Compiling SML to Java Bytecode

Peter Bertelsen, C917023

Master's Thesis

January 31, 1998

Department of Information Technology
Technical University of Denmark

Contents

Preface

This report concludes my M.Sc. project at the Department of Information Technology, Technical University of Denmark (IT/DTU). The work has been carried out at the Department of Mathematics and Physics, Royal Veterinary and Agricultural University of Denmark, in the period from February 1997 through January 1998, and under the supervision of associate professor Peter Sestoft. Professor Christian Gram and associate professor Hans Bruun, IT/DTU, have been co-supervisors during the project.

The report describes the development and implementation of a compiler for Standard ML[5, 6] (SML), based on the Moscow ML compiler written by Sergei Romanenko and Peter Sestoft[9, 10]. The compiler back-end of Moscow ML, generating (modified) Caml Light bytecode, has been replaced with a new back-end generating Java bytecode. The new compiler supports batch compilation only (no interactive top-level is provided). The new back-end, as well as the Moscow ML compiler, is implemented in SML.

The reader is assumed to be familiar with functional programming in SML, and with imperative, class-based object-oriented programming. The syntax of the Java programming language[2] is also assumed to be familiar, as a set of run-time support classes are implemented in Java, and small fragments of Java code are used in examples describing the design of the new back-end.

Detailed knowledge of the Java Virtual Machine[4] (JVM) and Java bytecode semantics[1] is not necessary to understand the design of the new back-end. However, acquaintance with high-level bytecode languages will probably help understanding the details of Java bytecode generation from the intermediate Lambda language of Moscow ML.

The report is organized as follows:

- Chapter 1 provides a brief introduction to SML and Java bytecode, and describes the existing Moscow ML compiler on which the new compiler is based.
- Chapter 2 describes the design of a translation from SML to Java, including the design of a set of run-time support classes for representation of SML values in Java.
- In Chapter 3, the 'SML-JVM toolkit' is presented. This toolkit consists of a set of modules for representing Java bytecode in SML, and for generating Java class files from abstract class declarations. Understanding the internals of the SML-JVM toolkit is not necessary to understand the implementation of the new compiler back-end. Readers who are already familiar with the Java bytecode instructions set, or who are not particularly interested in details of generating Java bytecode instructions and class files, may wish to go directly to Chapter 4.
- Chapter 4 describes the new back-end, including the general, unoptimized translation from intermediate Lambda code to Java bytecode.

- In Chapter 5, the optimizations that have been implemented to produce faster and/or smaller Java bytecode programs are presented.
- Chapter 6 describes how the compiler has been tested to verify that the expected bytecode is produced, and to evaluate the performance of the produced bytecode.
- In Chapter 7, various ideas for improving the compiler are discussed, and specific suggestions for further optimization of the generated bytecode are listed.
- Chapter 8 concludes.

The following appendices are provided:

- Appendix A presents a brief guide on how to download, install, and use the new compiler.
- Appendix B lists the assumptions on Java bytecode semantics that underly the implementation of the new back-end. The reason for this is that *The Java Virtual Machine Specification*[4] cannot be considered a complete specification of the JVM since many details in the semantics of Java bytecode programs are left unspecified or treated ambiguously.
- Appendix C shows an example of compilation of an SML program. The intermediate Lambda code generated by the front-end is listed, followed by the target Java class file generated by the new compiler back-end.
- Appendix D, E, F, and G list the source files for the run-time support classes, for the SML-JVM toolkit, for the new compiler back-end, and for the SML programs that have been used in testing the new compiler.

Peter Bertelsen
January 1998

Chapter 1

Introduction

1.1 Standard ML

Standard ML is a (mostly) functional programming language featuring:

- higher-order functions,
- strict evaluation,
- automatic garbage collection,
- static type checking and automatic type inference,
- polymorphic types,
- support for user-defined algebraic types,
- an advanced module system with parameterized modules[1],
- imperative features such as looping and mutable store (`ref` cells and arrays).

The language and its semantics is fully defined in [5], and has been revised in [6].

1.2 Java bytecode

Java is a class-based object-oriented programming language defined by Sun Microsystems[2]. A Java program declares a set of Java classes, which may be compiled into class files. A Java class file contains field and method declarations for the class plus bytecode instructions for each implemented method of the class.

A Java class file may be loaded and executed by a Java Virtual Machine[4] (JVM). When invoked on a class file, the JVM automatically invokes the `main` method of that class, which may in turn invoke other methods. Referenced classes are loaded dynamically (at run-time).

The Java bytecode instruction set includes instructions for:

- manipulating values of simple types (`int`, `double`, ...),
- manipulating the operand stack,
- accessing local variables, class variables, instance variables, and array components,
- creating class instances and arrays,
- invoking class methods and instance methods,
- branching (conditional and unconditional),

[1]The new compiler only supports 'flat' structures, as does Moscow ML.

- checking object types (at run-time),
- throwing exceptions.

The JVM maintains a frame stack, where each frame corresponds to a method activation record, comprising a local operand stack and local variable environment.

Java and the JVM also supports multi-threading and synchronization, but that is not considered in this project.

1.3 Moscow ML

Moscow ML is a compiler for SML, written in SML. It has been developed jointly by Sergei Romanenko, Russian Academy of Science in Moscow, and Peter Sestoft. The compiler back-end is based on the Caml Light system (developed at INRIA, France), whereas the compiler front-end owes a great deal to the ML Kit system (developed at DIKU, Copenhagen). The following description relates to Moscow ML version 1.42, on which the new compiler is based.

Moscow ML supports the core language of SML[5, 6] and a simple module system, described in the following section. Moscow ML also provides support for most of the required modules in the (forthcoming) SML Basis Library[3].

The Moscow ML distribution includes a batch compiler, as well as an interactive top-level. Only the batch compiler is considered in this report. It compiles source programs to (modified) Caml Light bytecode, which may be executed using a modified version of the Caml Light run-time system (also provided in the distribution).

1.3.1 Module System

The Moscow ML module system supports only a subset of the SML module system; functors and nested structures are not supported.

A Moscow ML structure `A` must be defined in the file `A.sml`. In case a file `A.sig` is present, it is expected to contain the signature for structure `A`. Compilation of `A.sig` produces the file `A.ui` containing the compiled signature for `A`[2]. Compilation of `A.sml` produces the file `A.uo` containing the bytecode for `A`. A number of `.uo` files may be linked to form an executable.

In order to support type-safe separate compilation, the compiled signature file `A.ui` includes a 128-bit stamp based on the signature of structure A, plus the stamps of all structures that A depends upon. For example, compilation of a file `A.sml`, containing the declaration

```
fun f x = B.g(x)
```

and a file `B.sml`, containing the declaration

```
fun g () = 13
```

will yield the files `A.ui`, `A.uo`, `B.ui`, and `B.uo`. `A.ui` contains the compiled signature for `A` and its signature stamp, plus a copy of the signature stamp for `B`, whereas `B.ui` contains the compiled signature for `B` and its signature stamp.

[2]In case there is no file `A.sig`, the inferred signature for structure `A` will be written to `A.ui` during compilation of `A.sml`.

When structure `A` is linked, the files `A.ui` and `A.uo` are loaded. Based on the reference to module `B` in `A.ui`, the files `B.ui` and `B.uo` are then loaded, and it is verified that the copy of `B`'s signature stamp contained in `A.ui` is identical to the actual signature stamp for `B`, as defined in `B.ui`.

1.3.2 Structure of the Compiler

The phases of the compiler are:

- lexical analysis and parsing,
- resolution of infix operators,
- elaboration of top-level declarations (i.e., type checking),
- resolution of overloaded constants and operators,
- translation to intermediate Lambda code,
- lifting of Lambda code (renumbering variables and lifting all functions to top-level),
- generation of target bytecode.

The function `Compiler.compileAndEmit` manages the compilation of a structure. It takes as argument a list of parsed top-level declarations, and invokes functions in the compiler front-end on each top-level declaration (in textual order) to produce intermediate Lambda code.

The resulting list of Lambda expressions is then passed to the code generator (function `Back.compileLambda`), which lifts the Lambda code and generates target bytecode for the top-level declaration in a 'backwards' traversal of the Lambda expression list. The generated bytecode is written to the target file (`.uo`), and compilation proceeds with the next top-level declaration.

Generating target bytecode 'backwards' from the intermediate Lambda code is convenient for implementing a number of local optimizations by pattern mathing on the 'continuation' of each Lambda expression. For example, it is possible to avoid jumps to jumps, remove unreachable code following an unconditional branch, etc.

1.3.3 Intermediate Lambda Language

The intermediate language produced by the Moscow ML compiler front-end is an extended version of the untyped λ-calculus. The Lambda clauses are represented as values of the data type `Lambda`, declared in the `Lambda` module of the compiler:

```
datatype Lambda =
    Lvar of int
  | Lconst of StructConstant
  | Lapply of Lambda * Lambda list
  | Lfn of Lambda
  | Llet of Lambda list * Lambda
  | Lletrec of Lambda list * Lambda
  | Lprim of primitive * Lambda list
  | Lcase of Lambda * (SCon * Lambda) list
  | Lswitch of int * Lambda * (BlockTag * Lambda) list
  | Lstaticfail
  | Lstatichandle of Lambda * Lambda
  | Lhandle of Lambda * Lambda
  | Lif of Lambda * Lambda * Lambda
  | Lseq of Lambda * Lambda
  | Lwhile of Lambda * Lambda
  | Landalso of Lambda * Lambda
  | Lorelse of Lambda * Lambda
  | Lunspec
  | Lshared of Lambda ref * int ref
  | Lassign of int * Lambda
```

The meaning of each kind of Lambda clause is:

`Lvar(`i`)` represents a reference to a variable. The Lambda language uses de Bruijn notation, in which a reference to a variable carries an index rather than a name: `Lvar(`i`)` refers to the variable bound in the i'th enclosing Lambda abstraction (`Lfn`), let-binding (`Llet`), or letrec-binding (`Lletrec`).

`Lconst` represents an atomic or structured constant (see page 5).

`Lapply(`e_0`, [`$e_1, e_2, \ldots, e_n$`])` represents application of a curried function e_0 to the arguments $e_1, e_2, \ldots, e_n$. The function may be over-saturated, i.e., may be provided with more arguments than it expects, or may not be saturated at all, i.e., may be provided with fewer arguments than it expects. For example, a function f, taking one argument and returning another function, may be applied to more than one argument. The first argument will be used by f, whereas the remaining arguments will be used by the function returned by f.

`Lfn(`e`)` represents a Lambda abstraction. Within the body e variable number 0 (`Lvar(0)`) is implicitly bound to the parameter of the abstraction. Also, all indices of variables bound in the context are shifted by one (as seen from the body of the abstraction). Hence, within the body e, `Lvar(1)` refers to the innermost enclosing variable binding, `Lvar(2)` refers to the second enclosing variable binding, and so forth.

`Llet([`$e_1, e_2, \ldots, e_n$`], `e_0`)` represents the let-binding of $e_1, e_2, \ldots, e_n$ with scope e_0. Each of the variables `Lvar(`i`)`, where $0 \leq i \leq n-1$, is implicitly bound to the corresponding right-hand side expression e_{n-i}. Also, all indices of variables bound in the context are shifted by n, similarly to `Lfn` above.

`Lletrec([`$e_1, e_2, \ldots, e_n$`], `e_0`)` represents a letrec-binding of $e_1, e_2, \ldots, e_n$ with scope e_0. It is similar to the `Llet` expression described above, except that the right-hand sides $e_1, e_2, \ldots, e_n$, representing mutually recursive functions, may refer to each other. SML semantics guarantees that $e_1, e_2, \ldots, e_n$ are closure-building expressions.

`Lprim(`p`, [`$e_1, e_2, \ldots, e_n$`])` represents a primitive operation p taking arguments $e_1, e_2, \ldots, e_n$. The primitives are described on page 6.

`Lcase` is used in connection with `Lstatichandle`[3]: `Lstatichandle(Lcase(`e_0`,`es`), `e_f`)` represents pattern matching on the value of e_0, assumed to be of simple type; es is a list of keys and corresponding right-hand sides, whereas e_f is the 'default' right-hand side (which is chosen if the value of e_0 does not match any of the keys in es).

`Lswitch` is also used in connection with `Lstatichandle`: `Lstatichandle(Lswitch(`n`,`e_0`,`es`), `e_f`)` represents pattern matching on (the tag of) a constructed value or exception.

`Lstaticfail` represents a 'branch' to the default section of the innermost enclosing `Lstatichandle` Lambda expression. It is used only in connection with pattern matching, and is generated by the match compiler.

`Lhandle(`e_0`, `e_1`)` represents an SML exception handler covering the expression e_0; e_1 is the handler expression, that is, the expression to be evaluated in case an exception is raised during evaluation of e_0.

`Lif(`e_0`, `e_1`, `e_2`)` represents a two-way branch with conditional e_0. If e_0 evaluates to `true` then e_1 is evaluated, otherwise e_2 is evaluated.

`Lseq(`e_1`, `e_2`)` represents sequential evaluation of e_1 and e_2. The value of e_1 is discarded before evaluating e_2.

`Lwhile(`e_0`, `e_1`)` represents a loop with conditional e_0 and body e_1. The body is evaluated repeatedly as long as e_0 evaluates to `true`.

`Landalso(`e_1`, `e_2`)` represents an SML `andalso` expression: if e_1 evaluates to `true` then e_2 is evaluated, otherwise the result is `false`.

`Lorelse()` represents an SML `orelse` expression: if e_1 evaluates to `false` then e_2 is evaluated, otherwise the result is `true`.

`Lunspec` serves as a 'dummy' Lambda expression. It is used in connection with anonymous top-level declarations, e.g.

```
val _ = print "\n"
```

which translates to this Lambda code:

```
Llet([Lapply(Lprim(Pget_global ..., []),
             ["\n"]),
      Lunspec)
```

`Lshared(ref `e`, ref `i`)` represents a shared Lambda expression e. Shared expressions are generated by the match compiler in connection with certain forms of complex pattern matching. The integer i is used internally by the compiler back-end to ensure that the shared expression is processed only once (in each step of compilation).

`Lassign(`i`, `e`)` represents assignment to a local variable. This is used in optimization of code using local (i.e., non-escaping) `ref` cells[4].

Constants

The constants used in `Lconst` Lambda clauses are defined as:

[3]The `Lstatichandle` Lambda clause has nothing to do with exception handling.

[4]The `Lassign` Lambda clause is not used in the new compiler since optimization of local `ref` cells has not been implemented.

```
datatype StructConstant =
    ATOMsc of SCon
  | BLOCKsc of BlockTag * StructConstant list
  | QUOTEsc of obj ref
```

where

```
datatype SCon =
    INTscon of int
  | WORDscon of word
  | CHARscon of char
  | REALscon of real
  | STRINGscon of string
```

and

```
datatype BlockTag =
    CONtag of int * int             (* tag number & span *)
  | EXNtag of QualifiedIdent * int  (* constructor name & stamp *)
```

An `ATOMsc` clause represents an atomic constant, whereas a `BLOCKsc` clause represents a structured constant. The `QUOTEsc` clause is used only in the compilation of special functions for the interactive top-level. It is not considered any further in this report.

A structured constant represents an SML tuple or constructor, and carries a tag and a list of constant arguments. A `CONtag` tag indicates that the constant represents a tuple or value constructor, whereas an `EXNtag` tag is used for an exception constructor.

Primitives

The primitives of the Lambda language are represented as values of the data type `primitive`, declared in the module `Prim` of the compiler:

```
datatype primitive =
    Pidentity
  | Pget_global of QualifiedIdent * int
  | Pset_global of QualifiedIdent * int
  | Pdummy of int
  | Pupdate
  | Ptest of bool_test
  | Pmakeblock of BlockTag
  | Ptag_of
  | Pfield of int
  | Psetfield of int
  | Pccall of string * int
  | Praise
  | Pnot
  (* The next five are unsigned operations: *)
  | Paddint | Psubint | Pmulint | Pdivint | Pmodint
  | Pandint | Porint | Pxorint
  | Pshiftleftint | Pshiftrightintsigned | Pshiftrightintunsigned
  | Pintoffloat
  | Pfloatprim of float_primitive
  | Pstringlength | Pgetstringchar | Psetstringchar
  | Pmakevector | Pvectlength | Pgetvectitem | Psetvectitem
  | Psmlnegint | Psmlsuccint | Psmlpredint
  | Psmladdint | Psmlsubint | Psmlmulint | Psmldivint | Psmlmodint
  | Pmakerefvector
  | Patom of int
  | Psmlquotint | Psmlremint
  | Pclosure of int * int
  | Pswap
```

In the existing Moscow ML compiler, values of the data type `Prim.primitive` are used in the representation of Caml Light bytecode instructions, as well as in the `Lprim` Lambda clause. Some of the primitives are being used in the Caml Light code generator only.

The meaning of (some of) the primitives is:

`Pget_global` represents access to a global variable, that is, an identifier declared at top-level.

`Pset_global` represents the binding of a value to a top-level identifier. For each top-level identifier of the source program there will be exactly one `Pset_global` primitive in the intermediate Lambda code.

`Ptest` represents comparison of values, e.g. equality test.

`Pmakeblock` represents a tuple, a value constructor, a `ref` cell, or an exception constructor. It is similar to an `Lconst` Lambda expression representing a structured constant, but takes non-constant arguments.

`Pfield` represents access to a component of a tuple, value construtor, `ref` cell, or exception constructor.

`Psetfield` represents assignment to a `ref` cell.

`Pccall` represents a call to an ‘external’ function, e.g. a function implemented in the run-time system.

`Praise` represents the raising of an exception.

`Pnot` represents boolean negation.

`Patom` represents a null-constructor.
`Pclosure` represents a function that has been lifted to top-level. It is generated by the compiler back-end during lifting.

The primitives `Paddint`, `Psubint`, ..., `Pshiftrightintunsigned` represent operations on values of SML type `word`, whereas the primitives `Psmlnegint`, `Psmlsuccint`, ..., `Psmlmodint`, `Psmlquotint`, and `Psmlremint` represent operations on values of SML type `int`.

Exceptions

Two different kinds of exceptions are used in the Lambda code of Moscow ML: static and dynamic exceptions. Static exceptions include the predefined exceptions, e.g. `Bind` and `Match`, plus the exceptions declared at top-level in a program. Dynamic exceptions are exceptions declared in local scope, e.g. inside a `Llet` expression.

The compiler front-end generates no Lambda code for the declaration of a static exception (i.e., no Lambda code is generated for the exception declaration itself). When the exception is raised, a special constructor with no arguments, and with the name of the exception as tag, is created. For example, the declaration

```
local exception Paf
in
    fun f () = raise Paf
end
```

in module `A` translates to this Lambda code (in pretty-printed form):

```
prim (set_global Top.f/1) (fn (prim (raise) (BLOCK Top.Paf/1 )))
```

The declaration of a dynamic exception is translated to Lambda code creating a reference to a pair of strings, comprising the name of the exception and the name of the module. When a dynamic exception is raised, this is translated to a static exception with the name `General.Exception/0`, taking the dynamic exception as an argument, being raised. That is, an exception constructor with the reserved tag `General.Exception/0`, and with the dynamic exception (a reference to a pair) as argument, is created. For example, the expression

```
let exception Slam
in
    (raise Slam) : unit
end
```

in module `B` translates to this Lambda code (in pretty-printed form):

```
let (prim (makeblock 250:1) (BLOCK 0:1 "Slam" "N")) in (prim
(raise) (prim (makeblock General.(Exception)/0) var:0)) end
```

When an exception with arguments is raised, the arguments of the exception appear as arguments to the exception constructor being created. For example, the declaration

```
local exception Zap of int * int
in
    fun g n = raise Zap(n, n * 2)
end
```

in module `P` translates to this Lambda code (in pretty-printed form):

```
prim (set_global Top.g/2) (fn (prim (raise) (prim (makeblock
Top.Zap/2) var:0 (prim (smlmulint) var:0 2))))
```

And the expression

```
let exception Kapow of string * string
in
    (raise Kapow("Hello", "World")) : unit
end
```

translates to

```
let (prim (makeblock 250:1) (BLOCK 0:1
"Kapow" "Top")) in (prim (raise) (prim (makeblock
General.(Exception)/0) var:0 (BLOCK 0:1 "Hello" "World"))) end
```

Chapter 2

Design

This chapter describes the design of a general scheme for translation of SML programs into Java bytecode. The design is based on the intermediate Lambda code produced by the front-end of the Moscow ML compiler (cf. Section 1.3.3), and serves as a basis for the implementation of the new back-end described in Chapter 4.

A number of optimizations to the translation scheme described in this chapter have been designed and implemented; these are described in Chapter 5. Further optimizations and improvements are discussed in Chapter 7.

2.1 Representation of SML Values

The original back-end of Moscow ML produces Caml Light bytecode. Since SML is strongly typed, and Caml Light bytecode is untyped, SML values can be represented quite efficiently: no type information is required at run-time, and values of simple types can be represented unboxed, provided they fit in a single machine word.

Java bytecode is somewhat different from Caml Light bytecode, especially in that it is typed. This is because Java bytecode must be verified by the JVM before it is executed. For example, an integer value cannot be used as a reference (pointer), and integer operations cannot be used to manipulate a reference.

In a translation from SML to Java bytecode, SML values should be represented in a uniform way, regardless of their SML types. The reason for this is that a value may be passed as argument to polymorphic functions, or may be stored in tuples, data type constructors, arrays, etc. One solution is using a boxed representation for all SML types, that is, wrapping every SML value into a JVM object (an instance of a subclass of class `java.lang.Object`). Every operation on an SML value (of simple type) should then be enclosed in proper unwrap and wrap operations.

2.1.1 Simple Types

Values of the simple SML types `int`, `word`, `char`, `real`, and `string` may be represented as instances of the standard Java classes `Integer`, `Double`, and `String`[1].

[1]Java standard classes in package `java.lang` are generally referred to without the package qualifier. For example, `Object` implicitly means `java.lang.Object`.

The idea is that an SML value, e.g. of type `int`, should be 'wrapped' into an `Integer` object, and should be passed around as such[2]. When a primitive operation is to be performed, e.g. integer addition, the values of the operands should first be unwrapped, the addition should be performed on the unboxed integer values, and the result should then wrapped into a fresh `Integer` object. Similarly for `word`, `char`, and `real` values.

A value of SML type `word` or `char` may be represented as a 32-bit integer value wrapped into an `Integer` object. As the JVM represents a value of type `int` in 32-bit signed two's complement, `word` arithmetics may be encoded using JVM integer instructions, interpreting the result as an unsigned integer. Also, `char` operations may be encoded using JVM integer operations and masking out unused bits of the 32-bit value.

It would be more straightforward to represent values of SML type `char` as `Character` objects. Unfortunately, the Lambda code generated by the existing Moscow ML front-end in some cases treats `char` values as `int` values, e.g. in connection with tests. Hence it would not be possible to translate values of type `char` to `Character` objects without modifying the Lambda code to carry extra type information.

A value of SML type `real` may be represented as a JVM value of type `double` wrapped into a `Double` object. A `double` value is represented as a 64-bit IEEE double precision floating-point value internally by the JVM.

The most natural encoding of SML values of type `string` is as instances of class `String`, since both types represent immutable character sequences. Also, JVM string literals are represented as `String` objects, so comparison with literals may be implemented efficiently. However, encoding SML strings as `String` objects means that certain Basis Library functions[3], e.g. `String.implode`, cannot be implemented as efficiently as would be possible in a representation of strings using mutable character sequences (more copying is necessary with immutable character sequences).

Alternatively, an SML `string` could be represented as a mutable array of characters, i.e., a value of JVM type `char[]`, but this would not work in connection with polymorphic equivalence (cf. Section 2.2). Similarly, representing an SML `string` as an instance of the standard Java class `StringBuffer` would not work in connection with polymorphic equivalence[3]. In the new back-end, an SML `string` is compiled to an instance of class `String`.

Values of the simple SML types `bool` and `unit` are represented as SML constructors in the Lambda code (cf. Section 2.1.4 and 2.1.5).

2.1.2 Primitive Operators for Simple Types

In general, the primitive operators for the simple SML types may be translated to corresponding Java bytecode instructions. For example, addition of two values of type `real` may be encoded directly using the JVM instruction `dadd` (with the necessary unwrapping/wrapping from/to `Double` objects). Similarly for conversion between the simple SML types.

The semantics of the primitive SML integer operators, however, differs from that of the corresponding JVM integer instructions. Specifically, the SML operators +, -, *, div, and ~ must

[2]An `Integer` object contains a value of JVM type `int` (a 32-bit signed integer in two's complement).

[3]The `StringBuffer` class represents mutable character seqences. Unfortunately, the class does not implement method `equals`, which implies that a `StringBuffer` object inherits method `equals` of class `Object`. Moreover, method `equals` cannot be implemented in a subclass of `StringBuffer` since it is a `final` class.

raise `Overflow` in case the result cannot be represented as an `int` value, whereas overflow is ignored by the corresponding JVM instructions `iadd`, `isub`, `imul`, `idiv`, and `ineg`.

The correct semantics of the above SML integer operators may be encoded using the corresponding JVM instructions for 64-bit signed `long` values (`ladd`, `lsub`, `lmul`, `ldiv`, and `lneg`), and explicitly checking for overflow. That is, the `Integer` operands would first be converted to values of JVM type `long`, and the operation would be performed with the proper `long` instruction. It could then be checked whether overflow occurred: if the result can be represented as an `int` value, the result would be wrapped into a fresh `Integer` object; otherwise, the SML exception `Overflow` would be raised. Unfortunately, this encoding of integer operations may be rather inefficient.

Furthermore, the SML `int` and `word` division operators are required to raise exception `Div` in case of division by zero. The corresponding JVM instructions (`idiv` and `irem`) do in fact throw a JVM exception (of class `ArithmeticException`) in case of division by zero, but this should be mapped to the SML exception `Div` at run-time. Two possible solutions are:

1. Checking the divisor prior to every division, and raising SML exception `Div` if the divisor is zero.
2. Inserting a JVM exception handler for each `idiv` and `irem` instruction. The handler would catch an `ArithmeticException`, and raise SML exception `Div` (cf. Section 2.1.8).

The latter solution will be the most efficient since a JVM exception handler introduces no run-time overhead, as long as no exceptions are thrown. This is the approach that is taken with the new back-end.

2.1.3 Algebraic Types

The constructors of an algebraic data type must be represented in a way so they can be distinguished at run-time. Two possible representations are:

1. As in the Pizza language[7]: every data type becomes a class (with a tag instance field), and each constructor becomes a separate subclass of the data type class (with a corresponding tag value, and an instance field for each constructor argument). The constructor tag can be used in pattern matching against each of the constructor cases.
2. As instances of a general class `Constructor` with an instance tag and an instance array of arguments[4]:

```
class Constructor
{
  int tag;
  Object[] args;

  Constructor (int tag, Object[] args)
  { this.tag = tag;  this.args = args; }
}
```

[4]Note that class `Constructor` has no span instance field. The constructor span is used only in the compiler, not at run-time.

The former solution would be the natural encoding of SML constructors in an object-oriented language. It requires, however, that the intermediate Lambda code of the Moscow ML compiler be augmented with type information, in order to distinguish constructors of different data types. For example, the SML expression

```
1 :: (2 :: (3 :: []))
```

is translated to this Lambda code

```
BLOCK 1:2 1 (BLOCK 1:2 2 (BLOCK 1:2 3 (BLOCK 0:2 )))
```

where `BLOCK` is a `BLOCKsc` structured constant in pretty-printed form.

The first argument to `BLOCK` is the constructor tag and span ("1:2" means tag 1, span 2). The next argument to `BLOCK` is a list of constructor arguments. Note that there is no information in the Lambda code to tell that this is in fact a list of integers. Given the data type declaration

```
datatype 'a t = A | B of 'a * 'a t
```

the expression

```
B(1, B(2, B(3, A)))
```

translates to the exact same Lambda code as shown above for the integer list. Similarly, the Lambda expression `(BLOCK 0:2)` may represent `nil`, but the same expression may also represent an SML value of some other type, e.g. `false`.

Representing SML constructors as `Constructor` objects, corresponding to solution 2 above, has a number of advantages:

- The Lambda code of the existing Moscow ML compiler can be used without carrying extra type information along.
- Efficient switching on the `Constructor` tag is possible.
- The translation from Lambda code (and its primitives) to Java bytecode is rather direct. For example, this is the Lambda code for extracting the head of a list:

  ```
  prim (field 0) var:0
  ```

  and for extracting the tail of a list:

  ```
  prim (field 1) var:0
  ```

  These two Lambda expressions may be compiled directly into Java bytecode fetching component number 0 and 1, respectively, from the `args` field of a `Constructor` object.
- Fewer classes are generated for an SML program, and hence fewer class files must be loaded at run-time.

In the Java bytecode generated by the new back-end, SML constructors are compiled to `Constructor` objects.

2.1.4 Boolean Values

As mentioned in Section 2.1.1, SML values of type `bool` are represented as constructors in the Lambda code. This means that boolean values may be represented as `Constructor` objects in the Java bytecode:

`false`	$\Rightarrow$	`Constructor(0, null)`
`true`	$\Rightarrow$	`Constructor(1, null)`

In boolean expressions, and in tests, the boolean sub-expressions may be converted to control-flow representation by branching on the `Constructor` tag.

2.1.5 Tuples, Records, and Unit

In the Lambda code of Moscow ML, a tuple is represented as a `BLOCK` with tag 0 and span 1. Three possible solutions for representation of tuples in Java bytecode are:

1. As instances of class `Constructor`; the constructor tag is immaterial (could be 0, as in the Lambda code).
2. As arrays of objects; this would mean that component access is just array indexing.
3. As constructors without a tag.

The second solution would be more efficient than the other two since an extra indirection in connection with component access could be avoided. However, the Lambda code produced by the Moscow ML front-end would have to be extended with type information to distinguish tuple component access from constructor argument access. For example, the expression

```
(prim (field 1) var:0)
```

appears in the Lambda code for both of these SML functions

```
fun f (x, y)  = y
fun g (x::xr) = xr
```

The third solution is more elegant than the former two, and may be implemented like this, based on the Lambda code produced by the existing front-end:

```
abstract class Block
{
  Object[] args;
}

class Tuple extends Block
{
  Tuple (Object[] args)  { this.args = args; }
}

class Constructor extends Block
{
  int tag;

  Constructor (int tag, Object[] args)
  { this.tag = tag;  this.args = args; }
}
```

SML tuples would thus be represented as instances of class `Tuple`, whereas constructors would be represented as instances of class `Constructor`. This is the approach that is taken with the new back-end.

Note that the `args` instance field has been 'moved up' to a common superclass of `Tuple` and `Constructor`. This way, the arguments of a constructor or tuple may be accessed in a uniform way, namely, via the field declared as `Block.args`. Hence, a direct translation of tuple component access and constructor argument access to Java bytecode is possible.

An SML record is translated to a tuple by the Moscow ML front-end. Similarly, the value `() : unit` is translated to an empty tuple, i.e., a tuple with no components, and requires no special attention.

2.1.6 Ref Cells

In the Lambda code of Moscow ML, an SML `ref` cell is represented as a constructor (`BLOCK`) with tag 250 and span 1. Hence, references could be represented as `Constructor` objects with the same (reserved) tag value[5]. A more elegant solution would, however, be representing references as instances of a separate subclass of class `Block`, just as described for tuples in the previous section:

```
class Ref extends Block
{
  Ref (Object arg)
  { Object[] args = {arg};  this.args = args; }
}
```

In the Java bytecode generated by the new back-end, SML `ref` cells are compiled to `Ref` objects.

Note that an instance of class `Ref` has no tag, and carries only one argument, which may be accessed as `args[0]`.

2.1.7 Functions

Just like values of simple types, constructors, etc., SML functions may be passed as arguments, stored in tuples, data type constructors, arrays, etc. Hence an SML function should be represented in a uniform way, i.e., wrapped into an object. Three possible solutions are:

1. Using the Pizza 'homogeneous' translation[7] with a class `C$Closure` for every source program `C`. The closure class contains an instance `tag` field, an instance field holding an array of free variables, and an instance method `apply`. Every closure building expression in the source program is associated with a tag. The `apply` method switches on the instance tag, and the switch entry for a given tag implements the closure corresponding to that tag.
2. Generating a class `Cf` for every closure building expression `f` in the source program. Each class `Cf` implements an interface, e.g. denoted `Callable`, with a method `apply()`

[5]In Moscow ML, a number of tag values are reserved for special purposes: `ref` cells, strings, reals, abstract values, weak pointers, and finalized objects.

implementing the body of function `f`. Each `Cf` class also includes an instance field holding the free variables of the closure it represents.

3. Using a general `Closure` class:

   ```
   abstract class Closure
   {
     int tag;
     Object[] free;

     abstract Object apply (Object x);
   }
   ```

   The closure-building expressions in an SML program would be represented as instances of a subclass of `Closure`, implementing the `apply` method with a switch on the `Closure` tag. The switch entry for a specific tag would then implement the body of the closure corresponding to that tag.

The latter approach is the simplest, and leads to generation of fewer classes than the former solutions. This is the solution that is used in the new back-end.

As an example, the SML functions

```
fun f x = x + 1
fun g _ = ()
```

may be represented like this, corresponding to solution 3 above[6]:

```
class C extends Closure
{
  C (int tag, Object[] free)
  { this.tag = tag;  this.free = free; }

  static Object f, g;

  Object apply (Object x)
  {
    switch (tag)
      {
      case 0: return new Integer(((Integer)x).intValue() + 1);
      case 1: return new Tuple(null);
      ...
      }
  }

  static
  {
    f = new C(0, null);
    g = new C(1, null);
    ...
  }
}
```

[6]Java, rather than Java bytecode, is used as the target language in this and the following examples only for the sake of simplicity and clarity. The new back-end directly produces Java bytecode, in the form of Java class files.

The static initializer in class `C` binds fresh instances of `C` to the static fields `f` and `g`. Hence, passing `f` around, e.g. as argument to another function, is simply a matter of passing the object bound to `C.f`. Similarly, when `f` is to be invoked, this will take place via invocation of method `Closure.apply` on the object bound to `C.f`.

Note that the static fields `f` and `g` in the above class are declared as having type `Object`, not `Closure`. This is because the Lambda code produced by the existing Moscow ML front-end does not contain enough information to tell the type of a top-level name. Hence, the most general type must be used for all top-level declarations.

Translation of polymorphic functions to Java bytecode is straightforward, provided all values are represented as objects: a polymorphic function just takes an argument of type `Object`, and returns an `Object`.

Translation of higher-order functions in Java bytecode is also straightforward, provided all functions are represented as instances of class `Closure`. For example, the function declaration

```
fun F h = h 117
```

may be represented by a switch entry like this in method `apply` (as shown above):

```
case 7: return ((Closure)x).apply(new Integer(117));
```

where 7 is the closure tag for the function.

Curried functions are translated to nested `Lfn` clauses in the Lambda code, and require no special attention. However, application of a curried function to more than one of its arguments should be represented as a sequence of closure applications, each application using one of the arguments (cf. Section 4.6.3).

2.1.8 Exceptions

In Java bytecode, SML exceptions may be represented as instances of this class:

```
class Exception extends Block
{
  Object tag;

  Exception (Object tag, Object[] args)
  { this.tag = tag;  this.args = args; }
}
```

Making class `Exception` a subclass of class `Block` is convenient: in the Lambda code, exception component access is identical to constructor component access (cf. Section 2.3).

To be able to throw instances of class `Exception` as JVM exceptions, `Exception` must be a subclass of the standard Java class `Throwable`. In Java (and in the JVM), a class can only have one immediate superclass[7]. Hence, class `Exception` cannot be an immediate subclass of `Block` and `Throwable` at the same time. One solution is changing class `Block` to be a subclass of `Throwable`:

[7]Java (and the JVM) actually supports multiple inheritance via interfaces, which is a special kind of abstract classes: a class can have multiple immediate superinterfaces. However, class `Block` cannot be changed to be an interface since it declares an instance field, which is not allowed in an interface.

```
abstract class Block extends Throwable
{
  Object[] args;
}
```

This has the undesirable effect of turning all subclasses of `Block` into subclasses of `Throwable`. Hence, an instance of any subclass of `Block`, e.g. a `Constructor` object, could also be thrown as a JVM exception. However, provided only instances of class `Exception` are used for implementing SML exceptions, this should not be a problem.

For static exceptions, the exception tag may be represented as a 'named object', e.g.

```
static Object Zap = new String("A.Zap");
```

corresponding to this declaration in structure `A`:

```
exception Zap of int * int
```

When the exception is raised, an instance of class `Exception` should be created with the exception name as tag. For example, the expression

```
raise Zap(n, n * 2)
```

may be represented like this in Java bytecode:

```
Object[] args = {n, n * 2};
throw new Exception(Zap, args);
```

As described in Section 1.3.3, a dynamic exception is represented as a reference to a pair of strings in the Lambda code, and this compiles directly to Java bytecode. Raising a dynamic exception may be implemented by throwing an instance of class `Exception` with tag `General.(Exception)` and proper arguments. For example, the SML expression

```
let exception Kapow of string * string
in
    (raise Kapow("Hello", "World")) : unit
end
```

may be represented like this in Java bytecode:

```
Object[] args1 = {"Kapow", "A"};
Ref Kapow = new Ref(new Tuple(args1));
Object[] args2 = {"Hello", "World"},
         args3 = {Kapow, new Tuple(args2)};
throw new Exception(General.(Exception), args3);
```

where `A` is the module name.

