Fachbereich 14 Informatik
Universität des Saarlandes

An Implementation of the Programming Language DML in Java: Compiler and Runtime Environment

Diplomarbeit

Angefertigt unter der Leitung von Prof. Dr. Gert Smolka

Daniel Simon
Andy Walter

23. 12. 1999

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit zusammen mit Andy Walter / Daniel Simon selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Die folgende Tabelle zeigt die Autoren der einzelnen Kapitel. Gemeinsame Kapitel sind identisch in beiden Arbeiten enthalten; die Kapitel, die nur von einem der Autoren geschrieben wurden erscheinen nur in der jeweiligen Arbeit.

Kapitel 1 (Introduction)	beide je 50%
Kapitel 2–4	Andy Walter
Kapitel 5–7	Daniel Simon
Kapitel 8–10	beide je 50%
Anhang A (Compilation Scheme)	Andy Walter
Anhang B (Code of the Benchmarks)	beide je 50%

Saarbrücken, den 23. 12. 1999

Daniel Simon / Andy Walter

Abstract

DML is an experimental language that has emerged from the developement of the Oz dialect Alice. DML is dynamically typed, functional, and concurrent. It supports transients and provides a distributed programming model.

Subject of this work is the implementation of a compiler backend that translates DML programs to Java Virtual Machine code. Code-optimizing techniques and possibilities for the treatment of tail calls are described.

To translate DML to the Java Virtual Machine, a runtime environment is needed. This work presents a simple and secure implementation of the basic DML runtime classes and elaborates on relevant improvements. Pickling, a mechanism to make higher order values persistent, is provided on top of the Java Object Serialization. Finally, a high-level distributed programming model for DML is implemented based on Java's Remote Method Invocation architecture.

Finally, the implemented compiler and the runtime environment of DML are compared to similar projects.

Acknowledgements

First of all, I want to thank Prof. Dr. Gert Smolka for the interesting subjects of the theses.

I am indebted to Leif Kornstädt who supported us with patience during almost one year of research and programming. He implemented the compiler frontend together with Andreas Rossberg. Both Andreas and Leif were always open to our questions and suggestions concerning the compiler.

Further, I would like thank the people at the Programming Systems Lab for their answers to our numerous questions and all the fun we had during that time.

Contents

Chapter 1

Introduction

The subject of this work is the implementation of DML, an experimental, functional, concurrent, distributed, dynamically typed language with support for transients and first class threads. In the following we present a compiler backend and a runtime environment for translating DML programs to Java Virtual Machine code.

The goal of our work is a simple and secure implementation of DML for the JVM. We want further to investigate the question of efficiency. We try to estimate the influence of dynamic typing on the speed of the implementation by comparing our results with related projects. We elaborate what support the JVM gives us for our implementation and what features we are missing.

The compiler is written in Standard ML; the implementation follows general well-known compiler construction techniques as described, e.g., in [ASU86, WM92]. The runtime environment consists of Java classes that provide the basic functionality for the execution of DML programs on the Java Virtual Machine; parts of the Standard ML basis library are implemented directly in Java to improve efficiency.

This chapter gives an overview about the various programming languages relevant for this work. We describe the key features of Standard ML, Oz, and Java. Further, an overview of the important features of DML is given.

1.1 Standard ML

Standard ML is a functional language commonly used in teaching and research. ML is type safe, i.e., a program accepted by the compiler cannot go wrong at runtime because of type errors. The compile-time type checks result in faster execution and can help in the development process to avoid common mistakes. The type inference system of ML makes programs easier to write because the compiler tries to derive the type of expressions from the context.

SML supports polymorphism for functions and data types. Data-type polymorphism allows to describe lists of ints, lists of strings, lists of lists of reals, etc. with a single type declaration. Function polymorphism avoids needless duplication of code by permitting a single function declaration to work on polymorphic types. SML functions are higher order values; functions are dynamically created closures that encapsulate the environment in which they are defined. Functions can be returned as results of functions, they can be stored in data structures and passed to functions as arguments. Function calls in SML are call-by-value, i.e., the arguments of a function are evaluated before the body of the function is evaluated.

In SML, most variables and data structures are immutable, i.e., once created they can never be changed or updated. This leads to guarantees on data structures when different parts of a program operate on common data. Such unchangable data fits well into a functional context, where one tends to create new structures instead of modifying old ones. The automatic garbage collection of SML supports the functional style of programs and makes code simpler, cleaner, and more reliable. However, SML also has updatable reference types to support imperative programming.

SML comes with an exception-handling mechanism that provides dynamic nesting of handlers and provides — similar to other languages like C++, Java, Ada, etc. — the possibility to separate error handling from the rest of the code. The Standard ML language supports modules (called *structures*) and interfaces (called *signatures*). The signatures of modules specify the components and types from the module that are visible from outside.

The language and its module system are defined formally in [MTHM97]. A consequence of having the language definition in a formal notation is that one can prove important properties of the language, such as deterministic evaluation or soundness of type checking. There are several efficient implementations of Standard ML available: Moscow ML, SML/NJ, and others. Moscow ML is a light weight implementation; SML/NJ has more developer tools such as a compilation manager and provides a concurrent extension CML.

1.2 Oz

Since 1991 the programming language Oz has been developed at the Programming Systems Lab under the direction of Gert Smolka. Oz combines concepts of logic, functional, and object oriented programming. It features concurrent programming and logical constraint-based inference. The first implementation of Oz was officially released in 1995 as DFKI Oz 1.0 [Oz95]. Two years later the release of Oz 2.0 [Oz97] was completed. In January 1999, the successor of Oz 2.0, Mozart, was announced to the public. The current development of Mozart is a collaboration with the Swedish Institute of Computer Science (SICS) and the Université catholique de Louvain in Belgium.

The Mozart system [Moz99] provides distributed computing over a transparent network. The computation is extended across multiple sites and automatically supported by efficient protocols. Mozart further provides automatic local and distributed garbage collection.

Many features of Oz are inherited by DML and thus are explained in detail in the corresponding section. Among the features not shared with DML are constraints, encapsulated search, and objects.

Similar to Java, Oz is compiled to byte code that can be run on several platforms. Unlike Java, Mozart provides true network transparency without the need of changing the distribution structure of applications. Further, Oz is a data-flow language, i.e., computations are driven by availability of data. Finally, Mozart provides low-cost threads. Thus, it is possible to create thousands of threads within a process.

1.3 DML

The Amadeus project now develops a dialect of Oz, Alice, with its implementation called Stockhausen. DML is an experimental language that has emerged from the development process of Alice. The roots of DML are described in [MSS98, Smo98a, Smo98b].

DML stands for 'Dynamic ML'; the syntax is derived from Standard ML. Like Oz, DML is dynamically typed. Further, DML supports transients and concurrency with first class threads.

The transient model of DML is a mixture of Mozart's transient model and the Alice model. In DML, there are three different kinds of *transients*: *logic variables*, *futures* and *by-need futures*. In our context, logic variables are single assignment variables and futures are read-only views of logic variables. A by-need future is a future that has a reference to a nullary function. The function's application is delayed until the value of the by-need future is requested and then the by-need future is replaced by the function's return value. All transients become transparent after they have been bound.

Transients can be obtained by the operations

```
lvar   : unit -> 'a
future : 'a -> 'a
byNeed : ( unit -> 'a ) -> 'a
```

The operation

```
bind : ('a * 'a) -> 'a
```

assigns a value to a logic variable. The operation

```
future : 'a -> 'a
```

returns a future if the argument is a logic variable or otherwise it returns the argument as is. Requesting transients is always implicit.

Threads can be created by

```
spawn : (unit -> 'a) -> unit
```

and can be synchronized by using transients. DML allows recursive values, e.g.,

```
val rec x = 1 :: x
    and y = (x,y,z)
    and z = {a=y, b=z}
    and foo = ref baz
    and baz = ref foo
    and vec = #[foo,baz,vec]
```

is valid in DML. Similar to Oz, exceptions and exception handling are more liberal than in SML:

```
17 + ((raise 5) handle _ => 2)
```

evaluates to `19`.

DML has a component system that is implemented by *pickling*. The component system is illustrated in Section 2.3.1; pickling is explained in detail in Chapter 6. Also a high level distributed programming model adopted from Mozart is implemented (cf. Chapter 7) that makes the network completely transparent.

Like Java and Oz, DML is platform-independent. A DML pickle can be used on any Java capable platform.

1.4 Java

Java was originally called 'Oak' and has been developed by James Gosling et al. of Sun Microsystems. Oak was designed for embedded consumer-electronic systems. After some years of experience with Oak, the language was retargeted for the Internet and renamed to 'Java'. The Java programming system was officially released in 1995. The design principles of Java are defined in the Sun white papers [GM96].

Java is a general-purpose, concurrent, class-based, object-oriented language. It is related to C and C++, but has a different organization. A number of aspects of C and C++ are omitted and some ideas from other languages are adopted. Java is meant to be a production language, so new and untested features are excluded from the design.

Java is strongly typed and it is specified what errors may occur at runtime and what errors must be detected at compile time. Java programs are compiled into a machine-independent byte-code representation (*write once, run everywhere*). However, the details of the machine representation are not available through the language. Java includes automatic storage management to avoid the unsafeties of explicit memory deallocation. Java has distributed programming facilities and supports networking with the special aspect of Internet programming. A security model for execution of untrusted code [Gon98] is supplied.

The Java programming system consists of the object oriented programming language, the class libraries of the *Java API*, a compiler for the language, and the *Java Virtual Machine*. The Java language is defined in [GJS96]; the Java Virtual Machine is specified in [LY97]. The programmer's interface is documented in [GYT96a, GYT96b]. Java also comes with a documentation tool (`javadoc`), and a generated documentation of the API classes is available in HTML. The Java platform provides a robust and secure basis for object oriented and multi-threaded distributed programming.

Since 1995, Java has spread widely and the language has changed its former target architecture from embedded systems to other subjects. People implement applications in Java that are not restricted to run on a limited hardware (e.g., hand-held devices), but run as user interfaces for business applications. Java is also used for scientific computing, cf. [Phi98, PJK98].

One of the less common kind of project in Java is to implement other programming languages for the Java Virtual Machine (see [Tol99]): There are a lot of implementations for various Lisp dialects available; BASIC variants have been ported; there are Java variants of logic programming languages; other object oriented languages (Ada, COBOL, SmallTalk) can be translated to Java or JVM byte code. There are some efforts to extend Java with generic classes, higher order functions and pattern matching and transparent distribution [OW97, PZ97].

1.5 Organisation of the Paper

This is how this document is organized:

- Chapter 1 (this chapter) gives an introduction into the programming languages of interest and a general overview of the work and its goals.
- Chapter 2 states a naïve compilation scheme for DML. The features of the Java Virtual Machine and the DML runtime environment are described. An overview of the intermediate representation of the Stockhausen compiler and backend independent transformations on this representation is also given.
- Chapter 3 describes platform-dependent optimizations of the compiler backend.
- Chapter 4 specifies implementation details of the compiler backend and transformations on the generated JVM instructions.
- Chapter 5 introduces the Java classes that make up the core of the runtime implementation. First, the basic idea is presented and then we show how this can be improved in terms of running time and memory usage.
- Chapter 6 explains the idea of *pickling*, i.e., making a persistent copy of stateless entities. The implementation in Java is presented and how the current DML system makes use of it.

- Chapter 7 shows how the DML language can easily be extended for distributed programming issues.
- Chapter 8 summarizes related projects and compares the achievements of others with the DML system.
- Chapter 9 is about benchmarking issues. The execution speed of DML is compared to others. We compare implementations of related languages.
- Chapter 10 draws a conclusion, gives a lookout into future dos and don'ts, advantages and disadvantages of Java/object orientation resp. DML/functional programming.

Chapter 2

Compilation Scheme

This chapter describes a simple, unoptimized compilation scheme for DML. The first sections outline the basic knowledge of the JVM, the intermediate representation, and the DML runtime environment that are needed to understand the compilation scheme.

The DML frontend performs some transformations on the intermediate representation that are useful for most compiler backends: Pattern matching is represented as test graphs in order to avoid redundant tests. Function abstractions with tuple or record patterns are annotated accordingly, so the compiler backend can easily generate different methods for such parts of functions. The intermediate representation is described in Sections 2.3.1–2.3.3, the transformations on this representation can be found in Sections 2.3.4–2.3.6.

The remaining sections describe the compilation scheme properly. Similar to [OW97], a new class is created for each abstraction and closures are represented by instances of their class. The free variables of a function are stored in fields of the corresponding class. Functions have a virtual `apply` method that is invoked on function applications. Applications of primitive operations are mostly inlined or invoked by calls to static methods. This is possible because primitive operations usually have no free variables.

Record arities are statically known. They are computed at compilation time and stored in static fields. When records are used in pattern matching, pointer comparison on the arity suffices to decide whether a pattern matches or not. Pattern matching is also the only place where the compiler backend has to explicitly check for the presence of transients. Exception handling is mapped to the exception handling of Java.

2.1 The Java Virtual Machine

To understand the code generation of a compiler it is necessary to know the target platform. Our target platform is the Java Virtual Machine, JVM, which is described in detail in [LY97]. We decided to compile to JVM rather than Java for the following reasons:

- This enables us to store the line numbers of the DML source code into the generated class files.
- Machine code is easier to generate than Java programs.
- Shared code can be compiled more easily. There is a `goto` instruction in JVM code, but not in Java.
- After bootstrapping, a DML interpreter could be easily implemented using an 'eval' function that dynamically creates classes.

- No files need to be written, so the compiler could be run as an applet after bootstrapping.
- DML programmers don't need the Java development tools. Java runtime environment is enough.

However, there are also some disadvantages of generating byte code directly:

- The javac compiler performs peephole-optimizations and liveness analysis which we have to do manually now.
- Many programmers know the Java language, but few know the JVM instructions. Java programs would probably be easier to read for most programmers who want to understand how their DML programs are compiled.

This section is intended to give an overview about the concepts and specific features of the JVM, assuming basic knowledge of Java.

2.1.1 The Machine

The JVM is a stack machine with registers. There are two important stacks: The operand stack on which all operations compute and the call stack which stores all registers of the current method when another method is invoked. Because the JVM was developed as target platform for Java, the machine is object-oriented and supports multi-threading. The JVM is able to load and execute class files, which represent compiled Java class definitions. When a class file is loaded, the byte code verifier checks whether the code satisfies some security properties to ensure a well-defined execution. For example, registers (in Java terms 'local variables' of a method) must be initialized before they can be used.

Java and the JVM are quite similar. Most Java constructs can be expressed in a few JVM instructions. Nevertheless, there are some marginal differences that might confuse Java programmers. In Java it is possible to omit the package name when accessing a class of the current package or of an imported one. JVM expects all package names to be explicitly stored in the class file. As package separator slash ('`/`') is used instead of the Java package separator dot ('`.`'). Constructors are no longer implicitly created. Java implicitly initializes unused (local) variables to a default value (0, 0.0 or **`null`**, depending on the type), JVM does not. Within non-static methods, register 0 always contains a '**`this`**' pointer to the current object. The n parameters of a method are passed in registers 1 to n (or 0 to $n-1$ for static methods). All other registers have no special meaning.

2.1.2 Class Files

Each compiled Java class or interface definition is stored in a class file that contains both definitions of methods and fields and the JVM code of the (non-abstract) methods. For every method, there is an exception handle table with the code ranges where exceptions should be handled, the class of exceptions to handle and the code position of the handler routine. A method may further have a line number table where source code line numbers are stored for debugging reasons. Each class file has a *constant pool* where JVM constants such as strings, integers and float values are stored as well as method names, field names and type descriptors.

With the description in [LY97], it is possible to directly generate class files. We decided to let *Jasmin*, a Java byte code assembler that is documented in [Mey97], do this. Jasmin performs no optimizations on the generated code, but it compiles (easy-to-read) ASCII code into Java class files and manages the exception table, line number table and the constant pool for us. There are other Java assemblers such as the KSM of the KOPI project that can be obtained from [G$^+$99]. KSM performs some dead code analysis and branch optimization which we also do. It would have been easier to use the KSM instead of Jasmin for generating class files, but the first alpha version was released in May 1999, so it wasn't available when we started our project.

2.2 Typography for the Compilation Scheme

In this chapter, we use the following typography and abbreviation conventions:

Description	**Example**	**Comment**
JVM instructions	`invokespecial`	
DML source code	`val x=5`	
Class names	**`Function`**	Class names are expanded to **`de/uni-sb/ps/dml/runtime/Function`** if no package name is explicitly stated. To distinguish between classes and interfaces in this chapter, interfaces always start with a capital **`I`** like **`IValue`**.
Method names	`apply` or `<init>`	
Field names	`vals`	
Abbreviations for methods or field names of the DML runtime environment	<MGetBuiltin>	<MGetBuiltin> refers to the method **`Builtin`**/`getBuiltin`.
Signatures of Java methods	`(int)`→`IValue`	The example refers to a method that takes a (JVM) integer as parameter and returns an **`IValue`**.
Labels	***retry***	We use labels to make it easier to read this compilation scheme. In class files, the labels are removed and branches to labels are replaced by relative jumps (e.g., `goto`) or absolute addresses (e.g., in the exception handle table).
Variables	*stamp*	
Functions of the compiler backend	*expCode*	The example translates expressions from the intermediate language to JVM code.
Constructors of the intermediate representation	*`ConExp`*.	
Definitions	*constructed value*	

2.3 Intermediate Language

This section describes the intermediate representation which we use.

2.3.1 Components and Pickles

The DML compiler backend as described in this chapter translates DML *components*, our units of separate compilation, into *pickle* files. Pickles are persistent higher-order values. Generally speaking, components are lists of statements, usually declarations. We distinguish between the component body and function definitions. The component body contains all statements of a program not contained in a function or functor definition. We don't specially treat functors here because functors are just functions that return a new structure. The frontend transforms structures into records and functors into functions. Each function definition creates a new class. Function closures are instances of this class. Those function classes have an `apply` method which is called whenever this function is applied. The free variables of a function are stored in corresponding fields of the object.

The pickle file of a DML component contains the definitions of all exported functions and the *evaluated* component body. This corresponds to *evaluated components* in Alice. The saving and loading of pickle files is described in Chapter 6. If a component has a top level `main` function, the generated pickle can be executed via the `dml` command.

2.3.2 Statements

Any DML program consists of a list of statements, most of which are declarations. The intermediate grammar declares the following statement constructors:

```
datatype stm =
    ValDec of stamp * exp
  | RecDec of (stamp * exp) list
  | EvalStm of exp
  | RaiseStm of stamp
  (* the following must always be last *)
  | HandleStm of stm list * stamp * stm list * stm list * shared
  | EndHandleStm of shared
  | TestStm of stamp * test * stm list * stm list
  | SharedStm of stm list * shared
  | ReturnStm of exp
  | IndirectStm of stm list option ref
  | ExportStm of exp
```

The intermediate representation has a node *`ValDec`* for each non-recursive declaration. It is possible that multiple *`ValDec`*s for the same stamp occur in the graph but for a given path, each referred stamp is declared exactly once. Mutually recursive declarations are pooled in a *`RecDec`* node. It is necessary to distinguish between recursive and non-recursive declarations as we will see in Section 2.7.2. A special case of *`ValDec`* is *`EvalStm`* when the result can be discarded.

`RaiseStm`, *`HandleStm`* and *`EndHandleStm`* nodes are used to represent exception raising and exception handling in DML. The declaration lists of the *`RaiseStm`* represent:

- The catch body within which exceptions should be handled — if an exception is raised, it is bound to the given stamp,
- the handler routine for exceptions, and
- the continuation that is executed in any case after both the catch body and handler routine have finished executing. This 'finish' is stated explicitly by an *`EndHandleStm`* with the same reference as the *`HandleStm`*.

Function returns are represented explicitly as a *`ReturnStm`*, which is useful for most backends that generate code for imperative platforms. The intermediate representation has an *`ExportStm`* node which lists the identifiers that are visible in the top level scope. Those are the values that should be stored in the pickle file. Whenever it is obvious that the same subgraph occurs at two different positions of the intermediate representation, a *`SharedStm`* is created instead where *`shared`* is a reference that the backend uses for storage of the label where the code is located. *`SharedStm`*s are helpful for creating compact code when it comes to the compilation of pattern matching (see Section 2.3.4) and loops.

