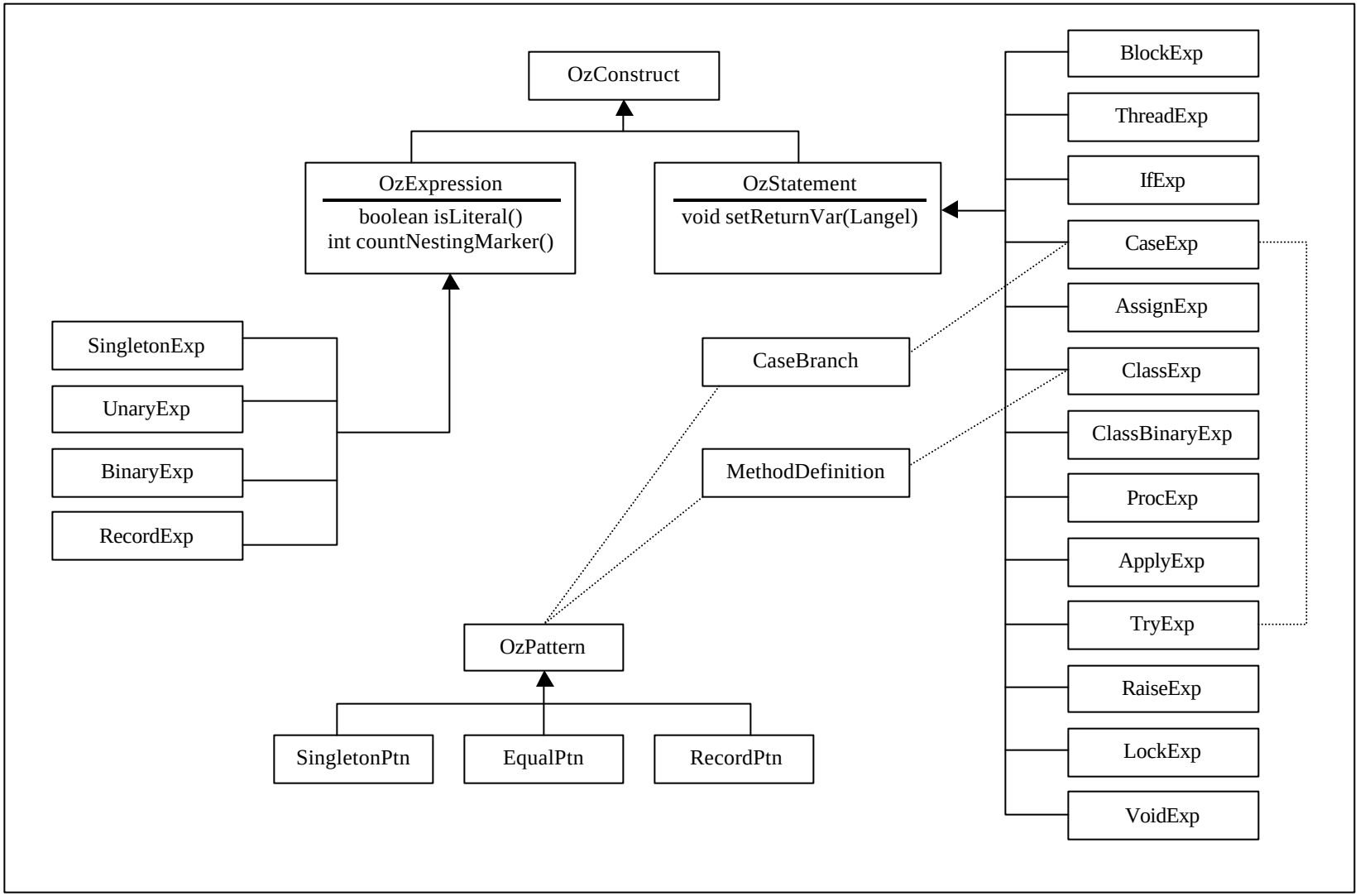


Figure 6-2: Object Diagram for Oz Constructs

6.5 Bytecode Generation

As mentioned in section 1.3, generated bytecode must conform to the Java class file format in order to be accepted by the verifier and executed [LY99]. The many restrictions required by the verifier make bytecode generation particularly complex and error-prone. Though there are a few bytecode-generating libraries available, not all of them offer the same degree of convenience and ease of use. In the end, we decided to use the Byte Code Engineering Library [Dahm99] for its clarity in design and ease of use, especially when it comes to generating branching instructions. In addition this library also offers facilities for certain bytecode-level optimizations, e.g. removing `NOP` instructions, or generating the shorter instructions when possible.