This solution requires a class `General` including a static field `(Exception)` to be present at run-time[8]:

[8]In Java, and in the JVM, a field name cannot include parentheses or any other punctuation characters. Instead, an encoding of such characters, e.g. by their hexadecimal ordinal value, may be used: `$28Exception$29`.

```
class General
{
  static Object (Exception) = new String("General.(Exception)"),
                Overflow    = new String("General.Overflow"),
                Div         = new String("General.Div");
  ...
}
```

The static fields `Overflow` and `Div` of class `General` may be used for implementing the corresponding 'predefined' static SML exceptions (cf. Section 2.1.2).

2.2 Polymorphic Equality

Traditionally, polymorphic equality is one of the more challenging features in implementations of functional languages. In an implementation of SML, testing for equality involves recursively traversing the sub-nodes of constructed data, except for references and arrays, which must be compared by pointer equality.

In Java bytecode, polymorphic equality may be implemented via the method `equals` defined in class `Object` (and thus inherited by all other classes). For values of the simple SML types, the corresponding standard Java classes directly implement method `equals` as equality of the values contained in the wrapper objects. For example,

```
new Integer(x).equals(new Integer(y))  ⇔  x == y
```

holds for all `int` values `x` and `y`. Similarly for instances of class `Double`. For instances of class `String`, method `equals` returns true iff the two objects represent identical sequences of characters.

In order to implement polymorphic equality for tuples, class `Tuple` should override method `equals`:

```
public boolean equals (Object x)
{
  if (this.args != null)
    {
      int i = this.args.length;

      while (--i >= 0)
        if (!this.args[i].equals(((Tuple)x).args[i]))
          return false;
    }

  return true;
}
```

This method compares the current `Tuple` object, a.k.a. `this`, to the argument `x`, which is assumed to be bound to an instance of class `Tuple`. The type system of SML guarantees that a tuple will only be compared to another tuple of the same arity, and with components of corresponding SML types. Hence, there is no need to verify that the argument arrays `this.args` and `x.args` have the same length. Note that the components of the two tuples are compared recursively.

Similarly, class `Constructor` should override method `equals` in order to implement polymorphic equality for constructors:

```
public boolean equals (Object x)
{
  if (this.tag == ((Constructor)x).tag)
    {
      if (this.args != null)
        {
          int i = this.args.length;

          while (--i >= 0)
            if (!this.args[i].equals(((Constructor)x).args[i]))
              return false;
        }

      return true;
    }
  else
    return false;
}
```

It is assumed that the argument `x` is an instance of class `Constructor`. The type system of SML guarantees that a constructor will only be compared to another constructor of the same SML data type. There is no need to verify that the argument arrays `this.args` and `x.args` have the same length: if the two `Constructor` objects have the same tag they must represent the same constructor within the data type, and hence have the same arity.

Polymorphic equality for SML `ref` cells need not be handled explicitly, provided references are represented as instances of class `Ref` (cf. Section 2.1.6). Equality on `Ref` objects is implicitly implemented via the `equals` method inherited from class `Object`. Since that method returns true for two object references iff they refer to the very same object, and since class `Block` and class `Ref` do not override method `equals`, comparing two `Ref` objects via method `equals` corresponds to pointer equality on `ref` cells.

In the type system of SML, functions and exceptions do not admit equality, and hence requires no special attention with respect to polymorphic equality.

2.3 Exception Handling

The JVM exception mechanism may be used to implement SML exceptions and exception handling. Every Java bytecode method may declare a set of exception handlers, each covering a specific sequence of instructions in the method, and handling a specific class of exceptions.

Each `Lhandle` clause of the Lambda code may thus be represented as one JVM exception handler, catching exceptions of class `Exception`. For example, the expression

```
fn () => ((2 + 2) handle Overflow => 5)
```

translates to this Lambda code:

```
(fn ((prim (smladdint) 2 2) handle ((switch:0 var:0 of
General.Overflow/0 : 5) statichandle (prim (raise) var:0))))
```

In Java bytecode, the body of the function may be represented like this:

```
try { return new Integer(2+2); }
catch (Exception e)
{
  if (e.tag == General.Overflow$0)
    return new Integer(5);
  else
    throw e;
}
```

Unfortunately, the semantics of JVM exceptions differs from that of SML exceptions: when a JVM exception is thrown, the operand stack of the current method is purged (all operands are removed from the local stack). This means that an SML exception handler appearing in a value context cannot be translated directly to a JVM exception handler. For example, the expression

```
37 + ((19 div 0) handle Div => 17)
```

could, hypothetically, be compiled to Java bytecode pushing the integer 37 onto the stack before evaluating the division operation. However, the value 37 would be removed from the operand stack once division by zero occurs (and the JVM exception `ArithmeticException` is thrown). Hence, when the exception handler pushes the integer 17, the first operand of the addition operation is no longer available.

Two possible solutions are:

1. Introducing bindings for all JVM stack operands to local variables prior to code covered by an exception handler appearing in a value context.
2. Lifting the body of an exception handler appearing in a value context into a separate method, and generating a JVM exception handler covering the instructions in that method. This way, the lifted handler body will be evaluated using a separate stack. If a JVM exception is thrown during evaluation of the handler body, only the local operand stack will be purged.

The latter approach would generally lead to shorter Java bytecode, imply less copying, and use fewer local variables than the former. This is the solution that is used with the new back-end.

2.4 Top-level Names and Exporting

In an SML module, the same name may be declared several times at top-level. To avoid ambiguity, Lambda code refers to top-level names via ‘unique identifiers’ comprising the top-level name and a stamp i, where i is the number of the declaration (counting from 1). For example, given these (valid) top-level SML declarations in a structure `S`

```
val a = 0
fun f () = a
val a = "foo"
fun g () = a
```

the Lambda code for the body of function `f` will be:

```
(prim (get_global S.a/1) )
```

whereas the Lambda code for the body of `g` will be:

```
(prim (get_global S.a/3) )
```

When a top-level name in another module is referred to, the unique identifier has stamp 0 in the Lambda code. This implicitly means that the 'exported' top-level declaration of that name is referenced. This corresponds to the SML semantics of subsequent declarations: redeclaration of a name overrides earlier declarations of the same name (in the same scope).

A unique identifier `foo/`*i* in the Lambda code may be represented as a static field `foo$`*i* in Java bytecode. Cross-module references may be implemented by introducing a set of extra 'export bindings' in the Java bytecode. For each exported identifier `foo/`*k* there would be a binding like this:

```
foo$0 = foo$k;
```

Alternatively, all references to an exported identifier `foo/`*k* from within the same module may be translated to references to the static field `foo$0`. This way, the above export bindings could be omitted. This is the approach that is taken with the new compiler back-end[9].

Top-level exception identifiers need not be exported under a different name. This is because the compiler front-end translates each reference to an exported exception from another module to Lambda code referencing the *internal* identifier of that exception. For example, if structure `A` contains the declarations

```
local exception A
in
    ...
end

exception A
```

where the second declaration of `A` has stamp 2 in the Lambda code, then the expression `raise A.A` in another module will be translated to this Lambda code:

```
(prim (raise) (BLOCK A.A/2))
```

In other words, exception exporting requires no further attention since this is handled by the compiler front-end.

Unfortunately, the compiler front-end does not properly distinguish top-level excetion identifiers from top-level variable identifiers. For example, a structure `B` including the SML declarations

```
val B = "slam"
val C = B
exception B
fun f () = (raise B) : unit
```

[9]Instead of translating all references to an exported top-level identifier `foo/`*k* to `foo$0`, the name `foo` could be used without any suffix. This would, however, lead to potential name clash between SML identifiers and other members of the target class, e.g. with respect to the `tag`, `free`, and `apply` instance members of class `Closure`.

is translated to the following Lambda code:

```
(prim (set_global B.B/1) "slam")
(prim (set_global B.C/2) (prim (get_global B.B/1) ))
(prim (set_global B.f/3) (fn (prim (raise) (BLOCK B.B/1 ))))
```

Note that the expression being assigned to `B.C/2`, as well as the body of function `B.f/3`, refers to `B.B/1`.

In order to avoid name clash in the target program, exception identifiers must be treated differently from top-level variable identifiers. One solution is translating all references to an exception identifier `E/`k into references to the static field `E$x`k. This is the solution implemented in the new back-end.

2.5 Modules

By following the design described in previous sections, a Moscow ML structure may be represented as one Java class file. In other words, a source file `A.sml` may be compiled to a class file `A.class`, implementing class `A`. Compilation of a signature file `A.sig` to a compiled signature file `A.ui` may be adopted directly and unchanged from Moscow ML[10].

Class `A` would be a subclass of the `Closure` class (cf. Section 2.1.7), and would include a static member declaration for each of the top-level declarations in the source file `A.sml`. Furthermore, class `A` may contain an 'empty' method declaration like this:

```
public static void main (String[] args) {}
```

A program consisting of one or more modules could thus be invoked by calling the special method `main` in the 'main' class of the target program. Normally, this is done by invoking the JVM with the name of the main class as argument. The JVM then loads the class file for that class, and recursively loads any classes that the main class depends upon (dynamically). This mechanism would replace the linking phase of the existing Moscow ML compiler.

The semantics of SML modules requires that execution of a program comprising the modules `A`, `B`, and `C` (in that order) should correspond to evaluation of `A`, `B`, and `C` in the same order. Unfortunately, this order of evaluation for the modules of a program would not be preserved in the approach described above. Instead, the modules of a program would be loaded and evaluated in the order which they are referenced, starting from the 'main' module. Presumably, most well-structured SML programs tend to have a main module, and do not rely on the specified evaluation order for modules, so this should not be a problem.[11]

Note that the above approach to module implementation includes no linking phase: references to other modules (classes) are represented as symbolic references in the generated class files, and these references are not resolved until load-time or run-time (depending on the JVM

[10]As in Moscow ML, the compiled signature file `A.ui` should be generated from the 'explicit' signature in file `A.sig`, if present. If no explicit signature is present, `A.ui` should be generated from the inferred signature for `A.sml`.

[11]The Compilation Manager in Standard ML of New Jersey has similar problems with respect to evaluation order for modules.

implementation). Hence, signature matching with respect to referenced modules will take place at compile-time only.

The link-time verification of signature stamps, as implemented in the existing Moscow ML compiler, may be mimicked by introducing a signature stamp in each generated class file. The signature stamp should be calculated from the signature of the module, e.g. by hashing the contents of the signature, and should be practically unique. The static initializer of a generated class should then compare the value of the signature stamp for each of the modules it depends upon against the 'expected value', calculated at compile-time.

For example, if module `A` depends on module `B`, a class declaration like this might be generated for `A`:

```
class A extends Closure
{
  final static long sigStamp;

  static
  {
    if (B.sigStamp != 12345678)
      throw new SmlError("signature for B has changed");
    ...
    sigStamp = ldots;   // initialize A.sigStamp
  }

  ...
}
```

where 12345678 is the compile-time signature stamp for `B`. This way, the actual signature stamp of module B, found in the class file `B.class` at load-time, would be checked against the 'expected' stamp when class `A` is loaded and initialized.

In case the signature of `B` had changed since `A` was compiled, the signature stamp `B.sigStamp` would no longer have the same value as that expected by the static initializer of class `A` (provided the signature of `B` had been recompiled, and provided the stamps are really unique).

Cyclic module dependencies are not permitted in SML. The above approach to signature stamp verification would ensure that generated class files with cyclic dependencies could not be loaded and initialized. If module A refers to module B, and B refers to A, then the static initializer in each of the generated classes would first have to check the signature stamp of the other class.

Note that the field `A.sigStamp` would be initialized only at the end of the static initializer of class A, and similarly for `B.sigStamp`. If class A was loaded first, its static initializer would attempt loading the value of `B.sigStamp`, thereby causing class B to be loaded and initialized. The static initializer of class B would then load the value of `A.sigStamp`, which had not yet been initialized; hence, `A.sigStamp` would have the default value 0, as seen from B[12]. Hence the signature stamp check would fail for B's reference to A (unless the expected signature stamp for A actually was 0).

A more precise error message in case of cyclic module dependencies could be generated by inserting a test like this prior to the stamp check in the static initializer of class A (and vice

[12]In the JVM, a static field has the default value for its type until it has been initialized (assigned a value). The default value for type `long` is 0.

versa in the static initializer of class B):

```
if (B.sigStamp == 0)
  throw new SmlError("module B depends on itself");
```

2.6 Printing Values

For the purpose of debugging, the `toString` method declared in class `Object` may be used to print SML values in user programs. Hence, the user may create a wrapper class, printing the string representation returned by method `toString` for a specific value.

The standard Java classes `Integer`, `Double`, and `String` already override method `toString`, returning a string representation of the contained value. Note, however, that SML values of type `word` and `char` will be presented as signed integer values.

The run-time support classes for representing SML constructors, `ref` cells, etc. should be extended to override method `toString`. For example, the method may be declared as follows in class `Tuple`:

```
public String toString ()
{
  String s = "Tuple(";

  if (args != null && args.length != 0)
    {
      s += args[0];

      for (int i = 1; i < args.length; ++i)
        s += ", " + args[i].toString();
    }

  return s + ")";
}
```

This will return the string "`Tuple(`$s_1, s_2, \ldots, s_n$`)`", where $s_1, s_2, \ldots, s_n$ are the string representations of the tuple components.

In class `Constructor` and `Exception`, method `toString` may be declared similarly to the above declaration for class `Tuple`, but should also include the constructor/exception tag.

Method `toString` in class `Closure` may simply be declared as:

```
public String toString ()
{
  return ("Closure(" + tag + ")");
}
```

In class `Ref`, method `toString` may be declared similarly to the above declaration for class `Tuple`, but this may cause problems in connection with circular references. A call to method `toString` on a `Ref` object that is part of a cycle in data will lead to infinite recursion (and, eventually, the JVM frame stack will overflow).

Two possible solutions are:

1. Introducing a flag in each `Ref` object, that is, a `boolean` instance field, to avoid revisiting the same node in the graph.
2. Maintaining a table of visited nodes in the graph.

In both cases, method `toString` in class `Ref` should check whether the current object has already been visited. If so, it should do nothing; otherwise, it should record that the object has been visited, and append the string representation of the current object (including the argument to the `ref` cell) to the result.

Also, in both cases, the 'state' of all `ref` cells will have to be reset before method `toString` is invoked on user data. With solution 1, this implies traversing the data and resetting the flag of each `Ref` object. With solution 2, it means removing all entries from the table. The latter will generally be the most efficient approach.

In the run-time support class `Ref` used in Java bytecode generated by the new back-end, the problem with circular references is ignored. Hence, invoking method `toString` on data containing a cycle will indeed cause infinite recursion (and frame stack overflow).

The above approach to conversion of SML values to printable form may be extended to support printing in general, i.e., not just for debugging purposes. For example, a top-level interactive loop could exploit this to print the result in each iteration. This would require, however, that the run-time support classes were extended to carry information about the type of the contained value. For example, a `Constructor` object should carry the name of its datatype and the name of the constructor it represents.

2.7 Summary

Values of the simple SML types `int`, `word`, `char`, `real`, and `string` may be represented as instances of the following standard Java classes:

SML type	Java class	Contained value
`int`	`Integer`	32-bit signed integer
`word`	`Integer`	32-bit signed integer
`char`	`Integer`	32-bit signed integer
`real`	`Double`	64-bit IEEE double precision floating-point value
`string`	`String`	Immutable character sequence

Values of other SML types may be represented as instances of the following classes:

SML type	Java class	Contained value(s)
$\alpha \times \beta \times \ldots$	`Tuple`	Tuple components
$\{l_1 : \alpha, l_2 : \beta, \ldots\}$	`Tuple`	Record components, sorted by label
`unit`	`Tuple`	None
(datatype)	`Constructor`	Constructor tag and arguments
`bool`	`Constructor`	Constructor tag 0 or 1
`ref`	`Ref`	Reference
$\alpha \to \beta$	`Closure`	Closure tag and free variables
`exn`	`Exception`	Exception tag and arguments

Note that SML functions are in fact represented as instances of subclasses of `Closure`.

The hierarchy of classes that may be used for representing SML values is:

```
java.lang.Object
   ▷ java.lang.Integer
   ▷ java.lang.Double
   ▷ java.lang.String
   ▷ java.lang.Throwable
       ▷ Block
           ▷ Tuple
           ▷ Constructor
           ▷ Ref
           ▷ Exception
   ▷ Closure
   ▷ General
```

Class `General` holds static exception tags for the 'predefined' SML exceptions.

A structure `S`, where the source for `S` is

```
fun f x = x + x
val a = f 37
```

is translated to this Lambda code by the Moscow ML front-end:

```
(prim (set_global S.f/1) (fn (prim (smladdint) var:0 var:0)))
(prim (set_global S.a/2) (app (prim (get_global S.f/1) ) 37))
```

The Lambda code may then be compiled to a class file having this structure:

```
class S extends sml.lang.Closure
{
  static Object f$0, a$0;

  S (int tag, Object[] free)
  { this.tag  = tag;  this.free = free; }

  Object apply (Object x)
  {
    switch (this.tag)
    {
    case 0: return new Integer(((Integer)x).intValue() +
                               ((Integer)x).intValue());
    default:  throw new SmlError("unmatched closure tag");
    }
  }

  static
  {
    f$0 = new S(0, null);
    a$0 = ((sml.lang.Closure)f$0).apply(new Integer(37));
  }
}
```

Note that the top-level names `f/1` and `a/2` in the Lambda code are exported as the static fields `f$0` and `a$0`.

Chapter 3

The SML-JVM Toolkit

This chapter presents the 'SML-JVM toolkit', a set of SML structures, datatypes and functions for representing Java bytecode and class files. It is used for generating Java bytecode and class files by the new compiler back-end described in Chapter 4.

The SML-JVM toolkit started out as a by-product of the compiler project, tailored for generating Java bytecode from the intermediate Lambda code of Moscow ML. It ended up being a separate library of modules, supporting representation and generation of the entire set of Java class file and bytecode constructs—also those that are not relevant in connection with compilation from SML.
The SML-JVM toolkit does *not* perform complete bytecode verification on generated classes. In other words, it is possible to generate class files that may not be accepted by Sun's bytecode verifier.

3.1 Toolkit Modules

These are the modules in the SML-JVM toolkit:

Classdecl	abstract class declaration
Jvmtype	JVM data types
Label	abstract labels (used for branch targets, etc.)
Localvar	abstract local variable indices
Bytecode	abstract bytecode instructions
Stackdepth	calculation of maximum operand stack depth (written by Peter Sestoft)
Constpool	abstract constant pool representation
Emitcode	conversion from abstract bytecode instructions to binary representation
Classfile	concrete class file structure

The source files for the SML-JVM toolkit modules are listed in Appendix E.

3.2 Class Representation

In the SML-JVM toolkit a Java class or interface is represented as a collection of field and method declarations. A class/interface declaration also specifies a set of 'access' flags, a direct superclass (except for class Object, the superclass of all other classes), a set of direct superinterfaces, and a set of attributes:

```
type class_decl =
    {flags:  access_flag list,  (* access flags *)
     this:   jclass,            (* this class/interface *)
     super:  jclass option,     (* direct superclass *)
     ifcs:   jclass list,       (* direct superinterfaces *)
     fdecls: field_decl list,   (* field declarations *)
     mdecls: method_decl list,  (* method declarations *)
     attrs:  attribute list}
```

A Java interface declaration is very similar to a class declaration; an interface simply carries the flag `ACCinterface`. Furthermore, the access flags specify whether the class/interface is public, final, and/or abstract.

An 'attribute' can be one of a number of standard attributes, e.g. specifying the source file of a class, but may also include non-standard information in arbitrary binary format. This permits the use of very flexible attributes for classes/interfaces, fields and methods. In general, a JVM must recognize the standard attributes defined in *The Java Virtual Machine Specification*[4], and must silently ignore any non-standard attributes that it does not recognize.

For each field declared in a class or interface the field name, type, access flags and attributes are specified:

```
type field_decl =
    {flags:  access_flag list,  (* access flags *)
     name:   string,            (* unqualified name *)
     ty:     jtype,             (* field type *)
     attrs:  attribute list}    (* field attributes *)
```

The field access flags specify whether the field is public, private or protected, and whether it is static, final, volatile, and/or transient. The field attributes may specify a constant value for the field, or may provide non-standard information about the field (in arbitrary format, as for classes/interfaces).

For example, a field declaration corresponding to the Java declaration

```
static final int answer = 42;
```

could be declared like this using the SML-JVM toolkit:

```
val x : field_decl =
    {flags = [ACCfinal, ACCstatic],
     name  = "answer",
     ty    = Tint,
     attrs = [CONSTVAL (Cint (Int32.fromInt 42))]}
```

Similarly, a method declaration specifies the method name, signature, access flags and attributes:

```
type method_decl =
    {flags:  access_flag list,  (* access flags *)
     name:   string,            (* unqualified name *)
     msig:   method_sig,        (* method signature *)
     attrs:  attribute list}    (* method attributes *)
```

The method signature specifies a list of parameter types and an optional return type; `NONE` means that the method does not return any value (a.k.a. `void`).

The method access flags specify whether the method is public, private or protected, and whether it is static, final, synchronized, native, and/or abstract. The method attributes may specify the Java bytecode for the method (if it is implemented by the class/interface), and may specify a set of exceptions that the method is declared to throw. Again, the method attributes may also include non-standard information about the method (in arbitrary format, as for classes/interfaces and fields).

For example, an 'empty' method `main` corresponding to the Java declaration

```
public static void main (String[] args) {}
```

could be declared like this using the SML-JVM toolkit:

```
val main : method_decl =
    {flags = [ACCpublic, ACCstatic],
     name  = "main",
     msig  = ([Tarray (Tclass String)], NONE),
     attrs = [CODE {stack  = 0,
                    locals = 1,
                    code   = [Jreturn],
                    hdls   = [],
                    attrs  = []}]
    }
```

Note that the `CODE` attribute includes a `stack` field, specifying the maximum operand stack depth for the method, and a `locals` field specifying the maximum number of local variables used by the method. The function `maxdepth` in the `Stackdepth` module (written by Peter Sestoft) can be used to calculate the maximum stack depth for the bytecodes of a method. Similarly, the function `Localvar.count` can be used to calculate the maximum number of local variables used by a method.

The `class_decl`, `field_decl`, and `method_decl` data types are defined in the `Classdecl` module of the SML-JVM toolkit.

3.3 Abstractions

The bytecodes for a method implementation is represented as a list of abstract bytecode instructions (of type `Bytecode.jvm_instr`), corresponding to an abstract view of the JVM instruction set. For example, the `jvm_instr` data type includes the pseudo-instruction `Jlabel` which has no counterpart in the actual instruction set of the JVM.

In the SML-JVM toolkit, labels are used for specifying target(s) of branching instructions, and for specifying scope and entry point of exception handlers. When the abstract bytecode instruction sequence is emitted the labels therein are converted to addresses in the resulting binary code (cf. Section 3.5).

Moreover, the `jvm_instr` data type contains a number of amalgamated JVM instructions. For example, the abstract instruction `Jiconst` represents all JVM instructions pushing a constant `int` value onto the operand stack. When the abstract bytecode sequence is emitted, the appropriate JVM instruction is chosen. The translation performed by the bytecode emitter is:

Abstract instruction	JVM instruction
`Jsconst`	`ldc` or `ldc_w`
`Jaload`	`aload_<n>`, `aload`, or `wide aload`
`Jastore`	`astore_<n>`, `astore`, or `wide astore`
`Jdconst`	`dconst_<n>` or `ldc2_w`
`Jdload`	`dload_<n>`, `dload`, or `wide dload`
`Jdstore`	`dstore_<n>`, `dstore`, or `wide dstore`
`Jfconst`	`fconst_<f>`, `ldc`, or `ldc_w`
`Jfload`	`fload_<n>`, `fload`, or `wide fload`
`Jfstore`	`fstore_<n>`, `fstore`, or `wide fstore`
`Jgoto`	`goto` or `goto_w`
`Jiconst`	`iconst_<i>`, `bipush`, `sipush`, `ldc`, or `ldc_w`
`Jiinc`	`iinc` or `wide iinc`
`Jiload`	`iload_<n>`, `iload`, or `wide iload`
`Jistore`	`istore_<n>`, `istore`, or `wide istore`
`Jjsr`	`jsr` or `jsr_w`
`Jlconst`	`lconst_<l>` or `ldc2_w`
`Jlload`	`lload_<n>`, `lload`, or `wide lload`
`Jlstore`	`lstore_<n>`, `lstore`, or `wide lstore`
`Jnewarray`	`newarray`, `anewarray`, or `multianewarray`
`Jret`	`ret` or `wide ret`
`Jreturn`	`areturn`, `dreturn`, `freturn`, `ireturn`, `lreturn`, or `return`

The bytecode emitter translates each of the above abstract bytecode instructions into the most compact of the corresponding JVM bytecode instructions.

By introducing this level of abstraction over the instruction set of the JVM, the user of the SML-JVM toolkit is relieved from worrying about details in the JVM instruction set, e.g. which of the JVM instructions `iconst_<i>`, `bipush`, `sipush`, and `ldc` should be used for pushing an integer constant onto the operand stack. The user just inserts an abstract `Jiconst` instruction with an `Int32.int` argument, and the bytecode emitter then chooses the most compact of the corresponding JVM instructions based on the value of the immediate integer argument.

All other abstract bytecode instructions than those listed above are mapped directly to the corresponding JVM instructions. For example, the abstract instruction `Jdup` simply maps to the JVM instruction `dup` (represented in binary form as opcode 89).

3.4 Generating A Class File

Generation of a physical Java class file from an abstract class/interface declaration is implemented in the function `Classfile.emit`. This function takes as arguments a 'writer' function, an initial constant pool, and a class declaration. The class declaration is then processed in two steps:

1. the abstract `class_decl` is converted to a more concrete `class_file` record,
2. the `class_file` record is output, e.g. to a physical file, using the 'writer' function for writing each single byte of output to the file.

A `class_file` record represents a class/interface declaration in binary format:

- abstract bytecode instructions for methods implemented in the class have been 'emitted' to binary form (cf. Section 3.5),
- constants and references to fields, methods and classes have been inserted into the constant pool (cf. Section 3.5.1),
- labels have been resolved to instruction addresses (cf. Section 3.5.2), and
- access flags for the class/interface and its members have been converted to binary representation.

The reason for generating a class file in two steps is that the constant pool for the resulting class file must be built before the file is actually written. The constant pool goes at the beginning of a binary class file, and its size is not known until it has been completed, i.e., until all constants and references have been inserted. Hence, the constant pool must be completed before the remaining parts of the class file can be output.

The constant pool argument to `Classfile.emit` is used as the basis of the constant pool of the resulting class file. This way, data referenced by non-standard attributes of the class/interface declaration can be included in the resulting class file. Such attributes would include, as part of their binary content, the constant pool indices of the referenced data. In case a class or interface uses only standard attributes, an empty constant pool should be passed as argument to `Classfile.emit`.

3.5 Code Emission

The conversion from abstract bytecode to binary representation of JVM bytecode instructions is implemented in the function `Emitcode.emit`. This function translates the abstract bytecode for a method to a byte vector holding the binary representation of the corresponding JVM bytecode instructions.
The translation involves mapping the abstract bytecode instructions to the actual JVM instructions (cf. Section 3.3), and for each abstract bytecode instruction choosing the most compact JVM instruction:

- For each `Jsconst` abstract instruction, the argument (a string literal) is inserted into the constant pool, and one of the JVM instructions `ldc` and `ldc_w` is emitted, with the resulting constant pool index as argument. If the constant pool index is less than 256 then `ldc` is used, otherwise `ldc_w` is used.
- For each `Jaload`, `Jdload`, `Jfload`, `Jiload`, and `Jlload` abstract instruction, if the index of the local variable being accessed is between 0 and 3, inclusive, then the corresponding JVM load instruction with an implicit immediate operand is emitted. Otherwise, if the index is between 4 and 255, inclusive, then the corresponding JVM load instruction with an immediate operand is emitted. Otherwise the corresponding `wide` JVM load instruction is emitted.
- For each `Jastore`, `Jdstore`, `Jfstore`, `Jistore`, and `Jlstore` abstract instruction, if the index of the local variable being accessed is between 0 and 3, inclusive, then the corresponding JVM store instruction with an implicit immediate operand is emitted. Otherwise, if the index is between 4 and 255, inclusive, then the corresponding JVM store instruction with an immediate operand is emitted. Otherwise the corresponding `wide` JVM store instruction is emitted.

- For each `Jdconst` abstract instruction, if the argument is 0.0 or 1.0 then the corresponding JVM instruction with an implicit immediate operand is emitted. Otherwise the argument is inserted into the constant pool, and the JVM instruction `ldc2_w` is emitted with the resulting constant pool index as argument.
- For each `Jfconst` abstract instruction, if the argument is 0.0, 1.0, or 2.0 then the corresponding JVM instruction with an implicit immediate operand is emitted. Otherwise the argument is inserted into the constant pool, and one of the JVM instructions `ldc` and `ldc_w` is emitted, depending on the resulting constant pool index.
- For each `Jgoto` abstract instruction, if the argument label has been resolved to an address, and if the instruction offset for that address is between -32768 and 32767, inclusive, then the JVM instruction `goto` is emitted. Otherwise the JVM instruction `goto_w` is emitted. See also Section 3.5.2.
- For each `Jiconst` abstract instruction, if the argument is between -1 and 5, inclusive, then the corresponding JVM instruction with an implicit immediate operand is emitted. Otherwise, if the argument is between -128 and 127, inclusive, then the JVM instruction `bipush` is emitted. Otherwise, if the argument is between -32768 and 32767, inclusive, then the JVM instruction `sipush` is emitted. Otherwise the argument is inserted into the constant pool, and one of the JVM instructions `ldc` and `ldc_w` is emitted, depending on the resulting constant pool index.
- For each `Jiinc` abstract instruction, if the index of the local variable being accessed is between 0 and 255, inclusive, and if the increment argument is between -128 and 127, inclusive, then the JVM `iinc` instruction is emitted. Otherwise, the JVM instruction `wide iinc` is emitted.
- For each `Jret` abstract instruction, if the index of the local variable to be loaded is between 0 and 255, inclusive, then the JVM instruction `ret` is emitted. Otherwise the JVM instruction `wide ret` is emitted.
- For each `Jjsr` abstract instruction, if the argument label has been resolved to an address, and if the offset is between -32768 and 32767, inclusive, then the JVM instruction `jsr` is emitted. Otherwise the JVM instruction `jsr_w` is emitted. See also Section 3.5.2.
- For each `Jlconst` abstract instruction, if the argument is 0 or 1 then the corresponding JVM instruction with an implicit immediate operand is emitted. Otherwise the argument is inserted into the constant pool, and the JVM instruction `ldc2_w` is emitted with the resulting constant pool index as argument.
- For each `Jnewarray` abstract instruction, if more than one dimension of an array is to be created at once then the JVM instruction `multianewarray` is emitted. Otherwise one of the JVM instructions `newarray` and `anewarray` is emitted, depending on the component type of the array (if the component type is a simple type `newarray` is used, otherwise `anewarray` is used).
- For each `Jreturn` abstract instruction, the JVM method return instruction corresponding to the return type of the method is emitted[1].

The binary representation of the emitted JVM instructions and their arguments is collected in a byte array (`Word8Array.array`). This array is created with the maximum bytecode size

[1]The six different JVM method return instructions `areturn`, `dreturn`, `freturn`, `ireturn`, `lreturn`, and `return` are examples of redundancy in the JVM instruction set. These 'typed' instructions could readily be merged into one 'untyped' return instruction, and the JVM would then return a value of the proper type based on the return type of the method.

for a method: 65,535 bytes. When all abstract bytecode instructions for the method have been translated, a vector of JVM bytecodes is extracted from the byte array, corresponding to the part of the array that has actually been used. This vector of JVM bytecodes is output as method code when the class declaration is written to a class file (once bytecode has been generated for all methods of the class).

3.5.1 Building the Constant Pool

The `Constpool` module implements the `pool` data type and related operations. The implementation uses a hash table to achieve maximum sharing: when a constant or reference is inserted into the pool, it is first checked whether an identical value is already in the pool. If so, the index of that entry is returned; otherwise the value is inserted into the constant pool, and the fresh index of the new entry is returned.

In the process of translating the sequence of abstract bytecode instructions to a byte vector, the bytecode emitter builds the constant pool for the target class. As mentioned, emission of an `Jsconst`, `Jdconst`, `Jfconst`, `Jiconst`, or `Jlconst` abstract instruction may lead to a constant being inserted into the constant pool. In that case, one of the JVM instructions loading a constant pool entry is emitted, with the index of the constant pool entry as argument.

Similarly, all symbolic references to classes, fields and methods are inserted into the constant pool when the referencing instruction is emitted, and the index of the resulting constant pool entry is emitted as an argument to the instruction.

3.5.2 Label Resolution

The bytecode emitter also implements resolution of abstract bytecode labels to addresses in the target JVM bytecode. This is done via a hash table representing a map from a label to its 'target status', which is one of

`PENDING(ref phs)` meaning that a list `phs` of references to the label has been encountered, but the label has not yet been resolved to an address

`RESOLVED addr` meaning that the label has been resolved to the address `addr`.

When a reference to a label is encountered, if the label has already been resolved to an address, then the target JVM branching instruction is emitted with that address as argument. Otherwise the target JVM branching instruction is emitted with a dummy argument (placeholder), and a reference to the placeholder is added to the list of `PENDING` references to the label (if any).

When a `Jlabel` abstract bytecode instruction is encountered, if there are any `PENDING` references to the label, these are backpatched to the address of the following target JVM instruction. The label is then marked as `RESOLVED` in the label map. No target JVM bytecode is emitted for the `Jlabel` instruction itself.

The bytecode emitter is more conservative than strictly necessary with respect to the abstract instructions `Jgoto` and `Jjsr`: only in case the branch target lies before the branch instruction will the most compact of the JVM instructions `goto` and `goto_w` or `jsr` and `jsr_w` be chosen. When the branch target lies after the branching instruction itself the JVM instruction `goto_w` or `jsr_w` will always be used. This is because bytecode emission is implemented as a single pass over the abstract bytecode sequence. Using a more advanced multi-pass algorithm would make it possible to always choose the most compact JVM branching instruction.

3.6 Auxiliary Modules

In order to support the full range of constants defined in the Java class file format [4], 16-bit, 32-bit and 64-bit integer and floating-point constants are used in a number of places in the SML-JVM toolkit. An example of this was shown in the field declaration example of Section 3.2 where the integer argument to the `Cint` constructor is of type `Int32.int`.

Moscow ML provides only one `Int` structure (31-bit signed integers), one `Word` structure (31-bit unsigned integers), and one `Real` structure (64-bit floating-point values). Hence, a set of utility modules has been implemented:

Int32	32-bit signed integers
Int64	64-bit signed integers
Real32	32-bit floating-point values
Real64	64-bit floating-point values
Word16	16-bit unsigned integers
Word32	32-bit unsigned integers
Word64	64-bit unsigned integers

These modules provide a limited subset of the corresponding SML Basis Library modules[3], primarily conversion from the usual integer and real types (`Int.int`, `Word.word`, and `Real.real`). Additionally, functions for conversion to byte vectors are provided.

The auxiliary modules are used in the bytecode emitter, and in the conversion from abstract class declarations to concrete `class_file` records (implemented in `Classfile.emit`).

The source files for the auxiliary modules are listed in Appendix E along with the other source files for the SML-JVM toolkit.

Chapter 4

The New Back-End

This chapter describes the implementation of the new compiler back-end. The structure of the back-end is outlined, and the general, unoptimized compilation of intermediate Lambda code to Java bytecode is explained. Implemented optimizations are described in Chapter 5.

4.1 Back-End Modules

These are the modules in the new compiler back-end:

`Codeutil`	Utility functions for generating bytecode
`Coercion`	Coercion between different run-time representations of values
`Freeenv`	Free variable environment
`Genclass`	Generation of Java class file from intermediate target code
`Gencode`	Generation of Java bytecode from Lambda code
`Instantiate`	Utility functions for instantiation of run-time support classes
`Jvmcode`	Intermediate representation of target code
`Lift`	Lifting of Lambda code
`Localenv`	Local variable environment
`Runtype`	Representation of run-time types
`Smlclasses`	References to run-time support classes
`Tag`	Representation of closure tags

The source files for the new compiler back-end are listed in Appendix F.

The following modules of the existing Moscow ML compiler have been modified for the new compiler:

`Const`	Added function `compareUids` for comparing `unique_id` values
`Compiler`	Significant changes to function `compileAndEmit`
`Hasht`	Fixed a bug with function `peek`
`Lambda`	Changed arguments to `Lshared`
`Prim`	Changed argument to the `Pclosure` primitive
`Pr_lam`	Printing Lambda expressions directly to output
`Printexc`	Reporting uncaught exceptions
`Sigmtch`	Changed function `matchSignature`

The modified source files `Compiler.sml` and `Sigmtch.sml` are listed in Appendix F along with the source files for the new back-end.