2.3.3 Expressions

The intermediate representation defines the following constructors for expressions.

```
datatype exp =
    LitExp of lit
  | PrimExp of string
  | NewExp of string option * hasArgs
  | VarExp of stamp
  | ConExp of stamp * hasArgs
  | RefExp
  | TupExp of stamp list
  | RecExp of (lab * stamp) list
    (* sorted, all labels distinct, no tuple *)
  | SelExp of lab
  | VecExp of stamp list
  | FunExp of stamp * (stamp args * stm list) list
    (* all arities distinct; always contains a single OneArg *)
  | AppExp of stamp * stamp args
  | SelAppExp of lab * stamp
  | ConAppExp of stamp * stamp args
  | RefAppExp of stamp args
  | PrimAppExp of string * stamp list
  | AdjExp of stamp * stamp
```

Literals are represented by a *`LitExp`*. For each occurance of a primitive value, a *`PrimExp`* node is created. The creation of a new (constructor) name or constructor is denoted by a *`NewExp`* node. The boolean argument is used to distinguish between constructor names and constructors. Referring occurances of constructors and names are represented by *`ConExp`*. References have their own constructor *`RefExp`*. For referring occurances of variables, *`VarExp`* nodes are created.

Tuples are denoted by a *`TupExp`*, records by a *`RecExp`*. The record labels are sorted and distinct. Whenever a record turns out to be a tuple, e.g., for `{1=x, 2=y}`, a *`TupExp`* is created instead. Vectors have their own constructor *`VecExp`*. The select function for tuple and record entries is represented by a constructor *`SelExp`*.

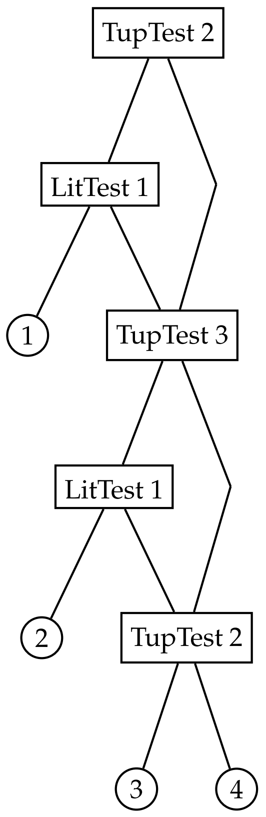
Functions are represented by a *`FunExp`* constructor. The *`stamp`* is used for identification of the function. Applications are denoted by an *`AppExp`* constructor. Section 2.3.5 describes the arguments and their treatment. Primitive functions and values that can be applied, have a special application constructor to make it easier for the backends to generate optimized code. These constructors are *`ConAppExp`* for constructed values, *`RefAppExp`* for creating reference cells, *`SelAppExp`* for accessing entries of records or tuples and *`PrimAppExp`* for applying builtin functions of the runtime.

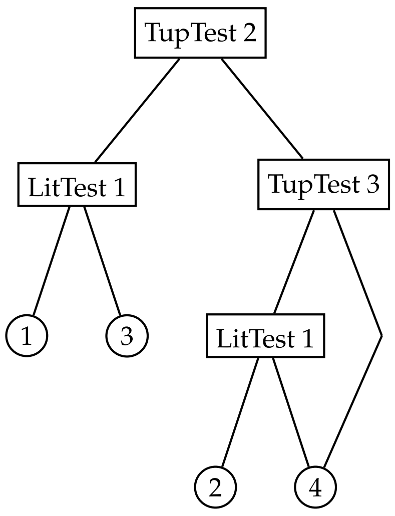
2.3.4 Pattern Matching

The compiler frontend supplies a pattern matching compiler that transforms pattern matching into a test graph. This is useful when testing many patterns because some tests may implicitly provide the information that a later test always fails or succeeds. These information yield in the test graph, as the following example demonstrates. Have a look at those patterns:

```
case x of
    (1,a)   => 1
  | (1,a,b) => 2
  | (a,b)   => 3
  | _       => 4
```

The illustration shows the naïve and the optimized test graph:





In case that the test expression is a 2-tuple, but its first value is not 1, pattern 2 never matches (because it is a 3-tuple), but pattern 3 always does. There are two nodes in the right graph where the code for expression 4 is expected. To avoid redundant code, *SharedStm*s are used there. The test nodes in the above graph each represent a single *TestStm*, so complex patterns don't increase the stack depth of the resulting code. Further, the frontend performs a linearization of the code. Instructions that follow a pattern are moved into the *TestStm* of each match as a *SharedStm*.

2.3.5 Function Arguments

DML functions take exactly one function argument. When more function arguments are needed, tuples or currying can be used. Because most destination platforms support multiple function arguments and because creating tuples and using higher order functions is comparatively expensive, the frontend splits different tuple and record patterns of a DML function into pairs of arguments and corresponding code.

```
datatype 'a args =
    OneArg of 'a
  | TupArgs of 'a list
  | RecArgs of (lab * 'a) list
    (* sorted, all labels distinct, no tuple *)
```

- *TupArgs* are used when a pattern performs a test on tuples, i.e., the function can take a tuple as argument. *TupArgs* provide a list of the stamps that are bound if the pattern matches.
- *RecArgs* are used when a function takes a record as argument. It has a list of labels and stamps. In case this pattern matches, the record's value at a label is bound to the corresponding stamp.

- `OneArg` is used when the pattern is neither a record nor a tuple. Functions always have exactly one `OneArg` section. The `OneArg` constructor has a stamp to which the value of the argument is bound.

Whenever a function is applied, the above argument constructors are also used. `TupArgs` and `RecArgs` are created whenever a function is applied directly to a tuple or record. The stamp of a `OneArg` constructor might also designate a tuple or record, e.g., if it is the result of another function application.

2.3.6 Constant Propagation

Future releases of the compiler frontend will propagate constants. Because this is not yet implemented, the compiler backend does a primitive form of constant propagation by now. A hash table maps stamps that are bound by `ValDec` and `RecDec` to expressions. Hereby, chains of declarations are resolved. This is needed for inlining the code of short and commonly used functions as described in Section 3.2.4.

2.4 A Short Description of the Runtime

This section gives a short description of the runtime from the compiler's point of view. For more detailed information about the runtime see Daniel Simon's Diplomarbeit.

2.4.1 Values

DML is dynamically typed, but the JVM is statically typed. Therefore we need a common interface for values, **`IValue`**. All other values are represented by implementing classes of this interface. For example, an integer value is represented by a wrapper **`Integer`** implementing **`IValue`**. As far as the compiler is concerned, the following values are of special interest.

Functions

In DML each function is represented by a corresponding class. All these classes inherit from a super class **`Function`** and have a method `apply` which implements the abstraction body. In DML as well as in SML an abstraction generates a closure. Variables that occur free in the function body can be replaced by their value at the time when the closure is built. Subclasses of **`Function`** have a field for each variable that occurs free in the abstraction body. Whenever a function definition is executed, i.e., its class is instantiated, the fields of the new instance are stored.

Variables bound within the function, such as parameters and local variables, can have different values each time the function is applied. We don't create fields for those variables but store them in the registers of the JVM.

Constructors

SML distinguishes unary and nullary constructors which have very little in common. To make clear which one we are talking about, we use the following terminology: *(Constructor) names* are equivalent to nullary constructors in SML. The unary SML constructors we call *constructors* and the value resulting from the application of a constructor and a value we call *constructed value*.

Records

The **`Record`** class contains a label array and a value array. Label arrays are sorted by the compiler and are represented by unique instances at runtime, so pattern matching on records can be realized as a pointer comparison of label arrays.

Exceptions

In DML it is possible to raise any value as an Exception. JVM only allows us to throw objects of type **`Throwable`**. The DML runtime environment provides a class **`ExceptionWrapper`** which contains an **`IValue`** and extends **`Throwable`**.

2.5 Helper Functions

For the compilation scheme in this chapter, there are some things we need really often. Therefore, we use the following helper functions.

2.5.1 Loading Values of Stamps onto the Stack

As described in section 2.4.1, free variables are stored in fields whilst variables that are bound in the current scope are stored in JVM registers. We often have to access a variable and don't want to distinguish where it has been bound. Sometimes we access builtins such as `nil` or `true` which are part of the runtime environment and are accessed via the `getstatic` command.

stampCode abstracts about the fetching of values:

stampCode (*stamp*) (* when stamp is that of a builtin value *) (* such as Match, false, true, nil, cons or Bind *)
`getstatic <BMatch>`

We use a table of builtins which maps builtin stamps to static fields of the runtime environment. Sometimes we access the current function itself, e.g., in

```
fun f (x::rest) = f rest
```

Since the `apply` method is an instance method of the function closure, we can get the current function closure via `aload_0` which returns the current object.

stampCode (*stamp*) (* when stamp is the current function *)
`aload_0`

Variables that are bound in the current scope are stored in registers and can be accessed via `aload`. The frontend makes sure that stamps are unique. We use the value of this stamp as the number of the register for now and do the actual register allocation later.

stampCode (*stamp*) (* when stamp has been defined within the current *) (* function closure or stamp is the parameter of the current function *)
`aload` *stamp*

All other variables are stored in fields of the current function closure and can be accessed via `getfield`:

stampCode (*stamp*) (* for all other cases *)
`aload_0`
`getfield` *curClass*`/field`*stamp* `IValue`

where *curClass* is the name of the class created for the current function closure.

Storing Stamps

In some (rare) cases, e.g., after a **`Transient`** has been requested, we store it (since we don't want to request it again). In general, we distinguish the same cases as for *stampCode* , but we don't need to care about builtins or the current function:

storeCode (*stamp*) (* when stamp has been defined within the current *) (* function closure or stamp is the argument of the current function *)
`astore` *stamp*

stampCode (*stamp*) (* for all other cases *)
`aload_0`
`swap`
`putfield` *curClass*`/field`*stamp* `IValue`

2.6 Compilation of Expressions

The evaluation of an expression leaves the resulting **`IValue`** on top of the stack. Note that the JVM provides some instructions such as `iconst_`i, with $-1 \leq i \leq 5$, `bipush` and `sipush` for integers and `fconst_`j for floats, with $0 \leq j \leq 2$, to access some constants faster than via `ldc`. To keep this description brief, we always use `ldc` in this document. However, in the implementation of the compiler backend, the specialized instructions are used when possible.

2.6.1 Constructors and Constructor Names

New constructors or names are generated by instantiating the corresponding class:

expCode (`NewExp` (*hasArgs*))
`new` *classname*
`dup`
`invokespecial` *classname*`/<init> ()`$\rightarrow$`void`

where *classname* is **`Constructor`** if *hasArgs* is true and **`Name`**, if not. We don't need to distinguish between occurances of constructor names and occurances of constructors when accessing them. The value is loaded from a JVM register or a field, depending on whether the constructor occurs free or bound.

expCode (`ConExp` *stamp*)
stampCode (*stamp*)

2.6.2 Primitive Operations

Primitive operations are managed by the runtime. The function <MGetBuiltin> returns a primive function.

expCode (`PrimExp` *name*)
`ldc '`*name*`'`
`invokestatic <MGetBuiltin> (java/lang/String)→IValue`

We chose to use a runtime function rather than a table in the compiler backend so that adding new functions doesn't entail changes in both runtime and backend. Usually, the function <MGet-Builtin> is called during the last compilation phase before the pickle is written, so we don't lose runtime performance.

2.6.3 Applications

Whenever a value is applied to another one, the first one should be a function or constructor. We needn't check this here, because every **`IValue`** has an apply method which raises a runtime error if necessary.

expCode (`AppExp`(*$stamp_0$ $stamp_1$*))
stampCode (*$stamp_0$*)
stampCode (*$stamp_1$*)
`invokeinterface` **`IValue`**`/apply (IValue)→IValue`

Applications of Primitive Operations

When we know that we are about to apply a primitive operation, we can do one of the following:

- inline the code,
- make a static call to the function, or
- get the primitive operation via <MGetBuiltin> as usual and make a virtual call.

Inlining usually results in faster, but larger code. Therefore, we inline some important and small primitive operations only.

Invoking static calls is possible, because primitive operations usually don't have free variables, so we don't need an instance in this case. Static calls are way faster than virtual ones, so this is the normal thing we do when primitive operations are called.

expCode (`PrimAppExp`(*name*, $stamp_0, \ldots, stamp_{n-1}$))
stampCode (*$stamp_0$*)
⋮
stampCode (*$stamp_{n-1}$*)
`invokestatic` *classname*`/sapply (IValue ...)→IValue`

The obvious disadvantage of using static calls is that adding primitive operations entails changes on both runtime and compiler, because the compiler needs to know the *classname* that corresponds to *name*. If the compiler doesn't know a primitive operation, we use <MGetBuiltin>.

expCode (`PrimAppExp`(*name*, *stamp*))

```
ldc 'name'
invokestatic <MGetBuiltin> (java/lang/String)→IValue
stampCode (stamp)
invokeinterface IValue/apply (IValue)→IValue
```

In case there is more than one argument, we store all arguments into a single **`Tuple`** and pass this tuple to the `apply` method.

2.6.4 Abstraction

When compiling an abstraction, we create a new subclass of **`Function`** with an `apply` method that contains the body of the abstraction. Then we instantiate this new class and copy all variables that occur free in the function body into this instance:

expCode (`FunExp`(*funstamp*,*body*))

```
new classname
dup
invokespecial classname/<init> ()→void
 (* populate the closure now:  *)
dup
stampCode (fv_0)
putfield classname/fv_0 IValue
⋮
dup
stampCode (fv_{n-1})
putfield classname/fv_{n-1} IValue

stampCode (fv_n)
putfield classname/fv_n IValue
```

$fv_0 \ldots fv_n$ are the variables that occur free within the function body. Furthermore, we need a bijective projection *className* which maps the function's *funstamp* to *classname*.

2.6.5 Literals

DML literals are int, real, word, char and string. The generated code for these looks quite similar: A new wrapper is constructed which contains the value of the literal.

expCode(`LitExp`*(lit))*

```
new cls
dup
ldc v
invokespecial cls/<init> (type)→void
```

where *v*, *cls* and *type* depend on *lit* as follows:

varlit	*cls*	*type*
`CharLit` *v*	**`Char`**	`c`
`IntLit` *v*	**`Integer`**	`i`
`RealLit` *v*	**`Real`**	`f`
`StringLit` *v*	**`String`**	**`java/lang/String`**
`WordLit` *v*	**`Word`**	`i`

2.6.6 Records

Record labels cannot change at runtime and therefore may be created statically. Record labels are always sorted. We load the label array from a static field, create an **IValue** array with the content of the record and invoke the Java constructor:

expCode(`RecExp`*[(*lab_0*,*$stamp_0$*),(*lab_1*,*$stamp_1$*),. . . ,(*lab_n*,*$stamp_n$*)])*

```
new Record
dup
getstatic curClassFile/arity IValue[]
ldc n + 1
anewarray IValue

dup
ldc 0
stampCode (stamp_0)
aastore
⋮
dup
ldc n
stampCode (stamp_n)
aastore

invokespecial Record/<init> (java/lang/Object[] * IValue[])→void
```

The static *arity* field must be created in the current class file *curClassFile* and is initialized when creating the top level environment.

2.6.7 Other Expressions

There are no new concepts introduced for the compilation of other expressions. However, the complete schemes can be found in the appendix (see Section A.1).

2.7 Compilation of Statements

After a statement has been executed, the stack is in the same state as before.

2.7.1 Non-Recursive Declarations

With non-recursive declarations, an expression is evaluated and stored into a JVM register:

decCode (`ValDec`(*stamp*, *exp*))

```
expCode (exp)
storeCode (stamp)
```

2.7.2 Recursive Declarations

With recursive declarations, function definitions like the following are possible:

```
fun odd 0 = false
  | odd n = even (n-1)
and even 0 = true
  | even n = odd (n-1)
```

Now `odd` occurs free in `even`, so the closure of `even` needs a pointer to `odd`. But `odd` also needs a pointer to `even` as `even` occurs free in `odd`. We solve this conflict by first creating empty closures, i.e., closures with `null` pointers instead of free variables. When all closures of the recursive declaration are constructed, we set the free variable fields.

decCode (`RecDec`[($stamp_0$,exp_0),($stamp_1$,exp_1),…,($stamp_n$,exp_n)])

`(* create empty closures:  *)`
emptyClosure (exp_0)
`astore` $stamp_0$
⋮
emptyClosure (exp_n)
`astore` $stamp_n$
`(* fill the closures now:  *)`
`aload` $stamp_0$
fillClosure (exp_0)
⋮
`aload` $stamp_n$
fillClosure (exp_n)

To create an empty closure, the corresponding class is instantiated. Variables, constructors, names and literals don't have a closure and can be directely evaluated. When all objects of a *`RecDec`* are created, the fields of functions, tuples, records, vectors, constructed values and references are stored via `putfield` instructions. Some expressions, such as *`AppExp`*, are not admissible in recursive declarations. However, we don't need to check admissibility here because illegal code shouldn't pass the frontend. The complete compilation scheme for these cases can be found in Section A.2.

2.7.3 Pattern Matching

As described in Chapter 2.3, pattern matching is transformed into *`TestStm`*s of the form *`TestStm`*(*teststamp*, *test*, *match*, *notmatch*) where *teststamp* is the identifier of the `case` statement, *test* the pattern to compare with, *match* the code that should be executed in case that *test* matches and *notmatch* the code to be executed if not.

When compiling a *`TestStm`*, we first check whether *teststamp* is an instance of the class corresponding to *test*. If so, we compare the content of *teststamp* to *test* and branch to *match* or *notmatch*. Otherwise, if *teststamp* is an instance of **`Transient`**, the test is rerun on the requested value of *teststamp*. The `request` of a **`Transient`** always returns a non-transient value, so this loop is executed at most twice. For performance reasons, it is possible to do the **`Transient`** check only once for chains of *`TestStm`*s with the same *teststamp*.

```
decCode (TestStm(teststamp, test, match, notmatch))
---------------------------------------------------
stampCode (teststamp)
retry:
testCode (test)
decListCode (match)
goto finished
wrongclass:
instanceof Transient
ifeq elsecase
stampCode teststamp
checkcast Transient
invokeinterface <MRequest> (Transient)→IValue
dup
storeCode (teststamp)
goto retry
popelsecase:
pop
elsecase:
decListCode (notmatch)
finished:
```

Depending on *test*, *testCode* may not only compare the content of *stampcode* to the *test* pattern but also bind one or more variables:

- **Literals**

  In our Java representation, `word`, `int` and `char` values are represented as wrapper classes that contain integer values. We check whether the test value is of the correct type and if so, we compare its content to the value of our pattern. The pattern matches if both are equal.

```
testCode (LitTest(IntLit v))
testCode (LitTest(WordLit v))
testCode (LitTest(CharLit v))
-----------------------------
dup
instanceof classname
ifeq wrongclass
checkcast classname
getfield classname/value type
ldc v
ificmpne elsecase
```

  classname is **`Word`**, **`Integer`** or **`Char`** for `word`, `int` or `char` patterns. `real` and `string` literals are compared exactly in the same way except that we use `fcmpl, ifne` ***elsecase*** resp. `invokevirtual <MEquals>, ifeq` ***elsecase*** to do the comparison.

- **(Constructor) Names**

  Because both the DML compiler and DML runtime guarantee that (constructor) name instances are unique, pointer comparison suffices here:

```
testCode (ConTest(stamp))
-------------------------
stampCode (stamp)
ifacmpne elsecase
```

- **Constructors**

  The comparison succeeds when the *teststamp* of the `TestStm`, which lies on top of the stack, is a constructed value and its constructor is equal to the *conststamp* of the `ConTest`. If this is the case, the content of the *teststamp* is stored into the *contentstamp* of the `ConTest`.

testCode (`ConTest`(*conststamp*, *contentstamp*))

```
dup
instanceof IConVal
ifeq wrongclass
checkcast IConVal
invokeinterface IConVal/getConstructor ()→Constructor
stampCode (conststamp)
ifacmpne elsecase
stampCode (teststamp)
checkcast IConVal
invokeinterface IConVal/getContent ()→IValue
astore contentstamp
```

- **References**

  Although, from the DML programmer's view, references are just special constructors with the extra feature of mutability, the generated code is somewhat different:

  The type of a 'constructed value' of a `ref` is no **`IConVal`**, but a special class **`Reference`**, so the constructor comparison is realized as an `instanceof`:

testCode (`RefTest`(*contentstamp*))

```
dup
instanceof Reference
ifeq wrongclass
checkcast Reference
invokevirtual Reference/getContent ()→IValue
astore contentstamp
```

- **Records**, **Tuples** and **Vectors**

  Records have a statically built arity that contains the sorted labels. Two records are equal if their record arities are equal in terms of pointer comparison. When the *teststamp* turns out to be a **`Record`**, we invoke a runtime function which compares the arities and returns an **`IValue`** array that contains the content of the record if the arities match or **`null`** if they don't.