The library, albeit powerful and expressive enough to generate all possible bytecode, is still not ideal. The main reason is that a substantial amount of bytecode format knowledge is still required to generate correct bytecode. Also in many cases, there is a strong coupling between different steps of bytecode generation. This makes it difficult to generate bytecode solely with local knowledge[1] – something we will discuss in section 6.6. As a result, a *bytecode generator* (represented by the class `JCompiler.java`) is introduced, which provides a layer of insulation between the intermediate language (constructs in the syntax tree) and the underlying bytecode library. The use of such a middle layer has the following advantages:

- It hides away details of the Java class file from the constructs to be compiled. This saves us from having to constantly refer to the internal details of Java class and semantics of bytecode instructions. As we have mentioned above, keeping generated code to conform to the verifier criteria is not simple. For example, when generating `try...catch...finally` blocks, care must be taken to ensure that the `finally` clause is always executed. Similarly when compiling or `synchronized` blocks, we must ensure the synchronized object is always released (even when there are exceptions, etc.). Consequently our code would be practically littered with Java class-specific details that are both error-prone and hard to debug. The use of a bytecode generator allows, for example, to simply compile a `try...catch...finally` block as follows (see [Dahm99] and [LY99] for how this is compiled without our generator):

  ```
  compiler.startTryBlock(...);
  // generate content of try clause
  compiler.startCatchBlock("berlioz.runtime.RuntimeException");
  // generate content of catch clause for RuntimeException
  compiler.startFinallyBlock();
  // generate content of finally clause
  compiler.endTryBlock();
  ```

- It eases the burden of housekeeping chores, thereby helps make the project more manageable and code more readable. For example when introducing a local variable, usually we must keep track of the index of the variable on local variable stack, to make sure it should be invalidated when the local variable goes out of scope. With these details taken care of by the generator, we can focus on semantics details of Oz, rather than housekeeping tasks required by the JVM. For example, consider this piece of Oz code:

  ```
  local X in
     X = 4
     {Show X}
  end
  ```

  In Berlioz, we compile the above code roughly as:

  ```
  compiler.createLocalVariable("berlioz.runtime.TaggedNode", "X");
  compiler.push(4); // Pushes 4 on operand stack
  compiler.storeLocalVariable("X");   // Assign X to 4
  compiler.loadLocalVariable("Show"); // Push Show on operand stack
  compiler.loadLocalVariable("X");    // Push X on operand stack
  compiler.callMethod(...);  // Call Show.apply with parameter X
  ```

[1] We will discuss this in great length in the next section. Briefly, this means to allow each intermediate construct to generate bytecode independently of its siblings or its parent.

```
compiler.destroyLocalVariable("X"); // Invalidate X, reclaim its slot
```

In this case, the direct use of the symbolic identifier ("X") apparently makes our code more readable and maintainable than when an index value has been used instead.

- The use of a wrapper class around the bytecode library facilitates the potential use of other libraries in the future. Though certain library-dependent details are inevitable (the library-defined constants, the design of branching instructions), the efforts required to replace the underlying library would be much more extensive had the library been used directly.

To aid reusability, the generator class (`berlioz.lang.JCompiler`) has been designed such that it can be used alone, independent of the rest of Berlioz package (though it still depends on the bytecode library for the actual code generation).

6.6 Compilation Scheme

We have briefly mentioned in section 6.1 that the compilation of the parsed syntax tree (of Oz constructs) into actual Java bytecode is strictly by delegation. This applies to all constructs: expressions, statements and patterns (for pattern matching). By *delegation*, we mean that each construct, when compiling its own Oz language semantics, simply delegates the job of compilation to its children (in the syntax tree). To reduce coupling, there should be no special instruction apart from whether generated bytecode should leave any value on the operand stack after executing. This allows for a simple yet extendible compilation scheme. Dependency between different language constructs is kept to a minimum, making code more maintainable. The downside of this is that most of the compilation decisions are made locally, which can lead to substantial inefficiency. Also each analysis over a number of parsed constructs requires tree traversal, and therefore this kind of analysis should be kept to a minimum. Below we shall see how efficiency can be improved. The way this is made possible, however, is slightly different between statements/expressions and patterns. As such, we will be discussing the compilation delegation interface for the *proper constructs* (we use this term for statements and expressions) and patterns in turn.