The modified function of `Compiler.compileAndEmit` is described in Section 4.2.

The type declaration

```
datatype shared_lam_state =
    TODO
  | LIFTED
  | COMPILED of Label.label
  | SPECIALIZED of Label.label
```

has been introduced in module `Lambda`. This is used in the `Lshared` clause of the data type `Lambda.Lambda`, which has been modified to carry a reference to a `shared_lam_state` mark rather than a reference to an integer:

```
| Lshared of Lambda ref * shared_lam_state ref
```

Initially, all `Lshared` expressions are marked as `TODO`. In the process of lifting the Lambda code, each `Lshared` clause is marked as `LIFTED` to indicate that the sub-expression has been lifted (see Section 4.3). The `COMPILED` mark is used in the code generator to indicate that code has been generated for the sub-expression (see Section 4.6.15). The `SPECIALIZED` mark is used only in connection with closure specialization; see Section 5.3.

The `Pclosure` clause of the data type `Prim.primitive` has been modified to carry a closure tag and additional closure information rather than a pair of integers:

```
| Pclosure of Tag.tag * Runtype.spec_clos
```

Using values of an abstract type such as `Tag.tag` instead of `int` values as closure tags seems appropriate. The extra closure information is used for closure specialization; see Section 5.3.

Function `Sigmtch.matchSignature` has been modified to return a list of Lambda expressions for various rebindings, rather than writing directly to the target file.

Compilation of an explicit signature, that is, generation of a `.ui` file from a `.sig` file, has been adopted directly and unchanged from the Moscow ML compiler.

4.2 Structure of the New Back-End

Compilation of a module (i.e., an `.sml` file) to a Java class file is controlled by the function `Compiler.compileAndEmit`. This function takes as arguments the name of the module, an optional signature specification, and a list of parsed top-level declarations. The following steps are performed to compile the module:

- Functions in the compiler front-end are invoked to compile each top-level declaration to a list of Lambda expressions. This results in a list of lists of Lambda expressions; the Lambda expressions appear in reversed order, as compared to the original top-level declarations of the module.
- The function `Compiler.checkSig` is invoked to check the signature of the module. If an explicit signature (i.e., a corresponding `.sig` file) was found, then the module is matched against this signature; otherwise an encoding of the inferred signature for the module

is written to the corresponding `.ui` file. Signature matching produces an export environment (see Section 4.4) and a possibly empty list of Lambda expressions for required rebindings of exported values in the module. The Lambda expressions are added to the list of lists of Lambda expressions produced by the front-end.

- The function `Lift.liftLambdas` is folded over the list of lists of Lambda expressions (see Section 4.3). The result of lifting is a list of top-level Lambda expressions and a list of 'pending' closures. The lifted Lambda expressions appear in reversed order, as compared to the original top-level declarations of the module. Each pending closure represents a lifted function.
- The function `Gencode.compileTopLevel` is invoked to generate Java bytecode for the top-level Lambda expressions.
- The function `Gencode.compileClosures` is invoked to generate Java bytecode for the pending closures.
- The function `Genclass.genClass` is invoked to generate a Java class file from the intermediate representation of target code; see Section 4.7.

The functions `Gencode.compileTopLevel` and `Gencode.compileClosures` serve as 'entry points' of the code generator, and both of these invoke the function `Gencode.compileExpr` to compile Lambda code to Java bytecode.

In `compileTopLevel`, `compileExpr` is folded over the list of top-level Lambda expressions to generate Java bytecode for the top-level expressions of the module. Code is generated backwards, and this implies that generated instructions may simply be added to the head of the list of target instructions. The resulting bytecode goes into method `<clinit>` of the target class[1].

In `compileClosures`, `compileExpr` is folded over the list of Lambda expressions for each pending closure. The resulting bytecode goes into method `apply` of the target class (cf. Section 2.7). Again, code is generated backwards.

4.3 Lifting

As in the original back-end of the Moscow ML compiler, the intermediate Lambda code of the new compiler is 'lifted' before code generation proper. In the new back-end, lifting is implemented in the function `Lift.liftLambda`. Lifting is done per Lambda expression, by recursively traversing the sub-expressions. The result is a Lambda expression where

- the de Bruijn index of each `Lvar` expression has been shifted to be relative to the innermost enclosing Lambda abstraction (`Lfn`),
- each `Lfn` expression has been lifted to top-level, and has been replaced by a corresponding `Pclosure` primitive,
- each `Lshared` expression has been marked as `LIFTED`,
- each `Lhandle` expression appearing in a value context has been 'eta-expanded' and lifted into the body of a separate function.

[1]Method `<clinit>` corresponds to the static initializer `static {...}` of a Java class. When the class is loaded, the JVM automatically invokes this method to initialize the class.

The resulting lifted Lambda expression thus represents only the 'top-level fraction' of the original Lambda expression.

A lifted expression `Lvar(0)` refers to the parameter of the enclosing Lambda abstraction; `Lvar(1)`, `Lvar(2)`, etc. refer to let-bound or letrec-bound variables within the Lambda abstraction; and `Lvar(`i`)`, where $i < 0$, refers to a free variable of the Lambda abstraction.

Lifting a Lambda abstraction `Lfn(`e`)` means replacing it with an expression like this:

```
Lprim (Pclosure (t, s), [v_1, v_2, ..., v_n])
```

where t is a fresh closure tag, s is the closure information for the function (used in optimizations), and $v_1, v_2, \ldots, v_n$ are the free variables of the function.

The bodies of lifted functions are collected in a list of 'pending' closures. This list includes for each function the closure tag, the number of arguments, the free variable environment, and the lifted Lambda code of the function body.

The marks `TODO` and `LIFTED` are used in the lifting of `Lshared` expressions. When an `Lshared` expression marked as `TODO` is encountered, the expression is marked as `LIFTED`, and the sub-expression is lifted (recursively). The next time this expression is encountered, via another `Lshared` clause referencing the same sub-expression, the `shared_lam_state` tag will already be `LIFTED`, and the `Lshared` clause remains unchanged.

An `Lhandle` Lambda expression may appear in a value context, as described in Section 2.3. In order to preserve the value context, the `Lhandle` expression is lifted into the body of a new function, and replaced by a Lambda expression like this:

```
Lapply(Lprim (Pclosure (t, s), [v_1, v_2, ..., v_n]), [])
```

where t is the closure tag corresponding to the new function, s is the closure information for the function (used in optimizations), and $v_1, v_2, \ldots, v_n$ are the free variables of the original `Lhandle` expression.

This way, an exception being raised in the sub-expression covered by the `Lhandle` expression will not affect the value context of the original `Lhandle` expression. Note that the new function is applied to an empty list of arguments[2].

4.4 Environments

A number of compile-time environments are used in the new compiler back-end. Some of these are used only to implement various optimizations, and are described in Chapter 5. The general compilation described in the following uses three environments: an export environment, *export*, a free variable environment, *free*, and a local variable environment, *local*.

The *export* environment maps an exported top-level identifier to the unique name under which it is to be exported. This environment is built prior to code generation, in connection with signature matching.

[2]When an `Lhandle` expression is lifted into a separate funtion to preserve the value context, the function is in fact compiled to a separate JVM method. This method may be invoked with `Jinvokestatic` and need not take any arguments, except for the free variables of the original `Lhandle` expression (if any).

The *free* environment maps a `Lvar` index of a free variable to its location (index) in the vector of free variables in the closure. A separate free variable environment is used in the compilation of each closure body. This environment is built in the lifting phase, in connection with the lifting of the original `Lfn` expression (cf. Section 4.3).

The *local* environment maps the `Lvar` index of Lambda variable to the corresponding JVM local variable index. Initially, the *local* environment is empty. The environment is updated during code generation every time a value is bound to a local variable, e.g. in connection with a `Llet` expression.

During generation of bytecode for top-level Lambda expressions, the first local variable binding uses JVM local variable index 0. When bytecode is generated for Lambda expressions in a closure body, the first local variable binding, i.e., the argument of the closure, uses JVM local variable index 1. This is because the JVM local variable at index 0 holds a reference to the closure itself (a.k.a. `this`).

4.5 Intermediate Target Code

During code generation the intermediate results are gathered in a record of this type, declared in module `Jvmcode`:

```
type jvm_code = {class     : Jvmtype.jclass,
                 names     : (string, name_info) Binarymap.dict ref,
                 exns      : Const.unique_id Binaryset.set ref,
                 clinit    : clinit_info ref,
                 apply     : apply_info ref,
                 ... }
```

The `class` component is the identifier of the target class.

The `names` component maps a top-level name to information about that name; this information is used for optimizations described in Chapter 5. The domain of the map is also used for determining which static field declarations the target class should include (cf. Section 4.7).

The `exns` component of the `jvm_code` record holds a set of identifiers for declared top-level exceptions. This is used for initializing exception tag fields in the target class (cf. Section 4.7).

The `clinit` component of the `jvm_code` record holds bytecode, local variable environment, and exception handler declarations for the `<clinit>` method.

The `apply` component of the `jvm_code` record holds bytecode, local variable environment, exception handler declarations, and closure body entries for the `apply` method (cf. Section 2.7)

The `jvm_code` record also holds other information used for implementing various optimizations; see Chapter 5.

4.6 Code Generation

Generation of abstract Java bytecode from Lambda code is implemented in the function `Gencode.compileExpr`. In the following sections, it is described how each Lambda clause is compiled to Java bytecode, using the abstract instruction set defined in the SML-JVM toolkit (cf. Section 3.3. The compilation shown is the general, direct translation without optimizations. A number of optimizations have also been implemented; these are described in Chapter 5.

The generated Java bytecode and Java class files expect the run-time support classes for representation of SML values to be located in the Java package `sml.lang`.

A relaxed notation is used in examples showing bytecode sequences generated by the code generator of the new back-end. The bytecodes are listed as sequences of pseudo-instructions akin to the abstract instructions of the SML-JVM toolkit.

4.6.1 Lvar

A Lambda expression `Lvar(`i`)`, where $i \geq 0$, that is, a locally bound variable, is compiled to:

```
Jaload local(i)
```

This bytecode instruction loads the local variable number *local*(i), where *local*(i) is the JVM local variable index that i maps to in the current local environment (cf. Section 4.4).

A Lambda expression `Lvar(`i`)`, where $i < 0$, that is, a free variable, is compiled to the following bytecode:

```
Jaload 0
Jgetfield Closure.free
Jiconst free(i)
Jaaload
```

This bytecode sequence performs the following steps:

1. Loads the value of local variable 0 onto the stack. Assuming that a reference to a free variable occurs within the body of a closure, local variable 0 holds a reference to the current `Closure` object (a.k.a. `this`).
2. Loads the value of the instance field `free` of the `Closure` object onto the stack. This is an array of free variables for the closure. The `Jgetfield` instruction consumes the object reference loaded in the previous step.
3. Pushes the index *free*(i) of the requested free variable onto the stack.
4. Loads component number *free*(i) of the array onto the stack. This consumes the array reference and integer constant loaded in previous steps.

The net effect of the above bytecode sequence is that of pushing the value of the free variable number i onto the stack.

4.6.2 Lconst

An `Lconst` clause represents an atomic or a structured constant. An atomic constant is compiled to bytecode creating a wrapper object for the constant. For example, the Lambda expression `Lconst (ATOMsc (INTscon 29))` is compiled to:

```
Jnew Integer
Jdup
Jiconst 29
Jinvokespecial Integer.<init>(int)
```

A structured constant is compiled to bytecode creating an instance of class `Tuple`, `Constructor`, or `Exception`, depending on the tag and span of the constant. If the constant has a constructor tag and span 1 then a `Tuple` object is created; if it has a constructor tag and span 2 or more then a `Constructor` object is created; otherwise, the constant has an exception tag, and an `Exception` object is created.

For example, the Lambda expression

```
Lconst(BLOCKsc(CONtag(1,2), [ATOMsc(INTscon 111), BLOCKsc(CONtag(0,2),[])]))
```

corresponding to the SML expression

```
111::[]
```

is compiled to:

```
Jnew Constructor
Jdup
Jiconst 1
Jiconst 2
Jnewarray {elem = Object, dim = 1}
Jdup
Jiconst 0
Jnew Integer
Jdup
Jiconst 111
Jinvokespecial Integer.<init>(int)
Jaastore
Jdup
Jiconst 1
Jnew Constructor
Jdup
Jiconst 0
Jinvokespecial Constructor.<init>(int)
Jaastore
Jinvokespecial Constructor.<init>(int,Object[])
```

This bytecode sequence creates a `Constructor` instance, and invokes the initialization method `<init>(int,Object[])` to initialize it. The arguments to the initialization method are the tag value 1 and an array holding the actual arguments to the `Constructor` object. An `Integer` object, holding the value 111, is stored into index 0 of the array, and another `Constructor` object, initialized with tag value 0 and no arguments[3], is stored into index 1 of the array.

4.6.3 Lapply

A Lambda expression `Lapply(`e_0`, [`e_1`])`, representing application of a function to an argument, is compiled to[4]:

[3]A special initialization method, `Constructor.<init>(int)`, is used for initializing a `Constructor` object corresponding to an SML constructor with no arguments. This method takes only one argument, namely, the constructor tag. Using this method for initializing a constructor with no arguments is more efficient than invoking the general initialization method `<init>(int,Object[])` of class `Constructor`. See also section 5.2.

[4]The notation <x> is used for indicating the instruction sequence evaluating the value of expression x.

```
<e0>
Jcheckcast Closure
<e1>
Jinvokevirtual Closure.apply(Object)
```

This bytecode sequence applies the value of e_0, assumed to be a closure, to the value of the argument e_1. Note the `Jcheckcast` instruction which ensures (at run-time) that the value of e_0 is really a `Closure` object. In general, this is necessary to make Sun's bytecode verifier accept the generated code. See also Section 7.2.

More generally, a Lambda expression `Lapply(`e_0`, [`$e_1, e_2, \ldots, e_n$`])` is compiled to bytecode applying the value of e_0, assumed to be a closure, to the values of the arguments $e_1, e_2, \ldots, e_n$:

```
<e0>
Jcheckcast Closure
<e1>
Jinvokevirtual Closure.apply(Object)
Jcheckcast Closure
<e2>
Jinvokevirtual Closure.apply(Object)
...
Jcheckcast Closure
<en>
Jinvokevirtual Closure.apply(Object)
```

Note that application of a curried function to two or more of its arguments ($n > 1$) is translated to a binary application scheme: $(\ldots((e_0 e_1)e2)e_n)$. The value of e_0 is first applied to the value of e_1; the result of this application, assumed to be another closure, is then applied to the value of e_2, and so forth.

4.6.4 Lfn

The code generator does not compile an `Lfn` Lambda clause into Java bytecode. This is because lifting should have replaced each `Lfn` expression with a `Pclosure` primitive (cf. Section 4.3). Hence, if the code generator encounters an `Lfn` expression, an exception indicating an internal error is raised.

4.6.5 Llet

A Lambda expression `Llet([`e_1`], `e_0`)` is compiled to:

```
<e1>
Jastore local(d)
<e0>
```

This bytecode sequence stores the value of e_1 into local variable number *local*(d), where d is the first unused variable index in the context of the `Llet` expression, and then evaluates e_0.

Before generating the bytecode for e_0, the current local variable environment is augmented with a binding for variable number d. This is necessary for enabling references in e_0 to the value of e_1 via variable number d.

In general, a Lambda expression `Llet([`$e_1, e_2, \ldots, e_n$`], `e_0`)` is compiled to bytecode storing the values of $e_1, e_2, \ldots, e_n$ into local variables before evaluating e_0:

```
<$e_1$>
Jastore $local(d)$
<$e_2$>
Jastore $local(d+1)$
...
<$e_n$>
Jastore $local(d+n-1)$
<$e_0$>
```

Prior to generation of bytecode for e_0, the current local variable environment is augmented with bindings for variable d through $d+n-1$, where d is the first unused variable index in the context of the `Llet` expression.

4.6.6 Lletrec

A Lambda expression `Lletrec([`e_1`], `e_0`)`, where e_1 is assumed to be a `Pclosure` primitive with tag t_1 and free variables $v_1, v_2, \ldots, v_k$, is compiled to:

```
Jnew $c$
Jdup
Jiconst $t_1$
Jinvokespecial $c$.<init>(int)
Jdup
Jastore $local(d)$
Jiconst $k$
Jnewarray {elem = Object, dim = 1}
Jdup
Jiconst 0
<$v_1$>
Jaastore
Jdup
Jiconst 1
<$v_2$>
Jaastore
...
Jdup
Jiconst $k-1$
<$v_k$>
Jaastore
Jputfield Closure.free
<$e_0$>
```

This bytecode sequence first creates an instance of the target class c, assumed to be a subclass of `Closure`, and initializes it with tag t_1 and no free variables[5]. The fresh closure object is then stored into local variable number $local(d)$, where d is the first unused variable index in the context of the `Lletrec` expression. An array containing the free variables of the closure is then created and stored into the instance field `free` of the closure object. Finally, e_0 is evaluated.

[5]A special initialization method, c.`<init>(int)`, is used for initializing the 'intermediate' closure object. This method takes only one argument, namely, the closure tag. Using this method for initializing a closure object with no free variables is more efficient than invoking the general initialization method `<init>(int,Object[])`.

Before generating the bytecode for initializing the free variables of the closure, and for evaluating e_0, the current local variable environment is augmented with a binding for variable number d. This way, a closure referring to itself via one of its free variables, i.e., representing a recursive function, may be created.

More generally, a Lambda expression `Lletrec([`$e_1, e_2, \ldots, e_n$`], `e_0`)`, binding mutually recursive functions, is compiled to bytecode creating intermediate closure objects for $e_1, e_2, \ldots, e_n$, binding these to JVM local variables, initializing the free variables of the closures[6], and then evaluating e_0:

```
Jnew c
Jdup
Jiconst t_1
Jinvokespecial c.<init>(int)
Jastore local(d)
Jnew c
Jdup
Jiconst t_2
Jinvokespecial c.<init>(int)
Jastore local(d + 1)
...
Jnew c
Jdup
Jiconst t_n
Jinvokespecial c.<init>(int)
Jdup
Jastore local(d + n − 1)
<free vars array for closure t_n>
Jputfield Closure.free
...
Jaload local(d + 1)
<free vars array for closure t_2>
Jputfield Closure.free
Jaload local(d)
<free vars array for closure t_1>
Jputfield Closure.free
<e_0>
```

SML semantics guarantees that $e_1, e_2, \ldots, e_n$ are closure-building expressions, that is, `Pclosure` primitives. It is assumed that the corresponding closure tags are $t_1, t_2, \ldots, t_n$.

Prior to generation of bytecode for initializing the free variables and evaluating e_0, the current local variable environment is augmented with bindings for variable d through $d+n-1$, where d is the first unused variable in the context of the `Lletrec` expression.

Binding all intermediate closure objects to local variables before initializing the free variables of the closures is necessary for creating closures that refer to each other, that is, closures for mutually recursive functions.

[6]The notation `<free vars array for closure `t_i`>` is used as an abbreviation for a bytecode sequence that creates an array holding the free variables for the closure with tag t_i.

4.6.7 Lprim

In general, a Lambda expression `Lprim(`p`, [`$e_1, e_2, \ldots, e_n$`])` is compiled to bytecode performing the following steps:

1. Evaluating and unwrapping each of the arguments $e_1, e_2, \ldots, e_n$.
2. Executing one or more instructions corresponding to the primitive p.
3. Wrapping the resulting value (if any) into an object.

Pget_global and Pset_global

A Lambda expression `Lprim(Pget_global` *uid*`, [])` is compiled to bytecode loading the value of a static field f:

```
Jgetstatic f
```

where f is a reference to the static field corresponding to the unique top-level identifier *uid*. However, if *uid* is an exported top-level identifier, then the field reference f will be based on the identifier *export*(*uid*) that *uid* maps to in the export environment (cf. Section 4.4).

The names of top-level identifiers must be mangled because the Java class file format does not permit all valid SML identifiers. The translation is simple: alphanumeric characters are left unchanged, whereas each non-alphanumeric character is replaced by `$xx`, where `xx` is the hexadecimal representation of the ordinal value of the character. Moreover, the stamp of the SML identifier is appended to the name after an additional `$` character. For example, the SML top-level identifer `Foo.bar'` with stamp 13 is translated to the field name `Foo.bar$27$13`. This name mangling scheme is safe in the sense that it cannot introduce name clashes.

A Lambda expression `Lprim(Pset_global` *uid*`,` e`)` is compiled to bytecode storing the value of expression e into a static field f:

```
<e>
Jputstatic f
```

where f is a reference to the static field corresponding to the unique identifier *uid*, as above. Again, if *uid* is an exported top-level identifier, then f will be based on the identifier *export*(*uid*) instead.

For each `Pset_global` primitive in the Lambda code, the name of the static field being accessed is recorded, in order to produce the required field declarations in the target class (cf. Section 4.7).

Ptest

A `Ptest` primitive is compiled to bytecode performing the test and pushing a constructor value corresponding to `false` or `true` onto the stack. For example, a Lambda expression `Lprim(Ptest (Pint_test PTeq), [`e_1, e_2`]))` is compiled to:

```
<e1>
Jcheckcast Number
Jinvokevirtual Number.intValue()
<e2>
Jcheckcast Number
Jinvokevirtual Number.intValue()
Jif_icmpne lbl'
Jgetstatic Constructor.One
Jgoto lbl
Jlabel lbl'
Jgetstatic Constructor.Zero
Jlabel lbl
```

It is assumed that e_1 as well as e_2 evaluates to a `Number` object[7]. Note that the 'inverted' test is used in the bytecode: instead of testing for equality, the generated bytecode branches to label lbl' if the values of e_1 and e_2 differ. If the values are equal, the static field `Constructor.One`, representing `true`, is pushed onto the stack; otherwise `Constructor.Zero` is stacked[8]. The labels lbl and lbl' are fresh labels.

If the test appears as the conditional of an `Lif` or `Lwhile` expression, or appears inside an `Landalso` or `Lorelse` expression, then the above bytecode sequence may be optimized; see Section 5.4.1.

Similarly, a Lambda expression `Lprim(Ptest (Pfloat_test PTeq), [`e_1, e_2`]))` is compiled to:

```
<e1>
Jcheckcast Number
Jinvokevirtual Number.doubleValue()
<e2>
Jcheckcast Number
Jinvokevirtual Number.doubleValue()
Jdcmpg
Jifne lbl'
Jgetstatic Constructor.One
Jgoto lbl
Jlabel lbl'
Jgetstatic Constructor.Zero
Jlabel lbl
```

The only significant difference is that two instructions must be used for comparison of values of JVM type `double`[9].

[7] The standard Java class `Number` is a common superclass of the `Integer` and `Double` classes. Class `Number` declares the instance methods `intValue` and `doubleValue` for extracting the wrapped value as an `int` value and a `double` value, respectively. Hence, `Number.intValue` may be used for unwrapping the integer value of an `Integer` object, for converting a `Double` object to an integer.

[8] A number of `Constructor` instances with no arguments are 'predefined' in class `Constructor`; see Section 5.1.1.

[9] The instruction `dcmpg` compares two `double` values and pushes the integer value -1, 0, or 1, corresponding to the first value being less than, equal to, or greater than the second value.

Block Primitives

A Lambda expression `Lprim(Pfield` n`, [`e`])` is compiled to bytecode loading the n'th argument from the value of e, assuming that e evaluates to a `Block` object, i.e., an instance of class `Tuple`, `Constructor`, `Ref`, or `Exception`:

```
<e>
Jcheckcast Block
Jgetfield Block.args
Jiconst n
Jaaload
```

As mentioned in Section 4.6.3, the `Jcheckcast` instruction is necessary to make Sun's bytecode verifier accept the code; see also Section 7.2.

A Lambda expression `Lprim(Psetfield` n`, [`e_1, e_2`])`, corresponding to assignment to a `ref` cell in SML, is compiled to:

```
<e1>
Jcheckcast Block
Jgetfield Block.args
Jiconst n
<e2>
Jaastore
Jgetstatic Tuple.Unit
```

This bytecode sequence stores the value of e_2 into the n'th component in the value of e_1, and loads the empty tuple[10]. Again, it is assumed that e_1 evaluates to a `Block` object, but the `Jcheckcast` is necessary to suit Sun's bytecode verifier.

The compilation of the `Pmakeblock` primitive is very similar to that of a `Lconst` Lambda clause representing a structured constant. The only difference is that the arguments to the `Pmakeblock` primitive are Lambda expressions rather than constants.

Pccall

In the Lambda code produced by the compiler front-end, polymorphic equality operations are represented as `Pccall` primitives.

In the new back-end, a Lambda expression `Lprim(Pccall ("sml_equal", 2), [`e_1, e_2`])` is compiled to bytecode comparing the values of e_1 and e_2 via the instance method `Object.equals`:

```
<e1>
<e2>
Jinvokevirtual Object.equals(Object)
Jifeq lbl'
Jgetstatic Constructor.One
Jgoto lbl
Jlabel lbl'
Jgetstatic Constructor.Zero
Jlabel lbl
```

[10]The static field `Unit` in class `Tuple` holds a 'predefined' empty tuple, a.k.a. `(): unit` in SML; see Section 5.1.1.

It is assumed that the `equals` method returns 0 or 1, corresponding to the logical values false and true, respectively. If the result from `equals` is 0, then the value of `Constructor.Zero`, corresponding to `false`, is pushed onto the stack; otherwise the value of `Constructor.One`, corresponding to `true`, is stacked.

The bytecode generated for a Lambda expression `Lprim(Pccall ("sml_not_equal", 2),` `[`e_1, e_2`])` is quite similar, except that the result from `Object.equals` is negated before branching:

```
<e1>
<e2>
Jinvokevirtual Object.equals(Object)
Jiconst 1
Jixor
Jifeq lbl'
Jgetstatic Constructor.One
Jgoto lbl
Jlabel lbl'
Jgetstatic Constructor.Zero
Jlabel lbl
```

No other `Pccall` primitives (i.e., external function calls) have been implemented in the new compiler back-end.

Pnot

A Lambda expression `Lprim(Pnot, [`e`])` is compiled to bytecode producing a constructor value corresponding to the boolean negation of the value of e:

```
<e>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jiconst 1
Jixor
Jifeq lbl'
Jgetstatic Constructor.One
Jgoto lbl
Jlabel lbl'
Jgetstatic Constructor.Zero
Jlabel lbl
```

It is assumed that e evaluates to a `Constructor` object, and that the value of its tag is either 0 or 1. If the tag is 0, the value of `Constructor.Zero` is pushed onto the stack, otherwise the value of `Constructor.One` is stacked.

If the boolean negation appears as the conditional of an `Lif` or `Lwhile` expression, or appears inside an `Landalso` or `Lorelse` expression, then the above bytecode sequence may be optimized; see Section 5.4.1.

Arithmetic Primitives

A Lambda expression `Lprim(Psmlnegint, [`e`])` is compiled to:

```
Jnew Integer
Jdup
<e>
Jcheckcast Number
Jinvokevirtual Number.intValue()
Jineg
Jinvokespecial Integer.<init>(int)
```

It is assumed that e evaluates to a `Number` object. The value of e is unwrapped and negated, and the result is wrapped into a fresh `Integer` object.

Similarly, a Lambda expression `Lprim(Pfloatprim Psmladdfloat,` $[e_1, e_2]$`)` is compiled to:

```
Jnew Double
Jdup
<e1>
Jcheckcast Number
Jinvokevirtual Number.doubleValue()
<e2>
Jcheckcast Number
Jinvokevirtual Number.doubleValue()
Jdadd
Jinvokespecial Double.<init>(double)
```

This bytecode sequence unwraps the values of e_1 and e_2 as `double` values, adds them, and wraps the sum into a fresh `Double` object.

String, Vector and Array Primitives

A Lambda expression `Lprim(Pstringlength,` $[e]$`)` is compiled to bytecode calculating the length of the value of e, assuming that e evaluates to a `String` object, and wrapping the result into an `Integer` object:

```
Jnew Integer
Jdup
<e>
Jcheckcast String
Jinvokevirtual String.length()
Jinvokespecial Integer.<init>(int)
```

A Lambda expression `Lprim(Pgetstringchar,` $[e_1, e_2]$) is compiled to:

```
Jnew Integer
Jdup
<e1>
Jcheckcast String
<e2>
Jcheckcast Number
Jinvokevirtual Number.intValue()
Jinvokevirtual String.charAt()
Jinvokespecial Integer.<init>(int)
```

This bytecode sequence loads the character at index e_2 in the value of e_1, assuming that e_1 evaluates to a `String` object and that e_2 evaluates to an `Integer` object. Note that the resulting character is wrapped into an `Integer` object.

In Moscow ML, the primitive `Psetstringchar` is used to implement certain library functions for manipulating character sequences. However, since an SML value of type `string` is represented as an immutable `String` object with the new back-end, this primitive is not implemented in the new compiler back-end.

The compilation of the vector/array primitives to bytecode is quite similar to that of string primitives, except that an SML vector or array is represented as a JVM array of objects[11].

Pclosure

The `Pclosure` primitive represents a closure-building Lambda expression (`Lfn`) that has been lifted to top-level (cf. Section 4.3).

A Lambda expression `Lprim(Pclosure` t`, [])`, i.e., a closure primitive with tag t and no free variables, is compiled to:

```
Jnew c
Jdup
Jiconst t
Jinvokespecial c.<init>(int)
```

This is quite similar to the creation of intermediate `Closure` objects shown for `Lletrec`.

A Lambda expression `Lprim(Pclosure` t`, [`$e_1, e_2, \ldots, e_n$`])` is compiled to:

```
Jnew c
Jdup
Jiconst t
Jiconst n
Jnewarray {elem = Object, dim = 1}
Jdup
Jiconst 0
<e1>
Jaastore
Jdup
Jiconst 1
<e2>
Jaastore
...
Jdup
Jiconst n − 1
<en>
Jaastore
Jinvokespecial c.<init>(int,Object[])
```

This bytecode sequence allocates a new instance of the target class c, assumed to be a subclass of `Closure`, and initializes it with tag t and an array holding the free variables $e_1, e_2, \ldots, e_n$.

[11]The representation of vectors must be changed to correctly implement the SML semantics of equality on vectors. Using a JVM array of objects is a mistake, because polymorphic equality of vectors thus corresponds to pointer equality, which is not the required semantics in SML.

Other Primitives

The compilation of a number of other primitives to bytecode is also implemented in the prototype compiler. The translation of those primitives is similar to the translations shown above, e.g.

- `Lprim(Pidentity, [`e`])` is compiled to bytecode evaluating e,
- `Lprim(Praise, [`e`])` is compiled to bytecode throwing the value of e as a JVM exception, assuming that e evaluates to an `Exception` object,
- `Lprim(Patom `n`, [])` is compiled to bytecode pushing a constructor with tag n and no arguments on the operand stack,
- `Lprim(Pswap, [`e_1,e_2`])` is compiled to bytecode evaluating e_1 and e_2, and swapping their values on the operand stack.

The primitives `Pdummy`, `Pupdate`, `Ptag_of`, `Psmlsuccint`, and `Psmlpredint` have not been implemented in the new compiler back-end, as they are not being used in the front-end of the compiler.

4.6.8 Lcase, Lswitch, Lstaticfail, and Lstatichandle

In the following the abbreviation es is used for a 'switch list' of pairs comprising a key and a related Lambda expression: `[`$(k_1, e_1), (k_2, e_2), \ldots, (k_n, e_n)$`]`.

A Lambda expression `Lstatichandle(Lswitch(`n, e_0, es`), `e_f`)` is compiled to bytecode branching on the constructor tag of the value of e_0:

```
<$e_0$>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jlookupswitch {default = $lbl_f$,
               cases   = [($k'_1$, $lbl_1$), ($k'_2$, $lbl_2$), ... ($k'_n$, $lbl_n$)]}
Jlabel $lbl_n$
<$e_n$>
Jgoto $lbl'$
...
Jlabel $lbl_2$
<$e_2$>
Jgoto $lbl'$
Jlabel $lbl_1$
<$e_1$>
Jgoto $lbl'$
Jlabel $lbl_f$
<$e_f$>
Jlabel $lbl'$
```

where and $k'_1, k'_2, \ldots, k'_n$ are the `int` values corresponding to the block tag keys $k_1, k_2, \ldots, k_n$.

An `Lcase` clause with atomic `INTscon`, `WORDscon`, or `CHARscon` constants as branch keys is compiled similarly to the `Lswitch` clause above, except that the value of e_0, which is assumed to be an `Integer` object, is unwrapped before executing the `Jlookupswitch` instruction.

An `Lcase` clause with atomic `REALscon` or `STRINGscon` constants as keys is compiled to a series of tests. For example, the expression `Lstatichandle(Lcase(`e_0,es`), `e_f`)`, where the keys $k_1, k_2, \ldots, k_n$ in es are `REALscon` atomic constants, is compiled to:

```
<$e_0$>
Jcheckcast Number
Jinvokevirtual Number.doubleValue()
Jdup2
Jdconst $k'_1$
Jdcmpg
Jifne $lbl_2$
Jpop2
<$e_1$>
Jgoto $lbl$
Jlabel $lbl_2$
Jdup2
Jdconst $k'_2$
Jdcmpg
Jifne $lbl_3$
Jpop2
<$e_2$>
Jgoto $lbl$
...
Jlabel $lbl_n$
Jdconst $k'_n$
Jdcmpg
Jifne $lbl_f$
<$e_n$>
Jgoto $lbl$
Jlabel $lbl_f$
<$e_f$>
Jlabel $lbl$
```

where $k'_1, k'_2, \ldots k'_n$ are the `real` values corresponding to the keys $k_1, k_2, \ldots, k_n$.

Note that the expression e_0 will be evaluated only once. The resulting value must be duplicated (using `Jdup2`) between successive tests since each `Jdcmpg` instruction consumes the two topmost values on the operand stack (and pushes a result of -1, 0, or 1). In order to keep the operand stack consistent, the duplicated value must also be removed (using `Jpop2`) after the corresponding `Jifne` instruction.

4.6.9 Lhandle

A Lambda expression `Lhandle(`e_0`, `e_1`)` is compiled to:

```
Jlabel $lbl$
<$e_0$>
Jlabel $lbl'$
Jgoto $lbl'''$
Jlabel $lbl''$
Jastore $local(d)$
<$e_1$>
Jlabel $lbl'''$
```

This bytecode sequence evaluates e_0 and branches to label lbl''', thereby 'bypassing' the handler code of e_1. Additionally, a JVM exception exception handler, covering the instructions from label lbl to lbl' is installed, with label lbl'' as entry point and catching JVM exceptions of class

`Exception`.

In case an `Exception` object is thrown during evaluation of e_0, the current operand stack will be purged, the exception is pushed onto the stack, and execution continues at label *lbl″*. The `Exception` object is then stored into a local variable, and e_1 is evaluated.

Before bytecode is generated for e_1 the current local variable environment is augmented with a binding for variable number d, where d is the first unused variable index in the context of the `Lhandle` expression.

4.6.10 Lif

A Lambda expression `Lif(`e_0`, `e_1`, `e_2`)` is compiled to bytecode branching on the constructor tag of the value of e_0, assuming that e_0 evaluates to a `Constructor` object:

```
<e0>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl
<e1>
Jgoto lbl'
Jlabel lbl
<e2>
Jlabel lbl'
```

If the constructor tag is 0, corresponding to the SML value `false`, then e_2 is evaluated, otherwise e_1 is evaluated.

4.6.11 Lseq

A Lambda expression `Lseq(`e_1`, `e_2`)` is compiled to bytecode that evaluates e_1, discards the resulting value, and then evaluates e_2:

```
<e1>
Jpop
<e2>
```

It is assumed that evaluation of e_1 leaves the value of that expression on the stack, i.e., that there is something to pop.

4.6.12 Lwhile

A Lambda expression `Lwhile(`e_0`, `e_1`)` is compiled to bytecode that first tests the constructor tag of the value of e_0, assuming that e_0 evaluates to a `Constructor` object. If the tag is non-zero then e_1 is evaluated, its value is discarded, and the bytecode again evaluates the expression e_0 before testing the constructor tag of its value:

```
Jgoto lbl'
Jlabel lbl
<e1>
Jpop
Jlabel lbl'
<e0>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifne lbl
Jgetstatic Tuple.Unit
```

This loop is repeated until e_0 evaluates to a `Constructor` object with a tag value of 0. Once the loop stops, the value `(): unit` is pushed on the stack.

4.6.13 Landalso and Lorelse

A Lambda expression `Landalso(`e_1`, `e_2`)` is compiled to:

```
<e1>
Jdup
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl
Jpop
<e2>
Jlabel lbl
```

This bytecode sequence branches on the constructor tag of the value of e_1, assuming that e_1 evaluates to a `Constructor` object. If the tag is 0, corresponding to the SML value `false`, then the value of e_1 is left on the stack as the value of the `andalso` expression; otherwise e_2 is evaluated.