  Pattern matching on Tuples and Vectors works exactly in the same way with the difference that the arity is an integer value that equals the size of the value array of the **`Tuple`** or **`Vector`**.

testCode (`RecTest`$[(name_0, stamp_0), \ldots, name_n, stamp_n])$
testCode (`TupTest`$[stamp_0, \ldots, stamp_n])$
testCode (`VecTest`$[stamp_0, \ldots, stamp_n])$

```
dup
instanceof classname
ifeq wrongclass
checkcast classname
cmpArity ()

 (* Now bind the values:  *)
dup
ldc 0
aaload
astore stamp_0
⋮
dup
ldc n−1
aaload
astore stamp_{n−1}

ldc n
aaload
astore stamp_n
```

classname is **`Record`**, **`Tuple`** or **`Vector`**. The arity comparison is compiled as follows:

cmpArity (* for Records *)

```
getstatic curClassFile/arity java.lang.String[]
invokevirtual Record/checkArity (java.lang.String[])→IValue[]
dup
ifnull popelsecase
```

The static *arity* field must be created in the current class file *curClassFile* and is initialized when creating the top level environment. See Section 2.3.1 for details.

cmpArity (* for Tuples and Vectors *)

```
getfield classname/vals ()→IValue[]
dup
arraylength
ldc n+1
ificmpne popelsecase
```

The field `vals` of a **`Tuple`** or **`Vector`** contains an array with its content.

- **Labels**

  To match a single label of a **`Vector`**, **`Tuple`** or **`Record`**, we invoke a runtime method of **`ITuple`**, `get`, which returns the value that belongs to a label or **`null`** if there is no such label in this **`ITuple`**. If `get` returns **`null`**, the pattern fails and we branch to the next one. Otherwise, the pattern matches and the value is bound.

testCode (`LabTest`(*lab*, *stamp*))

```
dup
instanceof ITuple
ifeq wrongclasscase
checkcast ITuple
ldc lab
invokeinterface ITuple/get (java.lang.String)→IValue
dup
ifnull popelselabel
astore stamp
```

2.7.4 Shared Code

*`SharedStm`*s contain statements that are referred to at different places of the intermediate representation. The code for each set of *`SharedStm`*s is only created once and branched to from the other positions. The branch can be done by the JVM instruction `goto` because

- equal *`SharedStm`*s never occur within different functions, so their code is always generated in the same method of the same class.
- JVM stack size is equal for all of the *`SharedStm`*s. (Indeed, the stack is always 0 before and after *any* statement)
- *`SharedStm`*s are always constructed to be the last statement in a list, so we don't have to return to the place where we branched.

decCode (`SharedStm`(*body*, *shared*)) (* if shared = 0 *)

```
shared := new label
shared:
decListCode (body)
```

decCode (`SharedStm`(*body*, *shared*)) (* if shared ≠ 0 *)

```
goto shared
```

2.7.5 Exceptions

To raise an exception, a new **`ExceptionWrapper`** which contains the value is built and raised.

decCode (`RaiseStm`(*stamp*))

```
new ExceptionWrapper
dup
stampCode (stamp)
invokespecial ExceptionWrapper/<init> (IValue)→void
athrow
```

When compiling a *`HandleStm`*, we create a new label in front of the *contbody* and set *shared* to this value. For each *`EndHandleStm`*, this label is branched to.

```
decCode (HandleStm(trybody, stamp, catchbody, contbody, shared))
-----------------------------------------------------------------
shared := cont
try:
decListCode (trybody)
to:
invokevirtual ExceptionWrapper/getValue ()→IValue
astore stamp
decListCode (catchbody)
cont:
decListCode (contbody)
```

```
decCode (EndHandleStm(shared))
------------------------------
goto shared
```

To indicate to the JVM to call the *catchbody* when an **`ExceptionWrapper`** is raised, we create an entry in the exception index table of the class file. Within this exception index table, order is important. Whenever an exception occurs, the JVM uses the first entry that handles the correct exception (which is, for DML, always **`ExceptionWrapper`**). Therefore, with nested exception handles it is important that the inner one comes first in the exception handle table. We create an entry like

```
catch (ExceptionWrapper, try, to, to)
```

after generating the instructions above which maintains the correct order. The catch entry means that any occurance of an **`ExceptionWrapper`** between ***try*** and ***to*** is handled by the code at ***to***.

2.7.6 Evaluation Statement

Evaluation statements are expressions whose value doesn't matter but must be evaluated because they may have side effects. Think of the first expression of a sequentialization, for example. We generate the code for the included expression, then `pop` the result:

```
decCode (EvalStm(exp))
----------------------
expCode (exp)
pop
```

2.7.7 Returning from Functions

DML is a functional programming language, so functions always have a return value of the type **`IValue`**. We therefore always return from a function via `areturn`.

```
decCode (ReturnStm(exp))
------------------------
expCode (exp)
areturn
```

2.7.8 Exports

Values are pickled by the runtime method <MPickle>.

decCode (`ExportStm`(*exp*))
expCode (*exp*)
`ldc` *picklename*
`invokestatic <MPickle>`
`pop`

where *picklename* is the name of the file that should be created. We use the basename of the source file with the extension '.pickle'.

2.8 Summary

This chapter described the unoptimized compilation of DML to JVM code. A relatively simple scheme emerged and helped to identify the concepts for which the translation is less straightforward since they cannot directly be mapped to JVM constructs: Special care has to be taken for first-class functions, records, pattern matching, and recursive value bindings.

Chapter 3

Optimizations

Starting from inefficiencies uncovered in the naïve compilation scheme from the preceeding chapter, we devise optimizations to improve performance of the generated code: The creation of literal objects is performed at compilation time. At runtime, literals are loaded from static fields. As far as possible, we avoid creating wrapper objects at all and use unboxed representation whenever possible.

Functions with tuple pattern get special `apply` methods, so creating and matching tuples can be avoided in many cases. We make two approaches for the treatment of tail calls: A CPS-like version which is a variant of the ideas in [App92] decreases performance but operates on a constant stack height for all kinds of tail calls and another solution that merges tail recursive functions into a single method, which makes tail recursion even faster but doesn't cope with higher order functions.

Finally, sequences of pattern matching on integer values are sped up by using dedicated switch instructions of the JVM.

3.1 The Constant Pool

Due to the fact that DML is dynamically typed, the Java representation of DML literals is rather expensive and slow because a new object is created for each literal. We speed up the access to constants by creating the objects at compilation time and storing them into static fields of the class in which the constant is used. Now the sequence

```
new Integer
dup
aconst_1
invokespecial Integer/<init> (int)→void
```

which creates a new integer constant 1 can be replaced by a simple

```
getstatic currentclass/litn Integer
```

where *currentclass* is the name of the current class and n a number which identifies the literal in this class. This optimization results in a performance gain of about 30 percent on arithmetic benchmarks, such as computing Fibonacci numbers (on Linux with Blackdown JDK. This result should be similar on other systems).

3.2 Functions and Methods

As DML is a functional programming language, a typical DML program has many function applications. Applications are compiled into virtual method invocations which are, because of late binding of JVM methods, rather expensive. Fortunately, function applications have high potential for code optimizations.

3.2.1 n-ary Functions

DML functions have exactly one argument. When more arguments are needed, functions can be curried and/or take a tuple as argument. Both techniques are quite common in functional programming languages, but not in imperative or object-oriented languages and therefore rather expensive on the JVM. On the other hand, JVM functions can have multiple arguments.

Tuple Arguments

All functions are instances of subclasses of **`Function`** which implements **`IValue`**. When a function is applied, generally **`IValue`**`/apply (IValue)`$\rightarrow$`IValue` is invoked and the JVM chooses at runtime the `apply` of the corresponding function class. It often occurs that tuples are created just to apply a function, where the tuple is split by pattern matching immediately afterwards. As with any object, creating **`Tuple`**s is both time and memory consuming.

The DML compiler and runtime support special apply methods for the common cases of functions with an arity of less than 5. Those `apply0`, `apply2`, `apply3` and `apply4` functions take 2, 3 and 4 arguments, so we save the time of creating tuples and pattern matching. Because every function has those `apply`n methods, this optimization can be used even with higher order function applications, when the function is not known at compilation time (e.g., if the function is loaded from a pickle file or is distributed over the network). The default implementation of `apply`n creates an n-ary tuple and invokes `apply`. This is necessary in case of polymorphic functions which don't expect a tuple as argument, but can cope with one (e.g., `fn x => x`). For non-polymorphic functions that don't expect tuples, `apply`n can raise <BMatch>.

Currying

For the compiler backend, there is little that can be done to optimize curried function calls. However, if the function that was created by currying is applied immediately, the frontend can convert such curried applies into Cartesian ones, i.e., use tuples (which can be treated as shown above). For instance,

```
fun add x = fn y => x+y
```

can be replaced by

```
fun add (x, y) = x+y
```

if the result of `add` is always applied immediately and `add` is not exported into a pickle file. Of course, all corresponding applications have also to be changed from

```
add a b
```

to

```
add (a,b)
```

3.2.2 Tail Recursion

Tail recursion is one of the fundamental concepts of functional programming languages. The only functional way to implement a (possibly infinite) loop is to use one or more functions that recursively call itself/each other. Whenever a function application is in *tail position*, i.e., it is the last instruction within a function, the result of this function is the result of the application and therefore a return to the current function is not necessary. Like most functional languages, DML supports tail recursion. This is necessary because otherwise, loops would have a linear need of stack (instead of a constant one) and thus many programs would crash with a stack overflow exception after a certain number of recursions. The simplest form, self tail calls, can be easily compiled into a `goto` statement to the beginning of the method. But `goto` cannot leave the scope of one method, so we have to find another solution for mutually recursive functions and for tail calls to higher order functions.

[App92] describes a way to compile applications of functional programming languages into machine code without using a call stack at all. The trick is to transform the programs into continuation passing style (CPS) which means that all functions f take a function c as an extra parameter where the program continues after f has been executed. In case that f again contains an application a, a new function is created that contains a sequence s of the part of f after the application and a call of c. Now we can jump to c with continuation s without having to return afterwards. Of course, it is quite inefficient to create so many functions. Appel describes lots of optimizations that make CPS rather fast as the (native) SML/NJ compiler shows. [TAL90] describes a compiler for ML programs into C code that also uses CPS.

The restrictions of the JVM don't allow us the use of CPS in its original form because programs can neither directly modify the call stack nor call another method without creating a new stack-frame on top of its own one. On the other hand, we don't need the full functionality of CPS. We can use JVM's call stack for applications which are not in tail position and do something CPS-like for tail calls: For each tail call, the **`Function`** that should be applied is stored into a (static) class field `continuation`, e.g., of the runtime environment and the value on which that function should be applied is returned. All other applications are compiled into the usual `invokevirtual`, but afterwards set `continuation` to **`null`** and apply the old value of `continuation` on the return value until `continuation` is **`null`** after returning from the function. Now all tail calls have a constant call stack need. This approach slows down all applications by a `getstatic` instruction and a conditional branch and tail calls additionally by a `putstatic` to store the continuation and some stack-modifying instructions like `dup` and `swap`, but compared to the `invokevirtual` instruction which we use for applications anyway, these instructions can be executed relatively fast. Let's have a look at the recursive definition of `odd` again:

```
fun odd 0 = false
  | odd n = even (n-1)
and even 0 = true
  | even n = odd (n-1)
```

With these changes, the tail call to `odd` is compiled like follows.

```
stampCode (odd)
putstatic wherever/continuation
stampCode (<BMinus>)
stampCode (n)
ldc 1
invokeinterface IValue/apply (IValue)→IValue
areturn
```

All Non-tail calls now take care of the continuation field as an application of `odd n` demonstrates:

```
stampCode (odd)
stampCode (n)
loop:
invokeinterface IValue/apply (IValue)→IValue
getstatic wherever/continuation
dup
if_null cont
swap
aconst_null
putstatic wherever/continuation
goto loop
cont:
```

But alas! Because DML is multi-threaded, we have to take care that each thread has its own `continuation` field. This can be achieved by storing the `continuation` in a (non-static) field of the current thread. To read or write the `continuation` now we determine the current thread via static call of **`java/lang/Thread`**`/currentThread`. Depending on the JVM implementation, applications now take about twice as long, but we have a constant stack size when using tail calls. As this is not very satisfying, the DML compiler uses the following procedure instead. The code for functions that call each other recursively are stored in the same method `recapply`. This method takes 5 arguments: 4 **`IValue`**s, so it can ingest the special `apply`*n* methods as described in Section 3.2 and an **`int`** value that identifies the function. Now tail calls can be translated into `goto` statements which are fast and don't create a new stackframe. Nevertheless, we need a separate class for each function in case it is used higher order or it is exported into a pickle file. The apply functions for these classes invoke `recapply` with the corresponding function identifier.

This practice is a lot faster than the previous one and even faster than doing no special treatment for tail calls at all, but it is less general than the CPS-like variant. The calls are computed at compilation time, so higher order functions in tail position are not handled. We chose to use this design because it is way faster than the alternative and because it can cope with most kinds of tail calls. As a matter of fact, tail calls could be optimized more efficiently and probably a lot easier by the JVM.

3.2.3 Using Tableswitches and Lookupswitches

For the common case that several integer tests occur after each other on the same variable, the JVM provides the tableswitch and lookupswitch commands. We use these instructions whenever possible, thus saving repeated load of the same variable and restoring its integer value of the wrapper class. This results in both more compact and faster code.

3.2.4 Code Inlining

One of the slowest JVM instructions is the `invokevirtual` command. Therefore, we gain a good performance increase by inlining often used and short runtime functions. Longer functions are not inlined to keep class files and methods compact (The actual JVM specification [LY97] demands that the byte code of each method is less than 64KB).

3.2.5 Unboxed Representation

The main performance loss we have due to the boxing of primitive values which is necessary because DML is intended to be dynamically typed *and* the source of values may be a pickle file or network connection. The pickle as well as the network connection even might not yet exist when a DML file is compiled, so we can make no assumptions about the type in those cases. Furthermore, the current version of the DML frontend doesn't supply any type information to the backend at all.

To get an idea about the potential of an unboxed representation, we have implemented a naive constant propagation and use this information to omit the **`IValue`** wrapper when builtin functions of a known type are invoked. For example, the following function that computes Fibonacci numbers gains a performance boost by about 35% if only the wrapper for the 3 first constant literals are omitted and the `+`, `-` and `<` functions are inlined.

```
fun fib(n) =
    if (1<n)
        then fib(n-2) + fib(n-1)
    else 1
```

If the type of `fib` were known to be `(int)`$\rightarrow$`int`, the wrappers for `n` and the result of `fib` wouldn't have to be created. By editing the code accordingly by hand, we gain a performance boost by almost 90% compared to the version with only the described optimizations.

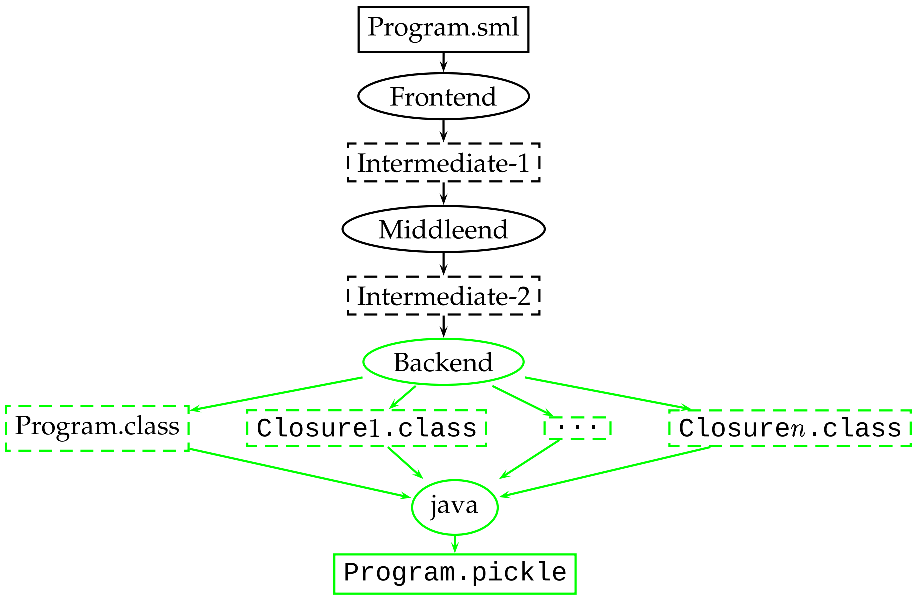
3.3 Summary

The optimizations as described in this chapter cause significant performance increases. Creating literal objects at compilation time speeds up arithmetic computations by about 30%. Omitting wrapper objects for inlined code results in another performance gain of circa 35% after all other optimizations are done. Manually editing the generated code shows us that the use of static type information could further improve performance by about 90%.

Chapter 4

Implementation

The DML compiler backend operates on the intermediate representation which is computed by the Stockhausen implementation for the Alice programming language. Stockhausen performs some platform-independant transformations that are useful for other compiler backends, too. The following figure shows the structure of the compilation process. The intermediate representation as described in Section 2.3 is marked as 'Intermediate-2'. After all classes are created, a Java process is executed on the main class to create a single pickle file that contains the evaluated component.



This chapter gives an overview of the implementation of the compiler backend. Before the class files are written, some optimizations are performed on the generated byte code: Sequences of `aload` and `astore` are omitted whenever possible. A liveness analysis is done and dead code is eliminated.

4.1 The Modules

The compiler backend is split into the following modules.

`Abbrev.sml`

This file defines a structure with abbreviations for the classes, methods and fields that are used in the backend. This avoids typos in the generated code.

`Common.sml`

defines a few functions, constants and structures that are used in most of the other parts of the backend.

`CodeGen.sml`

The main part of the compiler backend. This part transforms the intermediate representation into a list of JVM instructions which is written into Jasmin files by `ToJasmin.sml` afterwards. This is done by the following steps:

Constant Propagation

A hash table is used to map stamps to their value.

Computation of Target Classes

Generally, each function is stored in a separate class. As described in 3.2.2, we gain performance by merging functions that recursively call each other into one `recapply` method of a single class. When computing 'free variables', we are not really interested in the function where a variable occurs free, but in the method. A hash table maps pairs of function stamps and the number of arguments of that function to pairs of target classes and a label within this function. If that target class differs from the function stamp, the code of this function will be stored in the `recapply` method of the target class at the position of this label.

Computation of Free Variables

Remember that variables that are bound in the current method are stored in registers whereas free variables are stored in fields of the class.

Generate the Instruction List

This can be implemented straight forward as described in Chapter 2. In this compilation phase, we don't bother about register allocation. We operate on stamps instead of registers and postpone register allocation to liveness analysis in ToJasmin.sml. Sometimes we want to access some of the special registers 0 (the 'this' pointer), or 1 (the first argument of a method). We define some dummy stamps for these registers.

The optimizations as described in Chapter 3 demand the following changes:

- To realize the constant pool of DML literals, a hash table maps pairs of classes and constants to a number that uniquely identifies the literal in this class. A function inserts a literal into the hash table if necessary and returns the name of the static field associated with this literal in either case. Those static fields are created by a fold over this hash table. The initialization is done in the `<clinit>` method and is also created by a fold. The `<clinit>` method is executed by the JVM immediately after the Class Loader loaded this class, i.e., the class is used for the first time.

- Each generated function class file must have `apply`n methods with n arguments, where n=1...4. Those `apply`n methods create a **`Tuple`** out of their arguments and invoke `apply` by default. If a function has an explicit pattern matching for an n-tuple, i.e., a *`TupArgs`* constructors occurs in the *`FunExp`*, the `apply`n is generated like any other `apply` method.

- The merged `recapply` methods of mutually tail recursive functions differ slightly from ordinary `apply` functions. The number of **`IValue`** arguments of such a `recapply` is that of the function with the most arguments in this block, but not greater than 4. Apart from that, `recapply` takes an integer argument that identifies the function that should be called. The code for all of these functions is merged into one method which is introduced by a tableswitch instruction that branches to the correct function. For accessing registers 2, 3, 4 and 5, we need special stamps again.

  Now applications are compiled like this:

  - Tail call applications with the same target class as the calling method are compiled into a `goto` instruction to the beginning of the function in this method. The arguments are passed via register 1–4 (or 1–5 in case of `recapply`).
  - Any application of a function that resides in a `recapply` is done by invocation of this method.
  - Other applications, i.e., non-tail call applications or tail call applications to an unknown or another that the current method, are compiled into ordinary invocations of the `apply` or `apply`n method, depending on the number of parameters.