Proper constructs can be loosely defined to be those that can appear directly in the syntax tree. Generally, there is no coupling between a parent construct and its children. The only aspect that needs to be defined in the interface between parent-children is whether compiled bytecode should leave any value on the operand stack after executing. In other words, whether a child construct should compile itself as an expression or as a statement. There is another question, which arises because of the indirect nature of the `TaggedNode`. If a value should be left, should it be a wrapped value, or just a plain (primitive) value? Take for example, the following piece of code:

```
if (X + Y) > 0 then
      {Show 'Positive'}
end
```

A strictly-by-delegation compilation scheme just compiles all language constructs hierarchically, as follows:

Compile if statement
    Compile binary expression (condition)
        Compile binary expression (sum)
        Compile singleton expression (integer 0)
        Insert Java comparator instruction (>)
    Insert Java branching instruction
    Compile application statement

Note that in order to achieve our objectives (minimal coupling, clarity, simplicity, etc.), we have specifically removed the possibility of obtaining local knowledge (which may have helped making better, more efficient compilation decisions). Instead, we would have to always compile expressions to the generic, lowest denomination, that is the `TaggedNode`. But such code will be quite inefficient, since a lot of wrapping/unwrapping must be done unnecessarily. This is illustrated by the following list of operations:

Unwrap X (two Java instructions: `getfield` and `invokevirtual`)
Unwrap Y (two Java instructions: `getfield` and `invokevirtual`)
Perform addition (one Java instruction: `iadd`)
Wrap X + Y (one Java instruction: `invokevirtual`)

Wrap 0 (one Java instruction: `invokevirtual`)
Unwrap X + Y (two Java instructions)
Unwrap 0 (two Java instructions)
Perform comparison (one Java instruction: `icmp`)
Wrap comparison result into 'true' name (one Java instruction)
Comparison result and 'true' name (one Java instruction: `acmp`)
Perform branching (one Java instruction: `ifeq`)
…

With the above piece of code, if somehow we know that the result of `(X+Y)` is going to be used directly as integer, some optimizations are possible. We can simply leave the result unwrapped, load the integer 0, perform an integer comparison, and then using the same integer value (left on operand stack from comparison) to perform branching. (This is possible since the JVM considers Boolean values to be equivalent to integers). That way the number of instructions can be cut to about half or less. How could we do that without making all sorts of inter-construct inquiries (and housekeeping work)? The answer lies in what we call *compilation contract*, which is essentially a two-way means of communication during compilation between parent and child constructs. First, a number of *return type flags* are defined, which are possible value types that can be left on the operand stack (for example, VOID, VAR, INT, FLOAT, ATOM, etc.). These flags are set to correspond to specific bit positions on a 32-bit integer, which means they can be OR-ed together to form composite flags. When a parent wants its children to compile themselves, it gives each of them a *return directive* (this can be different among the children), which is composed of at least one, but possibly two or more *return type flags* OR-ed together. If there are more than one flag in the return directive, the parent construct essentially says: "try to compile to bytecode that leaves one of these value types. The more specific, the better."[1] The child construct in turns will examine the flags, and if it sees that it can satisfy at least one return type, compiles towards that type and returns the flag associated with the targeted type (the *directive fulfillment* value). If there are more than one flag that can be satisfied, it will favor the more specific one. If none of the flags can be satisfied, the child construct raises an exception. The parent construct, upon receiving the return value from its child, examines it to know the targeted type, and generates subsequent bytecode accordingly. Note that the *directive fulfillment* value must not be a composite return type flag (two or more flags OR-ed together), that will confuse the parent construct.

As an example, consider the compilation method for the `if...end` construct, which is defined roughly as:

```
public int compile(JCompiler cpl, CompileEnv env, int returnDirective)
{
      if (returnDirective != VOID)
            throw new Exception("Statement at expression value");

      // Specify that we expect either
      // a TaggedNode or an integer on operand stack
      int ret = expr.compile(cpl, env, VAR | INT);

      // Examines the returned value to generate appropriate bytecode
      if (ret == VAR) {
            // Branch out if value on operand stack is Oz name 'true'
      } else if (ret == INT) {
            // Branch out if value on operand stack is 1 (Java "true")
      }
      // Compile the body of this if statement

      // Compile the target of the branching out statement here.
      // This means the body of this 'if' statement
      // is bypassed when the 'if' condition fails

      return VOID; // Signify that this construct fulfills its directive
}
```

[1] "More specific" here means to require less unwrapping operations. For example, INT is more specific than VAR, VOID is more specific than INT, etc.

First, the return directive provided for the `if...end` construct is examined. If it is not of type VOID, an exception is raised (since `if...end` is a statement). Then the expression used for branching decision is compiled, with the directive being VAR | INT (here VAR is the generic `TaggedNode`). Depending on the return value (VAR or INT), appropriate bytecode is generated. If the child expression cannot compile to either VAR or INT, it raises a compilation exception.