Note that e_1 will be evaluated only once. The resulting value must be duplicated before the test on its tag, as the `Jifeq` instruction consumes the topmost stack operand. In order to keep the operand stack consistent, it is necessary to remove the duplicated value again after the `Jifeq` instruction.

If an `Landalso` expression appears as conditional of an `Lif` or `Lwhile` expression, or appears inside another `Landalso` or `Lorelse` expression, then the above bytecode sequence may be optimized; see Section 5.4.1.

The compilation of an `Lorelse` Lambda expression to bytecode is dual to that of an `Landalso` expression. The bytecode generated for the expression `Lorelse(`e_1, e_2`)` branches on the constructor tag of the value of e_1. If the tag is 0 then e_2 is evaluated; otherwise the value of e_1 is left on the stack as the value of the `Lorelse` expression.

4.6.14 Lunspec

The `Lunspec` Lambda expression is only expected to appear in the body of a `Llet` expression, and is otherwise ignored (no bytecode is generated).

4.6.15 Lshared

The first time a shared Lambda expression is encountered by the code generator, its status is assumed to be `LIFTED` (cf. Section 4.3). A Lambda expression `Lshared(ref` *e*`, ref LIFTED)` is compiled to:

```
Jlabel lbl
<e>
```

That is, the bytecode sequence for the shared expression is simply labelled with label lbl'. The `Lshared` clause is then marked as `COMPILED` *lbl* for later use.

A Lambda expression `Lshared(ref` *e*`, ref (COMPILED` *lbl*`))`, i.e., a shared expression for which bytecode has already been generated, is compiled to an unconditional branch:

```
Jgoto lbl
```

thereby effectively re-using the bytecode sequence for the shared expression.

4.6.16 Lassign

The `Lassign` Lambda clause is not used in the new compiler. Hence, if the code generator encounters an `Lassign` expression, an exception indicating an internal error is raised.

4.6.17 Arithmetic Exceptions

In order to properly implement SML semantics, integer operations must check for overflow and division by zero, as described in Section 2.1.2. To this end, two compiler flags are declared in the `Gencode` module of the new compiler-backend: `checkOverflow` and `checkDiv`.

If the `checkOverflow` compiler flag is true, then bytecode is generated for each of the primitives `Psmlnegint`, `Psmladdint`, `Psmlsubint`, `Psmlmulint`, and `Psmldivint` to explicitly check for overflow. For example, a Lambda expression `Lprim(Psmladdint, [`e_1, e_2`])` is compiled to:

```
Jnew Long
Jdup
<e₁>
Jcheckcast Number
Jinvokevirtual Number.longValue()
<e₂>
Jcheckcast Number
Jinvokevirtual Number.longValue()
Jladd
Jdup2
Jlconst Int32.minInt
Jlcmp
Jiflt lbl
Jdup2
Jlconst Int32.maxInt
Jlcmp
Jifle lbl'
Jlabel lbl
Jpop2
Jnew Exception
Jdup
Jgetfield General.Overflow$x0
Jinvokespecial Exception.<init>(Object)
Jathrow
Jlabel lbl'
Jinvokespecial Long.<init>(long)
```

provided `checkOverflow` is true.

This bytecode sequence unwraps the operands as values of JVM type `long`, adds them using the proper `long` instruction, and then examines their sum, checking whether it can be represented as a value of type `int`. If the sum is less than the minimum `int` value or larger than the maximum `int` value, then an `Exception` object with tag `Overflow` is thrown. Otherwise, the sum is wrapped into an instance of the standard Java class `Long` (another subclass of `Number`).

Note that the sum is duplicated before each `Jlcmp` instruction in the above bytecode sequence. This is necessary since the instruction consumes the two topmost stack operands.

If the `checkDiv` compiler flag is true, then an extra JVM exception handler will be generated for each of the primitives `Pdivint`, `Pmodint`, `Psmldivint`, `Psmlmodint`, `Psmlquotint`, and `Psmlremint`. This exception handler maps an `ArithmeticException`, which is thrown in case of division by zero, to the SML exception `Div`.

For example, a Lambda expression `Lprim(Psmldivint,` $[e_1, e_2]$`)` is compiled to:

```
Jnew Integer
Jdup
<e1>
Jcheckcast Number
Jinvokevirtual Number.intValue()
<e2>
Jcheckcast Number
Jinvokevirtual Number.intValue()
Jlabel lbl
Jidiv
Jlabel lbl'
Jgoto lbl'''
Jlabel lbl''
Jpop
Jnew Exception
Jdup
Jgetfield General.Div$x0
Jinvokespecial Exception.<init>(Object)
Jathrow
Jlabel lbl'''
Jinvokespecial Integer.<init>(int)
```

provided `checkDiv` is true. A JVM exception handler, covering the `Jidiv` instruction, catching exceptions of class `ArithmeticException`, and having label *lbl''* as entry point, is declared for the target method.

4.7 Class File Generation

Once code generation is complete, the compiler builds an abstract class declaration from the intermediate `jvm_code` representation of the target bytecode, and generates a class file using the SML-JVM toolkit (cf. Section 3.4).

Construction of the abstract class declaration from the intermediate target code is implemented in the function `Genclass.genClass`. This function performs the following steps:

- generates an 'empty' method `main`,
- completes the bytecode for method `<clinit>`,
- completes the bytecode for method `apply`,
- generates an instance initialization method `<init>(int)`,
- generates an instance initialization method `<init>(int,Object[])`,
- calculates maximum operand stack depth and maximum number of local variables used in each generated method,
- generates static field declarations corresponding to the top-level names declared in the source module,
- builds an abstract class declaration (`class_decl` record) holding the field and method declarations of the target class, and
- emits the abstract class declaration to a Java class file, using the function `Classfile.emit` of the SML-JVM toolkit.

The `main` method contains but one bytecode instruction:

```
Jreturn
```

The purpose of this method is enabling the target class to serve as a 'main class' for an application (cf. Section 2.5).

Completing the bytecode for method `<clinit>` means introducing instructions at the beginning of the method for initializing the static field corresponding to each top-level exception declared in the module. This is necessary to properly associate a JVM value with each top-level exception tag. Moreover, this must take place prior to evaluation of any top-level expressions.

Assuming that the source module declares top-level exceptions $id_1, id_2, \ldots, id_n$, the resulting bytecode for method `<clinit>` will be:

```
Jsconst $id'_1$
Jputstatic $f_1$
Jsconst $id'_2$
Jputstatic $f_2$
...
Jsconst $id'_n$
Jputstatic $f_n$
<top-level expr code>
Jreturn
```

where $id'_1, id'_2, \ldots, id'_n$ are string representations of the unique exception identifiers; $f_1, f_2, \ldots f_n$ are field references to the static fields declared for the exceptions; and `<top-level expr code>` represents the bytecode generated for top-level expressions of the module.

Note that (string representations of) the fully qualified and 'stamped' exception identifiers are used as exception tag values. This is necessary to ensure that exception tag values are unique.

Completing the bytecode for method `apply` means introducing a switch on the closure tag, branching to the bytecode for each of the closure bodies, at the beginning of the method. The resulting bytecode for method `apply` will be:

```
Jaload 0
Jcheckcast Closure
Jgetfield Closure.tag
Jtableswitch {default = lbl_f,
              offset  = <firstTag>,
              targets = [lbl_0, lbl_1, ..., lbl_{n-1}]}
Jlabel lbl_f
Jnew SmlError
Jdup
Jsconst "unmatched closure tag"
Jinvokespecial SmlError.<init>(String)
Jathrow
Jlabel lbl_{n-1}
<body of closure t_{n-1}>
Jreturn
...
Jlabel lbl_1
<body of closure t_1>
Jreturn
Jlabel lbl_0
<body of closure t_0>
Jreturn
```

The generated bytecode for the instance initialization method `<init>(int,Object[])` is:

```
Jaload 0
Jdup
Jinvokespecial Closure.<init>()
Jiload 1
Jputfield Closure.tag
Jreturn
```

This bytecode sequence invokes the superclass constructor on the (uninitialized) object, and stores the integer argument into the instance field `tag` of the object. This method may be used for initializing instances of the target class, i.e., closure objects, with no free variables.

The generated bytecode for the instance initialization method `<init>(int,Object[])` is:

```
Jaload 0
Jdup
Jinvokespecial Closure.<init>()
Jdup
Jiload 1
Jputfield Closure.tag
Jiload 2
Jputfield Closure.free
Jreturn
```

This bytecode sequence invokes the superclass constructor on the (uninitialized) object, stores the integer argument into the instance field `tag`, and stores the array argument into the instance field `free`.

Chapter 5

Implemented Optimizations

This chapter describes the optimizations that have been implemented in the new compiler back-end and in the support classes used for representation of SML values at run-time.

5.1 Run-time Support Classes

The implemented run-time support classes have undergone a number of minor changes and improvements, as compared to the design desribed in Chapter 2.

5.1.1 Predefined Constructors and Unit

As mentioned in Section 4.6.7, a number of `Constructor` instances with no arguments have been 'predefined' in the `Constructor` class. Specifically, the following field declarations have been introduced[1]:

```
public final static Constructor
  Zero  = new Constructor(0),
  One   = new Constructor(1),
  Two   = new Constructor(2),
  Three = new Constructor(3),
  Four  = new Constructor(4),
  Five  = new Constructor(5),
  Six   = new Constructor(6),
  Seven = new Constructor(7),
  Eight = new Constructor(8),
  Nine  = new Constructor(9);
```

The null-constructors will be created, initialized, and bound to the above static fields when class `Constructor` is first loaded and initialized.

Hence, bytecode may be generated for loading the value of these static fields, rather than creating and initializing similar instances of class `Constructor` over and over again.

As mentioned in Section 4.6.7, a similar 'predefined' instance has been introduced in class `Tuple`:

[1]The choice to 'predefine' null-constructors only for the tag values 0 through 9 is arbitrary; more null-constructors could have been declared, but, presumably, the smallest tag values (starting from 0) will most often be used.

```
public final static Tuple Unit = new Tuple(null);
```

The value of the static field `Tuple.Unit` thus corresponds to the empty tuple (a.k.a. `(): unit` in SML).

5.1.2 Specialized Initialization Methods

A special instance initialization method has been introduced in class `Constructor`:

```
public Constructor (int tag)
{
  this.tag = tag;
}
```

This method may be used for initializing a null-constructor, that is, a `Constructor` instance with no arguments.

A similar instance initialization method has been introduced in class `Exception`:

```
public Exception (Object tag)
{
  this.tag = tag;
}
```

Again, this method may be used for initializing an `Exception` with no arguments (i.e., a static exception in the Moscow ML sense).

5.1.3 Improved Equality Testing

The implementation of equality testing in the run-time support classes has been optimized, as compared to the naïve implementations of method `equals` presented in Section 2.2. For example, the implementation of method `equals` in class `Tuple` has been changed to:

```
public boolean equals (Object x)
{
  Object[] thisArgs = args,
           xArgs    = ((Tuple)x).args;

  if (thisArgs != null)
    {
      int i = thisArgs.length;
      // no need to check args.length == xArgs.length,
      // assuming arity(this) == arity(x)

      while (--i >= 0)
        if (!thisArgs[i].equals(xArgs[i]))
          return false;
    }

  return true;
}
```

Note that the values of the instance fields `this.args` and `x.args` are bound to local variables before the loop. Presumably, loading the value of a local variable is faster than loading the value of an instance field.

Similar improvements have been introduced in the implementation of method `equals` in class `Constructor`.

5.2 Target Class Improvements

The generation of target classes has been improved in a number of ways:

- Generating a special instance initializer method `<init>(int)` for initializing instances with no free variables.
- Generating each of the instance initializer methods `<init>(int)` and `<init>(int,Object[])` only if that method is actually being invoked. The flags `usesInit0` and `usesInit` of the `jvm_code` intermediate target code are used to determine whether each instance initialization method is necessary in the target class.
- Making the target class a subclass of `Object` rather than `Closure` in case no instances (closures) are being created, that is, provided both `usesInit0` and `usesInit` are `false`. This way, the `apply` method can be omitted in the target class.
- Flagging the generated instance initialization methods (if any) as `ACC_PRIVATE`. This way, only methods of the target class may instantiate the class. Methods in other classes may still utilize instances of the target class via the public static fields of the class.
- Flagging generated fields and methods corresponding to non-exported top-level names of the module as `ACC_PRIVATE`.
- Optimizing the closure tag switch in method `apply` in case there is only one entry. Instead of generating a `Jtableswitch` instruction with a single entry, a `Jifeq` instruction, branching to the closure body, is generated.

5.3 Closure Specialization

The optimization described in this section is one of the most important and complex optimizations that have been implemented in the new compiler back-end. The purpose is that of calling manifest functions directly. For example, in this program

```
fun f n []      = n
  | f n (x::xr) = f (n+x) xr

val a = f 17 [1,2,3]
val sum = f 0
```

the recursive call inside function `f` should be compiled to a direct invocation of a Java bytecode method representing `f`, rather than bytecode performing the following steps:

- loading a closure object from a free variable,
- verifying that it is in fact an instance of class `Closure`, and
- invoking instance method `Closure.apply` on the object (implying dynamic dispatch).

Similarly, the call to function `f` in the right-hand side of the binding `val a = ...` should be compiled to a direct invocation of the method representing `f`, since the function is provided with all the (curried) arguments that it expects.

However, the call to function `f` in the right-hand side of the binding `val sum = ...` need not be optimized to a direct call, as the function is not provided with all the (curried) arguments that it expects. In other words, only saturated calls to manifest functions should be optimized.

To support direct calls to manifest functions, specialized versions of the relevant closures must be generated as separate Java bytecode methods, and information must be carried through the compiler back-end in order to recognize a saturated call to a manifest function.

5.3.1 Run-time Types

The following data types are declared in module `Runtype` of the new compiler back-end:

```
datatype runtype =
    RTnumber | RTint | RTlong | RTdouble
  | RTbool
  | RTstring | RTchararray
  | RTvector | RTarray
  | RTblock  | RTtuple | RTconstructor | RTref | RTexception
  | RTclosure
  | RTmethod of Tag.tag * spec_clos
  | RTobject
and spec_clos =
    GENERAL
  | SPEC of {freeTypes : runtype list,  (* types of free variables *)
             argTypes  : runtype list,  (* types of arguments *)
             resType   : runtype}       (* result type *)
```

The `runtype` data type is used for representing the run-time type of SML values at compile-time. In other words, the `runtype` of a value indicates its representation at run-time.

Only the `RTobject`, `RTclosure`, and `RTmethod` clauses are used in connection with closure specialization. The meaning of each of these clauses is:

`RTobject` represents a (boxed) value of unknown type; for example, this is used for the argument of a closure, which is declared to have JVM type `Object`.

`RTclosure` represents a `Closure` object that has not been specialized.

`RTmethod`(t, c) represents a `Closure` object with closure tag t for which a specialized closure has been generated. The 'specialization information' c should be a `SPEC` value, specifying the run-time types of free variables, arguments, and result type of the specialized closure.

The `runtype` data type is used in a number of environments related to closure specialization; this is explained in Section 5.3.3.

5.3.2 Lifting

The lifting phase of the compiler back-end has been extended to collect a list of 'pending methods'. Each pending method represents a specialized closure taking one or more arguments. The list of pending methods includes the closure tag, free variable environment, number of (curried) arguments, 'context depth', and Lambda expressions for each of the pending methods.

5.3.3 Additional Environments

In order to recognize a saturated call to a manifest function, a number of extra environments have been introduced: a global environment, *global*, a method environment, *method*, and a closure environment, *closure*.

The *global* environment maps each top-level function identifier to its run-time type (`runtype` `RTclosure` or `RTmethod`).

For each top-level function, the *method* environment maps the corresponding closure tag to the unique identifier of the function. This is used in the compilation of each specialized closure to a Java bytecode method; see Section 5.3.5.

The global environment and the method environment are built in a separate pass over the lifted Lambda expressions of the module prior to code generation. This has been implemented in the function `Compiler.buildGlobalEnvs`.

The closure environment acts as an extension to the free variable environment used in the general compilation. The closure environment maps a closure tag to a free variable `runtype` environment for the closure; the latter environment maps each free variable of the closure to its `runtype`. The closure environment is built during code generation, based on the `runtype` of local variables being passed as free variables to each closure.

In addition to the information gathered in the above environments, the local variable environment *local* has been extended to carry the `runtype` of each function bound to a local variable.

5.3.4 Invoking Manifest Functions Directly

In the compilation of an `Lapply` expression, a manifest function can be recognized in three different ways:

- Direct application of a `Pclosure` primitive to an empty list of arguments. Such an application stems from an exception handler that has been lifted into a separate function because it appeared in a value context.
- Application of a variable of `runtype RTmethod` to k arguments, where $k \geq m$, and m is the number of (curried) arguments expected by the specialized closure bound to the local variable. The `runtype` of the local variable is retrieved from the *local* environment (for a locally bound variable) or from the *closure* environment (for a free variable).
- Application of top-level identifier of `runtype RTmethod` to k arguments, where $k \geq m$, and m is the number of (curried) arguments expected by the specialized closure bound to the top-level identifier. The `runtype` of the top-level identifier is retrieved from the *global* environment.

In each case, if the specialized closure being invoked has any free variables, then the array of free variables from the original, unspecialized closure is passed as the first argument to the specialized closure. A specialized closure is invoked using a `Jinvokestatic` instruction.

5.3.5 Compiling Specialized Closures to Methods

Generation of Java bytecode for specialized closure bodies is implemented in the function `Gencode.compileMethod`. This method invokes the code generator, `Gencode.compileExpr`, similarly to the compilation of top-level Lambda expression and (unspecialized) closure bodies (cf. Section 4.2).

The generated bytecode for each specialized closure is collected in the extra `methods` component of the `jvm_code` intermediate target code representation. The `methods` component holds the name, argument types, result type, exporting flag, generated bytecode, *local* environment, and declared exception handlers for each of the specialized closures.

The name of a specialized closure bound to a top-level identifier will be *method*(t) where t is the closure tag of the specialized closure. The name of 'anonymous' functions and functions bound to local variables will be `clos$$`t; again, t is the closure tag of the specialized closure.

The `methods` component of the `jvm_code` record is used in the `Genclass` module of the compiler back-end for generating a separate Java bytecode method for each of the specialized closures.

5.4 Code Generation

Some optimizations have been adopted from the original compiler back-end of Moscow ML:

- Re-using labels whereever possible rather than creating fresh ones (i.e., avoiding sequences of `Jlabel` instructions).
- Removing dead code; all instructions following an unconditional jump, a return instruction, or a `Jathrow` instruction are considered unreachable, up to the next `Jlabel` instruction.
- Avoiding jumps to unconditional jumps. Instead of jumping to an `Jgoto` instruction, the target of the `Jgoto` instruction is jumped to.
- Avoiding an unconditional jump to a `Jreturn` instruction; instead, a `Jreturn` instruction is generated.

5.4.1 Boolean Expressions in Control-Flow Context

In general, a boolean expression is compiled to Java bytecode pushing a `Constructor`, representing `false` or `true`, as shown in Section 4.6. However, if a boolean expression appears in a control-flow context, e.g. as the conditional an `Lif` or `Lwhile` expression, or inside an `Landalso` or `Lorelse` expression, then shorter and more efficient bytecode may be generated.

Ptest

In general, a Lambda expression `Lprim(Ptest (Pint_test PTlt), [`e_1, e_2`]))`, corresponding to the boolean expression $e_1 < e_2$, is compiled to:

```
<e₁>
Jcheckcast Number
Jinvokevirtual Number.intValue()
<e₂>
Jcheckcast Number
Jinvokevirtual Number.intValue()
Jif_icmpge lbl′
Jgetstatic Constructor.One
Jgoto lbl
Jlabel lbl′
Jgetstatic Constructor.Zero
Jlabel lbl
```

Note that the 'inverted' test is used in the above bytecode: control is transferred to label *lbl′* in case the integer value of e_1 is greater than or equal to the value of e_2.

If the test appears in a control-flow context then it is possible to generate more compact and efficient bytecode than that shown above. If the continuation is

```
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl
<C>
```

then the following bytecode is generated for the above integer test:

```
<e₁>
Jcheckcast Number
Jinvokevirtual Number.intValue()
<e₂>
Jcheckcast Number
Jinvokevirtual Number.intValue()
Jif_icmpge lbl
..................................
<C>
```

The bytecode above the dotted line is the code generated for the `Ptest (Pint_test PTlt)` primitive, whereas the code below the line is the resulting continuation. Note that the continuation has been reduced to C, and that the label *lbl* is branched to rather than *lbl′*.

Similarly, if the continuation is

```
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifne lbl
<C>
```

then the following bytecode is generated for the above integer test:

```
<e1>
Jcheckcast Number
Jinvokevirtual Number.intValue()
<e2>
Jcheckcast Number
Jinvokevirtual Number.intValue()
Jif_icmplt lbl
..................................
<C>
```

Note that the comparison operator corresponding to the test `PTlt` is used, and that the continuation has been reduced to C.

Pnot

As shown in Section 4.6.7, the unoptimized bytecode generated for a Lambda expression `Lprim(Pnot, [e]))` is:

```
<e>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jiconst 1
Jixor
Jifeq lbl'
Jgetstatic Constructor.One
Jgoto lbl
Jlabel lbl'
Jgetstatic Constructor.Zero
Jlabel lbl
```

However, if the continuation is

```
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifne lbl
<C>
```

then the following bytecode is generated for boolean negation:

```
<e>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl
..................................
<C>
```

Again, the bytecode above the dotted line is the code generated for the `Pnot` primitive, whereas the code below the line is the resulting continuation. Note that no integer instructions are used to perform the negation, and that the continuation has been reduced to C.

Similarly, if the continuation is

```
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl
<C>
```

then the following bytecode is generated for boolean negation:

```
<e>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifne lbl
.................................
<C>
```

Landalso and Lorelse

In general, the following bytecode is generated for a Lambda expression `Landalso(`e_1, e_2`)`, as shown in Section 4.6.13:

```
<e1>
Jdup
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl
Jpop
<e2>
Jlabel lbl
```

If the continuation is

```
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl
<C>
```

then the following bytecode is generated for the above `Landalso` expression:

```
<e1>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl
<e2>
.................................
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl
<C>
```

Again, the bytecode above the dotted line is the code generated for the `Landalso` expression, whereas the code below the line is the resulting continuation. Note that duplication of the value of e_1 is avoided, and that the continuation remains unchanged, including unwrapping.

Similarly, if the continuation is

```
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifne lbl
<C>
```

then the following bytecode is generated for the above Lambda expression:

```
<e₀>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifeq lbl′
<e₁>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifne lbl
Jlabel lbl′
.................................
<C>
```

Note that the continuation has been reduced to C.

Optimization of an `Lorelse` expression appearing in a control-flow context is dual to the optimization of an `Landalso` expression shown above.

5.4.2 Other Continuation-Based Optimizations

In addition to the optimization of boolean expressions described in the previous section, a number of other optimizations based on continuation matching has been implemented in the new code generator:

- Avoiding an instruction sequence where the wrapping of a value of simple type is immediately followed by a corresponding unwrapping operation (no wrap-unwrap code is generated). This is implemented in the function `Coercion.coerce`, which is used for generating bytecode for coercion from one `runtype` to another.
- Avoiding an instruction sequence where the creation of an object is immediately followed by a `Jcheckcast` instruction to the same class as that of the newly created object or one of its superclasses (no `Jcheckcast` instruction is generated).
- Avoiding an instruction sequence where a `Jaload` instruction loading a `Closure` object from a local variable is followed by a `Jcheckcast` instruction (no `Jcheckcast` instruction is generated).
- Avoiding an instruction sequence where the creation of a constant is immediately followed by a `Jpop` instruction (no code is generated).
- Avoiding an instruction sequence where a value is first stored into a local variable, then loaded from the same variable:

  ```
  Jastore j
  Jaload j
  ```

  Instead, the value is first duplicated, then stored (no `Jaload` instruction is generated):

  ```
  Jdup
  Jastore j
  ```

  Presumably, duplicating a stack operand is at least as efficient as loading a local variable.
- Avoiding an instruction sequence where a value is first stored into a static field, then loaded from the same static field:

```
putstatic f
getstatic f
```

Instead, the value is first duplicated, then stored (no `Jgetstatic` instruction is generated):

```
Jdup
Jputstatic f
```

Presumably, duplicating a stack operand is more efficient than loading a static field.
- Avoiding conversion from control-flow representation to constructor representation of boolean values, wherever possible. This is, however, not fully optimized for the first sub-expression in an `Landalso` or `Lorelse` Lambda expression (due to the use of a `Jdup` instruction).
- Avoiding `Jcheckcast Exception` inside an exception handler; the 'argument' will always be an `Exception` object, as this is the catch type of generated exception handlers.

5.4.3 Other Local Optimizations

Other local optimizations include:

- Using the 'predefined' values of class `Constructor` and `Tuple`, instead of creating often used null-constructors and the empty tuple over and over again (cf. Section 5.1.1). For example, the SML constants `nil`, `NONE`, and `false` are compiled to bytecode loading the static field `Constructor.Zero`, and all occurences of `(): unit` are compiled to bytecode loading the static field `Tuple.Unit`.
- Using the special instance initialization method `<init>(int)` for initializing instances of class `Constructor` and `Exception` with no arguments (cf. Section 5.1.2). Similarly for initialization of instances of the target class, that is, closure objects, with no free variables (cf. Section 5.2).
- Compiling an `Lcase` or `Lswitch` Lambda expression with only one entry to a conditional branch (`Jif`...) rather than a `Jlookupswitch` instruction with a single entry. For example, this situation could arise in compilation of a pattern match on an list:

```
case e0 of
    [] => e1
  | _  => e2
```

The above SML expression translates to this Lambda code:

```
Lstatichandle (Lswitch (2, e0, [(CONtag(0,2), e1)]),
               e2)
```

This is compiled to the following bytecode sequence:

```
<e_0>
Jcheckcast Constructor
Jgetfield Constructor.tag
Jifne lbl
<e_1>
Jgoto lbl'
Jlabel lbl
<e_2>
Jlabel lbl'
```

Chapter 6

Testing

The implemented compiler has been tested to verify that it produces the expected Java bytecode, and to investigate the performance of the generated bytecode. This chapter describes the tests that have been undertaken.

6.1 Generating Correct Bytecode

In the process of debugging the compiler, a set of small test programs have been implemented. Each of these programs have been used to verify that specific Lambda constructs are compiled correctly.

For example, the following program tests the compilation of an `Llet` Lambda clause:

```
(* let.sml *)

val _ =
    let val a = (util.println "a"; 1)
        val _ = (util.println "_"; 2)
        val b = (util.println "b"; 3)
    in
        util.println(a, b)
    end
```

Similar programs have been used to test that all Lambda clauses are in fact compiled correctly, i.e., leading to the expected bytecode. The compilation of all supported Lambda code primitives has been tested similarly. The source files for the test programs are listed in Appendix G.

6.1.1 The `util` module

Note that the above program refers to a `util` module. This is a simple utility class implemented in Java, which exports a number of static fields. The field `util.println` holds a closure wrapper, that is, an instance of the run-time support class `Closure`, representing the standard Java method `System.out.println`. When the closure is applied to an argument, it invokes `System.out.println` to print the argument.

In other words, the `util` class can be be used as a 'fake' SML module providing a number of special features, e.g. support for printing values to output. This is necessary, e.g. for the purpose of exploiting the Java output mechanism `System.out.println` from SML, as the SML program cannot refer directly to the standard Java libraries.

The source file `util.java` is listed in Appendix G along with the test programs.

6.1.2 The Exception Wrapper Class

One of the problems in debugging the compiler has been with unexpected and uncaught JVM exceptions thrown by the JVM due to errors. Unfortunately, Sun's JVM implementation does not report uncaught exceptions in a program; instead, the JVM simply aborts without any indication of error.

To this end, an exception wrapper utility has been developed and implemented in Java. The source file `exnWrapper.java` is listed in Appendix G. Method `main` of class `exnWrapper` performs the following steps:

- loads the class specified as the first command-line argument,
- invokes the `main` method of that class, passing to it the remaining command-line arguments, and
- catches any exception thrown in the previous steps, and displays the exception followed by a stack trace.

For example, the exception wrapper utility could be used to execute the above test program like this:

```
java exnWrapper let
```

This would invoke the JVM on the `exnWrapper` class, which would in turn load the class `let` and invoke its `main` method (with no arguments). In case any exception is thrown during the loading or initialization of class `let`[1], or during execution of its `main` method, the exception would be catched and displayed by the exception wrapper.

6.2 An Example

For the purpose of demonstrating the new compiler, this SML program has been compiled:

```
fun fib n =
    let fun fib' 0 k1 k2 = k1
          | fib' i k1 k2 = fib' (i-1) (k1+k2) k1
    in
        if n < 2 then n
        else fib' (n-2) 1 1
    end

val _ = (util.print "fib(10) = ";
         util.println(fib 10))
```

The intermediate Lambda code for the program and the target class file generated by the new compiler back-end are listed in Appendix C.

[1]A JVM initializes a Java class by executing method `<clinit>` of the class.

The Lambda code listing includes the Lambda expressions generated by the compiler front-end ("before lifting"), the lifted Lambda code ("after lifting"), plus lifted closures and specialized closures (methods). The listing of Lambda code was obtained by enabling the compiler flag `Compiler.showLambda` of the new compiler.

The generated target class was listed using the `javap` utility of Sun's JDK. The listing shows a number of the optimizations that have been implemented in the new compiler back-end. For example,

- the closure with tag 0 has been specialized to the method `fib$0`, taking one argument,
- the closure with tag 1 has been specialized to the method `clos$$1`, taking four arguments, the first of which is an array of free variables.

Moreover, the bytecode sequences show the effect of some of the implemented code generator optimizations. For example, this expression in the source program

```
if n < 2 then n
else ...
```

has been compiled to this bytecode sequence in method `fib$0`:

```
21 aload_0
22 checkcast #19 <Class java.lang.Number>
25 invokevirtual #23 <Method int intValue()>
28 iconst_2
29 if_icmpge 34
32 aload_0
33 areturn
...
```

With the general compilation shown in Chapter 4, the integer constant 2 would first be wrapped into an `Integer` object, and then be unwrapped before being compared to the value of `n`. However, the implemented optimization to avoid wrap-unwrap sequences has eliminated the wrapping and unwrapping of the integer constant.

The above instruction sequence also demonstrates the implemented optimization of boolean expressions in control-flow context. With the general compilation from Chapter 4, the test `2 < n` would be compiled to bytecode pushing a `Constructor` instance representing `false` or `true`, corresponding to the result of the test. Instead, the optimized bytecode directly branches to the `else` section of the `if` expression in the source program.

6.3 Benchmarks

In order to evaluate the performance of the bytecode generated by the compiler, some benchmark programs have been compiled and executed. The benchmarks were executed on the host `zuse.dina.kvl.dk`, a 250 MHz Sun UltraSparc machine with 512 MB of memory, running SunOS 5.5.1.

6.3.1 Fibonacci Benchmark

The first performance test is based on the fibonacci algorithm shown above in Section 6.2. A program containing the following SML functions has been compiled with the new compiler and with the existing Moscow ML compiler (version 1.42):

```
local
    fun fib' 0 f1 f2 = f1
      | fib' i f1 f2 = fib' (i-1) (f1+f2) f1
in
    fun fibCurry 0 = 0
      | fibCurry 1 = 1
      | fibCurry n = fib' (n-2) 1 1
end

fun fibRec 0 = 0
  | fibRec 1 = 1
  | fibRec n = fibRec(n-1) + fibRec(n-2)
```

The function `fibCurry` is quite similar to the `fib` function of the fibonacci program in Section 6.2, whereas the `fibRec` function is a naïve recursive variant of the same function.

Moreover, a Java program containing these methods have been compiled with the `javac` compiler of Sun's JDK 1.1.3:

```
private static int fib (int i, int f1, int f2)
{
  if (i == 0)
    return f1;
  else
    return fib(i-1, f1+f2, f1);
}

public static int fibCurry (int n)
{
  switch (n)
    {
    case 0:  return 0;
    case 1:  return 1;
    default: return fib(n-2, 1, 1);
    }
}

public static int fibRec (int n)
{
  switch (n)
    {
    case 0:  return 0;
    case 1:  return 1;
    default: return fibRec(n-1) + fibRec(n-2);
    }
}
```

The methods `fib` and `fibCurry` together correspond to the SML function `fibCurry` shown above, whereas method `fibRec` corresponds directly to the naïve recursive SML function `fibRec`.

The following table shows the results of running the above programs for `n` $= 0$ and `n` $= 30$.

The figures listed are the running times in seconds, measured with the Unix shell command `time` as the total of 'user' and 'system' time, averaged over 5 runs after an initial 'warm-up' run:

		n = 0	n = 30
mosmlc	fibRec	0.01	2.57
	fibCurry	0.02	0.03
mosmlc-jvm	fibRec	0.43	16.542
	fibCurry	0.45	0.44
javac	fibRec	0.45	2.26
	fibCurry	0.44	0.44

The programs compiled with `mosmlc` were executed with the modified Caml Light run-time system distributed with Moscow ML. The programs compiled with the new compiler, as well as the corresponding Java programs, were executed with the JVM implementation in Sun's JDK 1.1.3.

The column for `n` = 0 shows that it takes the JVM approximately 0.44 seconds to load and initialize the involved classes.

The column for `n` = 30 shows that only the naïve recursive versions of the fibonacci algorithm take any significant time to execute. This column also shows that the SML programs compiled with the new compiler run between 6 and 58 times slower than the same programs compiled with Moscow ML.

In other words, there appears to be potential for significant improvements on the performance of the bytecode generated by the new compiler back-end.

6.3.2 Heapsort Benchmark

As a somewhat more realistic benchmark, this program has been compiled with the new compiler and with the existing Moscow ML compiler:

```
datatype 'a heap = Empty | Node of 'a * 'a heap * 'a heap

fun siftdown (x : int) Empty Empty = Node(x, Empty, Empty)
  | siftdown x (t1 as Node(y, a, b)) Empty =
    if x >= y then Node(x, t1, Empty)
    else Node(y, siftdown x a b, Empty)
  | siftdown x Empty (t2 as Node(z, c, d)) =
    if x >= z then Node(x, Empty, t2)
    else Node(z, Empty, siftdown x c d)
  | siftdown x (t1 as Node(y, a, b)) (t2 as Node(z, c, d)) =
    if x >= y andalso x >= z then
        Node(x, t1, t2)
    else if y >= x andalso y >= z then
        Node(y, siftdown x a b, t2)
    else (* z >= x andalso z >= y *)
        Node(z, t1, siftdown x c d)
```

```
fun length xs =
    let fun h ([],    c) = c
          | h (_::xr, c) = h (xr, c+1)
    in h (xs, 0) end

fun heapify xs =
    let fun h 0 xs      = (Empty, xs)
          | h n (x::xr) =
            let val m = n div 2
                val (a, ys) = h m xr
                val (b, zs) = h (n-m-1) ys
            in (siftdown x a b, zs) end
          | h _ _       = raise Div
    in #1 (h (length xs) xs) end

fun del Empty Empty = Empty : int heap
          | del t1    Empty = t1
          | del Empty t2    = t2
          | del (t1 as Node(y, a, b)) (t2 as Node(z, c, d)) =
            if y >= z then Node(y, del a b, t2)
            else Node(z, t1, del c d)

fun extract heap =
    let fun h Empty             res = res
          | h (Node(x, t1, t2)) res = h(del t1 t2) (x :: res)
    in h heap [] end

fun heapsort xs = extract (heapify xs)
```

The program implements a 'heapsort' algorithm for sorting a list of integers. The algorithm first builds a balanced heap from the integer list; the biggest element is located at the top of the heap. The sorted list of integers is then built by:

- extracting the top-most element from the heap,
- restructuring the heap so that the biggest element in the remaining heap is located at the top, and
- repeating the above two steps until the heap is empty.

The following table shows the results of running the above heapsort program 100 times on a list of `n` pseudo-random integers. The figures listed are the running times in seconds, measured in the same way as for the Fibonacci benchmark above.

	n = 0	n = 1000
`mosmlc`	0.01	2.19
`mosmlc-jvm`	0.44	128.3

As for the Fibonacci benchmark, the column for `n` = 0 shows that it takes the JVM about 0.44 seconds to load and initialize the involved classes.

The results in the column for `n` = 1000 are also similar to those for the Fibonacci benchmark: the Java bytecode generated by the new compiler back-end executes about 58 times slower than the Caml Light bytecode generated by the existing back-end of Moscow ML.

It has been necessary to use repeated executions of the heapsort algorithm to achieve measurable running times for the Caml Light bytecode program generated with Moscow ML. At the same time, it has not been possible to increase the number of list elements because of the memory requirements for the Java bytecode program generated by the new compiler. For example, an attempt to use `n` = 5000 failed due to insufficient memory (required more than 512 MB).

The running times for the Java bytecode program shown in the above table were measured with a maximum heap size of 32 MB. For comparison, the Caml Light bytecode program uses approximately 1376 KB of memory.

6.3.3 Amlazy Benchmark

A lazy abstract machine has been implemented in SML by Peter Sestoft. The program consists of about 600 lines of SML code, and evaluates terms of the input language by rewriting.