- Many functional programs operate on lists. Our compilation so far is relatively inefficient when using lists, because for pattern matching the constructor name of the constructed value in question has to be loaded on the stack to compare it to the predefined '::'. The runtime defines a special class **`Cons`** for cons cells, so a simple `instanceof` instruction suffices to tell whether a value has been constructed by a '::' or not. This optimization is possible for any predefined constructor name, but not for user-defined ones. We could define a new class for the constructed values of each user-defined constructor names. But in the case of distributed programming when using names that were defined in an imported component, we cannot know the class name of the corresponding constructed values. For the same reason we cannot use integer values instead of constructors and constructor names.

- The runtime supplies special tuple classes **`Tuple`**n where $2 \leq n \leq 4$. The content fields of the tuples can be accessed via `getfield` commands. This is faster and less memory consuming than the normal `getContent` method we use for other tuples. See Section 5.3.1 for a description of this optimization.

When the code lists for all methods of a class are constructed, they are passed to the last module of the compiler backend, `ToJasmin.sml`.

`ToJasmin.sml`

This module does the register allocation for the stamps, computes the stack size required within each method and performs the following optimizations before the Jasmin files are generated.

Register Allocation.

The register allocation is done in 2 steps:

1. **Fuse `aload`/`astore` sequences.** A stamp is defined only once and is never bound to another value afterwards. The intermediate representation often binds one stamp to the value of another one. If this happens, a code sequence like `aload` l `astore` s is created. From now on, the stamps l and s contain the same value, so this code sequence can be omitted if further references to s are mapped to l. A hash table is used to map those stamps. This optimization can only be done for unconditional assignments. If more than one `astore` is performed on the same stamp within this method, those stamps must not be fused.

2. **Perform a liveness analysis.** We decided to do a simple but fast liveness analysis. Two hash tables map each stamp to its first defining and last using code position. Branches may influence the lifetime of a stamp as follows. When branching from within the (old) lifetime to a position before the first defining position of this stamp, the new defining position is set to the target of the branch. When branching from behind the (old) last usage to within the lifetime, the last using position is set to the origin of the branch. When all branches are taken into account, each stamp is assigned to the first register that is available within its lifetime.

Dead code elimination.

The code after unconditional branches to the next reachable label is omitted. If the first instruction after a (reachable) label is `goto`, all branches to this label are redirected to the target of this `goto`. If the first instruction after a label is an `athrow` or some kind of `return`, unconditional branches to this label are replaced by this command.

Chapter 5

Value Representation

The basic task of the runtime environment is to define a mapping of DML values to Java classes. Since Java doesn't provide corresponding constructs, we have to write classes that model the behavior of the DML values. The representation of the values is coupled with the primitive operations of the language.

In the following we discuss several implementation strategies and justify our basic design concept. We present a simple and secure implementation of DML values: literals, names, constructors, and tuples straightforward to model. DML threads are based on Java threads, the different kinds of transients can be used to synchronize the computation. The treatement of functions is adopted from Pizza [OW97]. The exception model of DML is implemented as light weight as possible. After representing a simple version of the runtime classes, we describe some optimizations that save access time and memory.

In Section 5.1 we explain and justify our basic class concept. Section 5.2 presents the implemenation of the runtime classes that will be refined and improved in Section 5.3. The features of the Java programming language the implementation relies on are explained as far as needed.

5.1 Basic Concept

The DML language has the following key properties:

1. the language is dynamically typed
2. concurrency with first class threads is supported
3. transients are supported
4. values are higher order
5. any value can be raised as an exception
6. state is represented in terms of references

The modeling of the features of DML into the Java language makes use of the concepts of Java's object oriented system (see Section 1.4). The key properties of DML induce the following constraints on the implementation.

- Property 1 enforces the usage of a common supertype, i.e., a common superclass or the implementation of an interface for all classes that represent DML values. Further, type tests are required at runtime and literals have to be boxed in wrapper classes. In particular this means that we can *not* use the Java Virtual Machine primitive types directly.

- Property 2 causes us to subclass the `Thread` class of Java. We will have to take care to synchronize parts of the code in order to preserve the semantic characteristics of references and transients.
- Property 3, the support of transients, requires additional runtime tests, since transients are used implicitly. This concerns amongst other things the `equals` method of the classes. This is another reason why the implementation has to provide wrapper classes for the literals.
- Property 4 implies that many values can be applied, i.e., many of the objects representing a value are applicable values.
- Property 5 induces us to implement exception raising and handling as light weight as possible; we have to choose between subclassing some Java exception class or using exception wrappers.

For the implementation different possibilities could have been chosen.

1. The first idea is that the superclass value should extend some exception class. Then Property 5 is fulfilled. But exception creation, i.e., the instantiation of an exception class, in Java is quite expensive even if the exception is not raised.
2. The second idea is to use the Java top class `java.lang.Object` as the superclass of all values. Then we will have to use an exception wrapper in order to be able to raise arbitrary values. First class threads can be provided by simply subclassing `java.lang.Thread`. The application of values can be done by explicitly type testing whenever an application occurs — and raise a type error if the object is not a function value.
3. The third idea is to define the supertype value as an interface that declares a method `apply`. This has several advantages with respect to 2. We can implement first class threads by subclassing `java.lang.Thread` and implementing the interface. We reduce the number of necessary type tests as follows. Values that are not applicable implement the `apply` method by raising a type error. This will speed up evaluation when there is no error condition. The usage of an interface makes the runtime more flexible for changes, it is easier to add further types and classes. The declaration is separated from the implementation.

We decided for the third idea, because this will be the most efficient way to implement the runtime. The resulting class hierarchy can be seen in Figure 5.1.

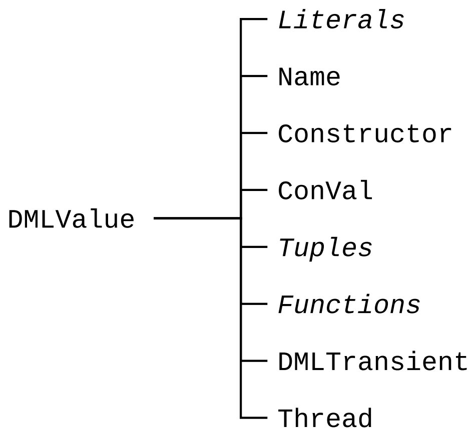


Figure 5.1: *The basic class hierarchy.*

5.2 Implementation

5.2.1 General Notes About The Implementation

Presented Code

In the sequel we present code to illustrate the runtime classes. This code is not 'pure Java' but can be read as pseudo code in the style of Java. Some sections of the actual code are replaced by comments that merely describe what happens instead of giving the hard-to-read Java reality. At other places, macros like `RAISE` are used; their meaning should be clear from the context.

Access Modifiers

Due to the extensions to SML, DML allows much more recursive values. Such recursive values must be constructed top-down. To achieve foreign function security, the content fields of recursive constructed objects should be private to the objects and accessible only via methods that check the semantical constraints. This has one obvious disadvantage: it is too slow. As a consequence, object fields that are immutable in the sense of DML are declared 'blank final' as far as possible. For the other fields security in that sense can only be achieved with high costs that we are not willing to pay. We relax security on demands of efficieny.

Threads and Synchronize

This is how Java understands threads. There is a shared main memory where all objects fields, class fields, and arrays are stored. In the main memory each object is associated with a *lock*. Every thread has its own working memory where it can execute the methods of a Java program. There the local variables of methods are stored; no other thread can see these variables. The working memory (cache) keeps working copies of the shared objects in the main memory. This situation is shown in Figure 5.2. Operations on the memory are executed in terms of *actions*, i.e., atomic instructions, see Figure 5.3. There are rules that define when a thread is obliged to transfer the content of the working copies to or from the shared memory. These rules can be found in [GJS96, Chapter 17] and in [LY97].

Consider the following code of a DML program.

```
let
    val r1 = ref 0
    val r2 = ref 0
in
    spawn (fn () => (r1 := 1; r2 := 2));
    (!r1, !r2)
end
```

According to the semantic rules of DML, there are three possible results of the expression above:

- `(0,0)` — the freshly spawned thread runs only after the references are dereferenced
- `(1,0)` — the spawned thread changes the value of `r1`, then the dereferenciation of both references takes place
- `(1,2)` — the spawned thread does all his work, then the dereferenciation is computed.

However, in Java we have to take care that there cannot be the result `(0,2)`. Such a situation could occur according to the Java rules for threads, cf. [GJS96, Chapter 17.10]. To avoid that outcome of the computation, all methods of the reference objects have to be synchronized, i.e., must aquire the lock of the reference object. This guarantees the correctness of the implementation.

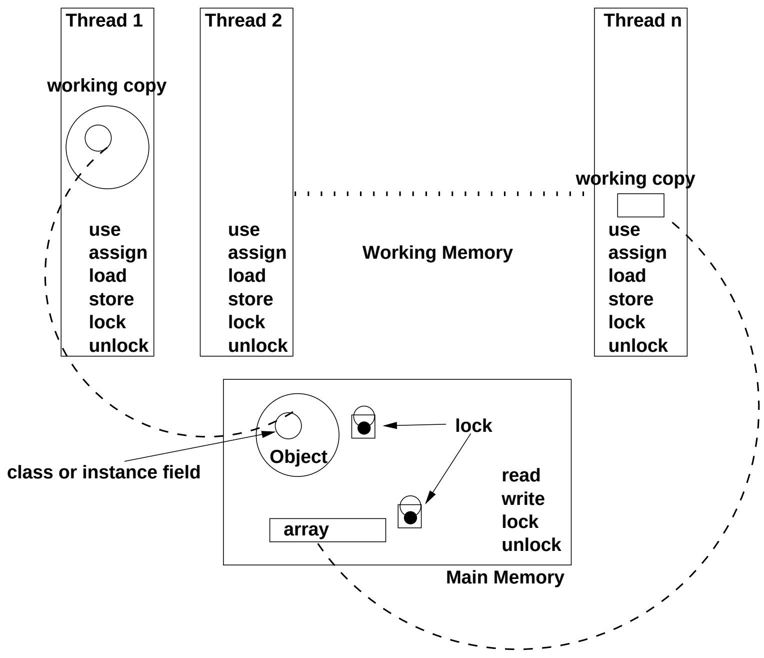


Figure 5.2: *How Java sees the memory of Threads.*

The method `toString`

Each of the classes overrides the `toString()` method in an appropriate way. There is also a method `toString(int)` that can be used to control the 'print depth' of recursive values. Those will not be shown explicitly in the following.

The method `equals`

For many of the objects in the runtime, `equals` is reduced to pointer equality. Because of the transients, we have to reconsider the `equals` method of each class. It usually looks similar to the following:

```
boolean equals(Object other) {
    if (this == other) {
        return true;
    } else if (other instanceof DMLTransient) {
        return other.equals(this);
    } else {
        return false;
    }
}
```

First the pointer equality is tested. If this test fails, we have to consider the `other` object to be a transient. If so, the `equals` of `other` will produce the correct result. Otherwise the objects cannot be equal. In the following we will present the code for `equals` methods only if they differ much from this concept and modulo the test for transients.

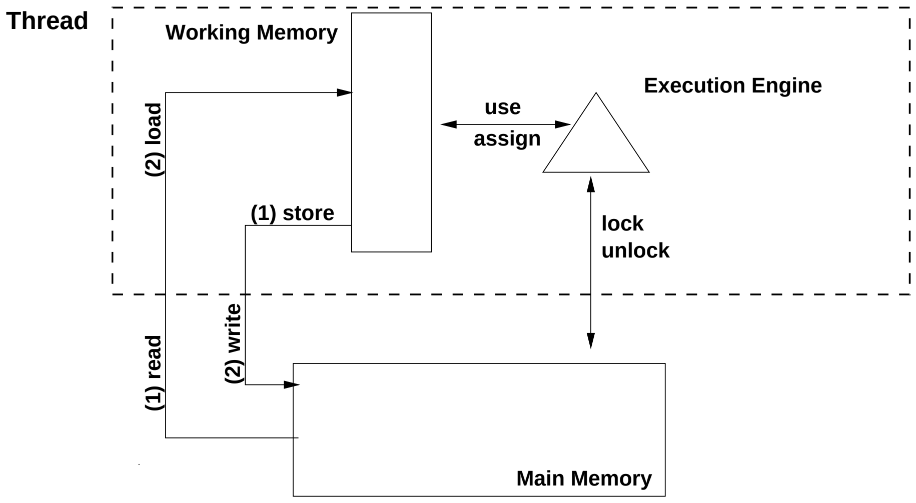


Figure 5.3: *Atomic actions threads may perform.*

5.2.2 DMLValue

The supertype *DMLValue* is provided by defining an interface `DMLValue`. This interface provides basic functions for all subtypes.

```
public interface DMLValue {
    public DMLValue apply(DMLValue val);
}
```

Since any value in DML may be an applicable value, the method `apply` has to be implemented. When invoked at runtime, late binding will resolve the method call to the implementing class and the appropriate object method will be executed.

5.2.3 Literals

In DML the literals consist of integers, words, reals, strings, and characters. These literals are implemented by wrapper classes. Since the corresponding classes `java.lang.Integer`, etc. are declared final, they cannot be extended for our purpose. Therefore we need our own wrapper classes. We present the character wrapper as an example.

```
public class Char implements DMLValue {
    final public char value;
    public Char(char c) {
        value = c;
    }
    public DMLValue apply(DMLValue val) {
        RAISE(runtimeError);
    }
}
```

We define an appropriate constructor for each of the literal classes. Each class has a field where the Java primitive value is stored. Two literals are equal if their primitive values are equal. The application of a literal value results in an error.

5.2.4 Names, Constructors and Constructed Values

Terminology

In SML, there are nullary and unary constructors and exceptions. In analogy we have the following situation in DML. *Names* are nullary constructors, e.g. `unit` and `true`. *Constructors* in terms of DML are unary constructors, e.g. `::`, `ref`, and `Option.SOME`. Since in DML any value can be raised as an exception, there is no need to have a special exception construct. If you apply a constructor to some arbitrary value, you obtain a *constructed value*; this corresponds to a value tagged with a constructor.

`Name`

The application of a name results in a type error. Equality is reduced to pointer equality of Java objects (modulo transients).

```
public class Name implements DMLValue {
    public Name() {}
}
```

`Constructor`

Again, equality is pointer equality. Applying a constructor results in a constructed value with a pointer to its constructor and the content value.

```
public class Constructor implements DMLValue {
    public Constructor() {}
    public DMLValue apply(DMLValue val) {
        return new ConVal(this,val);
    }
}
```

`ConVal`

A constructed value has a reference to the constructor by that it was constructed and a reference to the value. We define a unary constructor that is needed for recursive constructed values, and a constructor with constructor and value arguments.

```
public class ConVal implements DMLValue {
    public Constructor constructor;
    public DMLValue content = null;
    public ConVal(Constructor con) {
        constructor = con;
    }
    public ConVal(Constructor con, DMLValue ct) {
        constructor = con;
        content = ct;
    }
}
```

In the DML model, references are constructed values, too. Therefore we have to consider the following methods:

```
public boolean equals(Object val) {
    if (val instanceof ConVal &&
            ((ConVal) val).constructor != REFERENCE) {
            return
                (constructor == ((ConVal) val).constructor) &&
                content.equals(((ConVal) val).content);
        } else if (val instanceof DMLTransient) {
            return val.equals(this);
        } else {
            return false;
        }
    }
    public synchronized DMLValue exchange(DMLValue val) {
        if (constructor == REFERENCE) {
            DMLValue ret = content;
            content = val;
            return ret;
        } else {
            RAISE(runtimeError);
        }
    }
```

Equality is equality of the constructor and the value if the constructor is not the reference constructor. References have the property to be non-structural equal. The method `exchange` is `synchronized` to make the exchange of the content of a reference atomic.

5.2.5 Tuples and Records

Tuples are implemented by a Java class that stores the items of the tuple in an array.

```
public class Tuple implements DMLValue {
    public DMLValue vals[];
    public Tuple(DMLValue[] v) {
        vals = v;
    }
}
```

We must be aware that it is possible to dynamically create records with new labels. That means we cannot statically map record labels to numbers, i.e., we can't replace them by tuples. Records are somewhat more complex. The labels have to be stored with the values in order to be able to reference them again. Record labels are represented by Java strings and assumed to be sorted by the compiler backend.

```
public class Record implements DMLValue {
    protected Hashtable labels;
    public DMLValue[] values;
    public Record (String[] ls, DMLValue[] vals) {
        values = vals;
        // enter the labels with the values' index
        //  into the hashtable 'labels'
    }
    public DMLValue get(String label) {
        // lookup label's index in the hashtable
        // return 'value[index]'
    }
}
```

We use the hashtable `labels` to map labels to indices in the value array. In this naïve implementation equality testing is rather expensive — labels and associated values have to be compared. Also, records need much memory because each record keeps its own hashtable to lookup the values.

5.2.6 Functions

We proceed as follows:

```
abstract public class Function implements DMLValue {
    public boolean equals(Object val) {
        return getClass().equals(val.getClass());
    }
}
```

We consider two functions to be equal if their class names are the same. SML doesn't allow comparison of functions at all, so this can be seen as an extension.

5.2.7 Transients

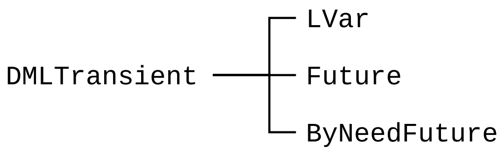


Figure 5.4: The transients of DML.

Terminology

The terminology used in the sequel is defined in Section 1.3.

Implementation

The class hierarchie of the transients is depicted in Figure 5.4. All the variants of transients have three operations in common: `request`, `getValue`, and `bind`. Therefore we define a new interface `DMLTransient` as follows:

```
public interface DMLTransient extends DMLValue {
    public DMLValue getValue();
    public DMLValue request();
    public DMLValue bind(DMLValue val);
}
```

Any concrete implementation of this interface contributes these three methods. The first implementation presented is `LVar`, representing a logic variable.

```
public class LVar implements DMLTransient {
    protected DMLValue ref = null;
    synchronized public DMLValue getValue() {
        if (ref == null) {
            return this;
        } else {
            if (ref instanceof DMLTransient) {
                ref = ((DMLTransient) ref).getValue();
            }
            return ref;
        }
    }
    synchronized public DMLValue request() {
        while (ref == null) {
            try {
                this.wait();
            } catch (java.lang.InterruptedException e) {
                // This should never happen!
            }
        }
        if (ref instanceof DMLTransient) {
            ref = ((DMLTransient) ref).request();
        }
        return ref;
    }
}
```

The `LVar` contains a reference to the not yet known value. The method `getValue` returns the content if known (and makes a path compression on the reference chain) or the logic variable itself. The method `request` either returns the known value (again with path compression) or suspends the current thread. The suspended threads are entered into the wait-set of the logic variable.

```
synchronized public DMLValue bind(DMLValue v) {
    if (ref != null) {
        RAISE(LVar.Fulfill);
    } else {
        while (v instanceof DMLTransient) {
            if (v == this) {
                RAISE(LVar.Fulfill);
            }
            DMLValue vv = ((DMLTransient) v).getValue();
            if (v == vv) {
                break;
            } else {
                v = vv;
            }
        }
        ref = v;
        this.notifyAll();
        return UNIT;
    }
}
```

By using `bind` on the logic variable the reference is bound and all threads in the wait-set are woken up. Equality is the equality of the referenced values. If the logic variable is used as a function, the `apply` method simply requests the referenced value and calls its apply.

`Future`s are essentially like `LVar`s, except that the `bind` operation will result in an error.

The third implemented variant are `ByNeedFuture`s. A `ByNeedFuture` contains a reference to an applicable value `closure` of type `unit -> 'a`. When the value is requested, `closure` is applied to `unit` and the reference of the function replaced by the result of the application. The `ByNeedFuture` knows three states: not yet evaluated function (`UNBOUND`), an exception occurred when the function was evaluated (`ERROR`), and successfully evaluated function (`BOUND`).

5.2.8 Threads

All threads we will run in a DML program are instances of the following class.

```
public class Thread extends java.lang.Thread implements DMLValue {
    final public DMLValue fcn;
    public Thread(DMLValue v) {
        fcn = v;
    }
    public void run() {
        fcn.apply(UNIT);
    }
}
```

A DML thread extends the class `java.lang.Thread`. The builtin function `spawn` uses the thread constructor with the `DMLValue` argument. The thread defines the `run` method that will be executed as its main loop. Equality is pointer equality. Uncaught Java exceptions and uncaught DML exceptions will be propagated up the call stack and they will not leave the thread context. The thread terminates, the exception is printed to the console.