For patterns, things are slightly different. The tree of the pattern constructs also compiles by delegation, but there is no compilation contract involved. This is because the semantics of pattern matching is already well-defined (see section 4.5). When a pattern compiles its own matching instructions, it always asks its children to compile to bytecode that leaves an integer (which is actually a Java `boolean`) value on operand stack. This value is in fact the result of pattern matching of each child pattern, and is then used as the flag for the subsequent branching instruction. Let us examine an example:

```
case X of 3|a then
      %% Body of the main branch ...
else
      %% Body of the else branch ...
end
```

The main pattern `3|a` in this case has two sub-patterns `3` and `a`. The actual operations generated are:

```
If X is not _|_ then jump to elseBranch
      If not X.1 = 3 then jump to elseBranch
            if not X.2 = a then jump to elseBranch
                  Execute body of main branch
                  Goto endCase
            end
      end
end
elseBranch:
      Execute body of else branch
endCase:
```

For each *If* statement, an integer is left on the operand stack as input for the branching instruction that follows. All of the branching instructions corresponding to each *if* statement is only followed when the condition fails, and all point to the `else` branch of pattern matching code. Thus if none of the branching path was taken, the pattern does match, control goes into the body of the pattern branch, and after that jumps over the `else` branch toward the end (of `case` statement code). If any of the condition fails, a branching path is followed and execution control immediately jumps to the `else` branch. When there are two or more pattern branches, the compilation process is similar. All *fail* paths of the first pattern point to the matching of the second pattern, the second points to the third, and so on. The *fail* paths of the last pattern point to the `else` branch if any, otherwise they point to the raising of a compilation exception.

6.7 Optimization

Current optimization measures are limited to the following:

- Avoid boxed value whenever possible. An example is the statement `Y = X + 2`. Usually this is compiled into:
  ```
  local Temp in
     Temp = 2
     Y = X + Temp
  end
  ```
  However we can avoid the unnecessary creation of a temporary variable (and with it, a unification operation) by simply pushing the integer 2 onto operand stack before performing the addition.
- Delay the initialization of variable until the variable is used. This is to avoid a call to the static method `unify()` when the first unification is also the first time the variable is used. For example with the statement:
  `local X in X = 2 ... end`,

We only need to reserve a slot (on local variable area) for X at the time of declaration, but no initialization needs to be done. Then when executing the assignment `X = 2`, we only need to store the value of 2 (wrapped in an appropriate class) in the slot reserved for X earlier. As a byproduct of this optimization, we are able to detect when a local variable is declared but not used

- Precompute some binary expressions whose operands are literal. If both operands are known at compile time, the compiler will replace the binary expression with the literal that is the result of computation (constant folding). For example `(X = 7 div 5)` will be replaced with `(X = 1)`.
- Make extensive use of local variable area. Our implementation tries to optimize the use of local variable slots. The use of the bytecode generator enables us to exploit the local variable slots efficiently. We can limit the scope of each temporary variable to the exact instructions where they are used. In this way, a large number of temporary variables can be used without increasing maximum stack size.

Together with the first and third optimizations above (avoid boxed values, delay variable initialization), we have been able to carry out some static flow analysis that allow us to detect at compile time certain type anomalies. For example, the expression `((X + 2) / Y)` is incorrect in Oz, because the operator '/' requires both operands to be float. Note that this form of analysis is still very basic. With some improvements coupled with a type-inference scheme, we expect to be able to arrive with better type checking at runtime, as well as more efficient boxing of literal values.

6.8 Programming Environment

Towards our aim of a development environment for Oz, we create a simple graphical programming environment. Its graphical interface allows simple editing of Oz source code, as well as the ability to parse Oz source into a statement-based core syntax, compiling into Java class files, or execute Oz code and produce result interactively. A snapshot of the Berlioz interface is presented below.