This program has been compiled with Moscow ML and with the new compiler. Execution of the resulting bytecode programs on sample input indicates that the Java bytecode generated by the new compiler back-end is up to 57 times slower than the Caml Light bytecode generated by the existing back-end of Moscow ML.

Chapter 7

Discussion

As described in the previous chapter, the performance of the Java bytecode generated by the new compiler back-end is quite poor, as compared to the corresponding Caml Light bytecode generated by the existing Moscow ML backend, and as compared to similar Java programs compiled to Java bytecode.

To some extent, the performance problems may be attributed to Sun's JVM implementation, on which the benchmarks have been performed. Observations of run-time behaviour of Java bytecode programs indicate that the garbage collector in Sun's JVM does not handle programs that allocate many intermediate objects in the heap very well.

However, studies of class files generated by the new compiler back-end reveals many potential optimizations. In the following sections, we describe the most important of those.

7.1 Tail-Call Optimization

Lack of tail-call optimization in the implemented compiler back-end is presumably the single most important reason for the performance problems with the generated Java bytecode, as compared to the Caml Light bytecode generated by the existing front-end of Moscow ML (cf. Section 6.3).

Currently, tail-calls cannot be implemented efficiently in Java bytecode: the JVM instruction set includes no specific tail-call instructions and tail-calls are not detected and optimized in existing implementations of the JVM.

However, tail-call optimization may be introduced in future JVM implementations, specifically to improve execution of recursive methods. Until then, a significant speed-up for Java bytecode generated from SML may be achieved by implementing directly tail-recursive functions as loops, i.e., by jumping. This solution would work only for directly tail-recursive functions since a Java bytecode method cannot jump to a bytecode instruction in another method.

7.2 Typed Lambda Code

Annotating the intermediate Lambda code with type information would make it possible to introduce two important optimizations in the new compiler back-end: minimizing run-time type checks and minizing boxing of values of simple type.

Since the JVM enforces a static typing discipline, InstrJcheckcast instructions, enforcing run-time type checking, must be generated in many situations to satisfy Sun's bytecode verifier, although run-time type checks are not necessary to ensure type-safety of the SML program. Moreover, since little type information is presently available to the code generator, more `Jcheckcast` instructions are generated than strictly necessary to suit Sun's bytecode verifier.

If more type information was available to the code generator, it would be possible to estimate more precisely when a `Jcheckcast` instruction may be omitted, while still having Sun's bytecode verifier accept the generated bytecode. Presumably, minimizing the number of `Jcheckcast` instructions generated by the new compiler back-end would have only marginal effect on the performance of generated bytecode[1]. It would, however, lead to somewhat shorter bytecode programs.

Boxing operations is probably a better candidate for performance improvements. Introducing more type information in the intermediate Lambda code would make it possible to omit coercions in a number of places. For example, it would be possible to store unboxed values of simple types into local and global variables, pass them as method arguments to specialized closures, etc.

Provided the Lambda code generated by the front-end is annotated with type information for functions and global variables, it should be possible to infer the necessary type information for the remaining Lambda clauses and primitives from the Lambda code itself. Type information corresponding to the `runtype` information described in Section 5.3.1 should suffice to minize the use of `Jcheckcast` instructions and coercions.

7.3 Calling External Functions

A mechanism for calling *static* Java methods may be implemented via the `prim_val` feature of Moscow ML (translated to a `Pccall` primitive in the intermediate Lambda code). For example, given the declaration

```
prim_val parseInt : string -> int = 1 "java.lang.Integer.parseInt"
```

the expression

```
parseInt "37"
```

would be translated to this Lambda code by the front-end:

```
Lprim(Pccall("java.lang.Integer.parseInt", 1), ["37"])
```

This Lambda expression could then be compiled directly into invocation of the mentioned method (with `Jinvokestatic`); all method arguments should always be represented as boxed values at run-time.

7.4 Representation of SML Strings

Instead of representing SML values of type `string` as `String` objects, they could be represented as instances of this class:

[1]An intelligent JVM should ignore any excessive `Jcheckcast` instructions.

```
public class CharVector
{
  public char[] chars;

  public CharVector (char[] chars  { this.chars = chars; }

  public boolean equals (Object x)  { ... }

}
```

An instance of class `CharVector` holds a mutable array of characters, `chars`. The idea is that the `CharVector` object serves as a 'wrapper' for the character array.

Since less copying is necessary to build a `CharVector` object, manipulate its contents, or extract the enclosed character array, efficient implementation of various library functions manipulating character sequences would be possible. At the same time, SML semantics would guarantee that a `string` value cannot not manipulated by user code (library functions would exploit special primitives to manipulate and build strings).

However, more copying would be necessary in connection with tests involving string literals, since a `CharVector` object cannot be compared directly to a `String` object. String literals go into the constant pool of the target class and are represented as `String` objects when they are loaded at run-time.

7.5 Improving Closure Specialization

The closure specialization implemented in the new compiler back-end may be improved in a number of ways, e.g. by:

- generating specialized subclasses of `Closure` (with 0, 1, ... n free variables),
- generating one subclass of `Closure` per closure-building expression (exception for those where only the specialized closure is ever invoked),
- calling manifest functions in other modules directly (using `Jinvokestatic`),
- directly creating a closure object corresponding to the actual number of supplied arguments in a non-saturated call to a curried function,
- storing the free variables of a closure in the local variable that the closure should have been bound to, in case the general version of the closure is not used; the free vars could then be loaded directly from the local variable or static field when the specialized closure (method) is to be invoked,
- re-using the `Object[]` free variable for a set of cascaded closures, by making it 'big enough' from the beginning; this way, the first entries in the array need not be copied over and over again,
- minimizing the number of free variables actually passed to a function by passing only the free variables that are actually used, that is, avoiding references to functions that are invoked directly (e.g. the function itself).

7.6 Improving the Representation of Values

The run-time support classes for representation of SML values may be improved by:

- introducing an instance initialization method `<init>(int, Object)` in class `Constructor`, for initializing constructor objects with a single argument,
- introducing an instance initialization method `<init>(int, Object)` in class `Exception`, for initializing exceptions with only one argument,
- using specialized `Tuple`, `Constructor` and `Exception` classes (with 0, 1, ..., n arguments).

7.7 Code Generation Improvements

The Java bytecode generated by the new compiler may be improved by:

- comparing with pre-allocated null-constructors in tests (using `Jif_acmpeq` or `Jif_acmpne`),
- using integer compare-with-zero instructions (`Jifeq`, `Jifne`, etc.) wherever possible,
- using `Jtableswitch` instead of `Jlookupswitch` when the switch is 'dense enough',
- eliminating common sub-expressions (using `Jdup`),
- placing the code for an SML exception handler at the end of the method (seperate from the code of the body of the handler); this way, we can spare the unconditional branch at the end of the handler body; but the handler itself must branch to the end of the handler body (unless it raises another exception or returns from the method).

7.8 Other Suggestions

- Generating an instance initializer method `<init>(int, Object)` in the target class. This could be used for initializing closures with only one free variable.
- Optimizing use of 'local' (i.e., non-escaping) `ref` cells via the `Lassign` Lambda clause, as in Moscow ML.
- Changing the compiler front-end to avoid using a `Llet` clause around each `Lcase` expression, unless actually necessary.
- Including function arity in compiled signature (`.ui`) files would facilitate implementation of direct invocation of functions in other modules.

7.9 Improving the SML-JVM Toolkit

The SML-JVM toolkit (cf. Chapter 3) already supports the entire Java class file format, as specified in *The Java Virtual Machine Specification*[4]. However, the toolkit could be improved in a number of ways, e.g. by

- changing the field declarations component of the `Classfile.class_decl` data type to be a map from field name to flags, type, and attributes,
- changing the method declarations component of the `Classfile.class_decl` data type to be a map from method name and signature to flags and attributes,
- enhancing the label resolution algorithm to always use the most compact branching instruction (cf. Section 3.5.2),
- implementing function `Classfile.scan` to read the contents of a physical class file and convert it to an abstract class declaration (`class_decl`),

- implementing verification of class declarations to ensure that generated class files comply with the rules in *The Java Virtual Machine Specification*[4], and to check that class files read from disk observe the same rules,
- adding descriptions to the SML-JVM toolkit signatures, in order to take advantage of the module documentation feature of Moscow ML, and
- modifying the SML-JVM toolkit to use only SML Basis Library facilities[3], thereby improving portability across SML implementations.

Chapter 8

Conclusion

A compiler for Standard ML (SML) has been developed, based on the Moscow ML compiler (version 1.42). The existing back-end of the Moscow ML compiler, generating Caml Light bytecode, has been replaced with a new back-end generating Java bytecode.

The new compiler supports the core language of SML and a simple module system (with 'flat' structures, as in Moscow ML). There is no support for Basis Library modules, and no interactive top-level is provided.

A few optimizations have been implemented, most notably:

- direct calls to manifest functions (via specialized closures), and
- avoiding wrap-unwrap sequences in generated bytecode.

Moreover, preparations for type-based optimizations have been made.

A toolkit for representating Java bytecode and Java class declarations in SML, and for generating binary Java class files, has been developed and implemented. This 'SML-JVM toolkit' provides an abstract view of the Java Virtual Machine instruction set, and of Java class files. The toolkit has proven useful in the implementation of a Java bytecode generator for the new compiler back-end.

Moreover, the SML-JVM toolkit may be used as a general library for representing Java bytecode and class files, as it is not tied to the new compiler. Extending the toolkit to read a class file from disk and convert it to an abstract class declaration should be rather straightforward.

The compiler has been tested to verify that the expected bytecode is generated, and that implemented optimizations lead to improved bytecode and target class files.

Benchmarks show that Java bytecode programs compiled with the new compiler execute 6 to 58 times slower than the same programs compiled with Moscow ML. The Java bytecode generated by the new compiler back-end for a sample program calculating Fibonacci numbers also executes 7 times slower than a similar program compiled from Java. These figures are, of course, rather disappointing.

However, some important potential optimizations have been identified:

- optimization of tail-calls,
- avoiding unnecessary run-time type checks, and
- avoiding boxing where possible.

Since tail-calls are not optimized in the new compiler, frame stack space is exhausted in connection with recursion; recursion may also lead to space leaks in the heap. In general, optimization of tail-calls requires support for this in the JVM. Unfortunately, this is not available in existing JVM implementations.

Presumably, a significant speed-up may be achieved, even with existing JVM implementations, by compiling directly tail-recursive functions to bytecode loops (i.e., by jumping rather than calling).

Minimizing run-time type checks and boxing requires more type information than is currently available in the intermediate language of the compiler. Introducing type information in the Lambda language, or inferring such information from the intermediate program itself, would presumably lead to significantly better performance of the generated bytecode.

Appendix A

User's Guide

The source files for the new compiler are available for anonymous FTP from:

`ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Bertelsen/mosml-jvm.zip`

This is an archive containing the source files for the compiler, the SML-JVM toolkit source files, and the run-time support classes. The compiler has been tested on Linux and Solaris, but, presumably, may be built on other platforms as well.

The files may be extracted from the archive using Info-Zip's `unzip` utility. This will create the following directory structure:

`mosml-jvm/bin/`	Compiler executables
`mosml-jvm/classes/sml/lang/`	Compiled run-time support classes
`mosml-jvm/src/classes/`	Run-time support class source files
`mosml-jvm/src/compiler`	Compiler source files
`mosml-jvm/src/sml-jvm-toolkit`	SML-JVM toolkit source files
`mosml-jvm/test/`	Test programs
`mosml-jvm/test/benchmark/`	Benchmark programs

Installing the compiler requires:

- Moscow ML for compiling the SML-JVM toolkit and the new compiler itself. Moscow ML is available for anonymous FTP from: `ftp://ftp.dina.kvl.dk/pub/mosml/`. The new compiler, as well as the SML-JVM toolkit, has been compiled using Moscow ML version 1.42.
- GNU make. The make files for the new compiler have been tested with GNU make version 3.74.
- Sun's *Java Runtime Environment* (JRE) for running target programs generated by the new compiler. Sun's JRE is available via `http://java.sun.com`. The run-time support classes have been tested with version 1.1.3 of Sun's JVM implementation.

To install the new compiler, change directory to the `mosml-jvm/src/` subdirectory, and invoke `make`. This should compile the SML-JVM toolkit and the compiler itself. Edit the script `mosml-jvm/bin/mosmlc` to suit your installation: `bindir` must equal the absolute path to the subdirectory `mosml-jvm/bin`, whereas `stdlib` must equal the absolute path to the `lib/` subdirectory of the Moscow ML installation.

In order to make the run-time support classes known to the JVM, include the absolute path to the subdirectory `mosml-jvm/classes/` in the `CLASSPATH` environment setting.

Assuming that installation completed normally, the compiler may now be invoked on an SML program, e.g.

```
dir/mosml-jvm/bin/mosmlc A.sml
```

where *dir* is the path to the directory in which you unpacked the `mosml-jvm.zip` archive. This will create the files `A.ui` and `A.class`, provided `A.sml` contains a valid SML program. The program may then be executed by invoking Sun's JVM on the generated class file:

```
java A
```

Note that the new compiler does not support the full SML module system; only 'flat' structures are supported, as in Moscow ML. Also, there is no interactive top-level; only batch compilation is supported.

Moreover, support for the SML Basis Library modules[3] is not available with the new compiler. In order to implement programs that produce any kind of output, the user will have to implement Java wrapper classes for invoking proper I/O methods in the standard Java classes. For an example on how to do this, see the file `mosml-jvm/test/util.java`.

Appendix B

Assumptions on JVM Semantics

The Java Virtual Machine Specification[4] is intended as a complete specification of the JVM and its semantics. Unfortunately, the specification is not complete: many details in the constraints on Java class files contents, and in the semantics of JVM instructions, are left unspecified or treated ambiguously[8].

Although the SML-JVM toolkit does not perform complete verification of classes (cf. Chapter 3), generated class files must comply with the required structure of Java class files. In practice this means that generated class files must be accepted by Sun's bytecode verifier in order to be executed by Sun's JVM implementation[1]. To this end the implemented compiler back-end generates Java bytecode and class files satisfying the following requirements:

1. The Java bytecode of a class file must obey a static typing discipline. That is, given the declared types of fields, method arguments, and method return values, it must be possible to verify statically that each bytecode instruction is applied to arguments of proper types. In case a bytecode instruction requires an operand of a more specific type than what can be inferred from static type information, a `checkcast` instruction must be present to ensure the proper dynamic type check.
2. An entry in the `methods` table of a class file need not include a `Code` attribute for the method. That is, if the method is flagged as `ACC_ABSTRACT` or `ACC_NATIVE` no bytecode implementation of the method should be included.
3. An entry in the `methods` table of a class file need not include an `Exceptions` attribute for the method.
4. A class file cannot contain two or more field declarations with identical names.
5. A class file cannot contain two or more method declarations with identical names and signatures.
6. The order of entries in the `exception_table` of a `Code` method attribute is significant. When an exception is thrown the exception handlers of the method are searched in declared order.
7. There must never be a reference to an uninitialized object on the operand stack or in a local variable when one of these instructions are executed with a negative branch offset argument:

[1]According to *The Java Virtual Machine Specification*[4], Sun's JVM implementation is *not* a reference implementation. However, since all other JVM implementations are measured against Sun's implementation, it must be considered a de facto reference implementation.

- `if_acmpeq` or `if_acmpne`
- `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpge`, `if_icmpgt`, or `if_icmple`
- `ifeq`, `ifne`, `iflt`, `ifge`, `ifgt`, or `ifle`
- `ifnonnull` or `ifnull`
- `tableswitch` or `lookupswitch`
- `goto` or `goto_w`

8. The instruction `aastore` may not be used to store a reference to an uninitialized object into an array component.
9. The operand stack manipulation instructions (`dup`, `dup_x1`, `dup_x2`, `dup2`, `dup2_x1`, `dup2_x2`, `pop`, `pop2`, and `swap`) may be used on references to uninitialized objects.
10. The `ireturn` instruction must be used to return from methods of return type `boolean`, `byte`, `char`, `short`, and `int`.
11. The class of an object loaded from or stored into by a `getfield` or `putfield` instruction must be the class declaring the accessed instance field or a subclass thereof.
12. The `invokestatic` instruction must be used for invoking a method flagged as `ACC_PRIVATE` and `ACC_STATIC`.
13. Each `invokespecial` instruction must refer to an `<init>` method, an instance method of the current class, or an instance method of a superclass of the current class.
14. The class of the object that an `invokespecial` or `invokevirtual` instruction is invoked on must be the same as the class declaring the invoked instance method or a subclass thereof.
15. Each instance initialization method, except for `java.lang.Object.<init>`, must invoke another instance initialization method on the (uninitialized) object reference in local variable 0 before instance members of the object are accessed. The invoked instance initialization method must be a member of the current class or its immediate superclass.
16. The special value `null` may be stored into a field of reference type, and may be passed as a method argument if and only if the corresponding formal parameter of the invoked method is of reference type.
17. The `areturn` instruction may be used to return the special value `null` from a method whose return type is a reference type.
18. Access permissions to classes and members correspond to those defined for the Java language[2].
19. Array bounds must be checked at every access to an array component. Valid indices are 0 through $n - 1$, where n is the length of the array.
20. When a method return instruction is executed, the current frame is discarded. If that frame was the last on the frame stack of the current thread, then the thread terminates. If at least one frame remains on the frame stack, control is transferred to the top-most frame, and the instruction counter is set to the address of the instruction immediately following the method invocation instruction causing the frame of the method return instruction to be created.
21. When the `invokevirtual` instruction is executed on an object, the methods of that object are searched for an implementation of the referenced method, starting at the class of the object, and recursively following the reference to the immediate superclass.
22. When a method invocation instruction is executed, the current frame may be discarded, provided

- the method invocation instruction is immediately followed by a method return instruction corresponding to the return type of the current method,
- the return type of the invoked method is the same as that of the current method (or the invoked method, as well as the current method, returns no value), and
- the method invocation instruction is not covered by an exception handler,
- the current method is not flagged as `ACC_SYNCHRONIZED`.

23. If an exception is thrown during execution of the `<clinit>` method of a class or interface, and the exception is not caught by an exception handler of the method, then the current class/interface is considered uninitialized and an exception is thrown in the method that caused the current class/interface to be loaded and initialized.

Appendix C

An Example: Fibonacci Numbers

C.1 Source Program

```
(* fibonacci.sml *)

fun fib n =
    let fun fib' 0 k1 k2 = k1
          | fib' i k1 k2 = fib' (i-1) (k1+k2) k1
    in
        if n < 2 then n
        else fib' (n-2) 1 1
    end

val _ = (util.print "fib(10) = ";
         util.println(fib 10))
```

C.2 Lambda Code

```
===== before lifting =====
(prim (set_global fibonacci.fib/1) (fn letrec (fn (fn (fn ((case var:2
of 0 : var:1) statichandle let (prim (smlsubint) var:2 1) in (app
var:4 var:0 (prim (smladdint) var:2 var:1) var:2) end)))) in if(prim
(test:lt) var:1 2) then (var:1) else (app var:0 (prim (smlsubint)
var:1 2) 1 1) end))

let ((app (prim (get_global util.print/0) ) "fib(10) = "); (app (prim
(get_global util.println/0) ) (app (prim (get_global fibonacci.fib/1)
) 10))) in unspec end

===== after lifting =====
(prim (set_global fibonacci.fib/1) (prim (closure 0) ))

let ((app (prim (get_global util.print/0) ) "fib(10) = "); (app (prim
(get_global util.println/0) ) (app (prim (get_global fibonacci.fib/1)
) 10))) in unspec end

--- closure 0, depth 1, [] free ---
letrec (prim (closure 1) var:1) in if(prim (test:lt) var:0 2) then
(var:0) else (app var:1 (prim (smlsubint) var:0 2) 1 1) end

--- closure 1, depth 1, [ ~1] free ---
(prim (closure 2) var:~1 var:0)
```

```
--- closure 2, depth 1, [ ~2 ~1] free ---
(prim (closure 3) var:~2 var:~1 var:0)

--- closure 3, depth 1, [ ~3 ~2 ~1] free ---
((case var:~2 of 0 : var:~1) statichandle let (prim (smlsubint) var:~2
1) in (app var:~3 var:1 (prim (smladdint) var:~1 var:0) var:~1) end)

--- method 0, depth 1, [] free, 1 args ---
letrec (prim (closure 1) var:1) in if(prim (test:lt) var:0 2) then
(var:0) else (app var:1 (prim (smlsubint) var:0 2) 1 1) end

--- method 1, depth 1, [ ~1] free, 3 args ---
((case var:~2 of 0 : var:~1) statichandle let (prim (smlsubint) var:~2
1) in (app var:~3 var:1 (prim (smladdint) var:~1 var:0) var:~1) end)
```

C.3 Generated Java Class File

```
Compiled from fibonacci.sml
public synchronized class fibonacci extends sml.lang.Closure
    /* ACC_SUPER bit set */
{
    public static java.lang.Object fib$0;
    public static java.lang.Object fib$0(java.lang.Object);
    private static java.lang.Object clos$$1(java.lang.Object[], java.lang.Object, java.lang.Object
java.lang.Object);
    private fibonacci(int);
    private fibonacci(int,java.lang.Object[]);
    public java.lang.Object apply(java.lang.Object);
    public static void main(java.lang.String[]);
    static static {};
}

Method java.lang.Object fib$0(java.lang.Object)
   0 new #2 <Class fibonacci>
   3 dup
   4 iconst_1
   5 invokespecial #11 <Method fibonacci(int)>
   8 dup
   9 astore_1
  10 iconst_1
  11 anewarray class #13 <Class java.lang.Object>
  14 dup
  15 iconst_0
  16 aload_1
  17 aastore
  18 putfield #17 <Field java.lang.Object free[]>
  21 aload_0
  22 checkcast #19 <Class java.lang.Number>
  25 invokevirtual #23 <Method int intValue()>
  28 iconst_2
  29 if_icmpge 34
  32 aload_0
  33 areturn
  34 aload_1
  35 getfield #17 <Field java.lang.Object free[]>
  38 new #25 <Class java.lang.Integer>
  41 dup
  42 aload_0
  43 checkcast #19 <Class java.lang.Number>
  46 invokevirtual #23 <Method int intValue()>
  49 iconst_2
  50 isub
  51 invokespecial #26 <Method java.lang.Integer(int)>
  54 new #25 <Class java.lang.Integer>
  57 dup
  58 iconst_1
  59 invokespecial #26 <Method java.lang.Integer(int)>
  62 new #25 <Class java.lang.Integer>
  65 dup
  66 iconst_1
  67 invokespecial #26 <Method java.lang.Integer(int)>
  70 invokestatic #30 <Method java.lang.Object clos$$1(java.lang.Object[], java.lang.Object,
java.lang.Object, java.lang.Object)>
  73 areturn

Method java.lang.Object clos$$1(java.lang.Object[], java.lang.Object, java.lang.Object,
java.lang.Object)
   0 aload_1
   1 checkcast #19 <Class java.lang.Number>
   4 invokevirtual #23 <Method int intValue()>
   7 ifne 12
  10 aload_2
  11 areturn
  12 new #25 <Class java.lang.Integer>
  15 dup
  16 aload_1
  17 checkcast #19 <Class java.lang.Number>
  20 invokevirtual #23 <Method int intValue()>
  23 iconst_1
  24 isub
  25 invokespecial #26 <Method java.lang.Integer(int)>
  28 astore 4
  30 aload_0
  31 iconst_0
  32 aaload
  33 checkcast #4 <Class sml.lang.Closure>
  36 getfield #17 <Field java.lang.Object free[]>
  39 aload 4
  41 new #25 <Class java.lang.Integer>
  44 dup
  45 aload_2
  46 checkcast #19 <Class java.lang.Number>
  49 invokevirtual #23 <Method int intValue()>
  52 aload_3
  53 checkcast #19 <Class java.lang.Number>
  56 invokevirtual #23 <Method int intValue()>
  59 iadd
  60 invokespecial #26 <Method java.lang.Integer(int)>
  63 aload_2
  64 invokestatic #30 <Method java.lang.Object clos$$1(java.lang.Object[], java.lang.Object,
java.lang.Object, java.lang.Object)>
  67 areturn

Method fibonacci(int)
   0 aload_0
   1 dup
   2 invokespecial #34 <Method sml.lang.Closure()>
   5 iload_1
   6 putfield #38 <Field int tag>
   9 return

Method fibonacci(int,java.lang.Object[])
   0 aload_0
   1 dup
   2 invokespecial #34 <Method sml.lang.Closure()>
   5 dup
   6 iload_1
   7 putfield #38 <Field int tag>
  10 aload_2
  11 putfield #17 <Field java.lang.Object free[]>
  14 return

Method java.lang.Object apply(java.lang.Object)
   0 aload_0
   1 getfield #38 <Field int tag>
   4 tableswitch 0 to 4: default=40
           0: 205
           1: 179
           2: 144
           3: 50
           4: 40
  40 new #42 <Class sml.lang.SmlError>
  43 dup
  44 ldc #44 <String "unmatced closure tag">
  46 invokespecial #47 <Method sml.lang.SmlError(java.lang.String)>
  49 athrow
  50 aload_0
  51 getfield #17 <Field java.lang.Object free[]>
  54 iconst_1
  55 aaload
  56 checkcast #19 <Class java.lang.Number>
  59 invokevirtual #23 <Method int intValue()>
  62 ifne 72
  65 aload_0
  66 getfield #17 <Field java.lang.Object free[]>
  69 iconst_2
  70 aaload
  71 areturn
  72 new #25 <Class java.lang.Integer>
  75 dup
  76 aload_0
  77 getfield #17 <Field java.lang.Object free[]>
  80 iconst_1
  81 aaload
  82 checkcast #19 <Class java.lang.Number>
  85 invokevirtual #23 <Method int intValue()>
  88 iconst_1
  89 isub
  90 invokespecial #26 <Method java.lang.Integer(int)>
  93 astore_2
  94 aload_0
  95 getfield #17 <Field java.lang.Object free[]>
  98 iconst_0
  99 aaload
 100 checkcast #4 <Class sml.lang.Closure>
 103 getfield #17 <Field java.lang.Object free[]>
 106 aload_2
 107 new #25 <Class java.lang.Integer>
 110 dup
 111 aload_0
 112 getfield #17 <Field java.lang.Object free[]>
 115 iconst_2
 116 aaload
 117 checkcast #19 <Class java.lang.Number>
 120 invokevirtual #23 <Method int intValue()>
 123 aload_1
 124 checkcast #19 <Class java.lang.Number>
 127 invokevirtual #23 <Method int intValue()>
 130 iadd
 131 invokespecial #26 <Method java.lang.Integer(int)>
 134 aload_0
 135 getfield #17 <Field java.lang.Object free[]>
 138 iconst_2
 139 aaload
 140 invokestatic #30 <Method java.lang.Object clos$$1(java.lang.Object[], java.lang.Object,
java.lang.Object, java.lang.Object)>
 143 areturn
 144 new #2 <Class fibonacci>
 147 dup
 148 iconst_3
 149 iconst_3
 150 anewarray class #13 <Class java.lang.Object>
 153 dup
 154 iconst_0
 155 aload_0
 156 getfield #17 <Field java.lang.Object free[]>
 159 iconst_0
 160 aaload
 161 aastore
 162 dup
 163 iconst_1
 164 aload_0
 165 getfield #17 <Field java.lang.Object free[]>
 168 iconst_1
 169 aaload
 170 aastore
 171 dup
 172 iconst_2
 173 aload_1
 174 aastore
 175 invokespecial #49 <Method fibonacci(int,java.lang.Object[])>
 178 areturn
 179 new #2 <Class fibonacci>
 182 dup
 183 iconst_2
 184 iconst_2
 185 anewarray class #13 <Class java.lang.Object>
 188 dup
 189 iconst_0
 190 aload_0
 191 getfield #17 <Field java.lang.Object free[]>
 194 iconst_0
 195 aaload
 196 aastore
 197 dup
 198 iconst_1
 199 aload_1
 200 aastore
 201 invokespecial #49 <Method fibonacci(int,java.lang.Object[])>
 204 areturn
 205 new #2 <Class fibonacci>
 208 dup
 209 iconst_1
 210 invokespecial #11 <Method fibonacci(int)>
 213 dup
 214 astore_2
 215 iconst_1
 216 anewarray class #13 <Class java.lang.Object>
 219 dup
 220 iconst_0
 221 aload_2
 222 aastore
 223 putfield #17 <Field java.lang.Object free[]>
 226 aload_1
 227 checkcast #19 <Class java.lang.Number>
 230 invokevirtual #23 <Method int intValue()>
 233 iconst_2
 234 if_icmpge 239
 237 aload_1
 238 areturn
 239 aload_2
 240 getfield #17 <Field java.lang.Object free[]>
 243 new #25 <Class java.lang.Integer>
 246 dup
 247 aload_1
 248 checkcast #19 <Class java.lang.Number>
 251 invokevirtual #23 <Method int intValue()>
 254 iconst_2
 255 isub
 256 invokespecial #26 <Method java.lang.Integer(int)>
 259 new #25 <Class java.lang.Integer>
 262 dup
 263 iconst_1
 264 invokespecial #26 <Method java.lang.Integer(int)>
 267 new #25 <Class java.lang.Integer>
 270 dup
 271 iconst_1
 272 invokespecial #26 <Method java.lang.Integer(int)>
 275 invokestatic #30 <Method java.lang.Object clos$$1(java.lang.Object[], java.lang.Object,
java.lang.Object, java.lang.Object)>
 278 areturn

Method void main(java.lang.String[])
   0 return

Method static {}
   0 new #2 <Class fibonacci>
   3 dup
   4 iconst_0
   5 invokespecial #11 <Method fibonacci(int)>
   8 putstatic #54 <Field java.lang.Object fib$0>
  11 getstatic #59 <Field java.lang.Object print$0>
  14 checkcast #4 <Class sml.lang.Closure>
  17 ldc #61 <String "fib(10) = ">
  19 invokevirtual #63 <Method java.lang.Object apply(java.lang.Object)>
  22 pop
  23 getstatic #66 <Field java.lang.Object println$0>
  26 checkcast #4 <Class sml.lang.Closure>
  29 new #25 <Class java.lang.Integer>
  32 dup
  33 bipush 10
  35 invokespecial #26 <Method java.lang.Integer(int)>
  38 invokestatic #68 <Method java.lang.Object fib$0(java.lang.Object)>
  41 invokevirtual #63 <Method java.lang.Object apply(java.lang.Object)>
  44 return
```

Appendix D

Run-time Support Classes

```
package sml.lang;

public abstract class Block extends Throwable
{
  public Object[] args;

  protected abstract StringBuffer header (int n);

  public String toString ()
  {
    int n = (args == null)? 0 : args.length;
    StringBuffer s = header(n);

    if (n != 0)
      {
        s.append(args[0]);

        for (int i = 1; i < n; ++i)
          s.append(", " + args[i]);
      }

    return s.append(")").toString();
  }
}
```

```
package sml.lang;

public class Tuple extends Block
{
  public final static Tuple Unit = new Tuple(null);

  public Tuple (Object[] args)
  {
    this.args = args;
  }

  public boolean equals (Object x)
  {
    Object[] thisArgs = args,
             xArgs    = ((Tuple)x).args;
    if (thisArgs != null)
      {
        int i = thisArgs.length;
        // no need to check args.length == xArgs.length,
        // assuming arity(this) == arity(x)

        while (--i >= 0)
          if (!thisArgs[i].equals(xArgs[i]))
            return false;
      }

    return true;
  }

  public String toString ()
  {
    int n = (args == null)? 0 : args.length;
    StringBuffer s = new StringBuffer("Tuple(");

    if (n != 0)
      {
        s.append(args[0]);

        for (int i = 1; i < n; ++i)
          s.append(", " + args[i]);
      }

    return s.append(")").toString();
  }

  public StringBuffer header (int n)
  {
    return new StringBuffer("Tuple(");
  }
}
```

```
package sml.lang;

public class Constructor extends Block
{
  public final static Constructor
    Zero  = new Constructor(0),
    One   = new Constructor(1),
    Two   = new Constructor(2),
    Three = new Constructor(3),
    Four  = new Constructor(4),
    Five  = new Constructor(5),
    Six   = new Constructor(6),
    Seven = new Constructor(7),
    Eight = new Constructor(8),
    Nine  = new Constructor(9);

  public int tag;

  public Constructor (int tag)
  {
    this.tag = tag;
  }

  public Constructor (int tag, Object[] args)
  {
    this.tag = tag;
    this.args = args;
  }

  public boolean equals (Object x)
  {
    Constructor X = (Constructor)x;

    if (tag != X.tag)
      return false;

    Object[] thisArgs = args;

    if (thisArgs != null)
      {
        Object[] xArgs = X.args;
        int i = thisArgs.length;
        // no need to check args.length == xArgs.length,
        // assuming tag == X.tag => arity(this) == arity(X)

        while (--i >= 0)
          if (!thisArgs[i].equals(xArgs[i]))
            return false;
      }

    return true;
  }

  public StringBuffer header (int n)
  {
    StringBuffer s = new StringBuffer("Constructor(" + tag);
    return (n > 0) ? s.append(": ") : s;
  }
}
```

```
package sml.lang;

public class Ref extends Block
{
  public Ref (Object arg)
  {
    Object[] args = {arg};
    this.args = args;
  }

  // NOTE: since class Ref does not override method Block.equals,
  // equality on Ref objects corresponds to pointer equality, as
  // required in SML.

  public StringBuffer header (int n)
  {
    return new StringBuffer("Ref(");
  }
}
```

```
package sml.lang;

public class Exception extends Block
{
  public Object tag;

  public Exception (Object tag)
  {
    this.tag = tag;
  }

  public Exception (Object tag, Object[] args)
  {
    this.tag = tag;
    this.args = args;
  }

  // It would be appropriate to override method equals here, in order
  // to throw a JVM exception in case polymorphic equality is invoked
  // on an Exception object.  This should never happen since
  // exceptions do not admit equality in SML.  In Java, however, we
  // cannot override method equals to throw an exception since
  // Object.equals is not declared to throw any exceptions.

  public StringBuffer header (int n)
  {
    StringBuffer s = new StringBuffer("Exception(" + tag);
    return (n > 0)? s.append(": ") : s;
  }
}
```

```
package sml.lang;

public abstract class Closure
{
  public int tag;
  public Object[] free;

  public Closure () {}

  public abstract Object apply (Object x) throws Throwable;

  // It would be appropriate to override method equals here, in order
  // to throw a JVM exception in case polymorphic equality is invoked
  // on a Closure object.  This should never happen because function
  // types do not admit equality in SML.  In Java, however, we cannot
  // override method equals to throw an exception since Object.equals
  // is not declared to throw any exceptions.

  public String toString ()
  {
    return ("Closure(" + tag + ")");
  }
}
```

```
package sml.lang;

public class General
{
  public final static Object
    Bind$x0            = new String("General.Bind"),
    Chr$x0             = new String("General.Chr"),
    Div$x0             = new String("General.Div"),
    Domain$x0          = new String("General.Domain"),
    Match$x0           = new String("General.Match"),
    Ord$x0             = new String("General.Ord"),
    Overflow$x0        = new String("General.Overflow"),
    Fail$x0            = new String("General.Fail"),
    $28Exception$29$x0 = new String("General.(Exception)");
}
```