We could have solved the problem of tail recursion with a field `tail` in the class Thread where the continuation would have been passed. The tail recursion is explained in detail in the backend (see Section 3.2.2).

5.2.9 Exceptions

Speaking in Java terms, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Exception are intended to be used when an error condition occurs. This has the advantages that

- the programmer can separate the error handling from the normal control flow
- errors are propagated up the call stack
- errors can be grouped and differentiated

Java has different types of exceptions, including I/O exceptions, runtime exceptions, and exceptions of one's own creation. The exception can either be *checked* or *unchecked*. Unchecked exceptions are subclasses of the class `RuntimeException`. Runtime exceptions can occur anywhere in a program and in a typical Java program can be very numerous. The cost of checking for runtime exceptions often exceeds the benefit of catching or specifying them. Thus the compiler does not enforce that you catch or specify runtime exceptions, although you can do so.

In DML we want to be able to raise any value as an exception. Since we don't want our values to extend some of the Java exception classes, we need to wrap the values to be raised into a wrapper class. Because in DML an exception can be raised everywhere and will be handled explicitly in some other (Java) context, it is a good idea to make our exception wrapper class a runtime exception.

```
public class ExceptionWrapper extends RuntimeException {
    final public DMLValue value;
    public ExceptionWrapper(DMLValue val){
        value = val;
    }
}
```

The value to be raised is an argument to the constructor of the wrapper class.

5.2.10 Miscellaneous Types

Literals

There is one additional class `JObject` that represents Java API objects. With some builtin functions, any Java object can be treated as a DML value. The `JObject`s cannot be seen as a real interface to the Java API. This would require a support by the frontend, cf. [BKR99].

Ports, Vectors, Arrays

One of the more sophisticated types is the `Port`. Ports implement a many-to-many communication. They use a stream, i.e., a list with a logic variable at the end. If a thread wants to write to a port, the logic variable at the end of the list is replaced by a 'cons' cell with the message as the 'car' and a fresh logic variable as the 'cdr'.

```
public class Port implements DMLValue {
    LVar last = null;
    public Port(LVar f) {
        last = f;
    }
    public DMLValue send(DMLValue msg) {
        LVar newLast = new LVar();
        synchronized (last) {
            last.bind(new Cons(msg, new Future(newLast)));
            last = newLast;
        }
        return UNIT;
    }
}
```

Receiving messages from a port is an implicit operation; it is not distinguishable from a usual list. Appropriate primitive functions for creation of ports and sending messages are supplied:

```
type 'a port
val newPort : unit -> 'a port * 'a list
val send : 'a port * 'a  -> unit
```

The SML Basis Library defines Arrays and Vectors. For reasons of efficiency these additional types are implemented in Java. The implementation is straightforward, and the code is not represented here.

5.3 Enhancements

All the enhancements implemented in this section result in the class tree shown in Figure 5.5.

5.3.1 Tuples

In the case of tuples there can be done several things to improve efficiency. First, we implement special classes `Tuple`*i* for tuples of arity 2, 3, and 4. We avoid the indirection of a Java array and achieve faster access to the subvalues by defining 2, 3 resp. 4 object fields.

Since all these `Tuple`*i*s are of type tuple, we define a common interface `DMLTuple`. To preserve the former class hierarchy position of the record class, the interface also declares a method `get(String)` that can be used to fetch values by a label.

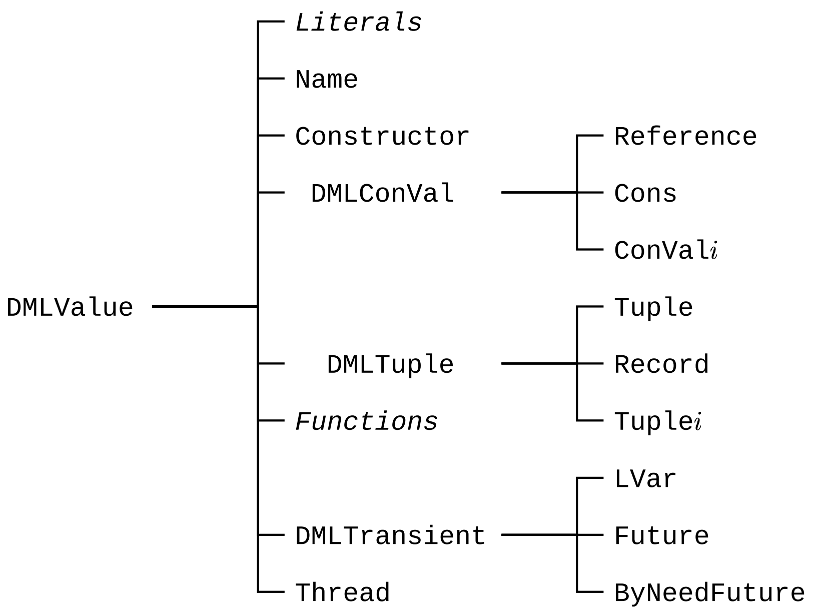


Figure 5.5: The class hierarchy after considering the enhancements.

Records are more efficient if the labels (we assume labels are always sorted) are stored in a global hashtable and the record only keeps a pointer to the so called `RecordArity`. The comparison of two records then is reduced to pointer equality of the records' arities. Only if the arities match, the values in the record have to be compared. Furthermore record values can be accessed by index (since the sorted labels are the same) rather than by label.

5.3.2 Constructed Values

Constructed values also can be specialized into `ConVal`i classes that avoid the indirection of a tuple but store the content in object fields. We define an interface `DMLConVal` similar to the `DML-Tuple` interface. Now a method `getConstructor` is necessary, since interfaces cannot define object fields.

Another point is the storage of the constructor: references and cons-cells know their constructor statically and don't need an object field for it; we define two special classes `Reference` and `Cons`.Since we have different objects for references and cons-cells we also have to take care of the constructors. The class `Constructor` is subclassed for each of `::` and `Reference`, and the default implementation of `apply` is changed. In both cases testing for the constructor can be reduced to an `instanceof` test.

5.3.3 Functions

In SML functions only have one argument. If a function is always called with tuples, we may avoid the wrapping and unwrapping of values into tuples. We can gain some speed by specializing `apply`. We define `apply0()`, `apply2(v,w)`,... Still, the compiler must produce a method `apply`, not only because the interface definition requires it, but because the function could be called at runtime with some other value than a tuple. If the closure of a function is empty, i.e., there are no free variables, we can additionally define static methods `sapply`i. There are of course changes to the `DMLValue` interface — now the `apply`s have to be defined. All the implementing classes have to provide implementations.

5.3.4 General Speedups

In Java, there are several things to do to speed up execution of methods. One thing is to declare methods to be class methods, i.e., `static` methods. Another improvement is the declaration of methods to be `final` whenever possible. It is also a good idea to avoid `synchronized` methods — these are much slower. If possible, classes should be declared final, too.

Since the Java Virtual Machine specification doesn't define the cost of method invocation or object creation or anything else, these enhancements vary in result on the concrete implementation of the virtual machine.

5.4 Summary

The DML runtime can be implemented in Java with a reasonable effort. The DML top level type is defined by an interface `DMLValue`. All classes representing DML values have to implement that interface. The `equals` method of any class has to be aware of transients. The method `apply` has to be reimplemented by all of the classes since Java provides no multiple inheritance or default implementations.

DML threads can be based on Java threads, and transients use Java's `wait` and `notifyAll` methods to synchronize the computation. The transients are not replaced by the value they are bound to — there are always reference chains, but the implementation provides the illusion of disappearing transients. To avoid that, transient support by the virtual machine would be needed.

New DML types can easily be added to the runtime environment. Any new type just has to implement the `DMLValue` interface and to contribute the implementation for equality testing and application.

Chapter 6

Pickling

Pickling is the process of making (higher order) stateless values persistent, i.e., to store them to some medium. Pickling plays a major role during the compilation process: each DML program is compiled into a pickle file. Execution then consists of depickling such a file and calling the `main` function of the pickle (see Chapter 2.3.1).

Pickling uses Java's Object Serialization mechanism [Sun98]. Object Serialization encodes an object, and objects reachable from it, into a stream of bytes and reconstructs the object graph from the stream when deserializing. The properties of Object Serialization have to be extended in order to achieve the semantic properties of pickling: while deserialization creates new objects, we want the objects to preserve their identity. Further, pickling abstracts away the existence of class files for user defined functions and stores the byte code with the function instances in the pickle. The size of the pickle files can be reduced by compression of the byte stream.

The Object Serialization of Java provides many possibilities to customize the default serialization behavior and the implementation of pickling requires only small changes to the runtime classes as presented in Chapter 5.

6.1 To Pickle Or Not To Pickle

In DML stateless values can be pickled. Stateful entities such as transients, references, ports, and threads cannot be pickled and pickling one of those results in a runtime error. For names and constructors, we have a special handling to preserve pointer equality: *globalization* and *localization*. The following table summarizes the behavior of the diverse values.

Value Type	**Pickled As**
Literals	themselves
`Name`	globalized name
`Constructor`	globalized constructor
`ConVal`	constructor globalized
`Tuple`, `Record`	themselves
`Function`	closure
`Reference`	runtime error
`Thread`	runtime error
`DMLTransient`	runtime error
`Array`, `Port`	runtime error

Pickling can be invoked in the following way

```
val _ = General.pickle ("filename",something)
```

and unpickling is simply

```
val v = General.unpickle "filename"
```

6.2 Pickling and Serialization

To implement pickling, Java Object Serialization is used. The Java Object Serialization process is specified in [Sun98]. Here we give a short summary of the serialization concept and its properties. The differences to pickling are elaborated, and the hooks are shown by which the pickling process can be customized.

The default properties of Java's serialization are

1. objects are optionally serializable
2. serialization/deserialization works *by copy* rather than *by reference*
3. the object graph is traversed
4. cyclic data structures and coreferences are preserved
5. the class code of objects is located by the class name which is attached to each object
6. static fields are not serialized with the object instances

Pickling needs some more sophisticated treatment. We want further

7. objects to enshrine their identity
8. the class code to reside in the pickle
9. static values of closures to be stored with the function's class. At the instantiation time of the closure, the transitive closure of all reachable data structures is built up. The external references of the code are made explicit as static fields of the function class.

As one can see there is some conflict between 2. and 7., 5., and 8. as well as between 6., and 9.. Considering this, the following has to be done:

- provide a mechanism for the pickled objects to have a unique identity
- modify the serialization so that class code will be attached to the objects in the pickle
- figure out how static properties of closures can be stored so that we don't have to compute them at unpickling time

Since serialization produces copies of objects, pointer equality is lost when serialization works on instances of `Name` or `Constructor`. Therefore, to pickle `Name` or `Constructor` values requires *globalizing* them during serialization, and respectively *localizing* them during deserialization. For the builtin names and constructors (`true`, `false`, `::`, etc.) we need *unique* names and constructors that are further equal across machines and pickling operations.

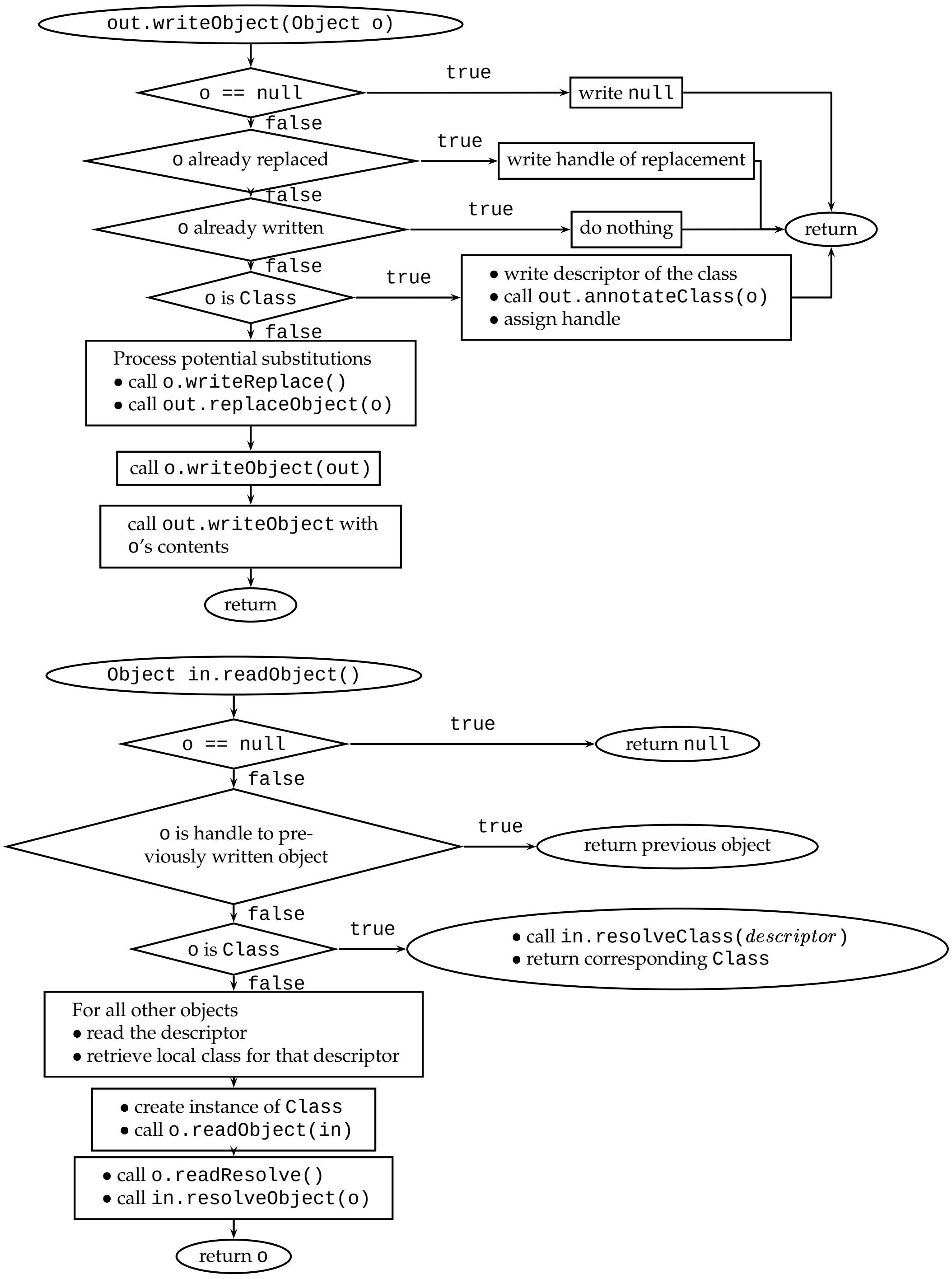

out.writeObject(Object o)
o == null
true
write null
false
o already replaced
true
write handle of replacement
false
o already written
true
do nothing
return
false
o is Class
true
• write descriptor of the class
• call out.annotateClass(o)
• assign handle
false
Process potential substitutions
• call o.writeReplace()
• call out.replaceObject(o)
call o.writeObject(out)
call out.writeObject with o's contents
return
Object in.readObject()
o == null
true
return null
false
o is handle to previously written object
true
return previous object
false
o is Class
true
• call in.resolveClass(descriptor)
• return corresponding Class
false
For all other objects
• read the descriptor
• retrieve local class for that descriptor
• create instance of Class
• call o.readObject(in)
• call o.readResolve()
• call in.resolveObject(o)
return o

Figure 6.1: *Java's Object Input and Output Routines*

6.2.1 Outline of Java Serialization

To serialize objects, the `java.io.ObjectOutputStream` class can be used. To write the objects to an underlying output stream the `ObjectOutputStream` provides a method `void writeObject(Object)`. To deserialize objects there is the `java.io.ObjectInputStream` class. It provides the input routine by the method `Object readObject()`[1]. All objects that want to use the default serialization mechanism have to be declared serializable by implementing the interface `java.io.Serializable`. This interface has no methods to implement, it serves only to identify the semantics of being serializable.

The algorithm for the output and the input of objectgraphs is depicted in Figure 6.1. Let `out` the outputstream and `in` the inputstream. The serialization starts with a call to `out.writeObject` with the object `o` that is to be serialized. The object's class must implement `Serializable`, or else an exception is thrown. Deserialization is performed by the `readObject` method. Each call to `readObject` returns one data structure. The depicted algorithm is simplified for the sake of understanding; the precise description can be found in [Sun98].

So, if a class implements `Serializable` and is written to resp. read from one of the provided streams, one has the default behavior with the following properties:

- data structures are serialized by traversal
- coreferences and cycles in the structures are preserved
- objects are written once to the stream
- each object is annotated with its corresponding class descriptor

In order to customize this behavior we can on the one hand implement several methods in the classes of the objects we want to serialize. On the other hand, we can subclass the default streams and customize the output algorithm. The following methods of the respective class influence serialization:

1. (a) `private void writeObject(ObjectOutputStream)`, and

   (b) `private void readObject(ObjectOutputStream)`

   can be overridden to provide special handling for the classes' fields. Note that the access modifier `private` is mandatory here.

2. (a) `Object writeReplace()`, and

   (b) `Object readResolve()`

   give the user a way of replacing and resolving objects by other objects. The `readResolve` is called after deserialization, but before the object is inserted into the object graph.

The following methods of the stream classes are overridden and modified in the implementation.

1. `void annotateClass(Class)` of `ObjectOutputStream` is invoked when the object's `Class` is written to the stream. This happens before the object itself is written. Therefore this class annotation will always be read from a stream before the corresponding instances.

2. `Class resolveClass(`*descriptor*`)` of `ObjectInputStream` is called when deserialization of an object starts and before the actual reading of the object itself.

[1]We are aware of the fact that using the default mechanism of Java is not the best solution considering speed or size of serialized objects. Efforts have been successfully made to improve the general behavior of the object serialization process, e.g., in [HP99]. Yet the improved versions don't have the full functionality (replacing objects is not possible), so we have to use the Java defaults.

3. `Object replaceObject(Object)` of the `ObjectOutputStream` replaces objects similar to the `writeReplace` method of the class to serialize. The `replaceObject` is called only if `enableReplaceObject(true)` method has been invoked on the stream.

The tasks of pickling can be fulfilled as follows. Globalizing and localizing uses the mentioned methods of the classes. Annotation of byte code to the objects uses `annotateClass` and `readResolve`. The `replaceObject` method will be used to check the admissibility of the pickling.

6.3 Implementation of Pickling

The tasks can now be specified as follows:

- modify the default serialization behavior of
  - transients, `Thread`, `Array`, `Port` etc. to raise errors
  - `Name` and `Constructor` to be globalized/localized
- annotate objects of type `Function` with their byte code and their static fields, i.e., store class properties with objects
- load the byte code of the classes from the pickle when deserializing, and
- provide a class loader that can handle pickled classes.

In order to use the default serialization of Java, the interface `DMLValue` looks like this:

```
public interface DMLValue extends java.io.Serializable {
    // ...
}
```

Most of the already mentioned value types are now ready to be serialized and nothing has to be changed in the classes, excepting `Name` and `Constructor`.

6.3.1 Globalization and Localization

Globalization and localization concerns names and constructors. It preserves pointer equality in terms of Java. The names and constructors must be globalized when serialized and localized when deserialized. The builtin names and constructors must be unique across machines and serialization processes. The implementation realizes localization by object replacement.

Therefore, we add a field to the classes where a so-called *GName* (for 'global name') is stored. The field is initially empty. A `GName` is an object that is globally unique across Java machines. The constructor and name objects can then be identified by their `GName`. A global hashtable maps the `GName`s to objects. The globalization takes place in the `writeObject` method of `Name` resp. `Constructor`: If the name or constructor is pickled for the first time, it gets a new `GName` and registers itself with the new `GName` in the global table `GName.gNames`. After that, the default mechanism is used for serialization.

The localization is implemented by `readResolve`: When unpickling a name or constructor object, the constructor is looked up by its `GName` in the global table. If there already is a reference, the current object reference is discarded and the one from the hashtable is used instead.

Unique names and constructors now can be realized by subclassing the `Name` and the `Constructor` class and rewriting the write and read routines. This is done by replacing the `GName` by a string. Now the `GName` is already initialized when the name or constructor is created. Therefore, nothing needs to be done in `writeObject`[2]. It cannot be assumed that such names and constructors are always available in the global hashtable since there is the possibility that a unique object is first encountered during unpickling[3]. Thus `readResolve` checks if the unique object is already present and returns either the value in the hashtable or the freshly created object.