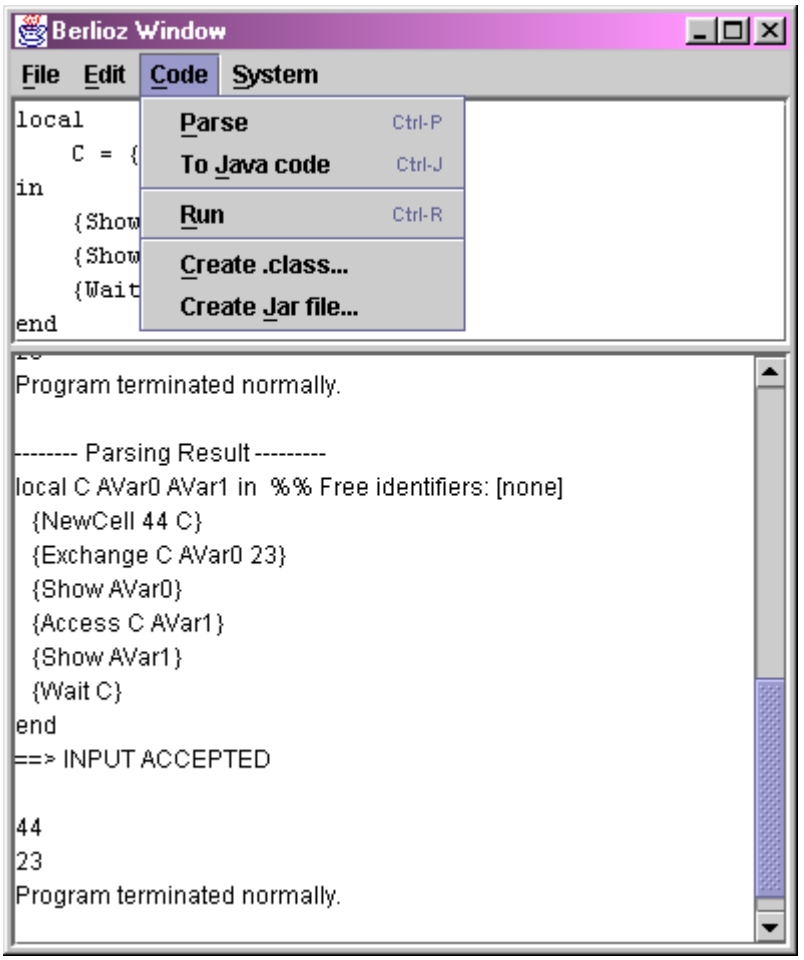


Figure 6-3: Berlioz Programming Environment

The top window is where Oz source code can be entered and edited. The bottom window is where results are displayed. This include output from parsing, compilation and runtime errors (if any), or the result of execution. Currently all global variables are initialized and loaded at start-up. This is convenient to programmers and simplifies development. However the disadvantage is that the loading can slow down start-up time slightly. In a proper functor implementation, these variables should only be initialized when needed (load-on-demand).

6.9 Summary

The functional Oz syntax is first transformed to a core relational language in a process called unnesting, before being compiled. Compilation is done strict strictly by hierarchical delegation, in which the constructs are compiled largely independently of each other. A compilation contract is the only interface between any two constructs, which specifies what value type, if any, should be left on the operand stack when generated bytecode is executed. Some simple static analysis is carried out at compile time as optimization. Finally, a programming environment is created to facilitate development and testing.

Chapter 7 Benchmarks and Conclusion

7.1 Examples and Benchmarks

These examples were tested on Berlioz:

- Fibonnaci numbers (recursive version).
- Takeuchi function.
- N-Queens problem

Overall the benchmark results have not been very encouraging. Compared to pure Java versions (programs written entirely in Java), Berlioz benchmark is usually between 40-120 times slower. Integer calculations suffer the heaviest setback: they are often about 100 times slower than their Java counterpart. This can be attributed to the following factors:

- Double boxing and unboxing: this affects mostly the integer calculation test (Fibonnaci & Takeuchi). As mentioned in section 2.3, each operations on integers and floats involve two boxing (or unboxing) each. This is a heavy overhead that renders Berlioz performance on integer tests disappointing.
- `Throwable` penalty: As we also mention in section 2.3, the trade-off for a simple and more natural implementation in which `TaggedNode` subclasses `Throwable` directly results in a runtime penalty [SW00]. In fact, a modified version of the Fibonnaci function in which `TaggedNode` does not subclasses `Throwable` gives a performance improvement of more than twofold.
- Function use: currently there is no special optimization towards function calls (not to mention tail-call optimization). In particular with recursive function, there is no special scheme to make use of the procedure object (the *this* pointer available in Java) to avoid unboxing performed on the procedure variable. Take the example of `{Fib N – 1} + {Fib N – 2}`. In this case, because this piece of code is inside the procedure Fib itself, the two procedure invocations could have been done directly on the *this* pointer, for example as `this.apply(...) + this.apply(...)`. Instead the current implementation compiles to the generic scheme: first the variable Fib is located, then checked if it is an instance of the `Apply1` interface. After that the interface is invoked, together with the error handling that follows, which would be in fact unnecessary had static analysis been carried out.

7.2 Future Works and Perspectives

While our objective of compiling a subset of the language Oz into Java bytecode was accomplished, many potential optimizations and improvements were not yet explored. Below we present our perspectives on these possible future works:

- Error reporting for the scanner and parser. The current error detecting and reporting of the scanner and parser is fairly primitive. This makes it difficult for debugging of syntax errors, and does not fit well our objective of making Berlioz a development environment.
- Oz as bootstrapping language: An alternative direction for implementing the compiler is to write it in Oz, have it compile itself into Java bytecode and then bootstrap it to run under the JVM. Potentially, this will allow Oz programmers to customize the compiler to suit their specific needs. However, there seems to be little performance advantage, since the resulting bytecode is still generated from the same process and executed under the same JVM.
- Better static analysis. We mentioned in section 6.7 that Berlioz does have some form of basic static analysis. There is room for improvement, however. Most importantly, program flow analysis can be carried out to detect the type of certain variables, thus avoiding runtime type error, or help provide more information for debugging.
- Tail-call optimization: though not implemented in Berlioz, this has been successfully implemented in other similar projects that compile declarative languages into Java bytecode.
- Optimized method dispatching: the current scheme using pattern matching may be improved by adopting a better dispatching strategy that can map the unique method names to their corresponding method. (Current implementation maps method names to their classes, and then the actual method is selected using pattern matching). Oz methods can be compiled to directly equivalent Java methods with distinct (and possibly artificial, since method names are first-class) names. It may be necessary to make use of Java reflection mechanism, to perform dispatching from method names to the actual Java object