Appendix E

SML-JVM Toolkit Source Files

```
(* Classdecl.sml
 *
 * Peter Bertelsen
 * September 1997
 *)

local
    open Label Bytecode Jvmtype
in
    datatype access_flag =
        ACCpublic        (* field/method/class/interface *)
      | ACCprivate        (* field/method *)
      | ACCprotected        (* field/method *)
      | ACCstatic        (* field/method *)
      | ACCfinal        (* field/method/class *)
      | ACCvolatile        (* field *)
      | ACCtransient        (* field *)
      | ACCsynchronized        (* method *)
      | ACCnative        (* method *)
      | ACCabstract        (* method/class/interface *)
      | ACCsuper        (* class/interface *)
      | ACCinterface        (* interface *)

    type exn_hdl =   (* exception handler declaration *)
        {start: label,               (* start of handler scope *)
         stop:  label,               (* end of handler scope *)
         entry: label,               (* handler entry point *)
         catch: jclass option}       (* class of handled exception; NONE = any *)

    datatype attribute =
        SRCFILE of string                           (* source file name *)
      | CONSTVAL of jvm_const                          (* constant value *)
      | CODE of {stack:  int,                          (* max operand stack depth *)
                 locals: int,                          (* max size of local vars *)
                 code:   jvm_instr list,        (* instruction sequence *)
                 hdls:   exn_hdl list,                   (* exception handlers *)
                 attrs:  attribute list}
      | EXNS of jclass list                          (* exception classes *)
      | LINENUM of {start: label,
                    line:  int} list
      | LOCALVAR of {from:  label,
                     thru:  label,
                     name:  string,
                     ty:    jtype,
                     index: Localvar.index} list
      | ATTR of {attr: string, info: Word8Vector.vector}

    type field_decl =        (* field declaration *)
        {flags:  access_flag list,       (* access flags *)
         name:   string,               (* unqualified name *)
         ty:     jtype,                       (* field type *)
         attrs:  attribute list}       (* field attributes *)

    type method_decl =        (* method declaration *)
        {flags:  access_flag list,       (* access flags *)
         name:   string,               (* unqualified name *)
         msig:   method_sig,               (* method signature *)
         attrs:  attribute list}       (* method attributes *)

    type class_decl =        (* class/interface declaration *)
        {flags:  access_flag list,       (* access flags *)
         this:   jclass,               (* this class/interface *)
         super:  jclass option,               (* direct superclass *)
         ifcs:   jclass list,             (* direct superinterfaces *)
         fdecls: field_decl list,        (* field declarations *)
         mdecls: method_decl list,       (* method declarations *)
         attrs:  attribute list}
end (* local *)
```

```
(* Jvmtype.sig
 *
 * Peter Bertelsen
 * December 1997
 *)

exception InvalidType of string

eqtype jclass

val class     : {pkgs: string list, name: string} -> jclass
val packages  : jclass -> string list
val className : jclass -> string
val qualName  : jclass -> string

datatype jtype =
    Tboolean
  | Tchar
  | Tfloat
  | Tdouble
  | Tbyte
  | Tshort
  | Tint
  | Tlong
  | Tarray of jtype
  | Tclass of jclass

val isSimple : jtype -> bool

val arrayOf   : int * jtype -> jtype
val arrayDim  : jtype -> int
val arrayBase : jtype -> jtype

type method_sig = jtype list * jtype option

val width          : jtype -> int
val typeDesc       : jtype -> string
val scanTypeDesc   : string -> jtype option
val methodDesc     : method_sig -> string
val scanMethodDesc : string -> method_sig option
```

```
(* Jvmtype.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

exception InvalidType of string

fun typeError s = raise InvalidType("Jvmtype." ^ s)

datatype jclass = CLASS of {pkgs: string list, name: string}

fun class (c as {pkgs, name}) =
    let fun h (#".", _) = false
          | h (#"/", _) = false
          | h (#"[", _) = false
          | h (#"]", _) = false
          | h (_,   ok) = ok

        fun validName s = Substring.foldl h true (Substring.all s)
    in
        if List.all validName pkgs andalso validName name then CLASS c
        else typeError "class: invalid class identifier"
    end

fun className (CLASS {name, ...}) = name

fun packages (CLASS {pkgs, ...}) = pkgs

fun revAppend s cs = CharVector.foldl (op ::) cs s

fun revAppendDelim (s, cs) = #"/" :: (revAppend s cs)

fun revQualName cs {pkgs, name} =
        revAppend name (List.foldl revAppendDelim cs pkgs)

fun qualName (CLASS c) = String.implode(rev (revQualName [] c))

(*
val insDots  = List.foldl (insDelim #".") []

fun javaQualName (CLASS {pkgs, name}) =
        String.implode(rev (revAppend name (insDots pkgs)))
  | javaQualName (ARRAY _) =
*)

datatype jtype =
    Tboolean
  | Tchar
  | Tfloat
  | Tdouble
  | Tbyte
  | Tshort
  | Tint
  | Tlong
  | Tarray of jtype
  | Tclass of jclass
```

```
fun isSimple t =
    (case t of
         Tboolean => true
       | Tchar    => true
       | Tfloat   => true
       | Tdouble  => true
       | Tbyte    => true
       | Tshort   => true
       | Tint     => true
       | Tlong    => true
       | _        => false)

type method_sig = jtype list * jtype option

fun arrayOf' (0, t) = t
  | arrayOf' (k, t) = arrayOf'(k-1, Tarray t)
fun arrayOf  (n, t) = if n >= 0 then arrayOf'(n, t)
                      else raise Domain

fun arrayDim' (Tarray t, res) = arrayDim'(t, res + 1)
  | arrayDim' (_,        res) = res
fun arrayDim (Tarray t) = arrayDim'(t, 1)
  | arrayDim   _        = raise Domain

fun arrayBase' (Tarray t) = arrayBase' t
  | arrayBase' t = t
fun arrayBase  (Tarray t) = arrayBase' t
  | arrayBase   _         = raise Domain

fun width t =
    (case t of
         Tboolean => 1
       | Tchar    => 1
       | Tfloat   => 1
       | Tdouble  => 2
       | Tbyte    => 1
       | Tshort   => 1
       | Tint     => 1
       | Tlong    => 2
       | Tarray _ => 1
       | Tclass _ => 1)

fun revDesc (t, cs) =
    (case t of
         Tboolean     => #"Z" :: cs
       | Tchar        => #"C" :: cs
       | Tfloat       => #"F" :: cs
       | Tdouble      => #"D" :: cs
       | Tbyte        => #"B" :: cs
       | Tshort       => #"S" :: cs
       | Tint         => #"I" :: cs
       | Tlong        => #"J" :: cs
       | Tarray t'    => revDesc(t', #"[" :: cs)
       | Tclass (CLASS c) => #";" :: (revQualName (#"L" :: cs) c))

fun typeDesc t =
    let val cs = revDesc(t, [])
    in
        String.implode(rev cs)
    end

fun methodDesc (argTs, retT) =
    let val cs   = foldl revDesc [#"("] argTs
        val cs'  = #")" :: cs
        val cs'' = (case retT of
                        NONE   => #"V" :: cs'
```

```
                    | SOME t => revDesc(t, cs'))
    in
        String.implode(rev cs'')
    end

local
    open Substring

    val splitQualName = splitl (fn c => c <> #"/" andalso c <> #";")

    fun scanQualName ids ss =
        let val (id, ss') = splitQualName ss
            val id'  = string id
            val ss'' = triml 1 ss'
        in
            if isEmpty ss' then
                typeError "scanQualName: invalid qualified name"
            else
                if sub(ss', 0) = #";" then (rev ids, id', ss'')
                else scanQualName (id' :: ids) ss''
        end

    fun scan' dim ss =
        let val ss' = triml 1 ss
        in
            (case sub(ss, 0) of
                 #"Z" => (dim, Tboolean, ss')
               | #"C" => (dim, Tchar,   ss')
               | #"F" => (dim, Tfloat,  ss')
               | #"D" => (dim, Tdouble, ss')
               | #"B" => (dim, Tbyte,   ss')
               | #"S" => (dim, Tshort,  ss')
               | #"I" => (dim, Tint,    ss')
               | #"J" => (dim, Tlong,   ss')
               | #"[" => scan' (dim + 1) (triml 1 ss')
               | #"L" => let val (pkgs, name, ss'') = scanQualName [] ss'
                            val class = CLASS {pkgs = pkgs, name = name}
                        in
                            (dim, Tclass class, ss'')
                        end
               | _ => typeError "scan: invalid type descriptor")
        end

    fun scan ss =
        let val (dim, t, ss') = scan' 0 ss
        in
            (arrayOf'(dim, t), ss')
        end

    fun scanArgs (ss, ts) =
        if sub(ss, 0) = #")" then
            (rev ts, triml 1 ss)
        else
            let val (t, ss')  = scan ss
            in
                scanArgs(ss', t :: ts)
            end
in
    fun scanTypeDesc s =
        let val (t, rest) = scan(all s)
        in
             if isEmpty rest then SOME t
             else NONE
        end

    fun scanMethodDesc s =
        let val ss = all s
        in
             if sub(ss, 0) = #"(" then
                 let val (args, ss') = scanArgs(triml 1 ss, [])
                 in
                     if sub(ss', 0) = #"V" then SOME(args, NONE)
                     else
                         let val (ret, ss'') = scan ss'
                         in
                             if isEmpty ss'' then SOME(args, SOME ret)
                             else NONE
                         end
                 end
             else NONE
        end
end
```

```
(* Label.sig
 *
 * Peter Bertelsen
 * September 1997
 *)

eqtype label
type   labels

val toString : label -> string

val freshLabels : labels
val newLabel    : labels -> labels * label
```

```
(* Label.sml
 *
 * Peter Bertelsen
 * August 1997
 *)

datatype label  = LBL of int
datatype labels = LBLS of int

fun toString (LBL n) = "label " ^ (Int.toString n)

val freshLabels = LBLS 0

fun newLabel (LBLS n) = (LBLS (n+1), LBL n)
```

```
(* Localvar.sig
 *
 * Peter Bertelsen
 * December 1997
 *)

exception LocalvarsFull
```

```
type locals
eqtype index

val this     : index
val fromInt  : int -> index
val toInt    : index -> int
val toString : index -> string
val compare  : index * index -> order

val freshLocals : locals
val count       : locals -> int   (* calculate the number of slots used *)
val nextVar1    : locals -> locals * index
val nextVar2    : locals -> locals * index

(* NOTE:
 * fromInt raises Domain if the specified index is invalid
 *
 * nextVar1 allocates a slot for a one-word variable
 *
 * nextVar2 allocates a slot for a two-word variable
 *)
```

```
(* Localvar.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

exception LocalvarsFull

datatype index  = VAR of int
datatype locals = LOCALS of int

val maxIndex1 = 0xfffe    (* maximum number of local vars is 65535 *)
val maxIndex2 = maxIndex1 - 1

val this = VAR 0

fun fromInt n =
    if n <= maxIndex1 then VAR n
    else raise Domain

fun toInt (VAR n) = n

fun toString (VAR n) = "var " ^ (Int.toString n)

fun compare (VAR m, VAR n) = Int.compare(m, n)

val freshLocals = LOCALS 0

fun count (LOCALS n) = n

fun nextVar1 (LOCALS n) =
         if n <= maxIndex1 then (LOCALS (n+1), VAR n)
        else raise LocalvarsFull

fun nextVar2 (LOCALS n) =
        if n <= maxIndex2 then (LOCALS (n+2), VAR n)
        else raise LocalvarsFull
```

```
(* Bytecode.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

exception InternalError of string
exception Unimplemented of string

datatype jvm_const =
    Cnull
  | Cint    of Int32.int
  | Cfloat  of Real32.real
  | Clong   of Int64.int
  | Cdouble of Real64.real
  | Cstring of string

local
    open Label Localvar Jvmtype
in
    datatype class_ref =
        CLASS of jclass   (* a class or interface *)
      | ARRAY of jtype    (* an 'array class' of the given element type *)

    type field_ref = {class : jclass,   (* not an array class *)
                      name  : string,
                      ty    : jtype}

    type method_ref = {class : jclass,  (* not an array class *)
                       name  : string,
                       msig  : method_sig}

    (* The type of JVM instructions, extended with labels *)
    datatype jvm_instr =
        Jlabel of label
      | Jsconst of string
      | Jaaload
      | Jaastore
      | Jaconst_null
      | Jaload of index
   (* | Jaload_0 | Jaload_1 | Jaload_2 | Jaload_3 *)
   (* | Janewarray of class_ref *)
   (* | Jareturn *)
      | Jarraylength
      | Jastore of index
   (* | Jastore_0 | Jastore_1 | Jastore_2 | Jastore_3 *)
      | Jathrow
      | Jbaload
      | Jbastore
   (* | Jbipush of int *)
      | Jcaload
      | Jcastore
      | Jcheckcast of class_ref
      | Jd2f
      | Jd2i
      | Jd2l
      | Jdadd
      | Jdaload
      | Jdastore
      | Jdcmpg | Jdcmpl
      | Jdconst of Real64.real
   (* | Jdconst_0 | Jdconst_1 *)
      | Jddiv
      | Jdload of index
```

```
(* | Jdload_0 | Jdload_1 | Jdload_2 | Jdload_3 *)
   | Jdmul
   | Jdneg
   | Jdrem
(* | Jdreturn *)
   | Jdstore of index
(* | Jdstore_0 | Jdstore_1 | Jdstore_2 | Jdstore_3 *)
   | Jdsub
   | Jdup
   | Jdup_x1
   | Jdup_x2
   | Jdup2
   | Jdup2_x1
   | Jdup2_x2
   | Jf2d
   | Jf2i
   | Jf2l
   | Jfadd
   | Jfaload
   | Jfastore
   | Jfcmpg | Jfcmpl
   | Jfconst of Real32.real
(* | Jfconst_0 | Jfconst_1 | Jfconst_2 *)
   | Jfdiv
   | Jfload of index
(* | Jfload_0 | Jfload_1 | Jfload_2 | Jfload_3 *)
   | Jfmul
   | Jfneg
   | Jfrem
(* | Jfreturn *)
   | Jfstore of index
(* | Jfstore_0 | Jfstore_1 | Jfstore_2 | Jfstore_3 *)
   | Jfsub
   | Jgetfield of field_ref
   | Jgetstatic of field_ref
   | Jgoto of label
(* | Jgoto_w of label *)
   | Ji2b
   | Ji2c
   | Ji2d
   | Ji2f
   | Ji2l
   | Ji2s
   | Jiadd
   | Jiaload
   | Jiand
   | Jiastore
   | Jiconst of Int32.int
(* | Jiconst_m1 | Jiconst_0 | Jiconst_1 | Jiconst_2 *)
(* | Jiconst_3 | Jiconst_4 | Jiconst_5 *)
   | Jidiv
   | Jif_acmpeq of label | Jif_acmpne of label
   | Jif_icmpeq of label | Jif_icmpne of label | Jif_icmplt of label
   | Jif_icmpge of label | Jif_icmpgt of label | Jif_icmple of label
   | Jifeq of label | Jifne of label | Jiflt of label
   | Jifge of label | Jifgt of label | Jifle of label
   | Jifnonnull of label | Jifnull of label
   | Jiinc of {var   : index,
               const : int}
   | Jiload of index
(* | Jiload_0 | Jiload_1 | Jiload_2 | Jiload_3 *)
   | Jimul
   | Jineg
   | Jinstanceof of class_ref
   | Jinvokeinterface of method_ref
   | Jinvokespecial of method_ref
```

```
      | Jinvokestatic of method_ref
      | Jinvokevirtual of method_ref
      | Jior
      | Jirem
   (* | Jireturn *)
      | Jishl
      | Jishr
      | Jistore of index
   (* | Jistore_0 | Jistore_1 | Jistore_2 | Jistore_3 *)
      | Jisub
      | Jiushr
      | Jixor
      | Jjsr of label
   (* | Jjsr_w of label *)
      | Jl2d
      | Jl2f
      | Jl2i
      | Jladd
      | Jlaload
      | Jland
      | Jlastore
      | Jlcmp
      | Jlconst of Int64.int
   (* | Jlconst_0 | Jlconst_1 *)
   (* | Jldc of    jvm_const *)
   (* | Jldc_w of  jvm_const *)
   (* | Jldc2_w of jvm_const *)
      | Jldiv
      | Jlload of index
   (* | Jlload_0 | Jlload_1 | Jlload_2 | Jlload_3 *)
      | Jlmul
      | Jlneg
      | Jlookupswitch of {default : label,
                          cases   : (Int32.int * label) list}
      | Jlor
      | Jlrem
   (* | Jlreturn *)
      | Jlshl
      | Jlshr
      | Jlstore of index
   (* | Jlstore_0 | Jlstore_1 | Jlstore_2 | Jlstore_3 *)
      | Jlsub
      |  Jlushr
      | Jlxor
      | Jmonitorenter
      | Jmonitorexit
   (* | Jmultianewarray of {elem : jtype,
                            dim  : int} *)
      | Jnew of jclass
   (* | Jnewarray of jtype *)
      | Jnewarray of {elem : jtype,
                      dim  : int}
      | Jnop
      | Jpop
      | Jpop2
      | Jputfield of field_ref
      | Jputstatic of field_ref
      | Jret of index
      | Jreturn
      | Jsaload
      | Jsastore
   (* | Jsipush of int *)
      | Jswap
      | Jtableswitch of {default : label,
                         offset  : Int32.int,   (* key of first target *)
                         targets : label Vector.vector}
   (* | Jwide of jvm_instr *)

end (* local *)

(* NOTE: Some JVM instructions have been omitted in the jvm_instr
 * datatype.  For example, there is no need to distinguish aload 0
 * from aload_0 until the binary byte code is emitted; the emitter can
 * choose the most compact aload variant.  These are the translations
 * made by the bytecode emitter (Emitcode.emit):
 *
 *   Jsconst   ==> ldc or ldc_w
 *   Jaload    ==> aload_<n>, aload, or wide aload
 *   Jastore   ==> astore_<n>, astore, or wide astore
 *   Jdconst   ==> dconst_<d> or ldc2_w
 *   Jdload    ==> dload_<n>, dload, or wide dload
 *   Jdstore   ==> dstore_<n>, dstore, or wide dstore
 *   Jfconst   ==> fconst_<f>, ldc, or ldc_w
 *   Jfload    ==> fload_<n>, fload, or wide fload
 *   Jfstore   ==> fstore_<n>, fstore, or wide fstore
 *   Jgoto     ==> goto or goto_w
 *   Jiconst   ==> iconst_<i>, bipush, sipush, ldc, or ldc_w
 *   Jiinc     ==> iinc or wide iinc
 *   Jiload    ==> iload_<n>, iload, or wide iload
 *   Jistore   ==> istore_<n>, istore, or wide istore
 *   Jjsr      ==> jsr or jsr_w
 *   Jlconst   ==> lconst_<l> or ldc2_w
 *   Jlload    ==> lload_<n>, lload, or wide lload
 *   Jlstore   ==> lstore_<n>, lstore, or wide lstore
 *   Jnewarray ==> newarray, anewarray, or multianewarray
 *   Jret      ==> ret or wide ret
 *   Jreturn   ==> areturn, dreturn, freturn, ireturn, lreturn, or return
 *)


(* Utility functions *)
fun intConst   i = Jiconst (Int32.fromInt i)
fun wordConst  w = Jiconst (Int32.fromInt(Word.toIntX w))
fun charConst  c = Jiconst (Int32.fromInt(Char.ord c))
fun realConst  r = Jdconst (Real64.fromReal r)
fun checkCast  c = Jcheckcast (CLASS c)
fun checkArray t = Jcheckcast (ARRAY t)
```

```
(* Stackdepth -- Peter Sestoft   1997-07-29
 *            -- Peter Bertelsen 1997-08-01
 *)

val maxdepth : Bytecode.jvm_instr list -> Label.label list -> int

(* [maxdepth code hdlrs] computes the maximal local stack depth
 * reached by any computation of the given code, and with the given
 * exception handler entry points hdlrs.
 *)
```

```
(* Stackdepth -- Peter Sestoft   1997-07-03, 1997-07-27
              -- Peter Bertelsen 1997-08-01, 1997-08-07, 1997-10-10

Computes the maximal stack depth of a well-formed JVM bytecode
sequence.

Restrictions on local subroutines (implemented by jsr and ret),
enforced by the code below:

  - a subroutine immediately stores the return address (from the stack)
    in a local variable;
  - the subroutine does not subsequently modify that variable (not even
    by calling other subroutines);
  - that variable must be used in all the subroutine's ret instructions;
  - subroutines can call subroutines, but they cannot be recursive (neither
    directly nor indirectly);
  - subroutines can be entered and exited only by means of jsr and ret;
  - hence every instruction belongs either to the main program or to a
    single subroutine;
  - this is the `color' of the instruction: NONE if it belongs to the
    main program, SOME lbl if it belongs to the subroutine that begins
    with label lbl;
  - a subroutine may fail to return by entering an infinite loop, by
    throwing an exception, or by terminating the containing method.
*)

(* This belongs elsewhere ******************************* *)

exception Impossible of string

(* End of This belongs elsewhere ******************************* *)

fun bug s = raise Impossible ("Stackdepth." ^ s)

local
    open Jvmtype
in
    fun fieldDelta t = width t

    (* Compute difference between method return type size and
       arguments type size *)

    fun methodDelta (argTs, retT) =
        let val argsSize = List.foldl (fn (t, tot) => width t + tot) 0 argTs
        in
            case retT of
                NONE   => ~argsSize
              | SOME t => width t - argsSize
        end
end


(* Manipulating sets of modified local variables *)

abstype lvarset = Lvarset of Intset.intset
with
    open Intset
    val emptyset = Lvarset empty

    fun addtoset1 lvar (Lvarset s) =
        Lvarset (add(s, Localvar.toInt lvar))

    fun addtoset2 lvar (Lvarset s) =
        let val i = Localvar.toInt lvar
        in
```

```
            Lvarset (add(add(s, i), i+1))
        end

    fun inset lvar (Lvarset s) =
        Intset.member(s, Localvar.toInt lvar)

    fun union (Lvarset a) (Lvarset b) =
        Lvarset(Intset.union(a, b))
end

(* What is known about the code at a given label:
    PENDING     - label not reached
    LBLRESOLVED - label reached, stack depth and color resolved
    SRPENDING   - subroutine called, not returned from, still resolving
    SRPARTIAL   - subroutine called and returned from, still resolving
    SRRESOLVED  - subroutine called and fully resolved
*)

datatype lblinfo =
    PENDING of Bytecode.jvm_instr list
  | LBLRESOLVED of { dep : int, col : Label.label option }
  | SRPENDING
  | SRPARTIAL of { after : int, modifref : lvarset ref }
  | SRRESOLVED of { dep : int, after : int option,
                    modif : lvarset, maxdep : int }

fun maxdepth code hdlrs =
    let open Bytecode

        (* Create a table with information for all labels in the program *)

        val exnFind = Fail "Stackdepth.maxdepth: undefined label"
        val labelinfo = Polyhash.mkPolyTable(1021, exnFind)
        fun update lbl info = Polyhash.insert labelinfo (lbl, info)
        fun lookup lbl = Polyhash.find labelinfo lbl
        fun buildpending [] = ()
          | buildpending (Jlabel lbl :: rest) =
            (update lbl (PENDING rest); buildpending rest)
          | buildpending (_ :: rest) = buildpending rest
        val _ = buildpending code

        (* Record the stack depth at return from local subroutine lbl *)

        fun jsrupdate lbl depth modif =
            case lookup lbl of
                SRPARTIAL {after, modifref} =>
                    if depth <> after then
                        raise Fail "Inconsistent stack depths at ret"
                    else
                        modifref := union (!modifref) modif
              | SRPENDING =>
                    update lbl (SRPARTIAL {after=depth, modifref=ref modif})
              | _ => bug "jsrupdate"

        fun color NONE = NONE
          | color (SOME(_, lbl)) = SOME lbl

        (* Record the stack depth at an ordinary label *)

        fun resolve depth srCol modif (lbl, maxdepth) =
            case lookup lbl of
                PENDING code =>
                    (update lbl (LBLRESOLVED {dep=depth, col=color srCol});
                     finddepth code srCol modif depth maxdepth)
              | LBLRESOLVED {dep, col} =>
                    if col <> color srCol then
                        raise Fail "Inconsistent subroutine colors at label"
                    else if depth <> dep then
                        raise Fail "Inconsistent stack depths at label"
                    else maxdepth
              | _ => raise Fail "Subroutine label used as ordinary label"

        (* Get the stack depth after subroutine lbl; resolve if necessary *)

        and jsrlookup depth lbl =
            case lookup lbl of
                PENDING [] => raise Fail "No code in subroutine"
              | PENDING (Jastore lvar :: rest) =>
                    let val _ = update lbl SRPENDING
                        val maxdep = finddepth rest (SOME (lvar, lbl))
                                               emptyset (depth-1) depth
                        val (after, modif0) =
                            case lookup lbl of
                                SRPENDING =>                (* no reachable Jret *)
                                    (NONE, emptyset)
                              | SRPARTIAL {after, modifref} =>
                                    (SOME after, !modifref)
                              | _ => bug "jsrlookup"
                        val modif = addtoset1 lvar modif0
                    in
                        if inset lvar modif0 then
                            raise Fail "Subroutine overwrites return address"
                        else
                            (update lbl (SRRESOLVED {dep=depth, after=after,
                                                     modif=modif,
                                                     maxdep=maxdep});
                             (after, modif, maxdep))
                    end
              | PENDING _ => raise Fail "Subroutine should start with Jastore"
              | LBLRESOLVED _ =>
                    raise Fail "Ordinary label used as subroutine label"
              | SRRESOLVED { dep, after, modif, maxdep } =>
                    if depth <> dep then
                        raise Fail "Inconsistent stack depths at subroutine"
                    else
                        (after, modif, maxdep)
              | _       => raise Fail "Subroutine call loop"

        (* Find maxdepth by code execution; resolve labels and subroutines *)

        and finddepth [] srCol modif depth maxdepth = maxdepth
          | finddepth (code as ins1 :: rest) srCol modif depth maxdepth =
            let fun finddepth1 modif delta =
                    let val depth = depth + delta
                        val maxdepth = Int.max(depth, maxdepth)
                    in
                        finddepth rest srCol modif depth maxdepth
                    end
                fun finddepth2 modif delta lbl =
                    let val depth = depth + delta
                        val maxdepth = Int.max(depth, maxdepth)
                    in
                        finddepth rest srCol modif depth
                                 (resolve depth srCol modif (lbl, maxdepth))
                    end
            in
            case ins1 of
                 Jathrow   => maxdepth
               | Jreturn   => maxdepth
               | Jgoto lbl => resolve depth srCol modif (lbl, maxdepth)
               | Jlookupswitch {default, cases} =>
                    let val depth' = depth - 1

                        fun resolve' ((_, lbl), maxdepth) =
                            resolve depth' srCol modif (lbl, maxdepth)

                        val maxdepth' = List.foldl resolve' maxdepth cases
                    in
                        resolve depth' srCol modif (default, maxdepth')
                    end
               | Jtableswitch {default, targets, ...} =>
                    let val depth = depth - 1
                        val maxdepth' =
                            Vector.foldl (resolve depth srCol modif)
                                         maxdepth targets
                    in
                        resolve depth srCol modif (default, maxdepth')
                    end
               | Jlabel lbl => resolve depth srCol modif (lbl, maxdepth)
               | Jjsr lbl =>
                    let val (afterOpt, modif', maxdepth') =
                            jsrlookup (depth+1) lbl
                        val modif'' = union modif modif'
                        val maxdepth'' = Int.max(maxdepth, maxdepth')
                    in
                        case afterOpt of
                            SOME after =>
                                finddepth rest srCol modif'' after maxdepth''
                            | NONE => maxdepth''
                    end
               | Jret lvar =>
                    (case srCol of
                         NONE => raise Fail "Local ret not within subroutine"
                       | SOME (lvar', lbl) =>
                             if lvar <> lvar' then
                                 raise Fail "Wrong lvar in local ret"
                             else
                                 (jsrupdate lbl depth modif;
                                  maxdepth))
               | Jastore j      => finddepth1 (addtoset1 j modif) ~1
               | Jdstore j      => finddepth1 (addtoset2 j modif) ~2
               | Jfstore j      => finddepth1 (addtoset1 j modif) ~1
               | Jistore j      => finddepth1 (addtoset1 j modif) ~1
               | Jlstore j      => finddepth1 (addtoset2 j modif) ~2
               | Jsconst _      => finddepth1 modif 1
               | Jaaload        => finddepth1 modif ~1
               | Jaastore       => finddepth1 modif ~3
               | Jaconst_null   => finddepth1 modif 1
               | Jaload j       => finddepth1 modif 1
               | Jarraylength   => finddepth1 modif 0
               | Jbaload        => finddepth1 modif ~1
               | Jbastore       => finddepth1 modif ~3
               | Jcaload        => finddepth1 modif ~1
               | Jcastore       => finddepth1 modif ~3
               | Jcheckcast i   => finddepth1 modif 0
               | Jd2f           => finddepth1 modif ~1
               | Jd2i           => finddepth1 modif ~1
               | Jd2l           => finddepth1 modif 0
               | Jdadd          => finddepth1 modif ~2
               | Jdaload        => finddepth1 modif 0
               | Jdastore       => finddepth1 modif ~4
               | Jdcmpg         => finddepth1 modif ~3
               | Jdcmpl         => finddepth1 modif ~3
               | Jdconst _      => finddepth1 modif 2
               | Jddiv          => finddepth1 modif ~2
               | Jdload j       => finddepth1 modif 2
               | Jdmul          => finddepth1 modif ~2
               | Jdneg          => finddepth1 modif 0
               | Jdrem          => finddepth1 modif ~2
               | Jdsub          => finddepth1 modif ~2
               | Jdup           => finddepth1 modif 1
               | Jdup_x1        => finddepth1 modif 1
               | Jdup_x2        => finddepth1 modif 1
               | Jdup2          => finddepth1 modif 2
               | Jdup2_x1       => finddepth1 modif 2
               | Jdup2_x2       => finddepth1 modif 2
               | Jf2d           => finddepth1 modif 1
               | Jf2i           => finddepth1 modif 0
               | Jf2l           => finddepth1 modif 1
               | Jfadd          => finddepth1 modif ~1
               | Jfaload        => finddepth1 modif ~1
               | Jfastore       => finddepth1 modif ~3
               | Jfcmpg         => finddepth1 modif ~1
               | Jfcmpl         => finddepth1 modif ~1
               | Jfconst _      => finddepth1 modif 1
               | Jfdiv          => finddepth1 modif ~1
               | Jfload j       => finddepth1 modif 1
               | Jfmul          => finddepth1 modif ~1
               | Jfneg          => finddepth1 modif 0
               | Jfrem          => finddepth1 modif ~1
               | Jfsub          => finddepth1 modif ~1
               | Jgetfield {ty, ...} =>
                     finddepth1 modif (fieldDelta ty - 1)
               | Jgetstatic {ty, ...} =>
                     finddepth1 modif (fieldDelta ty)
               | Ji2b           => finddepth1 modif 0
               | Ji2c           => finddepth1 modif 0
               | Ji2d           => finddepth1 modif 1
               | Ji2f           => finddepth1 modif 0
               | Ji2l           => finddepth1 modif 1
               | Ji2s           => finddepth1 modif 0
               | Jiadd          => finddepth1 modif ~1
               | Jiaload        => finddepth1 modif ~1
               | Jiand          => finddepth1 modif ~1
               | Jiastore       => finddepth1 modif ~3
               | Jiconst _      => finddepth1 modif 1
               | Jidiv          => finddepth1 modif ~1
               | Jif_acmpeq lbl => finddepth2 modif ~2 lbl
               | Jif_acmpne lbl => finddepth2 modif ~2 lbl
               | Jif_icmpeq lbl => finddepth2 modif ~2 lbl
               | Jif_icmpne lbl => finddepth2 modif ~2 lbl
               | Jif_icmplt lbl => finddepth2 modif ~2 lbl
               | Jif_icmpge lbl => finddepth2 modif ~2 lbl
               | Jif_icmpgt lbl => finddepth2 modif ~2 lbl
               | Jif_icmple lbl => finddepth2 modif ~2 lbl
               | Jifeq lbl      => finddepth2 modif ~1 lbl
               | Jifne lbl      => finddepth2 modif ~1 lbl
               | Jiflt lbl      => finddepth2 modif ~1 lbl
               | Jifge lbl      => finddepth2 modif ~1 lbl
               | Jifgt lbl      => finddepth2 modif ~1 lbl
               | Jifle lbl      => finddepth2 modif ~1 lbl
               | Jifnonnull lbl => finddepth2 modif ~1 lbl
               | Jifnull lbl    => finddepth2 modif ~1 lbl
               | Jiinc args     => finddepth1 modif 0
               | Jiload j       => finddepth1 modif 1
               | Jimul          => finddepth1 modif ~1
               | Jineg          => finddepth1 modif 0
               | Jinstanceof i  => finddepth1 modif 0
               | Jinvokeinterface {msig, ...} =>
                     finddepth1 modif (methodDelta msig - 1)
               | Jinvokespecial {msig, ...} =>
```

```
                     finddepth1 modif (methodDelta msig - 1)
                | Jinvokestatic {msig, ...} =>
                     finddepth1 modif (methodDelta msig)
                | Jinvokevirtual {msig, ...} =>
                    finddepth1 modif (methodDelta msig - 1)
                | Jior           => finddepth1 modif ~1
                | Jirem          => finddepth1 modif ~1
                | Jishl          => finddepth1 modif ~1
                | Jishr          => finddepth1 modif ~1
                | Jisub          => finddepth1 modif ~1
                | Jiushr         => finddepth1 modif ~1
                | Jixor          => finddepth1 modif ~1
                | Jl2d           => finddepth1 modif 0
                | Jl2f           => finddepth1 modif ~1
                | Jl2i           => finddepth1 modif ~1
                | Jladd          => finddepth1 modif ~2
                | Jlaload        => finddepth1 modif 0
                | Jland          => finddepth1 modif ~2
                | Jlastore       => finddepth1 modif ~4
                | Jlcmp          => finddepth1 modif ~3
                | Jlconst _      => finddepth1 modif 2
                | Jldiv          => finddepth1 modif ~2
                | Jlload j       => finddepth1 modif 2
                | Jlmul          => finddepth1 modif ~2
                | Jlneg          => finddepth1 modif 0
                | Jlor           => finddepth1 modif ~2
                | Jlrem          => finddepth1 modif ~2
                | Jlshl          => finddepth1 modif ~2
                | Jlshr          => finddepth1 modif ~2
                | Jlsub          => finddepth1 modif ~2
                | Jlushr         => finddepth1 modif ~2
                | Jlxor          => finddepth1 modif ~2
                | Jmonitorenter  => finddepth1 modif ~1
                | Jmonitorexit   => finddepth1 modif ~1
                | Jnew i         => finddepth1 modif 1
                | Jnewarray {dim, ...} => finddepth1 modif (1 - dim)
                | Jnop           => finddepth1 modif 0
                | Jpop           => finddepth1 modif ~1
                | Jpop2          => finddepth1 modif ~2
                | Jputfield {ty, ...} =>
                    finddepth1 modif (~(fieldDelta ty) - 1)
                | Jputstatic {ty, ...} =>
                    finddepth1 modif (~(fieldDelta ty))
                | Jsaload        => finddepth1 modif ~1
                | Jsastore       => finddepth1 modif ~3
                | Jswap          => finddepth1 modif 0
            end
    in
        finddepth code NONE emptyset 0
                  (List.foldl (resolve 1 NONE emptyset) 0 hdlrs)
    end
```

```
(* Constpool.sig
 *
 * Peter Bertelsen
 * December 1997
 *)

exception InvalidEntry of string

type pool
```

```
eqtype index

datatype entry =
    CPutf8 of string
  | CPint of Int32.int
  | CPfloat of Word8Vector.vector
  | CPlong of Int64.int
  | CPdouble of Word8Vector.vector
  | CPclass of index
  | CPstring of index
  | CPfieldref of {class: index, nameType: index}
  | CPmethodref of {class: index, nameType: index}
  | CPimethodref of {class: index, nameType: index}
  | CPnametype of {name: index, desc: index}
  | CPunused  (* placeholder *)

val create : unit -> pool

val makeIndex  : int -> index
val indexValue : index -> int

val insUtf8       : pool -> string              -> index
val insInt        : pool -> Int32.int           -> index
val insFloat      : pool -> Real32.real         -> index
val insLong       : pool -> Int64.int           -> index
val insDouble     : pool -> Real64.real         -> index
val insClass      : pool -> Jvmtype.jclass      -> index
val insArrayClass : pool -> Jvmtype.jtype       -> index
val insString     : pool -> string              -> index
val insNameType   : pool -> {name : string,
                              desc : string}     -> index
val insFieldref   : pool -> Bytecode.field_ref  -> index
val insMethodref  : pool -> Bytecode.method_ref -> index
val insIMethodref : pool -> Bytecode.method_ref -> index
val insConst      : pool -> Bytecode.jvm_const  -> index

val lookup : pool -> index -> entry
val emit   : (Word8.word -> unit) -> pool -> unit
```

```
(* Constpool.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

exception InvalidEntry of string

fun entryError s = raise InvalidEntry("Constpool." ^ s)

datatype index = IDX of int

val maxPool = 0xffff

fun makeIndex k = if 0 <= k andalso k < maxPool then IDX k
                  else raise Overflow

fun indexValue (IDX i) = i

(* NOTE: index values produced by the insert functions will never
 * exceed maxPool-1, due to the use of Array.sub in insert.
 *)

datatype entry =
    CPutf8 of string
  | CPint of Int32.int
  | CPfloat of Word8Vector.vector
  | CPlong of Int64.int
  | CPdouble of Word8Vector.vector
  | CPclass of index
  | CPstring of index
  | CPfieldref of {class: index, nameType: index}
  | CPmethodref of {class: index, nameType: index}
  | CPimethodref of {class: index, nameType: index}
  | CPnametype of {name: index, desc: index}
  | CPunused  (* placeholder *)

(* NOTE: float and double entries are stored as byte vectors, rather
 * than Real32.real/Real64.real values, since polymorphic equality is
 * used in the constant pool implementation.
 *)

datatype pool =
    CP of {pool  : entry Array.array ref,
           cache : (entry, index) Polyhash.hash_table,
           count : int ref}

(* NOTE: the first valid entry in the constant pool is pool[1]. *)

val exnFind = Bytecode.InternalError
              "Constpool.insert: unexpected use of Polyhash.find"

fun create () =
    CP {pool  = ref (Array.array(maxPool, CPunused)),
        cache = Polyhash.mkPolyTable(1021, exnFind),
        count = ref 1}

fun insert size (CP {pool, cache, count}) entry =
    let val count' = !count
        val index  = IDX count'
    in
        case Polyhash.peekinsert cache (entry, index) of
            NONE => (Array.update(!pool, count', entry);
                     count:= count' + size;
                     index)
          | SOME index' => index'
    end

val insert1 = insert 1   (* insert constant occupying one entry *)
val insert2 = insert 2   (* insert constant occupying two entries *)

fun insUtf8 cp str = insert1 cp (CPutf8 str)

fun insInt cp k = insert1 cp (CPint k)

fun insFloat cp r = insert1 cp (CPfloat (Real32.toBytes r))

fun insLong cp k = insert2 cp (CPlong k)

fun insDouble cp r = insert2 cp (CPdouble (Real64.toBytes r))

fun insString cp str =
    let val ins = insert1 cp
    in
        ins(CPstring (ins(CPutf8 str)))
    end
```

```
fun insClass' ins class =
    let val name = Jvmtype.qualName class
    in
        ins(CPclass (ins(CPutf8 name)))
    end

fun insClass cp = insClass'(insert1 cp)

fun insArrayClass cp t =
    let open Jvmtype
        val ins = insert1 cp
        val desc = (case t of
                        Tarray t'    => typeDesc t
                      | Tclass class => qualName class
                      | _ => entryError "insArrayClass: invalid array class")
    in
        ins(CPclass (ins(CPutf8 desc)))
    end

fun insNameType' ins name desc = ins(CPnametype {name = ins(CPutf8 name),
                                                 desc = ins(CPutf8 desc)})

fun insNameType cp {name, desc} = insNameType' (insert1 cp) name desc

fun insFieldref cp {class, name, ty} =
    let val ins  = insert1 cp
        val desc = Jvmtype.typeDesc ty
    in
        ins(CPfieldref {class    = insClass' ins class,
                        nameType = insNameType' ins name desc})
    end

fun insMethodref cp {class, name, msig} =
    let val ins  = insert1 cp
        val desc = Jvmtype.methodDesc msig
    in
        ins(CPmethodref {class    = insClass' ins class,
                         nameType = insNameType' ins name desc})
    end

fun insIMethodref cp {class, name, msig} =
    let val ins = insert1 cp
        val desc = Jvmtype.methodDesc msig
    in
        ins(CPimethodref {class    = insClass' ins class,
                          nameType = insNameType' ins name desc})
    end

local
    open Bytecode
in
    fun insConst cp value =
        (case value of
             Cint i    => insInt cp i
           | Cfloat f  => insFloat cp f
           | Clong l   => insLong cp l
           | Cdouble d => insDouble cp d
           | Cstring s => insString cp s
           | _ => entryError "insConst: invalid constant")
end

fun lookup (CP {pool, count, ...}) (IDX i) =
    if 1 <= i andalso i < !count then
        Array.sub(!pool, i)
    else
        raise Subscript

fun charToUtf8 (c, res) =  (* convert character to UTF-8 representation *)
    let val k = ord c
    in
        if 1 <= k andalso k <= 127 then
            (Word8.fromInt k) :: res
        else
         (* if k <= 2047 then *)
                let
                    val x = 0xc0 + (k div 0x40)
                    val y = 0x80 + (k mod 0x40)
                in
                    (Word8.fromInt x) :: (Word8.fromInt y) :: res
                end
         (* else
                let
                    val k' = k div 0x40
                    val x  = 0xe0 + (k' div 0x40)
                    val y  = 0x80 + (k' mod 0x40)
                    val z  = 0x80 + (k  mod 0x40)
                in
                    (Word8.fromInt x) :: (Word8.fromInt y) ::
                    (Word8.fromInt z) :: res
                end *)
         (* NOTE: the latter section will be necessary if WideChar.char is
          * used for characters
          *)
    end

fun emit out (CP {pool, count, ...}) =
    let val emitInt  = Int32.emit out
        val emitLong = Int64.emit out