6.3.2 Annotation of Class Code

Since functions are instances of classes we have to care about their byte code definition. One of the purposes of pickling is to make the presence of class files transparent for the programmer. To achieve transparency the byte code of the class files is stored with the instance of the class. It would be a waste of resources to annotate all of the function classes because much of the byte code is defined in the runtime's basis library and exists on all machines. Therefore we split up functions in two kinds: user defined *functions* and runtime defined *builtins*. In the following, we annotate only objects of type `Function`. The resulting class tree is shown in Figure 6.2.

The annotation with class code cannot be implemented by modifying the methods of the `Function` objects — at the time of the deserialization the code needed to countermand the annotation of the object is the concrete implementation; what we have here is a variant of the 'chicken-egg-problem'. So the annotation of the classes is deferred to the output stream.

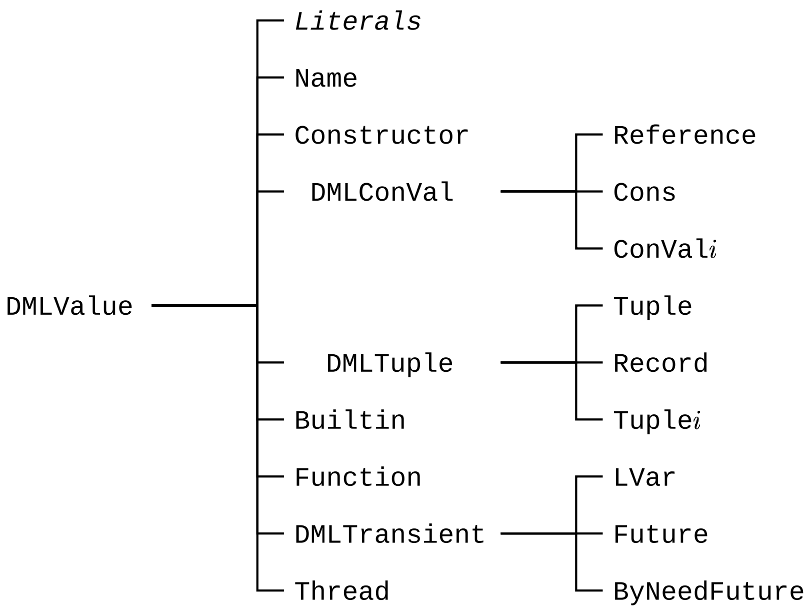


Figure 6.2: *The class hierarchy after the introduction of pickling.*

We define a `PickleOutputStream` subclassing `ObjectOutputStream`. The constructor of the class enables replacement of objects. We annotate the `Class` object of the `Function` objects. During the serialization process, Java writes a `Class` handle for each object before the object itself is written (see Figure 6.1). The annotation is implemented in the `annotateClass` method. We do not present the detailed code here because it is rather tricky to find out where to get the class code from and how to write it to the stream. The concept of the method is as follows: The method

[2]The method is *not* inherited from the superclass because of the required `private` access.

[3]On the other side, the runtime takes care that the builtin names and constructors are created only if they do not already exist.

`replaceObject` gives us the possibility to clean up references of transients and the chance to wait for transient values to be bound. This is the place where an error is raised if someone tries to pickle a `Thread`, `Array`, `Port` or `Reference`. We prevent the pickling of transients or threads etc. Note that this does *not* prevent those objects from being serialized — it's just not possible to *pickle* them. We will need this feature later, when we use distribution, cf. Chapter 7.

6.3.3 Class Loading

To deserialize pickles we need a class loader that can handle the pickled class code. To create such a class loader, the `java.lang.ClassLoader` class is extended and adapted for our purpose. The class loader contains a hashtable where the byte code is stored by the corresponding class name. The method `findClass` is modified so that it can define classes that are stored in the hashtable. The `PickleClassLoader` will be invoked when annotated classes are detected to define the classes in the VM.

6.3.4 Reading Pickles

To read pickles we adapt `java.io.ObjectInputStream` for our purpose. When the `Object-StreamClass` of an object is read from the input stream, the `resolveClass` method of the stream is called. At that time, we determine if the class was annotated with byte code by the `PickleOutputStream`. The byte code of the class is then read and defined with the `Pickle-ClassLoader`. Then we read the values of the static fields from the stream.

6.4 Summary

Pickling of DML values is implemented using Java Serialization. The `DMLValue` interface extends `Serializable` so that all values use the default mechanism. Modification of the objects during serialization can be realized by implementing `readObject` and `writeObject`. The globalization and the localization are realized by the methods `readResolve` and `writeReplace`. The annotation of special classes with their byte code requires the usage of a customized `Ob-jectOutputStream` and `ObjectInputStream`. The annotation itself takes place in the `anno-tateClass` method. Furthermore a class loader has to be created that can handle the byte code from the pickle. That class loader is used during deserialization in the method `resolveClass`. As a by-product, pickling cleans up references of transients. The actual implementation of pickling further uses compression on the pickled files. We just use a `GZIPOutputStream` resp. a `GZIPInputStream` as an intermediate layer between the file stream and the pickle stream. Pickling simplifies the usage of the compiler since it abstracts away the class file handling.

Chapter 7

Distributed Programming

This chapter introduces DML's features of distributed programming and explains the basic concepts. We show how a high level distribution can be implemented in Java.

Today computers are no longer standalone machines but are usually connected together by networks of different kinds. Still, it is an often error-prone task to implement distributed applications, partly because the programmer has to explicitly deal with the network.

By making mobility control a primitive concept, one can abstract away the network and so release the programmer of the low level handling of transport layers etc. By defining the behavior of different data structures, the network is still there, but on a higher level. The key concepts are *network awareness* and *network transparency*[1]. The idea behind the distribution model of DML comes from Mozart [HVS97, HVBS98, VHB$^+$97].

Java's design goals include distributed programming in heterogenous networks, so we can hope to find a simple way of implementing distribution in DML. Java Remote Method Invocation (RMI) provides the mechanism by which servers and clients may communicate and pass information back and forth. One of the central features of RMI is its ability to download the code of a class if the class is not available on the receiver's virtual machine. Yet this involves the presence of a HTTP or FTP server; the distribution of DML removes this necessity by modifying some classes in the Java API.

This chapter is organized as follows. In Section 7.1 we explain the usage of distribution in DML. Section 7.2 gives a summary of Java's distribution model. The distributed semantics of DML is specified in Section 7.3 and the implementation of the DML model is presented in Section 7.4. To facilitate the handling of class code we have to modify some of the Java RMI classes; this is explained in Section 7.5. The results of the chapter are summarized in Section 7.6.

7.1 Establishing Connections in DML

The user of DML can provide values for other sites with

```
Connection.offer : 'a -> 'a ticket
```

This function takes a value and makes it available for access from some other machine. It returns a ticket by which the value can be taken from somewhere else with

```
Connection.take : 'a ticket -> 'a
```

[1]These terms first appeared in [Car95].

A `ticket` is a unique identifier that carries information about where the value is offered. Such a `ticket` is simply a string and can be sent to an other site via email or something like that.

7.2 Java RMI

Java has a distributed object model, implemented by using Remote Method Invocation *RMI*. RMI is a correspondent to Remote Procedure Call *RPC* in procedural languages. RMI abstracts away the communication protocol layer and tries to be both simple and natural for the Java programmer. Java RMI uses Java serialization; many of the properties discussed in Chapter 6 are maintained.

Speaking in Java terms a *remote object* is an object on one Java Virtual Machine that provides methods that can be invoked from another Java Virtual Machine, potentially on a different host. An object of that kind implements some *remote interface* that declares the methods that can be invoked remotely. A *remote method invocation* is the action of invoking such a method on an remote object. There is no difference in syntax for remote or non-remote methods. Similar to the non-distributed environment, references of remote objects can be passed as arguments to and as results from methods. One can still type cast a remote object along the implemented interfaces hierarchy. The `instanceof` operator of Java works as before.

To make objects remote one must define an interface that extends the interface `java.rmi.Remote`. The remote object implementation class must implement that interface. The remote object class must extend one of the `RemoteObject` classes provided in the `java.rmi` package, e.g., `UnicastRemoteObject`. It is further required to define an appropriate constructor for the class and to provide implementations for the remote methods in the interface.

There are some characteristics that the programmer has to keep in mind when using RMI. First, remote objects are usually referred to by their interface methods and never by their implementation. Second, non-remote arguments that are used in a remote method invocation are transferred using Java serialization and therefore passed by copy. In contrast, remote objects are passed by reference. Two further details have to be respected: some methods of `java.lang.Object` are specialized, and each remote method must declare to throw the checked exception `java.rmi.RemoteException`.

Java RMI applications usually are comprised of several programs, namely a server and a client. Server applications create remote objects and make them available for clients, which in turn invoke methods on those objects. The arguments of these methods (and the return values) can also be remote objects created by the client — eventually the client is a server, too. So there are the following tasks to comply with:

1. provide a naming service for remote objects,

2. organize the communication between remote objects

The naming service is provided by the so called *registry*. The communication between objects is implemented by RMI. Since in Java a subclass can be passed instead of the class itself we have to

3. load class code of objects for which the class code is not available locally

This is realized by RMI's class loading facility which annotates objects with the location of the corresponding class. The code can then be fetched from the web server denoted by the location. Figure 7.1 shows the Java model of distribution.

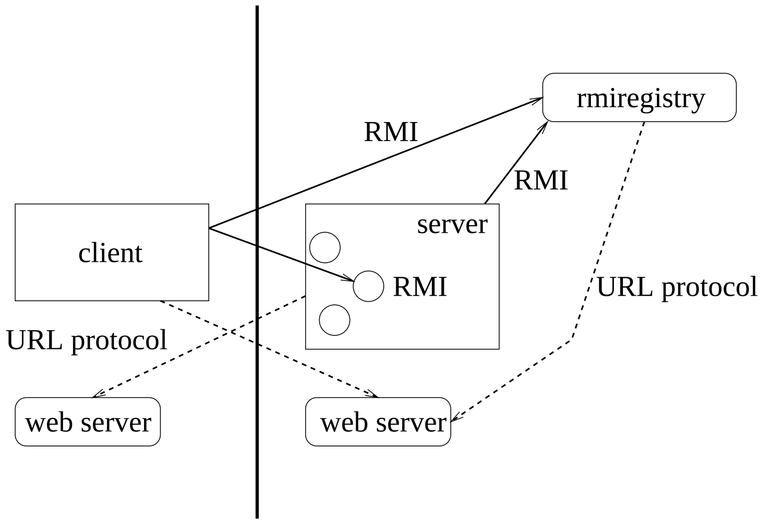


Figure 7.1: *RMI distributed computation model.*

7.3 Distributed Semantics of DML

To use the distribution correctly a semantics has to be specified for the primitive values. The semantics of DML tries to imitate the semantics of distributed Oz (see [HVS97]). A summary of the semantics is depicted in Figure 7.2.

Furthermore, while Java speaks in terms of *clients* and *servers*, the DML distribution model doesn't make such a distinction, we simply talk about *sites*. The *home site* of a value *V* is the site, where the value was created; other sites that have a reference to *V* are called *foreign sites*.

Literals, constructed values, names, constructors, records, tuples, and functions are *stateless* and can be replicated, i.e., if such a value is offered to another site, the requesting site simply gets a copy of that value.

Stateful values such as references, transients, threads, arrays, ports, and vectors are treated in a different way. Threads and arrays are not sent to another site, but are replaced by a *NoGood* which does not allow the foreign site to perform any operationson the value. `NoGood`s are globalized so that if a `NoGood` comes back to its creation site, it will be localized and replaced by its former entity, and the home site can use the respective value as before. Ports and transients are stationary on their creation site, and foreign sites only get a proxy for the requested value. All operations performed on such a proxy are forwarded to the home site and executed there.

References use a *Mobile Protocol* that works as follows. On each site, references have a proxy. Only one site can own the content in the proxy. There is one manager on the home site of the reference which is responsible for the coordination of the proxies. To request the cell content the proxy on the site asks the manager for it. The manager then takes the content from the actual owner and gives it to the requesting site. Subsequent requests are handled in a first-in-first-out-manner.

7.4 Implementation

To implement the distributed part of DML, Java's Remote Method Invocation is used. This has the advantages that

- there is much functionality implemented

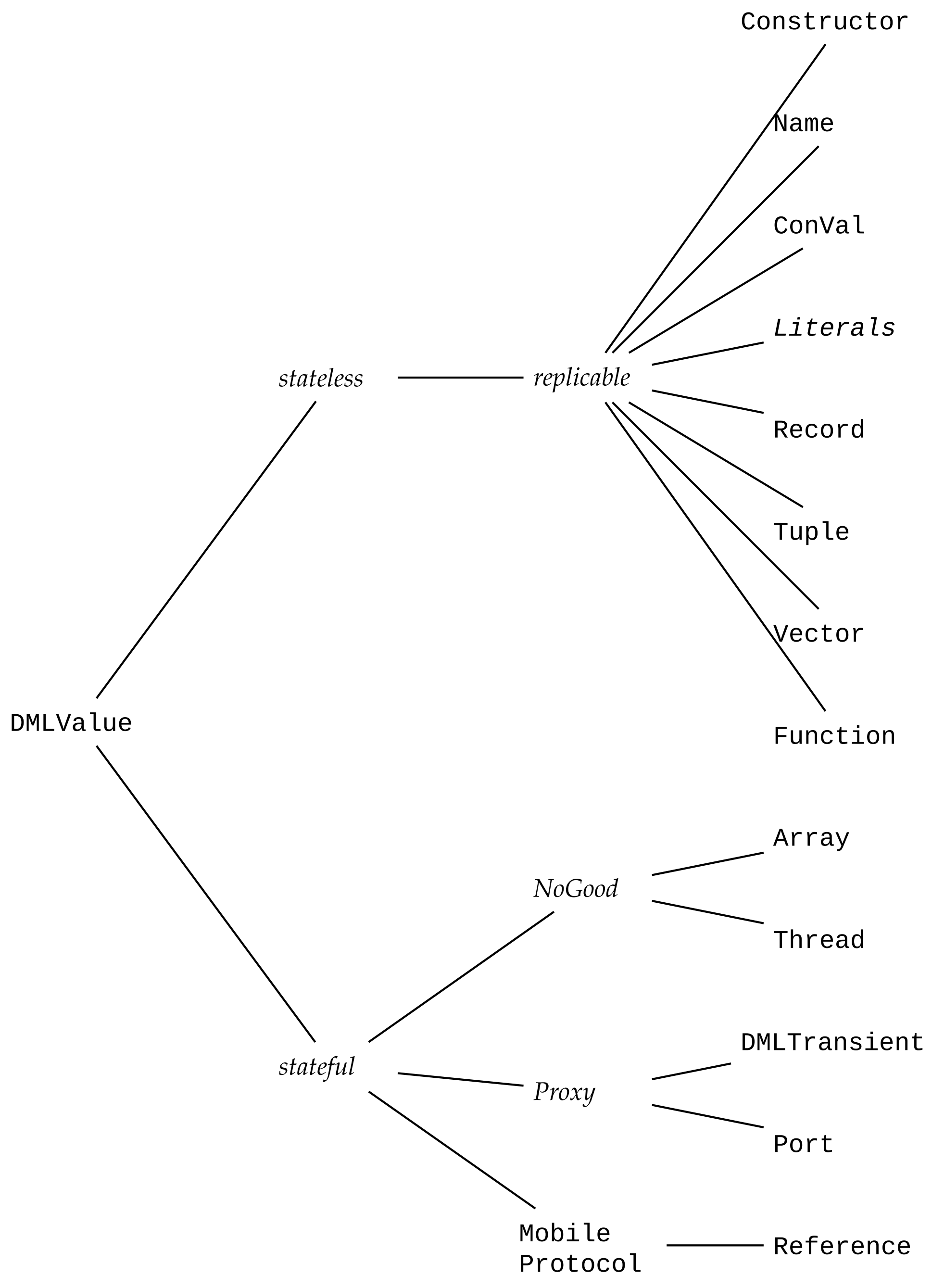


Figure 7.2: *DML values and their distributed semantics.*

- distribution is compact and secure
- distributed garbage collection is provided by RMI

There is one problem if we want to use the standard RMI. The class code for values must be provided via a web server working on the offering machine. To facilitate the handling and avoid the need for a web server, we have to reimplement some classes of the RMI API (see Section 7.5).

7.4.1 Providing Values

To facilitate the handling of tickets and values, all the interchange of values works via one bootstrap remote object. This bootstrap object is created and installed the first time one of the `Connection` functions is used. At that time, the registry is started on a specified port of the host and the special bootstrap object is bound in that registry. The bootstrap object is defined by the `Export` interface and the implementing `Exporter` class. All communication goes through the exporter; that object handles an internal hash table that maps tickets to values. This situation is illustrated in Figure 7.3. There is one registry per host, several JVMs may use one single registry.

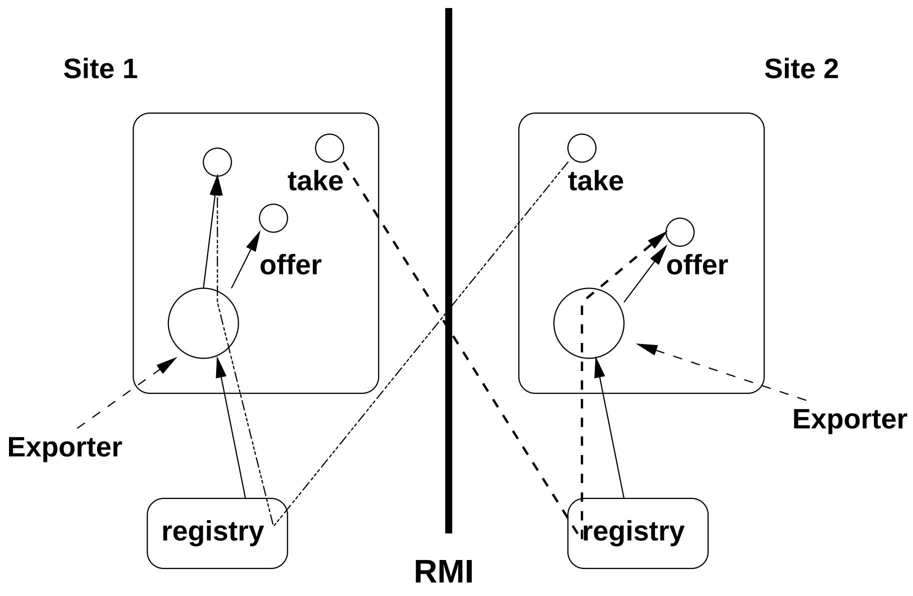


Figure 7.3: *Offering and Taking Connections*

7.4.2 Stateless Entities

All stateless values as listed in Figure 7.2 are replicable. Literals are easy to implement: nothing has to be changed. The same holds for tuples and records. Functions and builtins don't have to be changed either, since their default behavior is to be passed by copy, and this is the desired way.

7.4.3 Stateful Entities

`Array` and `Thread`

Values that are offered as `NoGood`s have to be changed slightly, e.g., the `Array` class.

```
public class Array implements DMLValue {
    private NoGood ng = null;
    private Object writeReplace() throws ObjectStreamException {
        if (ng == null) { // if serialized for the first time
            GName gn = new GName();
            ng = new NoGood(gn);
            GName.gNames.put(gn,ng);
            return ng;
        } else {
            return ng;
        }
    }
}
```

The `writeReplace` method replaces the values by an instance of the `NoGood` class. In order to achieve uniqueness, the `NoGood` has an associated `GName`. The actual value is stored under that `GName`, so that original value can be restored if the `NoGood` returns to its creation site.

`Transient`s and `Port`s

For the proxy-providing values the following has to be done. We define new interfaces `DMLTransient` and `DMLPort` for the respective classes. These interfaces extend the `Remote` interface of Java's RMI and our `DMLValue`.

```
public interface DMLTransient extends Remote, DMLValue {
    public DMLValue getValue() throws RemoteException;
    public DMLValue request() throws RemoteException;
    public DMLValue bind(DMLValue val) throws RemoteException;
}
```

The methods declared in the `DMLTransient` interface can be invoked remotely. No other methods of the transients can be invoked, since the remote site only sees the `DMLTransient` interface.

The remote interface for the `Port`s looks similar; `Port`s provide a method `send` and, to preserve the DMLValue type, a method `apply`.

In order to make objects stationary, it is required to extend one of the `RemoteObject` classes (remote objects are passed by reference). In our case this is realized as follows:

```
public class LVar extends UnicastRemoteObject
    implements DMLTransient {
    public LVar() throws RemoteException { }
    // ...
}
```

7.4.4 References

References implement the *Mobile Protocol*. On its home site, the reference is associated with a server manager. All sites communicate with the reference via a client manager. The server manager and the client manager are modeled by remote objects. The reference is a non-remote object and transferred by serialization.