of type `java.lang.reflect.Method`. It remains to be seen, however, whether the cost of Java reflection is enough to justify the potential improvement.

- Overall performance: Since Berlioz performance is still its main weakness, it is imperative that performance improvement is carried out in future. Possible directions are to make use of the shared interface approach used in DML [SW00] for value-representing classes, separate `TaggedNode` from `Throwable`, optimize boxing/unboxing operations (to eliminate when possible).
- Functor and distributed programming: Currently the functor extension syntax and distributed programming still remain to be supported.
- Constraint programming: Eventually we do hope to incorporate a constraint-based problem-solving mechanism into Berlioz. One way to achieve this is to design and develop a native constraint extension in Java. Alternatively, some existing constraint-based library can be adapted, which may prove to be simpler and more convenient.
- Development environment: The current graphical interface is primitive and does not have editing capabilities required for serious development. Either more effort is needed to improve the current editor and development environment, or some third party software is used as front end for development, in which case Berlioz will act as a back-end compiler engine and runtime environment.

7.3 Conclusion

We have built a robust, though somewhat under-optimized, compiler and development environment for Berli-Oz, a sub-language of Oz using the Java virtual machine. There are considerable differences between the Oz and Java. While Oz is a declarative language, Java is an imperative language. While Oz is dynamically typed, Java is statically typed. While Oz is a combination of functional, logic and object-oriented languages, Java is primarily object-oriented. Bridging this gap requires certain customizations of features supported by the Java runtime environment. Nevertheless, we find the effort to patch up the differences reasonable. In the end, we manage to arrive with a bytecode compiler and runtime environment that covers a significant sub-language of Oz, including logic variables (Chapter 2), concurrency (Chapter 3), first-class procedures (section 4.3), pattern matching (section 4.5), and the object system (Chapter 5). Numerous design decisions were taken along the way, which are summarized at the end of each section. Generally, similar features between Oz and Java (concurrency, exception, objects and classes) are exploited, while Oz features that are missing in Java (logic variables, first-class procedures, pattern matching, method dispatching) are approximated, though not without a noticeable decrease in efficiency.

Appendix A: The Adapted Oz Grammar

The adapted context-free syntax used in Berlioz is presented below as illustration for the combined definitions discussed in section 6.3. Some definitions have been deliberately expanded to reduce the size of the grammar. They are presented as part of our implementation and not as a definitive reference on Oz grammar. A full reference to the Oz language syntax and token definitions used in lexical scanning is described in detail in the Oz Notation [HK00]. In addition, the following items are not covered here and should be referred to [HK00]:

- Context-free syntax corresponding to functor and constraint extensions.
- The character class definitions used by the lexical scanner.
- Definitions for regular expressions `(variable)`, `(label)`, `(atom)`, `(int)` and `(float)`.

Throughout this grammar definition, the following symbol interpretation is used:

(w)	Set *w* of regular expressions
[w]	Either an element of set *w* or none
term or 'term'	*term* is to be substituted literally
{w}	Set of words containing zero or more elements of w
{w}+	Set of words containing one or more elements of w
w_1 w_2	Set of words containing an element of w_1 followed by an element of w_2
w_1 | w_2	An element of w_1 or an element of w_2