        val emitU2   = Word16.emit out
        val emitU2i  = emitU2 o Word16.fromInt

        fun emitIndex (IDX i) = emitU2i i

        fun emit' (CPutf8 s) =
                let val cs = CharVector.foldr charToUtf8 [] s
                in
                    out 0w1;   (* CONSTANT_Utf8 *)
                    emitU2i(length cs);
                    List.app out cs
                end
          | emit' (CPint i) =
                (out 0w3;   (* CONSTANT_Integer *)
                 emitInt i)
          | emit' (CPfloat bs) =
                (out 0w4;   (* CONSTANT_Float *)
                 Word8Vector.app out bs)
          | emit' (CPlong l) =
                (out 0w5;   (* CONSTANT_Long *)
                 emitLong l)
          | emit' (CPdouble bs) =
                (out 0w6;   (* CONSTANT_Double *)
                 Word8Vector.app out bs)
          | emit' (CPclass i) =
                (out 0w7;   (* CONSTANT_Class *)
                 emitIndex i)
          | emit' (CPstring i) =
                (out 0w8;   (* CONSTANT_String *)
                 emitIndex i)
          | emit' (CPfieldref {class = c, nameType = nt}) =
                (out 0w9;   (* CONSTANT_Fieldref *)
                 emitIndex c;
                 emitIndex nt)
          | emit' (CPmethodref {class = c, nameType = nt}) =
                (out 0w10;   (* CONSTANT_Methodref *)
                 emitIndex c;
                 emitIndex nt)
          | emit' (CPimethodref {class = c, nameType = nt}) =
                (out 0w11;   (* CONSTANT_IMethodref *)
                 emitIndex c;
                 emitIndex nt)
          | emit' (CPnametype {name = n, desc = d}) =
                (out 0w12;   (* CONSTANT_NameAndType *)
                 emitIndex n;
                 emitIndex d)
          | emit' CPunused = ()
    in
        emitU2i(!count);
        Array.appi (fn (_, e) => emit' e) (!pool, 1, SOME(!count))
    end
```

```
(* Emitcode.sig
 *
 * Peter Bertelsen
 * October 1997
 *)

exception InvalidCode of string

val emit : Constpool.pool -> Jvmtype.jtype option ->
           Bytecode.jvm_instr list -> Word8Vector.vector * (Label.label -> int)

val isU1    : int -> bool
val isU2    : int -> bool
val isByte  : int -> bool
val isShort : int -> bool
```

```
(* Emitcode.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

open Bytecode Jvmtype

(* NOTE: it is assumed that Constpool.indexValue and Localvar.toInt
 * return integers in the range [0, 0xffff].
 *)

exception InvalidCode of string

fun codeError s = raise InvalidCode("Emitcode." ^ s)

fun bug s = raise InternalError("Emitcode." ^ s)

type address = int

datatype placeholder =
    OFS16 of {addr: address, base: address}
```

```
  | OFS32 of {addr: address, base: address}

datatype target =
    PENDING of placeholder list ref
  | RESOLVED of address

fun typeKey t =
    case t of
        Tboolean => 0w4 : Word8.word
      | Tchar    => 0w5
      | Tfloat   => 0w6
      | Tdouble  => 0w7
      | Tbyte    => 0w8
      | Tshort   => 0w9
      | Tint     => 0w10
      | Tlong    => 0w11
      | _ => bug "typeKey: not a simple JVM type"

val exnFind = InvalidCode "Emitcode.emit: label not found"

val wide = 0w196 : Word8.word   (* opcode for JVM wide instruction prefix *)

fun isU1    k =      0 <= k andalso k <= 0xff
fun isU2    k =      0 <= k andalso k <= 0xffff
fun isByte  k =   ~128 <= k andalso k <= 127
fun isShort k = ~32768 <= k andalso k <= 32767

fun placeHolder16 addr = addr + 2
fun placeHolder32 addr = addr + 4

datatype method_kind = STATIC | NONSTATIC

fun argsWidth mkind (msig: method_sig) =
    let val extra =
        (case mkind of
             STATIC    => 0
           | NONSTATIC => 1)   (* include size of objectref (this) *)
    in
        List.foldl (fn (t, res) => width t + res) extra (#1 msig)
    end

fun emit cp returnTy code =
    let        val nextAddr = ref 0   (* the first instruction goes into code'[0] *)
        val targets  = Polyhash.mkPolyTable(1021, exnFind)
        val code'    = Word8Array.array(0xffff, 0w0)
                       (* max 0xffff bytes of code per method *)

        local
            open Constpool
        in
            val insInt'        = insInt cp
            val insFloat'      = insFloat cp
            val insLong'       = insLong cp
            val insDouble'     = insDouble cp
            val insClass'      = insClass cp
            val insArrayClass' = insArrayClass cp
            val insString'     = insString cp
            val insFieldref'   = insFieldref cp
            val insMethodref'  = insMethodref cp
            val insIMethodref' = insIMethodref cp
        end

        fun setU1 w addr = (Word8Array.update(code', addr, w);
                            addr + 1)

        (* NOTE: the set* functions store a value at a specific
           * address in the code' array, and return the next address;
           * the emit* functions also update the nextAddr counter, and
           * return () : unit.
           *)

          val setU1i = setU1 o Word8.fromInt

          fun emitU1 w = nextAddr:= setU1 w (!nextAddr)

        val setVec = Word8Vector.foldl (fn (w, a) => setU1 w a)

        fun setU2i k addr = setVec addr (Word16.toBytes(Word16.fromInt k))

        val setCpIndex = setU2i o Constpool.indexValue

        fun setInt k addr = setVec addr (Int32.toBytes k)

        val setInti = setInt o Int32.fromInt

        fun emitBipush k =   (* assuming isByte k holds *)
            let val addr = setU1 0w16 (!nextAddr)
            in
                nextAddr:= setU1i k addr
            end

        fun emitSipush k =   (* assuming isShort k holds *)
            let val addr = setU1 0w17 (!nextAddr)
            in
                nextAddr:= setU2i k addr
            end

        fun emitLdc index =
            let val index' = Constpool.indexValue index
            in
                if isU1 index' then
                    let val addr = setU1 0w18 (!nextAddr)   (* ldc *)
                    in
                        nextAddr:= setU1i index' addr
                    end
                else
                    let val addr = setU1 0w19 (!nextAddr)   (* ldc_w *)
                    in
                        nextAddr:= setU2i index' addr
                    end
            end

        fun emitLdc2_w index =
            let val addr = setU1 0w20 (!nextAddr)   (* ldc2_w *)
            in
                nextAddr:= setCpIndex index addr
            end

        fun emitIntConst i =
             (let val i' = Int32.toInt i
             in
                 case i' of
                     ~1 => emitU1 0w2   (* iconst_m1 *)
                   |  0 => emitU1 0w3   (* iconst_0 *)
                   |  1 => emitU1 0w4   (* iconst_1 *)
                   |  2 => emitU1 0w5   (* iconst_2 *)
                   |  3 => emitU1 0w6   (* iconst_3 *)
                   |  4 => emitU1 0w7   (* iconst_4 *)
                   |  5 => emitU1 0w8   (* iconst_5 *)
                           |  _ => if isByte i' then emitBipush i'
                           else if isShort i' then emitSipush i'
                                else emitLdc(insInt' i)
             end) handle Overflow => emitLdc(insInt' i)

        local
            open Real32
         (* val bytes_0 = toBytes(fromReal 0.0) *)
            val bytes_0 = Word8Vector.fromList[0w0, 0w0, 0w0, 0w0]
            val bytes_1 = toBytes(fromReal 1.0)
            val bytes_2 = toBytes(fromReal 2.0)
        in
            fun emitFloatConst f =
                let val bytes = toBytes f
                in
                    if bytes = bytes_0 then
                        emitU1 0w11   (* fconst_0 *)
                    else if bytes = bytes_1 then
                             emitU1 0w12   (* fconst_1 *)
                         else if bytes = bytes_2 then
                                  emitU1 0w13   (* fconst_2 *)
                              else
                                  emitLdc(insFloat' f)
                end
        end

        local
            val zero = Int64.fromInt 0
            val one  = Int64.fromInt 1
        in
            fun emitLongConst l =
                if l = zero then
                    emitU1 0w9   (* lconst_0 *)
                else
                    if l = one then
                        emitU1 0w10   (* lconst_1 *)
                    else
                        emitLdc2_w(insLong' l)
        end

        local
            open Real64
             val bytes_0 = toBytes(fromReal 0.0)
            val bytes_1 = toBytes(fromReal 1.0)
        in
            fun emitDoubleConst d =
                let val bytes = toBytes d
                in
                    if bytes = bytes_0 then
                        emitU1 0w14   (* dconst_0 *)
                    else
                        if bytes = bytes_1 then
                            emitU1 0w15   (* dconst_1 *)
                        else
                            emitLdc2_w(insDouble' d)
                end
        end

        fun emitVarAccess instr j =   (* assuming isU2 j holds *)
            let val addr = !nextAddr
            in
                if isU1 j then
                    let val addr' = setU1 instr addr
                    in
                        nextAddr:= setU1i j addr'
                    end
                else   (* emit wide instruction variant *)
                    let val addr'  = setU1 wide  addr
                        val addr'' = setU1 instr addr'
                    in
                        nextAddr:= setU2i j addr''
                    end
            end

        fun immVarAccess {instr, imm_0, imm_1, imm_2, imm_3} index =
            let val index' = Localvar.toInt index
            in
                case index' of
                    0 => emitU1 imm_0
                  | 1 => emitU1 imm_1
                  | 2 => emitU1 imm_2
                  | 3 => emitU1 imm_3
                  | _ => emitVarAccess instr index'
            end

        fun emitBranch16 instr lbl =
            let val base = !nextAddr
                val addr = setU1 instr base
            in
                case Polyhash.peek targets lbl of
                    NONE => let val phs  = [OFS16 {addr = addr, base = base}]
                                val trgt = PENDING (ref phs)
                            in
                                Polyhash.insert targets (lbl, trgt);
                                nextAddr:= placeHolder16 addr
                            end
                    | SOME (PENDING phs) =>
                        (phs:= OFS16 {addr = addr, base = base} :: (!phs);
                         nextAddr:= placeHolder16 addr)
                    | SOME (RESOLVED trgt) =>
                        let val ofs = trgt - base
                        in
                            if isShort ofs then nextAddr:= setU2i ofs addr
                            else codeError "emit: branch offset out of range"
                        end
            end

        fun setOffset32 base lbl addr =
             (case Polyhash.peek targets lbl of
                  NONE => let val phs = [OFS32 {addr = addr, base = base}]
                         in
                             Polyhash.insert targets (lbl, PENDING (ref phs));
                             placeHolder32 addr
                         end
               | SOME (PENDING phs) =>
                     (phs:= OFS32 {addr = addr, base = base} :: (!phs);
                      placeHolder32 addr)
               | SOME (RESOLVED trgt) => setInti (trgt - base) addr)

        fun emitBranch16Or32 {instr, instr_w, lbl} =
            let val base = !nextAddr
            in
                case Polyhash.peek targets lbl of
                    NONE => let val addr = setU1 instr_w base
                                val phs  = [OFS32 {addr = addr, base = base}]
                            in
                                Polyhash.insert targets
                                                (lbl, PENDING (ref phs));
                                nextAddr:= placeHolder32 addr
                            end
                    | SOME (PENDING phs) =>
                        let val addr = setU1 instr_w base
                        in
                            phs:= OFS32 {addr = addr, base = base} :: (!phs);
                            nextAddr:= placeHolder32 addr
                        end
                    | SOME (RESOLVED trgt) =>
                        let val ofs = trgt - base
                        in
                            if isShort ofs then
                                let val addr = setU1 instr base
                                in
                                    nextAddr:= setU2i ofs addr
                                end
                            else
                                let val addr = setU1 instr_w base
                                in
                                    nextAddr:= setInti ofs addr
                                end
                        end
            end

        fun backPatch lbl =
            let val trgt = !nextAddr

                fun patch ofs =
                    (case ofs of
                         OFS16 {addr, base} =>
                             let val ofs = trgt - base
                             in
                                 if isShort ofs then setU2i ofs addr
                                 else codeError
                                      "emit: branch offset out of range"
                             end
                       | OFS32 {addr, base} => setInti (trgt - base) addr;
                     ())
            in
                (case Polyhash.peek targets lbl of
                     NONE => ()
                   | SOME (PENDING phs) => List.app patch (!phs)
                   | _ => codeError "emit.backPatch: label already resolved");
                Polyhash.insert targets (lbl, RESOLVED trgt)
            end

        fun emitIinc {var, const} =
            let val j    = Localvar.toInt var
                val addr = !nextAddr
            in
                if isU1 j andalso isByte const then
                    let val addr'  = setU1 0w132 addr   (* iinc *)
                        val addr'' = setU1i j addr'
                    in
                        nextAddr:= setU1i const addr''
                    end
                else if isShort const then
                         let val addr'   = setU1 wide addr
                             val addr''  = setU1 0w132 addr'
                             val addr''' = setU2i j addr''
                         in
                             nextAddr:= setU2i const addr'''
                         end
                     else codeError "emit: iinc constant out of range"
            end

        fun emitClassRef instr class =
            let val addr  = setU1 instr (!nextAddr)
                val index =
                    (case class of
                         CLASS c    => insClass' c
                         | ARRAY elem => insArrayClass'(Tarray elem))
            in
                nextAddr:= setCpIndex index addr
            end

        fun emitFieldAccess instr args =
            let val addr  = setU1 instr (!nextAddr)
            in
                nextAddr:= setCpIndex (insFieldref' args) addr
            end

        fun emitInvokeinterface (mref as {msig, ...}) =
            let val nargs = argsWidth NONSTATIC msig
            in
                if isU1 nargs then
                    let val addr   = setU1 0w185 (!nextAddr)
                        val addr'  = setCpIndex (insIMethodref' mref) addr
                        val addr'' = setU1i nargs addr'
                    in
                        nextAddr:= setU1 0w0 addr''
                    end
                else codeError "emit [Jinvokeinterface]: nargs out of range"
            end

        fun emitInvokeMethod mkind instr (mref as {msig, ...}) =
            if isU1 (argsWidth mkind msig) then
                let val addr  = setU1 instr (!nextAddr)
                in
                    nextAddr:= setCpIndex (insMethodref' mref) addr
                end
            else codeError
                 "emit [Jinvoke{special,static,virtual}]: nargs out of range"

        fun emitNewarray {elem, dim} =
            if dim = 1 then
                if isSimple elem then
                    let val addr = setU1 0w188 (!nextAddr)   (* newarray *)
                    in
                        nextAddr:= setU1 (typeKey elem) addr
                    end
                else
                    let val addr = setU1 0w189 (!nextAddr)   (* anewarray *)
                    in
                        nextAddr:= setCpIndex (insArrayClass' elem) addr
                    end
            else
                if isU1 dim then   (* multinewarray *)
                    let val addr  = setU1 0w197 (!nextAddr)
                        val index = insArrayClass'(Tarray elem)
                        val addr' = setCpIndex index addr
                    in
                        nextAddr:= setU1i dim addr'
                    end
                else codeError "emit [Jnewarray]: dim out of range"

        fun wordAlign addr = addr + (~addr mod 4)

        fun emitLookupswitch {default, cases} =
            let val base   = !nextAddr
                val addr   = setU1 0w171 base
                val addr'  = setOffset32 base default (wordAlign addr)
                val addr'' = setInti (length cases) addr'

                fun addCase ((key, lbl), map) = Binarymap.insert(map, key, lbl)

                val caseMap =
                    List.foldl addCase (Binarymap.mkDict Int32.compare) cases
                fun setCase (key, lbl, a) =
                    setOffset32 base lbl (setInt key a)
            in
                nextAddr:= Binarymap.foldl setCase addr'' caseMap
            end

        fun emitTableswitch {default, offset, targets} =
            let val base    = !nextAddr
                val addr    = setU1 0w170 base
                val addr'   = setOffset32 base default (wordAlign addr)
                val addr''  = setInt offset addr'
                val len     = Vector.length targets
                val high    = Int32.toInt offset + len - 1
                   (* NOTE: high should be calculated using Int32.+ and Int32.- *)
                val addr''' = setInti high addr''
                val setOffsets =
                    Vector.foldl (fn (lbl, a) => setOffset32 base lbl a)
            in
                nextAddr:= setOffsets addr''' targets
            end

        fun returnInstr () : Word8.word =
             (case returnTy of
                  NONE   => 0w177   (* return *)
                 | SOME t =>
                      (case t of
                           Tboolean => 0w172   (* ireturn *)
                         | Tchar    => 0w172   (* ireturn *)
                         | Tfloat   => 0w174   (* freturn *)
                         | Tdouble  => 0w175   (* dreturn *)
                         | Tbyte    => 0w172   (* ireturn *)
                         | Tshort   => 0w172   (* ireturn *)
                         | Tint     => 0w172   (* ireturn *)
                         | Tlong    => 0w173   (* lreturn *)
                         | Tarray _ => 0w176   (* areturn *)
                         | Tclass _ => 0w176   (* areturn *)))

        fun emitInstr instr =
             (case instr of
                 Jlabel lbl         => backPatch lbl
                 | Jsconst s          => emitLdc(insString' s)
                 | Jaaload            => emitU1 0w50
                 | Jaastore           => emitU1 0w83
                 | Jaconst_null       => emitU1 0w1
                 | Jaload j           => immVarAccess
                                                {instr = 0w25,
                                                 imm_0 = 0w42, imm_1 = 0w43,
                                                 imm_2 = 0w44, imm_3 = 0w45} j
                 | Jarraylength       => emitU1 0w190
                 | Jastore j          => immVarAccess
                                                {instr = 0w58,
                                                 imm_0 = 0w75, imm_1 = 0w76,
                                                 imm_2 = 0w77, imm_3 = 0w78} j
                 | Jathrow            => emitU1 0w191
                 | Jbaload            => emitU1 0w51
                 | Jbastore           => emitU1 0w84
                 | Jcaload            => emitU1 0w52
                 | Jcastore           => emitU1 0w85
                 | Jcheckcast class   => emitClassRef 0w192 class
                 | Jd2f               => emitU1 0w144
                 | Jd2i               => emitU1 0w142
                 | Jd2l               => emitU1 0w143
                 | Jdadd              => emitU1 0w99
                 | Jdaload            => emitU1 0w49
        | Jdastore           => emitU1 0w82
        | Jdcmpg             => emitU1 0w152
        | Jdcmpl             => emitU1 0w151
        | Jdconst d          => emitDoubleConst d
        | Jddiv              => emitU1 0w111
        | Jdload j           => immVarAccess
                                  {instr = 0w24,
                                   imm_0 = 0w38, imm_1 = 0w39,
                                   imm_2 = 0w40, imm_3 = 0w41} j
        | Jdmul              => emitU1 0w107
        | Jdneg              => emitU1 0w119
        | Jdrem              => emitU1 0w115
        | Jdstore j          => immVarAccess
                                  {instr = 0w57,
                                   imm_0 = 0w71, imm_1 = 0w72,
                                   imm_2 = 0w73, imm_3 = 0w74} j
        | Jdsub              => emitU1 0w103
        | Jdup               => emitU1 0w89
        | Jdup_x1            => emitU1 0w90
        | Jdup_x2            => emitU1 0w91
        | Jdup2              => emitU1 0w92
        | Jdup2_x1           => emitU1 0w93
        | Jdup2_x2           => emitU1 0w94
        | Jf2d               => emitU1 0w141
        | Jf2i               => emitU1 0w139
        | Jf2l               => emitU1 0w140
        | Jfadd              => emitU1 0w98
        | Jfaload            => emitU1 0w48
        | Jfastore           => emitU1 0w81
        | Jfcmpg             => emitU1 0w150
        | Jfcmpl             => emitU1 0w149
        | Jfconst f          => emitFloatConst f
        | Jfdiv              => emitU1 0w110
        | Jfload j           => immVarAccess
                                  {instr = 0w23,
                                   imm_0 = 0w34, imm_1 = 0w35,
                                   imm_2 = 0w36, imm_3 = 0w37} j
        | Jfmul              => emitU1 0w106
        | Jfneg              => emitU1 0w118
        | Jfrem              => emitU1 0w114
        | Jfstore j          => immVarAccess
                                  {instr = 0w56,
                                   imm_0 = 0w67, imm_1 = 0w68,
                                   imm_2 = 0w69, imm_3 = 0w70} j
        | Jfsub              => emitU1 0w102
        | Jgetfield  a       => emitFieldAccess 0w180 a
        | Jgetstatic a       => emitFieldAccess 0w178 a
        | Jgoto lbl          => emitBranch16Or32
                                  {instr = 0w167, instr_w = 0w200,
                                   lbl   = lbl}
        | Ji2b               => emitU1 0w145
        | Ji2c               => emitU1 0w146
        | Ji2d               => emitU1 0w135
        | Ji2f               => emitU1 0w134
        | Ji2l               => emitU1 0w133
        | Ji2s               => emitU1 0w147
        | Jiadd              => emitU1 0w96
        | Jiaload            => emitU1 0w46
        | Jiand              => emitU1 0w126
        | Jiastore           => emitU1 0w79
        | Jiconst i          => emitIntConst i
        | Jidiv              => emitU1 0w108
        | Jif_acmpeq lbl     => emitBranch16 0w165 lbl
        | Jif_acmpne lbl     => emitBranch16 0w166 lbl
        | Jif_icmpeq lbl     => emitBranch16 0w159 lbl
        | Jif_icmpne lbl     => emitBranch16 0w160 lbl
        | Jif_icmplt lbl     => emitBranch16 0w161 lbl
        | Jif_icmpge lbl     => emitBranch16 0w162 lbl
        | Jif_icmpgt lbl     => emitBranch16 0w163 lbl
        | Jif_icmple lbl     => emitBranch16 0w164 lbl
        | Jifeq lbl          => emitBranch16 0w153 lbl
        | Jifne lbl          => emitBranch16 0w154 lbl
        | Jiflt lbl          => emitBranch16 0w155 lbl
        | Jifge lbl          => emitBranch16 0w156 lbl
        | Jifgt lbl          => emitBranch16 0w157 lbl
        | Jifle lbl          => emitBranch16 0w158 lbl
        | Jifnonnull lbl     => emitBranch16 0w159 lbl
        | Jifnull lbl        => emitBranch16 0w160 lbl
        | Jiinc a            => emitIinc a
        | Jiload j           => immVarAccess
                                    {instr = 0w21,
                                     imm_0 = 0w26, imm_1 = 0w27,
                                     imm_2 = 0w28, imm_3 = 0w29} j
        | Jimul              => emitU1 0w104
        | Jineg              => emitU1 0w116
        | Jinstanceof class  => emitClassRef 0w193 class
        | Jinvokeinterface a => emitInvokeinterface a
        | Jinvokespecial   a => emitInvokeMethod NONSTATIC 0w183 a
        | Jinvokestatic    a => emitInvokeMethod STATIC 0w184 a
        | Jinvokevirtual   a => emitInvokeMethod NONSTATIC 0w182 a
        | Jior               => emitU1 0w128
        | Jirem              => emitU1 0w112
        | Jishl              => emitU1 0w120
        | Jishr              => emitU1 0w122
        | Jistore j          => immVarAccess
                                    {instr = 0w54,
                                     imm_0 = 0w59, imm_1 = 0w60,
                                     imm_2 = 0w61, imm_3 = 0w62} j
        | Jisub              => emitU1 0w100
        | Jiushr             => emitU1 0w124
        | Jixor              => emitU1 0w130
        | Jjsr lbl           => emitBranch16Or32
                                    {instr = 0w168, instr_w = 0w201,
                                     lbl   = lbl}
        | Jl2d               => emitU1 0w138
        | Jl2f               => emitU1 0w137
        | Jl2i               => emitU1 0w136
        | Jladd              => emitU1 0w97
        | Jlaload            => emitU1 0w47
        | Jland              => emitU1 0w127
        | Jlastore           => emitU1 0w80
        | Jlcmp              => emitU1 0w148
        | Jlconst l          => emitLongConst l
        | Jldiv              => emitU1 0w109
        | Jlload j           => immVarAccess
                                    {instr = 0w22,
                                     imm_0 = 0w30, imm_1 = 0w31,
                                     imm_2 = 0w32, imm_3 = 0w33} j
        | Jlmul              => emitU1 0w105
        | Jlneg              => emitU1 0w117
        | Jlookupswitch a    => emitLookupswitch a
        | Jlor               => emitU1 0w129
        | Jlrem              => emitU1 0w113
        | Jlshl              => emitU1 0w121
        | Jlshr              => emitU1 0w123
        | Jlstore j          => immVarAccess
                                    {instr = 0w55,
                                     imm_0 = 0w63, imm_1 = 0w64,
                                     imm_2 = 0w65, imm_3 = 0w66} j
        | Jlsub              => emitU1 0w101
```

```
              | Jlushr             => emitU1 0w125
              | Jlxor              => emitU1 0w131
              | Jmonitorenter      => emitU1 0w194
              | Jmonitorexit       => emitU1 0w195
              | Jnew class         => emitClassRef 0w187 (CLASS class)
              | Jnewarray a        => emitNewarray a
              | Jnop               => emitU1 0w0
              | Jpop               => emitU1 0w87
              | Jpop2              => emitU1 0w88
              | Jputfield  a       => emitFieldAccess 0w181 a
              | Jputstatic a       => emitFieldAccess 0w179 a
              | Jret j             => emitVarAccess 0w169 (Localvar.toInt j)
              | Jreturn            => emitU1(returnInstr())
              | Jsaload            => emitU1 0w53
              | Jsastore           => emitU1 0w86
              | Jswap              => emitU1 0w95
              | Jtableswitch a     => emitTableswitch a
            )

        val codeVec = (List.app emitInstr code;
                       Word8Array.extract(code', 0, SOME (!nextAddr)))

        fun labelMap lbl =
            (case Polyhash.peek targets lbl of
                 NONE => codeError "emit.labelMap: unknown label"
               | SOME (RESOLVED trgtAddr) => trgtAddr
               | _ => codeError "emit.labelMap: unresolved label")
    in
        (codeVec, labelMap)
    end
```

```
(* Classfile.sig
 *
 * Peter Bertelsen
 * October 1997
 *)

exception InvalidClass of string

val emit : (Word8.word -> unit) ->
           Constpool.pool -> Classdecl.class_decl ->  unit

val scan : (unit -> Word8.word) -> Classdecl.class_decl * Constpool.pool
```

```
(* Classfile.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

open Classdecl Constpool

(* NOTE: it is assumed that Constpool.indexValue and the labelMap
 * returned by Emitcode.emit return integers in the range
 * [0, 0xffff].
 *)

exception InvalidClass of string
```

```
fun classError s = raise InvalidClass("Classfile." ^ s)

fun bug s = raise Bytecode.InternalError("Classfile." ^ s)

type exn_hdl =
    {start : Word16.word,
     stop  : Word16.word,
     entry : Word16.word,
     catch : index option}

type line_number_info =
    {start : Word16.word,
     line  : Word16.word}

type local_var_info =
    {start  : Word16.word,
     length : Word16.word,
     name   : index,
     desc   : index,
     index  : Word16.word}

datatype attribute =
    SRCFILE of {attr : index,
                file : index}
  | CONSTVAL of {attr  : index,
                 value : index}
  | CODE of {attr   : index,
             stack  : Word16.word,
             locals : Word16.word,
             code   : Word8Vector.vector,
             hdls   : exn_hdl list,
             attrs  : attribute list}
  | EXNS of {attr : index,
             exns : index list}
  | LINENUM of {attr  : index,
                lines : line_number_info list}
  | LOCALVAR of {attr : index,
                 vars : local_var_info list}
  | ATTR of {attr : index,
             info : Word8Vector.vector}

type member =
    {flags : Word16.word,
     name  : index,
     desc  : index,
     attrs : attribute list}

type class_file =
    {magic   : Word32.word,
     minor   : Word16.word,
     major   : Word16.word,
     pool    : pool,
     flags   : Word16.word,
     this    : index,
     super   : index option,
     ifcs    : index list,
     fields  : member list,
     methods : member list,
     attrs   : attribute list}

val magic' = Word8Vector.fromList[0wxca, 0wxfe, 0wxba, 0wxbe]
val magic  = valOf(Word32.fromBytes magic')
val minor  = Word16.fromWord 0w3
val major  = Word16.fromWord 0w45

fun toWord' flag =
         (case flag of
              ACCpublic       => 0wx0001
             | ACCprivate      => 0wx0002
             | ACCprotected    => 0wx0004
             | ACCstatic       => 0wx0008
             | ACCfinal        => 0wx0010
             | ACCvolatile     => 0wx0040
             | ACCtransient    => 0wx0080
             | ACCsynchronized => 0wx0020
             | ACCnative       => 0wx0100
             | ACCabstract     => 0wx0400
             | ACCsuper        => 0wx0020   (* same as ACCsynchronized *)
             | ACCinterface    => 0wx0200)

fun toWord (flag, w) = Word.orb(toWord' flag, w)

val flagsToWord = Word16.fromWord o (List.foldl toWord 0w0)

val isU2 = Emitcode.isU2

fun fromClassDecl cp {flags, this, super, ifcs, fdecls, mdecls, attrs} =
    let val insUtf8'  = insUtf8  cp
        val insConst' = insConst cp
        val insClass' = insClass cp

        fun insExnHandler labelMap {start, stop, entry, catch} =
            let val labelMap' = Word16.fromInt o labelMap
            in
                {start = labelMap' start,
                 stop  = labelMap' stop,
                 entry = labelMap' entry,
                 catch = (case catch of
                              NONE       => NONE
                              | SOME class => SOME (insClass' class))}
            end

        fun insLnumInfo labelMap {start, line} =
            if isU2 line then
                {start = Word16.fromInt(labelMap start),
                 line  = Word16.fromInt line}
            else classError
                  "fromClassDecl.insLnumInfo: line number out of range"

        fun insLvarInfo labelMap {from, thru, name, ty, index} =
            let val start' = labelMap from
                val length = labelMap thru - start'
                val index' = Localvar.toInt index
            in
                if isU2 length then
                    if isU2 index' then
                        {start  = Word16.fromInt start',
                         length = Word16.fromInt length,
                         name   = insUtf8' name,
                         desc   = insUtf8'(Jvmtype.typeDesc ty),
                         index  = Word16.fromInt index'}
                    else classError
                         "fromClassDecl.insLvarInfo: index out of range"
                else classError
                      "fromClassDecl.insLvarInfo: length out of range"
            end
        fun insAttr' _ _ (Classdecl.SRCFILE file) =
                SRCFILE {attr = insUtf8' "SourceFile",
                         file = insUtf8' file}
            | insAttr' _ _ (Classdecl.CONSTVAL value) =
                CONSTVAL {attr  = insUtf8' "ConstantValue",
                          value = insConst' value}
            | insAttr' _ ty (Classdecl.CODE {stack, locals, code, hdls, attrs}) =
                let val (code', labelMap) = Emitcode.emit cp ty code
                    val insExnHandler'    = insExnHandler labelMap
                    val insAttr''         = insAttr' labelMap ty
                in
                    if isU2 stack then
                        if isU2 locals then
                            CODE {attr   = insUtf8' "Code",
                                  stack  = Word16.fromInt stack,
                                  locals = Word16.fromInt locals,
                                  code   = code',
                                  hdls   = List.map insExnHandler' hdls,
                                  attrs  = List.map insAttr'' attrs}
                        else classError
                              "insAttr [CODE]: max_locals out of range"
                    else classError "insAttr [CODE]: max_stack out of range"
                end
            | insAttr' _ _ (Classdecl.EXNS exns) =
                EXNS {attr = insUtf8' "Exceptions",
                      exns = List.map insClass' exns}
            | insAttr' labelMap _ (Classdecl.LINENUM lines) =
                LINENUM {attr  = insUtf8' "LineNumberTable",
                         lines = List.map (insLnumInfo labelMap) lines}
            | insAttr' labelMap _ (Classdecl.LOCALVAR vs) =
                LOCALVAR {attr = insUtf8' "LocalVariableTable",
                          vars = List.map (insLvarInfo labelMap) vs}
            | insAttr' _ _ (Classdecl.ATTR {attr, info}) =
                ATTR {attr = insUtf8' attr,
                      info = info}

        fun emptyLblMap _ = classError
                             "fromClassDecl.insAttr: invalid nested attributes"

        val insAttr = insAttr' emptyLblMap

               (* NOTE: when insAttr is used for inserting a method
               * attribute, the second argument specifies the (optional)
               * return type for the method; when insAttr is used for
               * inserting a field or class attribute, the faked `return
               * type' is NONE.
               *)

        fun insField {flags, name, ty, attrs} =
               {flags = flagsToWord flags,
             name  = insUtf8' name,
             desc  = insUtf8'(Jvmtype.typeDesc ty),
             attrs = List.map (insAttr NONE) attrs}

        fun insMethod {flags, name, msig, attrs} =
            let val (_, returnTy) = msig
            in
                {flags = flagsToWord flags,
                 name  = insUtf8' name,
                 desc  = insUtf8'(Jvmtype.methodDesc msig),
                 attrs = List.map (insAttr returnTy) attrs}
            end
    in
           (* NOTE: cp need not be completed before we start building the
         * resulting class_file since Constpool.pool is imperative. *)
        {magic   = magic,
         minor   = minor,
         major   = major,
         pool    = cp,
         flags   = flagsToWord flags,
         this    = insClass' this,
         super   = (case super of
                       NONE => NONE
                     | SOME class => SOME (insClass' class)),
         ifcs    = List.map insClass' ifcs,
         fields  = List.map insField fdecls,
         methods = List.map insMethod mdecls,
         attrs   = List.map (insAttr NONE) attrs}
    end

fun toClassDecl cf = raise Bytecode.Unimplemented "Classfile.toClassDecl"

val word16_0 = Word16.fromWord 0w0
val word32_2 = Word32.fromWord 0w2

fun emitClassFile out {magic, minor, major, pool, flags, this,
                       super, ifcs, fields, methods, attrs} =
    let val emitU2      = Word16.emit out
        val emitU2i     = emitU2 o Word16.fromInt
        val emitU4      = Word32.emit out
        val emitU4i     = emitU4 o Word32.fromInt
        val emitCpIndex = emitU2i o Constpool.indexValue

        fun emitCpIndexOpt  NONE    = emitU2 word16_0
          | emitCpIndexOpt (SOME i) = emitCpIndex i

        fun emitExnHdl {start, stop, entry, catch} =
            (emitU2 start;
             emitU2 stop;
             emitU2 entry;
             emitCpIndexOpt catch)

        fun emitLineNumInfo {start, line} =
                (emitU2 start;
                 emitU2 line)

        fun emitLocalVarInfo {start, length, name, desc, index} =
                (emitU2 start;
                 emitU2 length;
                 emitCpIndex name;
                 emitCpIndex desc;
                 emitU2 index)

        fun attrSize (LINENUM {lines, ...}) = 8 + List.length lines * 4
          | attrSize (LOCALVAR {vars, ...}) = 8 + List.length vars * 10
          | attrSize (ATTR {info, ...}) = 2 + Word8Vector.length info
          | attrSize _ = classError "emit.attrSize: invalid nested attributes"

        fun attrsSize attrs =
            List.foldl (fn (a, sz) => attrSize a + sz) 0 attrs

        fun emitAttr (SRCFILE {attr, file}) =
                (emitCpIndex attr;
                 emitU4 word32_2;
                 emitCpIndex file)
          | emitAttr (CONSTVAL {attr, value}) =
                (emitCpIndex attr;
                 emitU4 word32_2;
                 emitCpIndex value)
```

```
  | emitAttr (CODE {attr, stack, locals, code, hdls, attrs}) =
        let val codeLength = Word8Vector.length code
            val hdlsCount  = List.length hdls
        in
            if isU2 hdlsCount then
                (emitCpIndex attr;
                 emitU4i(2 + 2 + 4 + codeLength +
                         2 + hdlsCount * 8 +
                         2 + attrsSize attrs);
                 emitU2 stack;
                 emitU2 locals;
                 emitU4i codeLength;
                 Word8Vector.app out code;
                 emitU2i hdlsCount;
                 List.app emitExnHdl hdls;
                 emitAttrs attrs)
            else classError
                 "emit.emitAttr [CODE]: too many exception handlers"
        end
  | emitAttr (EXNS {attr, exns}) =
        let val exnCount = List.length exns
        in
            if isU2 exnCount then
                (emitCpIndex attr;
                 emitU4i(2 + exnCount * 2);
                 emitU2i exnCount;
                 List.app emitCpIndex exns)
            else classError "emit.emitAttr [EXNS]: too many exceptions"
        end
  | emitAttr (LINENUM {attr, lines}) =
        let val lineCount = List.length lines
        in
            if isU2 lineCount then
                (emitCpIndex attr;
                 emitU4i(2 + lineCount * 4);
                 emitU2i lineCount;
                 List.app emitLineNumInfo lines)
            else classError "emit.emitAttr [LINENUM]: too many lines"
        end
  | emitAttr (LOCALVAR {attr, vars}) =
        let val varCount = List.length vars
        in
            if isU2 varCount then
                (emitCpIndex attr;
                 emitU4i(2 + varCount * 10);
                 emitU2i varCount;
                 List.app emitLocalVarInfo vars)
            else classError
                 "emit.emitAttr [LOCALVAR]: too many variables"
        end
  | emitAttr (ATTR {attr, info}) =
        (emitCpIndex attr;
         Word8Vector.app out info)
and emitAttrs attrs =
    let val len = List.length attrs
    in
        if isU2 len then (emitU2i len;
                          List.app emitAttr attrs)
        else classError "emit.emitAttrs: too many attributes"
    end

fun emitMember {flags, name, desc, attrs} =
    (emitU2 flags;
     emitCpIndex name;
     emitCpIndex desc;
     emitAttrs attrs)
```

```
        val ifcsCount    = List.length ifcs
        val fieldsCount  = List.length fields
        val methodsCount = List.length methods
    in
        if isU2 ifcsCount then
            if isU2 fieldsCount then
                if isU2 methodsCount then
                    (emitU4 magic;
                     emitU2 minor;
                     emitU2 major;
                     Constpool.emit out pool;
                     emitU2 flags;
                     emitCpIndex this;
                     emitCpIndexOpt super;
                     emitU2i ifcsCount;
                     List.app emitCpIndex ifcs;
                     emitU2i fieldsCount;
                     List.app emitMember fields;
                     emitU2i methodsCount;
                     List.app emitMember methods;
                     emitAttrs attrs)
                else classError "emit: too many methods"
            else classError "emit: too many fields"
        else classError "emit: too many direct superinterfaces"
    end

fun emit out cp decl =
    emitClassFile out (fromClassDecl cp decl)

fun scanClassFile src = raise Bytecode.Unimplemented "Classfile.scanClassFile"

fun scan src = toClassDecl(scanClassFile src)
```