The manager classes implement remote interfaces, `SManager` and `CManager`. These have the following definitions respectively.

```
public interface SManager extends Remote {
    public DMLValue request(CManager requester) throws RemoteException;
}

public interface CManager extends Remote {
    public DMLValue release() throws RemoteException;
}
```

The interfaces are implemented by `ServerManager` resp. `ClientManager`:

```
public class ServerManager extends UnicastRemoteObject
    implements SManager {
    CManager contentOwner;
    public ServerManager(CManager initial) throws RemoteException {
        contentOwner = initial;
    }
    public synchronized DMLValue request(CManager iWantIt)
        throws RemoteException {
        DMLValue val = contentOwner.release();
        contentOwner = iWantIt;
        return val;
    }
}
```

The `request` method takes the requesting client manager as its argument. It enforces the former owner to release the content of the reference and registers the requesting client as the new content owner. Then it returns the content value of the reference.

The client manager provides the possibility to delete the content of the reference and return it to the server manager.

```
public class ClientManager extends UnicastRemoteObject
    implements CManager {
    Reference ref = null;
    public ClientManager(Reference r) throws RemoteException {
        ref=r;
    }
    final public DMLValue release() throws RemoteException {
        return ref.release();
    }
}
```

The `Reference` class now looks as follows:

```
public class Reference implements DMLConVal {
    DMLValue content = null;
    SManager mgr     = null;  // Homesite-Manager
    CManager cmgr    = null;  // Clientsite-Manager
    // ...
}
```

The object has fields for the homesite manager and a client manager, both are initially empty. As long as no distribution takes place, we have no overhead in the code. The content of a reference is now removable:

```
public DMLValue release() {
    DMLValue t = content;
    content = null;
    return t;
}
```

The `release` method is used by the server manager to fetch and delete the content from a client reference.

The content of a reference can be accessed by:

```
synchronized public DMLValue getContent() throws RemoteException {
    if (content == null) {
        content = mgr.request(cmgr);
    }
    return content;
}
```

Before we can return the content of the reference, we have to make sure that the reference on which the method is invoked owns the content or else we ask the manager to request it.

The managers of a `Reference` are only created if the reference is used in an distributed context. Since `Reference` is a non-remote class, the distribution is performed by serialization, i.e., the `writeObject` method of the `Reference` object is used.

```
private void writeObject(ObjectOutputStream out) throws IOException {
    try {
        // the client manager is not written to the stream
        // since the readObject method will install a new one
        CManager CMGR = null;
        if (cmgr == null) { // i.e., we are on the home site
            CMGR = new ClientManager(this);
        } else { // previously installed by readObject
            CMGR = cmgr;
        }
        cmgr = null;
        // the content will not be transferred immediately
        DMLValue t = content;
        content = null;
        // install and export the server manager
        if (mgr == null) { // i.e., we are at home
            mgr = new ServerManager(CMGR);
        }
        // write Reference to stream and restore the client
        // manager and the content
        out.defaultWriteObject();
        cmgr = CMGR;
        content = t;
    } catch (RemoteException e) {
        System.err.println(e);
    }
}
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    cmgr = new ClientManager(this);
}
```

When the reference is written to the RMI output stream, the managers are installed. The client manager and the content are not written to the stream, since the site that deserializes the reference installs its own client manager. Note that if you re-distribute references the new server manager remains the same.

To make clear that the references have the desired behavior, look at the following. Imagine we have two sites *A* and *B*. On site *A* we create a reference R. At first nothing happens at all —

the reference class is designed such that there is no overhead if the user doesn't use distribution. Then site *A* offers `R` to the world, using

```
val ticket = Connection.offer R
```

Then the ticket is somehow transferred to site *B*, and *B* uses the ticket with

```
val myR = Connection.take ticket
```

At this point in time several things happen. First of all, *B* connects via a TCP/IP connection to *A*, asking for the value with the ticket `ticket`. *A* now serializes `R` and uses the `writeObject` method, i.e., installs a ClientManager, a `ServerManager` and transfers a copy of the `Reference` without the content, but with a reference to the `ServerManager`. Site *B* receives the reference and uses the `readObject` method of `Reference` to deserialize the object. It knows nothing about the content, but receives the reference of the `ServerManager`. If site *B* now wants to do anything with `myR`, the methods of `myR` enforce the connection to the ServerManager and the request of the content. This example is illustrated in Figure 7.4.

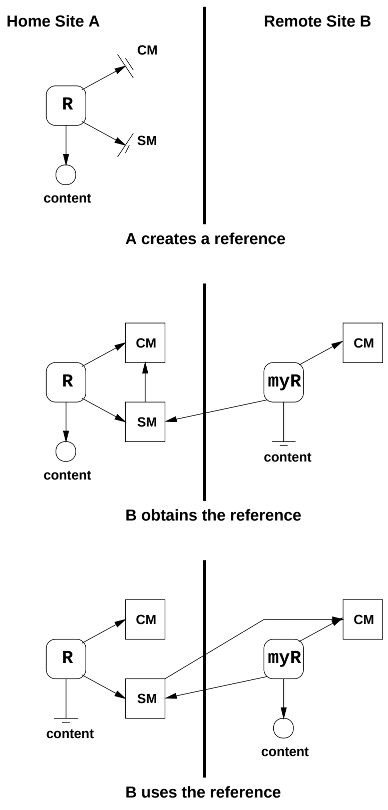


Figure 7.4: *The Mobile Protocol of* `Reference`.

7.5 Reimplementing RMI

After having a look at the sources of RMI, we are faced with the following situation: Java RMI uses specialized subclasses of `MarshalInputStream` resp. `MarshalOutputStream` for the serialization of non-remote objects in the remote environment. The solution is to reimplement these classes and tell the Virtual Machine to use the modified implementation instead. The virtual machine then has to be invoked with the `-Xbootclasspath` command line option as follows:

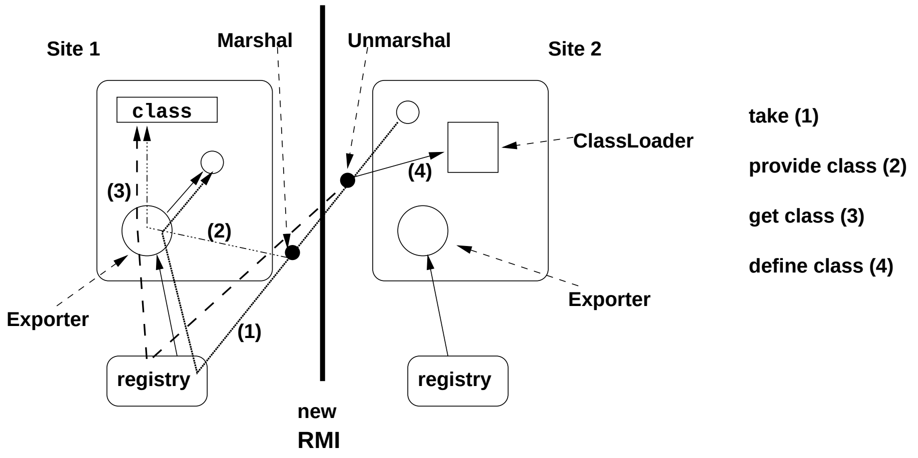


Figure 7.5: *The patched RMI without a web server and URL protocol.*

```
%  java -Xbootclasspath:$PATCH:$RT -classpath $CLASSPATH ...
```

The environment variable `PATCH` contains the path to the patched class files and the variable `RT` points to Java's default runtime classes (usually something like `/opt/jdk/jre/lib/rt.jar`).

The modifications are the following. In the output class we have to make sure the byte code of offered functions is provided. The server will make the code available via the export facility which itself simply looks up the code in the `PickleClassLoader`. So we modify the `annotateClass` method similar to the `annotateClass` of pickling. The unmarshalling has to check whether it has to load the class via net and is modified by overriding the `resolveClass` method.

That is all you have to do. All other parts of the runtime classes are left unchanged. The `Export` interface has to be extended with a method that provides class byte code and another method that has access to the static fields of classes.

Since the classes that we have modified are platform-dependant, the current implementation works with Linux. The source code for other platforms has not been available for me, but the modifications are straight forward and could easily be transferred to other operating systems.

The model of DML's distribution is sketched in Figure 7.5. In contrast to Java RMI's class loading concept, DML doesn't need a web server and the use of a URL protocol.

7.6 Summary

In DML distributed programming is implemented using Java Remote Method Invocation. We have to modify the default behavior of the marshaling classes to achieve a smart solution. The DML system provides distributed programming on a high level, many of the explicit Java constructs are avoided. In this respect, a similar approach is taken in [PZ97], where an add-on for Java RMI is presented.

Because we use RMI, we don't have to care about distributed garbage collection and low level protocols; no failure handling is implemented. The implementation is compact and reuses much of Java's facilities.

There are alternative implementations of RMI [NPH99], but we don't have the source code to adapt it to our input and output streams.

Chapter 8

Related Work

In this chapter we briefly describe projects that share some properties with our work. We sketch what other systems do and give references to these works. As mentioned in Chapter 1, there are already many programming languages that are translated to the Java VM. An always changing and useful list can be found at [Tol99].

8.1 Kawa

Kawa [Bot98] is a compiler for the LISP dialect Scheme to the JVM. It provides a higher-level programming interface with support for scripting and interactive read-eval-print loops. It gives a full integration between Java and Scheme. The author, Per Bothner, describes a possibility for generally treating tail calls in a CPS-like style. Just like our proposal in Section 3.2.2, this slows down all applications but operates on a constant call stack.

8.2 MLj

MLj is meant to be a complete system for compilation of Standard ML to Java byte code. It has a static type system with extensions for classes and objects. MLj provides a Java API interface with additional syntax. MLj performs whole program optimization and gains much speed because of that: Polymorphism is reduced to monomorphism by duplicating methods. Functions are decurried and boxing can be avoided rather efficiently. So far, only simple tail recursion is handled. See also [BKR99].

8.3 Bertelsen

In his Master Thesis Peter Bertelsen describes a new compiler backend for Moscow ML that compiles Standard ML to Java. In contrast to MLj, no API interface is given. There is no tail call handling implemented. Bertelsen performs no optimizations that go beyond MLj. See [Ber98] for details.

8.4 Java related software

For the Java programming language several software projects are concerned with issues we addressed in Chapters 5–7. For one, the serialization routines have been reimplemented to provide faster (and more compact) algorithms, cf. [HP99]. In correspondence, the Java RMI facilities have been improved. Java RMI uses Java serialization so it is useful to enhance both in conjunction. The reimplementation provides a much faster RMI, designed to work on high speed network connections rather than insecure and unstable Internet connections. This work is described in [NPH99].

Java's RMI needs much care to be used by a programmer such as exception handling etc. JavaParty [PZ97] gives transparent remote objects and further improvements to facilitate the usage of distributed programming in Java.

Chapter 9

Benchmarks

In this chapter we present some benchmarks and show how we use them to analyze the performance of the DML system. We compare the execution time of programs with SML/NJ and MLj to figure out the pros and conts of the way we implemented DML. The benchmark programs and the evaluation follows the evalutation of the Oz VM [Sch98].

To what other system should we compare DML? Of course, MLj and SML/NJ are good candidates for comparison. DML is dynamically typed and therefore needs many runtime type tests while SML/NJ and MLj don't; so we can hope to figure out the cost of dynamic typing. While both MLj and DML are interpreted by the Java Virtual Machine, SML/NJ is compiled to native code. DML supports transients and has the possibilty to dynamically read higher order values; the former induces further runtime type tests, and the latter prohibits some optimizations. Neither MLj nor SML/NJ have similar features.

Due to dynamic typing, we expect DML to be slower than MLj or SML/NJ. Further, because DML provides transients, pattern matching and primitive operations of the language are more expensive. If we compare DML to MLj, we can analyze the cost of our dynamic properties and the support for transients. The costs of concurrency and distribution cannot be compared because MLj does not provide similar features. By comparing DML to SML/NJ, we try to determine how much overhead is introduced by using a virtual machine to interpret the code instead of using native code. Again the costs of concurrency and the omitted static type information have to be considered. SML/NJ provides concurrency as a module and we will have a look at the costs of threads. The distributed programming facility of DML have no counterpart in SML/NJ.

Another system we compare DML with is Kawa. Similar to DML, Kawa programs are interpreted by the JVM. Kawa is dynamically typed and in this respect more similar to DML than MLj and SML/NJ. But because the source language differs from DML considering the translation of closures, the benchmark results are not so significant.

9.1 How We Measure

The time is measured with GNU Time v1.7. Let `X` denote the benchmark to be executed. In the case of DML, the execution time of

```
dml X
```

is measured. For MLj, the execution of the command

```
java -classpath X.zip X
```

is used. For both DML and MLj we start the JVM with identical runtime options. SML/NJ programs are started by

```
sml @SMLload=X
```

The execution of Java byte code can be accellerated by using so called Just In Time (JIT) compilers. For Linux, there are several JIT compilers available; we only consider 'tya1.5' [Kle99] and the default 'sunwjit' that comes with Blackdown's JDK [dSR+99]. The usage of JIT compilers is only for the profit of DML and MLj with respect to SML/NJ as SML/NJ has no such way of speeding up.

First we call each benchmark program without arguments and measure the time it takes to execute the dummy call. By doing this, we can get a good estimation of how much time is consumed before the actual benchmark is started; the startup time will be subtracted from the overall runtime. If we use a JIT compiler we have to take into account the time that is needed to JIT-compile the class code. In practice, this time span was infinitesimal with respect to the total time of the execution, so that we can simply neglect this aspect. Each benchmark program then is executed 25 times, the time listed in the result table is the arithmetic average of the overall time minus the startup time. The standard deviation was also computed in order to show that the results of the benchmarks don't vary too much. We do not show the values of the deviation here, we usually gained results where we had $\sigma \leq 5\%$.

$$\mu = (\frac{1}{25} \cdot \sum_{i=0}^{25} \texttt{time cmd}_i) - \texttt{startup}$$

The standard deviation is computed as follows:

$$\sigma = \sqrt{\frac{1}{25} \cdot \sum_{i=0}^{25} (\texttt{time cmd}_i - \mu)^2}$$

Since the running time measured in seconds is not so interesting as the ratio we give the results in the form 'SML/NJ : MLj : DML' resp. 'MLj : DML' normalized with respect to SML/NJ resp. MLj, i.e.,

$$1 : \mu_{\texttt{MLj}}/\mu_{\texttt{SML/NJ}} : \mu_{\texttt{DML}}/\mu_{\texttt{SML/NJ}}$$

resp.

$$1 : \mu_{\texttt{DML}}/\mu_{\texttt{MLj}}$$

The benchmarks on the JVM are run with 'green' threads. For Linux running on a single processor machine the choice of green threads results in a better running time; native threads are heavy weight. Note that Linux limits the number of threads per user to 256 by default; each Java native thread counts as one system thread. A normal user therefore is not able to create more than the system limit; some programs will not be able to run as expected. For green threads there is no such limit.

9.2 The Test Platform

The benchmarks programs are executed by using the following hardware and software:

Architecture	x86
Processor	Intel Celeron (Mendocino), 466 MHz
Main Memory	128 MB
OS	(Red Hat 6.1) Linux 2.2.12
Java	Blackdown JDK1.2.2-RC2
JIT	TYA1.5
JIT	sunwjit (included in JDK1.2.2-RC2)
MLj	Persimmon IT. MLj 0.2
Kawa	Kawa 1.6.62
SML	SMLofNJ Version 110.0.3
Time	GNU Time v1.7

9.3 Benchmark Programs

We perform and analyze the following benchmarks:

- Fib/Tak — The Fibonacci numbers are computed in order to measure how good function application and integer arithmetic works. The Takeushi function serves the same purpose with a different ratio of function calls and arithmetics.
- Deriv — This program computes symbolic derivations to show the costs of using constructors and pattern matching.
- NRev — By using naive reverse of lists, we want to find out the costs of transients and allocation of lists on the heap (see below)
- Concfib — Concfib measures the cost of thread creation and synchronization with transients in DML resp. channels in CML (the concurrent extension to SML/NJ).

The code of the benchmarks can be found in Appendix B.

9.4 Analysis

Fibonacci and Takeushi

The Fibonacci benchmark computes `fib 31` and thereby induces circa 2.1 million recursive calls. In each call of the function, one comparison on integers, one addition and one subtraction has to be performed. The Takeushi benchmarks computes `tak(24, 16, 8)` which causes about 2.5 million function calls. In this case, less arithmetic functions have to be performed.

The benchmark results for Fibonacci are the following:

$$1 \quad : 4 \quad : 40 \quad (\textit{no JIT})$$

$$1 \quad : 1.5 \quad : 25.5 \quad (\texttt{tya})$$

$$1 \quad : 0.6 \quad : 26.5 \quad (\texttt{sunwjit})$$

There are two things one notices at once: MLj code that is accelerated by a JIT compiler beats SML/NJ and DML falls back by the factor of about 10-12. One can image that the native code of SML/NJ is way faster than the interpreted byte code of Java, but this contradicts the fact that MLj is as fast as SML/NJ. So this can't be the reason why the DML code is so slow. What else could cause that breakdown? The analysis of the runtime behavior of the DML Fibonacci program shows that about 90% of the time is spent for arithmetics, particularly the creation of integer wrapper objects. To confirm that the missing type information (or better: statically unused type information) causes the high costs we try the following: we add and use type information 'by hand'. If we then use the enhanced Fibonacci we indeed gain about 95% of run time and are almost as fast as MLj. The benchmark ratio then is

1 : 0.6 : 0.7 (`sunwjit`)

The conclusion is that object creation makes the computation expensive; the high costs emerge from wrapping and unwrapping integers.

The Takeushi benchmark confirm these assumptions. As there are less arithmetic operations to perform the ratio looks slightly better for DML:

1 : 3 : 22 (*no JIT*)

1 : 1.2 : 15 (`tya`)

1 : 0.4 : 15 (`sunwjit`)

If static type information could be used, we could again be as fast as MLj.

Deriv

Computing symbolic derivations does not use arithmetic. So we can hope to have nicer results with that benchmark. Indeed the ratio is

1 : 2.5 : 6.5 (*no JIT*)

1 : 1.7 : 4.3 (`tya`)

1 : 3.4 : 4.3 (`sunwjit`)

Considering the case of `sunwjit`, we are almost as fast as MLj, the ratio MLj : DML is 1 : 1.2. The loss of efficiency can be put down to the presence of transients and DML's dynamic properties that prohibit the optimizations MLj may perform.

NRev

Naive reverse depends mainly on the `append` function; languages that support transients can write `append` in a tail recursive manner and should have some advantage over the others. So we have to compare the following functions to one another:

```
fun append' (nil, ys, p) = bind (p, ys)
  | append' (x::xr, ys, p) =
    let
        val p' = lvar ()
        val x' = x::p'
    in
        bind (p, x');
        append' (xr, ys, p')
    end
```

versus

```
fun append (nil, ys) = ys
  | append (x::xs, ys) = x :: (append (xs,ys))
```

Using `append'` results in

```
1  : 15    : 13.000  (no JIT)

1  : 11    : 14.000  (tya)

1  : 10    : 24.000  (sunwjit)
```

The disastrous result comes from the creation of logic variables. These objects are very expensive to create because they inherit from `UnicastRemoteObject` (for distribution). A way of speeding up is therefore to provide different transients for non-distributed programming. Indeed this speeds up the DML execution

```
1  : 13.5  : 91      (no JIT)

1  : 11    : 86      (tya)

1  : 10    : 98      (sunwjit)
```

The `append'` function creates twice as much objects as the corresponding function used with MLj. But since in Java object creation is more expensive than method invocation, we still are much slower than SML/NJ or MLj though the `append` can only be written with simple recursion. For MLj this has the consequence that the limit on the length of the lists is tighter than that of DML, because the stack depth is greater. As SML/NJ doesn't use a call stack there are no consequences for them.

If we compare the run times using the `append` function that is not tail recursive, the following ratio is achieved:

```
1  : 13.5  : 29      (no JIT)
```

The ratio is similar if we use JIT compilers.

As a consequence we propose the implementation of two transient variants — one that can supports distribution and one that can only be used on one site. Further it seems that the cost for object creation exceeds the benefits of saving a method invocation by far.

Concfib

Concfib repeatedly computes the value of `fib 12`. Each recursive invocation creates two fresh threads that compute the sub value. Results are given back via transients resp. channels.