The initial state when parsing an Oz program is `(statement_list)`, which is simply a list of proper Oz statements.

```
statement_list ::= (statement_list) (statement_list)
   | (statement)
   | (variable)

statement ::= local (in_statement) end
   | '(' (in_statement) ')'
   | proc '{' (expression) {(pattern)} '}' (in_statement) end
   | fun  '{' (expression) {(pattern)} '}' (in_statement) end
   | '{' (expression) { (expression) } '}'
   | if (expression) then (in_statement) [ (else_statement) ] end
   | case (expression) of (case_statement_clause)
      { '[]' (case_statement_clause) }
      [ (else_statement) ]
     end
   | lock (expression) then (in_statement) end
   | thread (in_statement) end
   | try (in_statement)
      [ catch (case_statement_clause) { '[]' (case_statement_clause) } ]
      [ finally (in_statement) ]
     end
   | raise (in_statement) end
   | (expression) '=' (expression)
   | skip
   | class (expression) {(class_descriptor)} {(method_definition)} end
   | (method_statement)

expression ::= (statement)
   | (expression) orelse (expression)
   | (expression) andthen (expression)
   | (monop) (expression)
   | (expression) (binop) (expression)
   | (feature) | (float) | '_' | '$'
```

```
   | self
   | (label) ‘(‘ { (subtree) } ‘)’
   | ‘[‘ { (expression) }+ ‘]’
   | (expression) ‘|’ (expression)
   | (expression) { ‘#’ (expression) }+

in_statement ::= [ (declaration) in ] [ (statement_list) ] (expression)

declaration ::= (statement_list)

else_statement ::=
     elseif (expression) then (in_statement) [ (else_statement) ]
   | elsecase (expression) of (case_statement_clause)
      { ‘[]’ (case_statement_clause) }
      [ (else_statement) ]
   | else (in_statement)

case_statement_clause ::=
     (pattern) [ andthen (in_statement) ] then (in_statement)

monop ::= ‘~’ | ‘@’

binop ::= ‘.’ | ‘^’ | ‘==’ | ‘\=’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’
   | ‘+’ | ‘-‘ | ‘*’ | ‘/’ | div | mod

subtree ::= [ (feature) ‘:’ ] (expression)

feature ::= (variable) | (atom) | (int) | unit | true | false

pattern ::= (label) ‘(‘ { (subpattern) } [‘...’] ‘)’
   | ‘[‘ { (pattern) }+ ‘]’
   | (pattern) ‘|’ (pattern)
   | (pattern) { ‘#’ (pattern) }+
   | (feature) | (float) | ‘_’ | ‘$’
   | (pattern) ‘=’ (pattern)
   | ‘(‘ (pattern) ‘)’

subpattern ::= [ (feature) ‘:’ ] (pattern)

class_descriptor ::= from { (expression) }+
   | prop { (expression) }+
   | attr { (feature) [‘:’ (expression) ] }+
   | feat { (feature) [‘:’ (expression) ] }+

method_statement ::= lock (in_statement) end
   | (expression) ‘<-‘ (expression)
   | (expression) ‘,‘ (expression)

method_definition ::=
     meth (method_head) [ ‘=’ (variable) ] (in_statement) end

method_head ::= (feature)
   | (label) ‘(‘ { (method_param) } [‘...’] ‘)’

method_param ::= [(feature) ‘:’] (method_variable) [‘<=’ (expression)]

method_variable ::= (variable) | ‘_’ | ‘$’
```

Appendix B: Compilation of Oz Constructs

In this section, we will briefly describe how the Oz constructs are actually compiled into Java bytecode. Only the more interesting constructs are mentioned here, the rest are trivial and generally do not need special treatment. Note that some of the steps here are only possible with the use of the bytecode generator (see section 6.5). Without the generator, the compilation of many of the following constructs would become much more complex and labor-intensive.

Java class	Oz construct	Compilation steps
`BlockExp`	Statement list	Reserve a slot on local variable stack for each variable (if any). Compile all `OzStatement`'s sequentially (directive = VOID).
`ThreadExp`	Thread creation and execution	Create a new class with `java.lang.Thread` as super class. Compile the thread body into `run()` method. Instantiate the newly created class. Invoke the method `Thread.start()` on the newly created object.
`IfExp`	if…end	Compile the branching `OzExpression` (directive = VAR | INT). Compile the branching instruction. Compile the main `OzStatement` body. Compile the `OzStatement` for else branch, as target for the earlier branching instruction.
`CaseExp`	Pattern matching	Compile the `OzExpression` for testing pattern matching. Compile the `OzPattern` for branches, each followed by the branch's `OzStatement` body.
`AssignExp`	Unification	Compile the `OzExpression` on the left side Compile the `OzExpression` on the right side. Invoke the static method `TaggedNode.unify()`.
`ClassExp`	Class creation	Create a new class inheriting from `berlioz.runtime.OzClass`. Compile the all the `MethodDefinition`'s into the `apply()` method. Instantiate the newly created class. Unify the newly created object and the class variable.
`ProcExp`	Procedure creation	Create a new class implementing `berlioz.runtime.Procedure` and the appropriate `apply()` method (see section 4.3). Compile the procedure body into `apply()` method. Instantiate the newly created class. Unify the newly created object and the procedure variable.
`ApplyExp`	Procedure application	Compile the procedure variable as expressions (directive = VAR | PROC) Narrow the return value down to the procedure object (if fulfillment = PROC). Compile all parameters as `OzExpression`'s. Invoke the appropriate interface method `apply()` on the procedure object (see section 4.3).
`TryExp`	try…catch …end	Compile the start of try block. Compile the `OzStatement` body of try block. Compile the `catch` clauses (if any) as pattern matching. Compile the `finally` clause (if any).
`RaiseExp`	Exception raising	Compile the `OzExpression` exception to be thrown (directive = VAR). Insert the bytecode instruction for throwing exception (`athrow`).
`LockExp`	lock…end	Compile the `OzExpression` to be used as lock. Check if it is of type LOCK.

		Starts a `synchronized` block with the lock on operand stack (Java instruction: `monitorenter`). Compile the `OzStatement` body. End the `synchronized` block (Java instruction: `monitorexit`).
`SingletonExp`	Variable, literal, '_'	Load (push) the variable or literal value onto operand stack. (For unnamed variable represented by '_', create a fresh `TaggedNode` and push it on operand stack).
`BinaryExp`	Binary expression	Compile the `OzExpression` on the left side Compile the `OzExpression` on the right side. Compile the appropriate inline bytecode instruction.
`RecordExp`	Record construction	Compile the label as `OzExpression`. Compile the features (if this record is not a tuple) as an array of `OzExpression`. Compile the fields as an array of `OzExpression`. Create the record from the label and arrays.

Bibliography

[Both98] Per Bothner, *Kawa: Compiling Scheme to Java*. List Users Conference ("Lisp in the Mainstream"), Berkeley, CA, 1998. Available at http://sources.redhat.com/kawa/

[Col82] Alain Colmerauer, Prolog and infinite trees. In K. Clark and S. Tarnlund, ed., *Logic Programming*, pp. 231-251, Academic Press, NY 1982.

[Dahm99] Markus Dahm, *Byte Code Engineering with the JavaClass API*, Technical Report B-17-98, ICS, Free University of Berlin, Berlin, July 1999. Available at http://bcel.sourceforge.net/

[GJS96] James Gosling, Bill Joy and Guy Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[Han97] Han B. Lee, and Benjamin G. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS97)*, pages 73-82, Monterey, CA, December 1997. Available at http://www.cs.colorado.edu/~hanlee/USITS97/USITS97.ps

[Henz97] Martin Henz, *Objects for Concurrent Constraint Programming*, The Kluwer International Series In Engineering And Computer Science, Vol. 426, Kluwer Academic Publishers, Boston, 1997.

[HK00] Martin Henz, Leif Kornstaedt, *The Oz Notation*, Feb 2000. Available at http://www.mozart-oz.org/.

[HS84] Seif Haridi, Dan Sahlin, Efficient Implementation of Unification of Cyclic Structures. In J. A. Campbell, ed., *Implementations of Prolog*, John Wiley, 1984.

[Hud97] Scott E. Hudson, *CUP LALR Parser Generator for Java v0.10j*, July 1997. Available at http://www.cs.princeton.edu/~appel/modern/java/CUP/

[Kal99] Kalberer, Kirmin, J-Eiffel - An Eiffel to Java compiler, 1999. Available at http://www.sourcepole.com/sources/programming/misc/

[Kle99] Klein G., *JFlex: The Fast Scanner Generator for Java*, August 1999. Available at http://jflex.de/

[LY99] Tim Lindholm and Frank Yellin, *The JVM Specification*, 2nd edition, Addison-Wesley, 1999.

[Mehl99] Michael Mehl, *The Oz Virtual Machine: Records, Transients and Deep Guards*, PhD dissertation, Technischen Fakultät, Universität des Saarlandes, Saarbrücken 1999.

[Mey97] Meyer, J., The Jasmin Java Assembler, March 1997. Available at http://www.cat.nyu.edu/meyer/

[MLj99] MLj: Compiler for standard ML, 1999. Available at http://www.dcs.ed.ac.uk/home/mlj/index.html

[Moz98] The Mozart Programming System, 1998. Available at http://www.mozart-oz.org/

[Sma] The SmalltalkJVM commercial system. Available at http://www.smalltalkjvm.com/

[Smo94] Gert Smolka, The Definition of Kernel Oz. In A. Podeski, ed., *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910, pp251-292, Springer-Verlag, Berlin, 1995. Available at http://www.ps.uni-sb.de/papers/abstracts/RR-94-23.html

[Smo95] Gert Smolka, The Oz Programming Model. In J. van Leeuwen, ed., *Computer Science Today*, Lecture Notes in Computer Science, Vol. 1000, pp324-343, Springer-Verlag, Berlin, 1995. Available at http://www.ps.uni-sb.de/papers/abstracts/volume1000.html

[SW00] Daniel Simon and Andreas Walter, *An Implementation of the Programming Language DML in Java*, Universität des Saarlandes, Saarbrücken 2000. Available at http://www.ps.uni-sb.de/papers/

[Ven99] Bill Venners, *Inside the Java Virtual Machine*, 2nd edition, McGraw-Hill, NY 1999.