MLj has no special feature for concurrency so we only compare DML to SML/NJ. The SML/NJ version of `Concfib` uses the CML facilities. The communication between threads is implemented by the use of *channels*. The Java-Linux Porting Team considers 'the usage of hundreds or even thousands of threads is bad design'. This point of view is reflected in the costs for threads as the ratio shows

```
1  : 124         (no JIT, green threads)
```

1	: 116	(`tya`, *green threads*)
1	: 105	(`sunwjit`, *green threads*)

For the native threads implementation we have

1	: 273	(*no JIT*)
1	: 281	(`tya`)
1	: 275	(`sunwjit`)

In this benchmark, we have used the cheaper versions of transients. By using the expensive versions, we obtain a result that is worse (due to the costs of creating transients that can be used distributedly).

Code Size

Besides the run time of the programs the size of the compiled code is worth a look. The size is given in bytes.

	SML/NJ	**MLj**	**DML**
Fibonacci	385960	5941	2866
Takeushi	384936	6004	2922
Derivate	388008	8448	6239
NRev	385960	6426	3709
Concfib	647432	–	4638

We do not consider the runtime systems here, i.e., for SML/NJ the size of the Linux heap image is listed; the heap image was produced by `exportFn` and can be executed via `run.x86-linux`. The Java runtime is used to execute MLj's output. MLj stores the generated class code in zip files. DML compiles the programs into pickles and additionally has runtime libraries. Note that the pickles are compressed using gzip output streams and are therefore somewhat reduced in size.

As one can see, the size of the files interpreted by the JVM are more compact by up to two orders of magnitude. This aspect becomes important if we consider distributed programming — the more data transferred, the slower the transmission and the slower the execution of distributed programs.

9.5 Dynamic Contest

Kawa is the only competitor that translates a dynamically typed language to the JVM. If we want to compare our implementation with Kawa, we have to rewrite the benchmarks in the Scheme language. We measured for all of the benchmarks that DML is faster than Kawa by a factor of about 3–4. This factor is achieved with each benchmark no matter which JIT is used. So we can conclude that we have the fastest implementation of a functional dynamically typed language executed on the JVM.

9.6 Summary

As expected, DML is usually slower than SML/NJ and MLj. The main performance loss comes from the boxing and unboxing of literals. If we avoid boxing (as done in the manually optimized Fibonacci benchmark), we achieve significantly better results. If we take advantage of type information, the Fibonacci benchmark runs even faster on the JVM than the native compilation of SML/NJ.

The transients of DML introduce only little overhead for pattern matching and primitive operations if they are not used. In the case of NRev, the cost for creating transients exceeds the benefits of constant call-stack size of tail recursive `append`. Transients that can be used in a distributed environment are very expensive to create; it is recommended to provide a further variant of transients that can only be used locally. Threads in DML cannot be used in the way Mozart or CML propagate the use of threads. It is not possible to create zillions of threads on a JVM because Java threads are valuable resources that are not suitable for short live cycles.

The component system and distribution facilities of DML prohibit whole program optimization. MLj can perform such optimizations at the expense of modularity and separate compilation. According to Benton, Kennedy, and Russell [BKR99] this is the most important step to achieve a reasonable performance.

DML is faster than Kawa in any benchmark we have tested. This is the result of Scheme's environment model that provides less possibilities for optimization.

The influence of JIT compilers for the JVM depends on the program executed; there are programs that cannot be run with a special JIT compiler, some run faster, some are slower. In general, the JIT compilers achieved better performance enhancements for MLj programs than for the DML counterparts.

Chapter 10

Conclusion

We have built a compiler and a runtime environment for translating a dynamically typed high-level language to the Java Virtual Machine.

The compiler produces pickle files that contain evaluated components. Pickle files can be executed on any Java capable platform in combination with the DML runtime that provides the value representation and the primitives of the language. The goal of a simple and secure implementation of DML for the JVM has been achieved.

The naïve implementation of the DML system is straightforward and compact. We take advantage of many Java features. DML's concurrency is based on Java threads, the pickling mechanism reuses Java Object Serialization, and the distribution model adopts the Java RMI infrastructure. The data-flow synchronization of threads by transients is implemented using Java synchronization primitives. Exceptions can directly use the corresponding JVM exceptions.

The implementation is enhanced by refining the representation of tuples and constructed values. Further, function closures have special support for tuple arguments. One of the problems of Java as a target platform for a dynamically typed source language is that the typed byte-code enforces the usage of expensive wrappers. The DML compiler omits boxing and unboxing of literals whenever possible. The representation of constant values is built at compilation time to avoid the creation of objects at runtime. Mutually recursive functions are merged into a single function to enable tail call optimization. The compiler performs code inlining for primitive functions.

It is interesting to look at the optimizations MLj performs on programs. MLj can avoid boxing and unboxing completely due to whole program optimization. Polymorphic functions are replaced by separate versions for each type instance for which they are used. The problem with this approach is that the code size can grow exponentially. Because MLj operates on the complete source code, it can limit the blowup by only generating the monomorphic functions that are actually used. In contrast, DML's component system prohibits that approach.

General tail-call optimization cannot be implemented reasonably without support of the JVM. Solutions that use CPS-like approaches are too slow. Without special treatment, tail recursion requires linear call-stack size and leads to stack overflow exceptions of the JVM. The current versions of MLj and Kawa have no special treatment for mutually tail recursive functions. This limitation may cease to exist in future versions of the JVM. The Java Virtual Machine specification limits the amount of code per method to 65535 bytes. This limit is no problem for most programs, yet the generated code of the DML parser exceeds that size and cannot be compiled. As a consequence, the compiler cannot be bootstrapped.

As the benchmark results show, future versions of DML should use static type information to further reduce boxing and runtime type checks. The current RMI model and implementation recommends the support for variants of transients: One version that is capable of being distributed and another one that is faster but can only be used locally. A good implementation of transients takes advantage of VM support. E.g., the Oz VM provides transient support at the level of the bytecode interpreter and can replace transients by their value as soon as they are bound. The DML implementation has to keep the wrapper objects instead.

Appendix A

Compilation Scheme

A.1 Compilation of Expressions

Section 2.6 describes how most expressions are compiled for the JVM. The compilation of references, constructed values, variables, tuples and vectors introduces no new concepts. For the sake of completeness they are described here.

A.1.1 References

A reference is a special constructor, but the contents of its constructed values are mutable. Constructors are equal if pointer comparison succeeds, so there must be only one reference constructor, which can be found in a static field of the runtime.

expCode (`RefExp`)
`getstatic <BRef>`

Applications of Constructors

When applied to a value, a constructor results in a constructed value. The **`ConVal`** class has a (Java) constructor which takes a constructor and content value as parameter.

expCode (`ConAppExp`(*constamp*, *stamp*))
`new` **`ConVal`**
`dup`
stampCode (*constamp*)
stampCode (*stamp*)
`invokespecial` **`ConVal`**`/<init> (IValue, IValue)→void`

Applications of References

The **`Reference`** class has a (Java) constructor with an **`IValue`** as its argument. The constructor of references is always `ref`; we don't have to explicitly store it.

expCode (`RefAppExp`(*stamp*))

`new` **`Reference`**
`dup`
stampCode (*stamp*)
`invokespecial` **`Reference`**`/<init> (IValue)`$\rightarrow$`void`

A.1.2 Variables

Variables are loaded from either a JVM register or a field depending on whether it occurs bound or free in the current function:

expCode(`VarExp`(stamp))
stampCode (*stamp*)

A.1.3 Tuples and Vectors

In DML, Tuples are records with labels [1,2,…,*n*]. Tuples are constructed just like records, but have no label array.

Vectors are treated similar to tuples. They supply some special functions and are constructed in the same way:

expCode(`TupExp`[$stamp_0$,$stamp_1$,…,$stamp_n$])
expCode(`VecExp`[$stamp_0$,$stamp_1$,…,$stamp_n$])

`new` *classname*
`dup`
`ldc` $n+1$
`anewarray` **`IValue`**
 	
`dup`
`ldc` 0
stampCode ($stamp_0$)
`aastore`
⋮
`dup`
`ldc` n
stampCode ($stamp_n$)
`aastore`
 	
`invokespecial` *classname*`/<init> (IValue[])`$\rightarrow$`void`

classname is either **`Tuple`** or **`Vector`**.

A.2 Recursive Declarations

General information about the compilation of recursive declarations can be found in Section 2.7.2. For the sake of completeness, a description about creating and filling closures follows here.

The empty instances of Functions, Tuples, Vectors, Constructor Applications and Reference Applications are constructed as follows. Regardless of the body or content of the expression, a new instance of the corresponding class is created:

emptyClosure (`FunExp`(*funstamp*,*exp*))
emptyClosure (`TupExp`(*content*))
emptyClosure (`VecExp`(*content*))
emptyClosure (`ConAppExp`(*content*))
emptyClosure (`RefAppExp`(*content*))
`new` *classname*
`dup`
`invokespecial` *classname* `()`→`void`

where *classname* is the classname of the *funstamp*, or **`Tuple`**, **`Vector`**, **`ConVal`** or **`Reference`**, in case the expression is `TupExp`, `VecExp`, `ConAppExp` or `RefAppExp`.

`Record`s are constructed just like **`Tuple`**s, but get an arity as parameter:

emptyClosure (`RecExp`(*content*))
`new` *classname*
`dup`
`getstatic` *curClassFile*`/`*arity* **`IValue`**`[]`
`invokespecial` **`Record`** `(`**`IValue`**`[])`→`void`

`VarExp`, `ConExp` and `LitExp` don't have a 'closure', so they can be directly evaluated.

emptyClosure (*exp*)
expCode (*exp*)

Some expressions, such as `AppExp`, are not admissible in recursive declarations. However, we don't need to check admissibility here because illegal code shouldn't pass the frontend.

The empty closures are filled as follows:

- **Functions**

  To fill the empty closure of a function, we load it on top of the stack, then store the values of the free variables into the closure:

fillClosure (*FunExp*(*funstamp*, *exp*))
`dup`
stampCode (fv_0)
`putfield` *classname*`/`fv_0 `IValue`
⋮
`dup`
stampCode (fv_{i-1})
`putfield` *classname*`/`fv_{i-1} `IValue`
stampCode (fv_i)
`putfield` *classname*`/`fv_i `IValue`

  where fv_0 to fv_i are the free variables of *funstamp* and *classname* is the name of the class corresponding to *funstamp*.

- **Tuples, Records, Vectors**

  Tuples, records and vectors are filled by creating an array of values and storing it into *classname*`/vals` where *classname* is **`Tuple`**, **`Record`** or **`Vector`**.

fillClosure (`TupExp`[$stamp_0$,$stamp_1$,…,$stamp_i$])
fillClosure (`RecExp`[(lab_0,$stamp_0$),(lab_1,$stamp_1$), …,(lab_i,$stamp_i$)])
fillClosure (`VecExp`[$stamp_0$,$stamp_1$,…,$stamp_i$])

```
ldc i+1
anewarray IValue

dup
ldc 0
stampCode (stamp_0)
aastore
⋮
dup
ldc i
stampCode (stamp_i)
aastore

putfield classname/vals IValue[]
```

- **Constructor Applications** and **ref Applications**

  `ConVal`s and **`Reference`**s have one single content field. This field is now assigned.

fillClosure (`ConAppExp`(*constamp*,*stamp*))
fillClosure (`RefAppExp`(*stamp*))

```
stampCode (stamp)
putfield classname/content IValue
```

Appendix B

The Code of the Benchmarks

This Chapter lists the code for the benchmarks performed in Chapter 9. We present the code for SML/NJ, MLj, and DML where is it identical, the differences are listed seperately.

B.1 Differences

SML/NJ wants the programmer to do the following to make a program 'executable':

```
fun foo n = ...
fun callfoo (_, [x]) =
    let
        val arg = Int.fromString x;
    in
        case arg of
            NONE => 1
          | SOME n => (foo n; 0)
    end
  | callfib _ = 2
val _ = SMLofNJ.exportFn ("Foo", callfoo)
```

To fetch the command line arguments, MLj programs have to look as follows:

```
structure Foo :
    sig
        val main : string option array option -> unit
    end
=
struct
    fun do_foo = ...
    fun main (env: string option array option) =
        case env of
            NONE => ()
          | SOME env' =>
                if Array.length env' = 0 then ()
                else
                    case Array.sub(env', 0) of
                        NONE => ()
                      | SOME str =>
                            case Int.fromString str of
                                NONE => ()
                              | SOME n => (do_foo n;  ())
```

```
end
```

In DML we have the following situation

```
fun foo n = ...
fun main [x] =
    let
        val n = valOf(fromString x)
    in
        foo n
    end
  | main _ = ()
```

B.2 Common Parts

Fibonacci

```
fun fib n =
       if (1 < n)
           then fib(n-2) + fib(n-1)
       else 1
```

Takeushi

```
fun tak(x,y,z) = if y<x
                        then tak(tak(x-1,y,z),
                                 tak( y-1,z,x),
                                 tak( z-1,x,y))
                    else z
```

Derivations

```
datatype expr = Plus  of expr * expr
     | Minus of expr * expr
     | Var of int
     | Const of int
     | Times of expr * expr
     | Div of expr * expr
     | Exp of expr * int
     | Uminus of expr
     | Log of expr;
   fun dotimes(0,p) = 0
     |  dotimes(n,p) = (p(); dotimes(n-1,p))
   fun deriv(Var(u),x)     = if u=x then Const(1) else Const(0)
     | deriv(Const(u),x)   = Const(0)
     | deriv(Plus(u,v),x)  = Plus(deriv(u,x),deriv(v,x))
     | deriv(Minus(u,v),x) = Minus(deriv(u,x),deriv(v,x))
     | deriv(Times(u,v),x) = Plus(Times(deriv(u,x),v),Times(u,deriv(v,x)))
     | deriv(Div(u,v),x)   = Div(Minus(Times(deriv(u,x),v),
                                       Times(u,deriv(v,x))),
                                 Exp(v,2))
     | deriv(Exp(u,n),x)   = Times(Times(deriv(u,x),Const(n)),Exp(u,n-1))
     | deriv(Uminus(u),x)  = Uminus(deriv(u,x))
```

```
    | deriv(Log(u),x)     = Div(deriv(u,x),u)
  fun nthderiv(0,exp,x) = exp
    | nthderiv(n,exp,x) = nthderiv(n-1,deriv(exp,x),x)
  fun goderiv n =
      dotimes(n, fn () => nthderiv(6,Exp(Div(Const(1),Var(1)),3),1));
```

Naive Reverse

```
fun append (nil, ys) = ys
    | append (x::xs, ys) = x :: (append (xs,ys))
  fun nrev(nil)   = nil
    | nrev(a::bs) = append(nrev(bs), a::nil);

  fun append’ (nil, ys, p) = fulfill (p, ys)
    | append’ (x::xr, ys, p) =
      let
          val p’ = lvar ()
          val x’ = x::p’
      in
          bind (p, x’);
          append’ (xr, ys, p’)
      end
  fun append (xs, ys) =
      let
          val p = lvar ()
      in
          append’ (xs, ys, p); p
      end
  fun rev nil = nil
    | rev (x::xr) = append (rev xr, x :: nil)
```

Concfib

Concfib is implemented using CML; threads communicate through channels:

```
  open CML
  fun fib 0 = 1
    | fib 1 = 1
    | fib n =
      let
          fun fib’ n =
              let
                  val res = channel ()
              in
                  spawn (fn () => send (res, fib n));
                  recv res
              end
      in
          fib’ (n-1) + fib’ (n-2)
      end
  fun loop 1 = (fib 12; ())
    | loop n = (fib 12; loop (n-1))
  fun loopit (_, [x]) =
      let
          val b = Int.fromString x
      in
```

```
        case b of
            NONE => 1
          | SOME i =>
                (RunCML.doit ((fn () => loop i), NONE);
                 print "Done.\n";
                 0)
    end
  | loopit _ = 2
```

In DML, Concfib uses transients. The threads communicate by binding logic variables:

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n =
    let
        fun fib' n =
            let
                val res = lvar ()
            in
                spawn (fn () => bind (res, fib n));
                future res
            end
    in
        fib' (n - 1) + fib' (n - 2)
    end
fun loop 1 = fib 12
  | loop n = (fib 12 ; (loop (n - 1)))
```

Bibliography

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, UK, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[Ber98] Peter Bertelsen. Compiling SML to Java Bytecode. Master's thesis, Department of Information Technology, Technical University of Denmark, Copenhagen, Denmark, January 1998. Available at `http://www.dina.kvl.dk/~pmb`.

[BKR99] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 129–140, Baltimore, MD, USA, June 1999. ACM Press.

[Bot98] Per Bothner. Kawa: Compiling Scheme to Java. Lisp Users Conference ("Lisp in the Mainstream" / "40th Anniversary of Lisp") in Berkeley, CA., November 1998. Available at `http://sourceware.cygnus.com/kawa/papers`.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[dSR$^+$99] Kars de Jong, Michael Sinz, Paul Michael Reilly, Cees de Groot, Scott Hutinger, Tod Matola, Juergen Kreileder, Karl Asha, and Stephen Wynne. Blackdown JDK1.2, December 1999. Available at `http://www.blackdown.org/java-linux`.

[G$^+$99] Vincent Gay-Para et al. The KOPI Project, 1999. Available at `http://www.dms.at`.

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

[GM96] James Gosling and Henry McGilton. The Java Language Environment, May 1996. Available at `http://java.sun.com/docs/white`.

[Gon98] Li Gong. *Java Security Architecture (JDK1.2)*. Sun Microsystems, Inc., 1.0 edition, October 1998. Available at `http://java.sun.com/products/jdk/1.2/docs`.

[GYT96a] James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface. Vol. 1: Core packages*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[GYT96b] James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface. Vol 2: Window Toolkit and Applets*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[HP99] Bernhard Haumacher and Michael Philippsen. More Efficient Object Serialization. In *Parallel and Distributed Processing, LNCS 1586*, pages 718–732, San Juan, Puerto Rico, April 1999. International Workshop on Java for Parallel and Distributed Computing.

[HVBS98] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming Languages for Distributed Applications. *New Generation Computing*, 16(3):223–261, 1998.

[HVS97] Seif Haridi, Peter Van Roy, and Gert Smolka. An Overview of the Design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[Kle99] Albrecht Kleine. TYA 1.5 JIT compiler, September 1999. Available at `ftp://gonzalez.cyberus.ca/pub/Linux/java`.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[Mey97] Jon Meyer. Jasmin, March 1997. Available at `http://www.cat.nyu.edu/meyer/jasmin`.

[Moz99] Mozart Consortium. The Mozart programming system, 1999. Available at `http://www.mozart-oz.org`.

[MSS98] Michael Mehl, Christian Schulte, and Gert Smolka. Futures and By-Need Synchronisation for Oz, Mai 1998. Draft. Available at `http://www.ps.uni-sb.de/~smolka/drafts/oz-futures.ps`.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, USA, 1997.

[NPH99] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A More Efficient RMI. In *ACM 1999 Java Grande Conference*, pages 152–159, Sir Francis Drake Hotel, San Francisco, California, June 12–14 1999.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.

[Oz95] The dfki oz programming system, 1995. Available at `http://www.ps.uni-sb.de/oz1`.

[Oz97] The dfki oz programming system (version 2), 1997. Available at `http://www.ps.uni-sb.de/oz2`.

[Phi98] Michael Philippsen. Is Java ready for computational science? In *Euro-PDS'98*, pages 299–304, Vienna, Austria, July 1998. 2nd European Parallel and Distributed Systems Conference.

[PJK98] Michael Philippsen, Matthias Jacob, and Martin Karrenbach. Fallstudie: Parallele Realisierung geophysikalischer Basisalgorithmen in Java. In *Informatik – Forschung und Entwicklung*, pages 72–78. Universität Karlsruhe, Germany, 1998.

[PZ97] Michael Philippsen and Matthias Zenger. JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997. Available at `http://wwwipd.ira.uka.de/~phlipp/partyd.ps.gz`.

[Sch98] Ralf Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. Dissertation, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, December 1998.

[Smo98a] Gert Smolka. Concurrent Constraint Programming Based on Functional Programming. In Chris Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 1381, pages 1–11, Lisbon, Portugal, 1998. Springer-Verlag.

[Smo98b] Gert Smolka. Concurrent Constraint Programming based on Functional Programming, April 1998. Available at `http://www.ps.uni-sb.de/~smolka/drafts/etaps98.ps`.

[Sun98] Sun Microsystems, Inc. *Java Object Serialization Specification*, November 1998. Available at `http://java.sun.com/products/jdk/1.2/docs/guide/serialization`.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, November 90.

[Tol99] Robert Tolksdorf. Programming Languages for the Java Virtual Machine, 1999. Available at `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`.

[VHB$^+$97] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau – Theorie, Konstruktion, Generierung*. Springer, Berlin, Germany, 1992.