Appendix F

Source Files for the New Back-End

```
(* Codeutil.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

local
    open Prim Lambda Const Jvmtype Bytecode Smlclasses Runtype Jvmcode Error

    val labels = ref Label.freshLabels
    val tags   = ref Tag.freshTags

    (* NOTE: labels and tags will be unique across target class
     * methods *)

    val charFmtHex = (Int.fmt StringCvt.HEX) o Char.ord

    fun toJvmChar (c, l) = if Char.isAlphaNum c then Char.toString c :: l
                           else "$" :: (charFmtHex c) :: l
    fun toJvmName' l s = String.concat(CharVector.foldr toJvmChar l s)

    fun member x = List.exists (fn y => y=x)
in
    fun resetLabels () = labels:= Label.freshLabels

    fun resetTags () = tags:= Tag.freshTags

    (* Generate a fresh label *)
    fun newLabel () =
        let val (labels', lbl) = Label.newLabel(!labels)
        in
            labels:= labels';
            lbl
        end

    (* Generate a fresh tag *)
    fun nextTag () =
        let val (tags', tag) = Tag.nextTag(!tags)
        in
            tags:= tags';
            tag
        end

    fun lastTag () = Tag.lastTag(!tags)

    (* Add a label to a list of instructions *)
```

```
fun labelCode C =
    (case C of
         Jgoto lbl :: _  => (lbl, C)
       | Jlabel lbl :: _ => (lbl, C)
       | _ => let val lbl = newLabel()
              in
                  (lbl, Jlabel lbl :: C)
              end)

(* Generate a branch to the given list of instructions *)
fun makeBranch C =
    (case C of
     Jreturn :: _                      => (Jreturn, C)
   | (branch as (Jgoto _)) :: _        => (branch, C)
   | Jlabel _ :: Jreturn :: _          => (Jreturn, C)
   | Jlabel lbl :: _           => (Jgoto lbl, C)
   | _ => let val lbl = newLabel()
          in
              (Jgoto lbl, Jlabel lbl :: C)
          end)

(* Discard all instructions up to the next label *)
fun dropDead C =
    (case C of
         [] => []
       | Jlabel _ :: _      => C
       (* | Kset_global _ :: _ => C *)   (* is this critical??? [PMB] *)
       | _ :: rest          => dropDead rest)


(* Avoid checkcast to the specified class *)
fun dropCast class C =
    (case C of
         Jcheckcast (CLASS c) :: C' => if c = class then C' else C
       | _ => C)

(* Avoid checkcast to any of the specified classes *)
fun dropCasts classes C =
    (case C of
         Jcheckcast (CLASS c) :: C' => if member c classes then C' else C
       | _ => C)

(* Generate code to store a value into a local variable and avoid
 * loading the same value immediately thereafter *)
fun storeInVar j C =
    (case C of
         Jaload j' :: C' =>
             if j = j' then Jdup :: Jastore j :: C'
             else Jastore j :: C
       | _ => Jastore j :: C)

(* Generate code to store a value into a static field and avoid
 * loading the same value immediately thereafter *)
fun putStatic f C =
    (case C of
         Jgetstatic f' :: C' =>
             if f = f' then Jdup :: Jputstatic f :: C'
             else Jputstatic f :: C
       | _ => Jputstatic f :: C)

(* Converting an identifier to a Bytecode.field_ref *)
val toJvmName = toJvmName' []

fun toFieldName id n = toJvmName' ["$", Int.toString n] id

fun exnToFieldName id n = toJvmName' ["$x", Int.toString n] id
fun toFieldref rt ({qual, id}, n) =
    let val class =
        (case qual of
             "General" => General
           | _         => class{pkgs = [], name = qual})
    in
        {class = class,
         name  = toFieldName id n,
         ty    = toJvmType rt} : field_ref
    end

fun exnToFieldref rt ({qual, id}, n) =
    let val class =
        (case qual of
             "General" => General
           | _         => class{pkgs = [], name = qual})
    in
        {class = class,
         name  = exnToFieldName id n,
         ty    = toJvmType rt} : field_ref
    end


fun toMethodref argTypes resType ({qual, id}, n) =
    {class = class{pkgs = [], name = qual},
     name  = toFieldName id n,
     msig  = toMethodSig argTypes resType} : method_ref

fun uidToString ({qual, id}, 0) = qual ^ "." ^ id
  | uidToString ({qual, id}, n) = qual ^ "." ^ id ^ "/" ^ Int.toString n

fun specMethodName tag = "clos$$" ^ Tag.toString tag

fun specMethodref class name {freeTypes, argTypes, resType} =
    let val argTypes' = if freeTypes = [] then argTypes
                        else RTarray :: argTypes
    in
        {class = class,
         name  = name,
         msig  = toMethodSig argTypes' resType} : method_ref
    end

fun globalMethodref {freeTypes, argTypes, resType} uid =
    let val argTypes' = if freeTypes = [] then argTypes
                        else RTarray :: argTypes
    in
        toMethodref argTypes' resType uid
    end

(* invert a bool_test *)
fun invertTest t =
    let val invert' =
        (fn PTeq         => PTnoteq
          | PTnoteq      => PTeq
          | PTnoteqimm _ => fatalError "Codeutil.invertTest: PTnoteqimm"
          | PTlt         => PTge
          | PTle         => PTgt
          | PTgt         => PTle
          | PTge         => PTlt)
    in
        case t of
            Peq_test         => Pnoteq_test
          | Pnoteq_test      => Peq_test
          | Pint_test t'     => Pint_test (invert' t')
          | Pfloat_test t'   => Pfloat_test (invert' t')
          | Pstring_test t'  => Pstring_test (invert' t')
          | Pword_test t'    => Pword_test (invert' t')
          | Pnoteqtag_test _ =>
              fatalError "Codeutil.invertTest: Pnoteqtag_test"
    end

(* Convert a prim_test to a JVM int test *)
val intTest =
    (fn PTeq         => Jif_icmpeq : Label.label -> jvm_instr
      | PTnoteq      => Jif_icmpne
      | PTnoteqimm _ => fatalError "Codeutil.intTest"
      | PTlt         => Jif_icmplt
      | PTle         => Jif_icmple
      | PTgt         => Jif_icmpgt
      | PTge         => Jif_icmpge)

(* Convert a prim_test to a JVM int test against zero *)
val zeroTest =
    (fn PTeq         => Jifeq : Label.label -> jvm_instr
      | PTnoteq      => Jifne
      | PTnoteqimm _ => fatalError "Codeutil.zeroTest"
      | PTlt         => Jiflt
      | PTle         => Jifle
      | PTgt         => Jifgt
      | PTge         => Jifge)


(* Check if a list of switch clauses contains exception tags
 * (i.e., tags that are not resolved until load-time) *)
fun containsExnTags clauses =
    List.exists (fn (EXNtag _, _) => true | _ => false) clauses

(* Maximum and minimum 32-bit integers *)
val minInt32 = valOf Int32.minInt
val maxInt32 = valOf Int32.maxInt

(* Maximum and minimum 32-bit integers represented as 64-bit values *)
val minInt32As64 = IntCvt.int32To64(valOf Int32.minInt)
val maxInt32As64 = IntCvt.int32To64(valOf Int32.maxInt)

(* Convert an int operator to a long operator *)
val longOper =
    (fn Jiadd => Jladd
      | Jisub => Jlsub
      | Jimul => Jlmul
      | Jidiv => Jldiv
      | Jirem => Jlrem
      | _ => fatalError "Codeutil.longOper")


(* Build a list with n copies of e *)
fun copy e n =
    let fun copy' 0 es = es
          | copy' k es = copy' (k-1) (e::es)
    in
        if n >= 0 then copy' n []
        else raise Domain
    end

(* The identity function *)
fun identity x = x


(* Bind an Lvar in the local environment, updating the environment *)
fun bindLvar localEnv n rt =
        let val (localEnv', j) = Localenv.bind (!localEnv) n rt
        in
            localEnv:= localEnv';
            j
        end


    (* Register an exception with a given uid *)
    fun registerExn ({class, names, exns, ...} : jvm_code)
        (uid as ({qual, id}, n))=
        if qual = Jvmtype.className class then
            let val name = exnToFieldName id n
                val nameInfo =
                    {runType = RTobject,
                     export  = true}
            in
                names:= Binarymap.insert(!names, name, nameInfo);
                exns := Binaryset.add(!exns, uid)
            end
        else ()

    (* Build a map from free variable index to runtype, given a list
     * of free variables and a local variable environment, and record
     * it in the closure environment: *)
    fun registerFreeRts tag free lvarRT closureEnv =
        let fun h _ [] m = m
              | h i ((Lvar n)::rest) m =
                    h (i+1) rest (Intmap.insert(m, i, lvarRT n))
              | h _ _ _ = fatalError "Codeutil.registerFreeRts"

            val m = h 0 free (Intmap.empty())
        in
            closureEnv:= Binarymap.insert(!closureEnv, tag, m)
        end

    (* Retrieve the free variable runtype environment corresponding to
     * the specified tag from the closure environment, and return a
     * lookup function for the free variable runtype environment: *)
    fun freeRuntype closureEnv tag =
        let val m = Binarymap.find(!closureEnv, tag)
            handle Binarymap.NotFound =>
                fatalError("Codeutil.freeRuntype: tag " ^ Tag.toString tag)
        in
            fn i => (Intmap.retrieve(m, i)
                     handle Intmap.NotFound =>
                         fatalError("Codeutil.freeRuntype (tag " ^
                                    Tag.toString tag ^ "): free var #" ^
                                    Int.toString i))
        end

end (* local *)
```

```
(* Coercion.sig
 *
 * Coercion between boxed and unboxed run-time representation of SML
 * values.
 *
 * Peter Bertelsen
 * December 1997
 *)

type wrapper = Bytecode.jvm_instr list ->
               (Bytecode.jvm_instr list -> Bytecode.jvm_instr list)
                * Bytecode.jvm_instr list

val coerce    : Runtype.runtype -> Runtype.runtype -> wrapper
```

```
(* Coercion.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

open Jvmtype Bytecode Smlclasses Codeutil Runtype Error

type wrapper = jvm_instr list ->
               (jvm_instr list -> jvm_instr list) * jvm_instr list

fun new class C = Jnew class :: Jdup :: C

fun wrap class init : wrapper =
    fn C => (new class, Jinvokespecial init :: C)

val wrapInteger' = wrap Integer integerInit
val wrapLong'    = wrap Long longInit
val wrapDouble'  = wrap Double doubleInit
val wrapString'  = wrap String stringInit
val wrapVector'  = wrap Vector vectorInit

fun wrapBoolean' C =
    let val (branch, C') = makeBranch C
        val lbl = newLabel()
    in
        (identity, Jifeq lbl :: Jgetstatic constructorOne :: branch ::
                   Jlabel lbl :: Jgetstatic constructorZero :: C')
    end

fun wrapInteger'' m C C' =
    if m = intValue then (identity, C')
    else if m = longValue then (identity, Ji2l :: C')
         else wrapInteger' C

fun wrapInteger (C as Jcheckcast (CLASS c) :: Jinvokevirtual m :: C') =
        if c = Number then wrapInteger'' m C C'
        else wrapInteger' C
  | wrapInteger (C as Jinvokevirtual m :: C') = wrapInteger'' m C C'
  | wrapInteger C = wrapInteger' C

fun wrapLong'' m C C' =
    if m = longValue then (identity, C')
    else if m = intValue then (identity, Jl2i :: C')
         else wrapLong' C
```

```
fun wrapLong (C as Jcheckcast (CLASS c) :: Jinvokevirtual m :: C') =
        if c = Number then wrapLong'' m C C'
        else wrapLong' C
  | wrapLong (C as Jinvokevirtual m :: C') = wrapLong'' m C C'
  | wrapLong C = wrapLong' C

fun wrapDouble (C as Jcheckcast (CLASS c) :: Jinvokevirtual m :: C') =
        if c = Number andalso m = doubleValue then (identity, C')
        else wrapDouble' C
  | wrapDouble (C as Jinvokevirtual m :: C') =
        if m = doubleValue then (identity, C')
        else wrapDouble' C
  | wrapDouble C = wrapDouble' C

fun wrapString (C as Jcheckcast (CLASS c) :: Jinvokevirtual m :: C') =
        if c = String andalso m = stringToCharArray then (identity, C')
        else wrapString' C
  | wrapString (C as Jinvokevirtual m :: C') =
        if m = stringToCharArray then (identity, C')
        else wrapString' C
  | wrapString (C as Jcheckcast (CLASS c) :: C') =
    (* this situation may occur, e.g. in connection with a string test *)
        if c = String then wrapString' C'
        else wrapString' C
  | wrapString C = wrapString' C

fun wrapVector (C as Jcheckcast (CLASS c) :: Jgetfield f :: C') =
        if c = Vector andalso f = vectorElems then (identity, C')
        else wrapVector' C
  | wrapVector (C as Jgetfield f :: C') =
        if f = vectorElems then (identity, C')
        else wrapVector' C
  | wrapVector C = wrapVector' C

fun wrapBoolean (C as Jcheckcast (CLASS c) :: Jgetfield f :: C') =
        if c = Constructor andalso f = constructorTag then (identity, C')
        else wrapBoolean' C
  | wrapBoolean (C as Jgetfield f :: C') =
        if f = constructorTag then (identity, C')
        else wrapBoolean' C
  | wrapBoolean C = wrapBoolean' C

fun castTo class C = (identity, checkCast class :: C)

fun castToArray elem C = (identity, checkArray elem :: C)

fun unwrapMethod class m : wrapper =
    let val check  = checkCast class
        val invoke = Jinvokevirtual m
    in
        fn C => (identity, check :: invoke :: C)
    end

fun unwrapField class f : wrapper =
    let val check = checkCast class
        val get   = Jgetfield f
    in
        fn C => (identity, check :: get :: C)
    end

val unwrapInteger = unwrapMethod Number intValue
val unwrapLong    = unwrapMethod Number longValue
val unwrapDouble  = unwrapMethod Number doubleValue
val unwrapBoolean = unwrapField Constructor constructorTag
val unwrapString  = unwrapMethod String stringToCharArray
val unwrapVector  = unwrapField Vector vectorElems
```

```
fun invokeVirtual m C = (identity, Jinvokevirtual m :: C)

fun getField f C = (identity, Jgetfield f :: C)

fun coerce src trgt C =
    if src = trgt then (identity, C)
    else
        (case (src, trgt) of
             (RTnumber, RTint)    => invokeVirtual intValue C
           | (RTnumber, RTlong)   => invokeVirtual longValue C
           | (RTnumber, RTdouble) => invokeVirtual doubleValue C

           | (RTint, RTnumber) => wrapInteger C
           | (RTint, RTlong)   => (identity, Ji2l :: C)
           | (RTint, RTdouble) => (identity, Ji2d :: C)
           | (RTint, RTobject) => wrapInteger C

           | (RTlong, RTnumber) => wrapLong C
           | (RTlong, RTint)    => (identity, Jl2i :: C)
           | (RTlong, RTdouble) => (identity, Jl2d :: C)
           | (RTlong, RTobject) => wrapLong C

           | (RTdouble, RTnumber) => wrapDouble C
           | (RTdouble, RTint)    => (identity, Jd2i :: C)  (* ??? *)
           | (RTdouble, RTlong)   => (identity, Jd2l :: C)  (* ??? *)
           | (RTdouble, RTobject) => wrapDouble C

           | (RTbool, RTconstructor) => wrapBoolean C
           | (RTbool, RTobject)      => wrapBoolean C

           | (RTstring, RTchararray) => invokeVirtual stringToCharArray C
           | (RTchararray, RTstring) => wrapString C

           | (RTvector, RTarray) => getField vectorElems C
           | (RTarray, RTvector) => wrapVector C

           | (RTblock, RTtuple)       => castTo Tuple C
           | (RTblock, RTconstructor) => castTo Constructor C
           | (RTblock, RTref)         => castTo Ref C
           | (RTblock, RTexception)   => castTo Exception C

           | (RTtuple, RTblock)       => (identity, C)
           | (RTconstructor, RTblock) => (identity, C)
           | (RTconstructor, RTbool)  => getField constructorTag C
           | (RTref, RTblock)         => (identity, C)
           | (RTexception, RTblock)   => (identity, C)

        (* | (RTclosure, RTmethod _) => ??? *)

           | (RTobject, RTnumber)      => castTo Number C
           | (RTobject, RTint)         => unwrapInteger C
           | (RTobject, RTlong)        => unwrapLong C
           | (RTobject, RTdouble)      => unwrapDouble C
           | (RTobject, RTbool)        => unwrapBoolean C
           | (RTobject, RTchararray)   => unwrapString C
           | (RTobject, RTstring)      => castTo String C
           | (RTobject, RTvector)      => castTo Vector C
           | (RTobject, RTarray)       => castToArray (Tclass Object) C
           | (RTobject, RTblock)       => castTo Block C
           | (RTobject, RTtuple)       => castTo Tuple C
           | (RTobject, RTconstructor) => castTo Constructor C
           | (RTobject, RTref)         => castTo Ref C
           | (RTobject, RTexception)   => castTo Exception C
           | (RTobject, RTclosure)     => castTo Closure C
        (* | (RTobject, RTmethod _)    => ??? *)
```

```
          | (_, RTobject) => (identity, C)
          | _ => fatalError "Coercion.coerce: invalid coercion")
```

```
(* Freeenv.sml -- applicative map from (negative) Lvar id to free var index
 *
 * Peter Bertelsen
 * December 1997
 *)

type free_env

val fresh      : free_env
val fromList   : int list -> free_env
val fromSet    : Intset.intset -> free_env
val add        : free_env -> int -> int -> free_env
val remove     : free_env -> int -> free_env
val lookup     : free_env -> int -> int option
val find       : free_env -> int -> int
val numItems   : free_env -> int
val app        : (int * int -> unit) -> free_env -> unit
val foldl      : (int * int * 'a -> 'a) -> 'a -> free_env -> 'a
val foldr      : (int * int * 'a -> 'a) -> 'a -> free_env -> 'a
val shift      : free_env -> free_env
val domain     : free_env -> int list

(* NOTE: remove and find raise Domain in case the specified index is
 * not found in the environment *)
```

```
(* Freeenv.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

open Lambda

datatype free_env = ENV of int Intmap.intmap

val fresh = ENV (Intmap.empty())

fun fromList xs =
    let fun h _ []      m = ENV m
          | h i (x::xr) m = h (i+1) xr (Intmap.insert(m, i, x))
    in
        h 0 xs (Intmap.empty())
    end

fun fromSet s =
    let fun h (i, (n, m)) = (n+1, Intmap.insert(m, i, n))

        val (_, m) = Intset.foldl h (0, Intmap.empty()) s
    in
        ENV m
    end

fun add (ENV m) i x = ENV (Intmap.insert(m, i, x))
fun remove (ENV m) i =
    let val (m', _) =
        (Intmap.remove(m, i)) handle Intmap.NotFound => raise Domain
    in
        ENV m'
    end

fun lookup (ENV m) i = Intmap.peek(m, i)

fun find (ENV m) i = (Intmap.retrieve(m, i))
    handle Intmap.NotFound => raise Domain

fun numItems (ENV m) = Intmap.numItems m

fun app f (ENV m) = Intmap.app f m

fun foldl f a (ENV m) = Intmap.foldl f a m

fun foldr f a (ENV m) = Intmap.foldr f a m

fun shift (ENV m) =
    let fun h (~1, x, m') = m'
          | h (i,  x, m') = Intmap.insert(m', i+1, x)
    in
        ENV (Intmap.foldl h (Intmap.empty()) m)
              (* or .foldr *)
    end

fun domain (ENV m) = Intmap.foldr (fn (i, _, l) => i::l) [] m
```

```
(* Genclass.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

local
    open Jvmtype Bytecode Classdecl Localenv Jvmcode Runtype Smlclasses
         Instantiate Codeutil

    (* Codeutil Error *)

    fun maxStack entries C = (Stackdepth.maxdepth C entries)
                     handle Fail s => (print(s ^ "\n");
                                       0xffff)

    val init0 : method_decl =
        let val (env, j) = bind envInstance 1 RTint
            val code = [Jaload Localvar.this,
                        Jdup,
                        Jinvokespecial closureInit,
                        Jiload j,
                        Jputfield closureTag,
                        Jreturn]
        in
            {flags = [ACCprivate],
             name  = initName,
             msig  = ([Tint], NONE),
             attrs = [CODE {stack  = maxStack [] code,
                            locals = maxLocals env,
                            code   = code,
                            hdls   = [],
                            attrs  = []}]
              }
        end

    val init : method_decl =
        let val (env,  j)  = bind envInstance 1 RTint
            val (env', j') = bind env 2 RTarray
            val code = [Jaload Localvar.this,
                        Jdup,
                        Jinvokespecial closureInit,
                        Jdup,
                        Jiload j,
                        Jputfield closureTag,
                        Jaload j',
                        Jputfield closureFree,
                        Jreturn]
        in
            {flags = [ACCprivate],
             name  = initName,
             msig  = ([Tint, Tarray (Tclass Object)], NONE),
             attrs = [CODE {stack  = maxStack [] code,
                            locals = maxLocals env',
                            code   = code,
                            hdls   = [],
                            attrs  = []}]
             }
        end

    val main : method_decl =
        {flags = [ACCpublic, ACCstatic],
         name  = "main",
         msig  = ([Tarray (Tclass String)], NONE),
         attrs = [CODE {stack  = 0,
                        locals = 1,
                        code   = [Jreturn],
                        hdls   = [],
                        attrs  = []}]
         }

    fun staticFlags export =
        if export then [ACCpublic, ACCstatic]
        else [ACCprivate, ACCstatic]

    fun makeStaticField (name, {runType, export}, fs) =
        {flags = staticFlags export,
         name  = name,
         ty    = toJvmType runType,
         attrs = []} :: fs : field_decl list

    fun makeCodeAttr code env hdls =
        CODE {stack  = maxStack (List.map #entry hdls) code,
              locals = maxLocals env,
              code   = code,
              hdls   = rev hdls,
              attrs  = []} : attribute

       (* NOTE: the list of exception handler declarations for each
       * method must be reversed to make nested handlers work
       * properly. *)

    fun makeStaticMethod ({name, argTypes, resType, export, code, env, hdls},
                          ms) =
        {flags = staticFlags export,
         name  = name,
         msig  = toMethodSig argTypes resType,
         attrs = [makeCodeAttr code env hdls]} :: ms : method_decl list

    fun makeClinit class exns {code, env, hdls} =
        let val class' = className class

            fun h (uid as ({qual, id}, n), C) =
                Jsconst (uidToString uid) ::
                Jputstatic (exnToFieldref RTobject uid) :: C

            val code' = Binaryset.foldl h code exns
        in
            {flags = [ACCstatic],
             name  = clinitName,
             msig  = ([], NONE),
             attrs = [makeCodeAttr code' env hdls]} : method_decl
        end

    fun makeApply {code, env, entries, hdls} =
        let val firstTag' = Tag.toInt Tag.firstTag
            val lastTag'  = Tag.toInt(lastTag())
            val failLbl   = newLabel()

               (* Generate a switch on the closure tag at the beginning
               * of method apply *)
            fun tagSwitch _ [] C = C
                | tagSwitch fail (lbls as (lbl::rest)) C =
                Jaload Localvar.this ::
                Jgetfield closureTag ::
                 (case rest of
                           [] => if firstTag' = 0 then Jifeq lbl :: C
                           else intConst firstTag' ::
                                Jif_icmpeq lbl :: C
                           | _  => Jtableswitch {default = fail,
                                                 offset  = Int32.fromInt firstTag',
                                                 targets = Vector.fromList lbls} :: C)

            fun sortEntries [] = []
                | sortEntries xs =
                let val a = Array.array(lastTag' - firstTag' + 1, failLbl)
                    fun h (tag, lbl) =
                        let val i = Tag.toInt tag - firstTag'
                        in
                            Array.update(a, i, lbl)
                        end
                in
                    List.app h xs;
                    Array.foldr (op ::) [] a
                end

            val code'  = Jlabel failLbl :: Jnew SmlError :: Jdup ::
                         Jsconst "unmatced closure tag" ::
                         Jinvokespecial smlErrorInit :: Jathrow :: code
            val code'' = tagSwitch failLbl (sortEntries entries) code'
        in
            {flags = [ACCpublic],
             name  = "apply",
             msig  = ([Tclass Smlclasses.Object],
                      SOME (Tclass Smlclasses.Object)),
             attrs = [makeCodeAttr code'' env hdls]} : method_decl
        end

    fun makeClassDecl srcName ({class, names, exns, methods, usesInit0,
                                usesInit, clinit, apply}: jvm_code) =
        let val usesInit0'  = !usesInit0
            val usesInit'   = !usesInit
```

```
            val anyClosures = usesInit0' orelse usesInit'
            val ms     = [main, makeClinit class (!exns) (!clinit)]
            val ms'    = if anyClosures then makeApply (!apply) :: ms else ms
            val ms''   = if usesInit'  then init :: ms'   else ms'
            val ms''' = if usesInit0' then init0 :: ms'' else ms''
        in
            {flags   = [ACCsuper, ACCpublic],
             this    = class,
             super   = SOME (if anyClosures then Closure else Object),
             ifcs    = [],
             fdecls  = Binarymap.foldl makeStaticField [] (!names),
             mdecls  = List.foldl makeStaticMethod ms''' (!methods),
             attrs   = [SRCFILE (Path.file srcName)]} : class_decl
        end
in
    fun genClass srcName trgtName jvmCode =
        let val outStream = BinIO.openOut trgtName
            fun out w     = BinIO.output1(outStream, w)
            val classDecl = makeClassDecl srcName jvmCode
            val constPool = Constpool.create()
        in
            (Classfile.emit out constPool classDecl;
             BinIO.closeOut outStream)
            handle x => (BinIO.closeOut outStream;
                         FileSys.remove trgtName;
                         raise x)
        end
end
```

```
(* Gencode.sig
 *
 * Peter Bertelsen
 * December 1997
 *)

type free_runtypes = Runtype.runtype Intmap.intmap
type closure_env   = (Tag.tag, free_runtypes) Binarymap.dict
type global_env    = (Const.unique_id, Runtype.runtype) Binarymap.dict
type export_env    = (Const.unique_id, Const.unique_id) Binarymap.dict
type method_env    = (Tag.tag, string * bool) Binarymap.dict

val resetCodeGen : unit -> unit
    (* reset label and tag generators *)

val compileTopLevel : global_env -> export_env -> Lambda.Lambda list ->
                      closure_env ref -> Jvmcode.jvm_code -> unit
    (* generate JVM code for top-level expressions *)

val compileClosures : global_env -> export_env -> Lift.pending_closure list ->
                      closure_env ref -> Jvmcode.jvm_code -> unit
    (* generate JVM code for lifted closure bodies *)

val compileMethods : global_env -> method_env -> export_env ->
                     Lift.pending_method list -> closure_env ref ->
                     Jvmcode.jvm_code -> unit
    (* generate JVM code for specialized closure bodies *)
```

```
(* Gencode.sml -- generating JVM bytecode from Lambda terms
 *               (based on Doug Currie's Back.sml)
 *
 * Peter Bertelsen
 * December 1997
 *)

open Mixture Lambda Prim Const Lift Bytecode Smlclasses Codeutil Runtype
     Coercion Instantiate Jvmcode

val checkOverflow = false  (* flag: generate code to check for Overflow? *)
val checkDiv      = true   (* flag: generate code for raising Div? *)


type free_runtypes = runtype Intmap.intmap
type closure_env   = (Tag.tag, free_runtypes) Binarymap.dict
type global_env    = (Const.unique_id, Runtype.runtype) Binarymap.dict
type export_env    = (Const.unique_id, Const.unique_id) Binarymap.dict
type method_env    = (Tag.tag, string * bool) Binarymap.dict


fun genCodeError s = Error.fatalError("Gencode." ^ s)

(* Reset the code generator state (label and closure tag counters) *)
fun resetCodeGen () = (resetLabels();
                       resetTags())


(* Compile the clauses of an Lcase or Lswitch statement; this function
 * cannot be declared as part of the large letrec in compileExpr since
 * it gives a typing problem (the type of intOfKey cannot be
 * generalized). *)
fun compClauses _ _ _ [] _ = genCodeError "compClauses"
  | compClauses compExpr' (SOME sfLbl) intOfKey [(k, e)] C =
    let val C' = compExpr' e C
        val k'  = intOfKey k
    in
        if k' = 0 then
            Jifne sfLbl :: C'
        else
            intConst k' :: Jif_icmpne sfLbl :: C'
    end
  | compClauses compExpr' sfOpt intOfKey (clause::rest) C =
    let val int32OfKey   = Int32.fromInt o intOfKey
        val (branch, C') = makeBranch C

        fun compClause ((k, e), (cases, C'')) =
            let val (lbl, C''') = labelCode(compExpr' e (branch :: C''))
            in
                ((int32OfKey k, lbl) :: cases, C''')
            end

        val (cases, C'') =
            let val (k, e) = clause
                val (lbl, C'') = labelCode(compExpr' e C')
            in
                List.foldl compClause ([(int32OfKey k, lbl)], C'') rest
            end

        (* NOTE: we should not generate a branch after the code for
         * the first clause (the one whose resulting bytecode comes
         * last); in case a branch is necessary, it will already be in
         * C, e.g. due to the context being a Lstatichandle term.
         * This is why we fold compClause over rest, not over all of
         * the clauses. *)
        val (lbl, cases') =
            (case (sfOpt, cases) of
                 (NONE, []) => genCodeError "compClauses (no staticfail)"
               | (NONE, (_, lbl)::xr) => (lbl, xr)
               | (SOME lbl, _) => (lbl, cases))

        (* NOTE: sfOpt = NONE should only happen when the case
         * statement is in fact exhaustive; in that case we use one of
         * the cases as the `default' target. *)
    in
        Jlookupswitch {default = lbl,
                       cases   = cases'} :: C''
    end


(* The translator from lambda terms to lists of instructions.
 *
 * globalEnv  : maps a global name (uid) to its runtype
 * exportEnv  : maps a global name (uid) to the exported name (uid)
 * closureEnv : maps a closure tag to a runtype environment for the free vars
 *              of the closure (side-effected)
 * jvmCode    : target JVM code and class info
 * hdls       : declared exception handlers (side-effected)
 * varIndex   : renumbering of Lvars (in compilation of specialized closures)
 * getLvar    : generating code to load the value of an Lvar
 * lvarRT     : maps an Lvar id to its runtype
 * localEnv   : maps an Lvar id to a Localvar index (side-effected)
 * sfOpt      : NONE or SOME(the label to which Lstaticfail must branch)
 * compShared : compilation of Lshared Lambda expressions
 * depth      : depth of the Front.sml stack model (excl. codegen temporaries)
 * expr       : lambda term to compile
 * C          : continuation, i.e., code that follows the code for expr
 *)

fun compileExpr globalEnv exportEnv (closureEnv : closure_env ref)
        (jvmCode: jvm_code) hdls varIndex getLvar lvarRT
        (localEnv : Localenv.local_env ref) sfOpt compShared =
    let fun compExpr depth expr C =
        (case expr of
             Lvar n               => compLvar n C
           | Lconst scon          => compConst scon C
           | Lapply (body, args)  => compApply depth body args C
           | Lfn body  => genCodeError "compExpr: Lfn"
           | Llet (args, body)    => compLet depth args body C
           | Lletrec (args, body) => compLetrec depth args body C
           | Lprim (prim, args)   => compPrim depth prim args C
           | Lcase (arg, clauses) => compCase depth arg clauses C
           | Lswitch (size, arg, cs) => compSwitch depth size arg cs C
           | Lstaticfail =>
                 (case sfOpt of
                      NONE     => genCodeError "compileExpr: Lstaticfail"
                    | SOME lbl => Jgoto lbl :: dropDead C)
           | Lstatichandle (body, handler) =>
                 compStaticHandle depth body handler C
           | Lhandle (body, handler) => compHandle depth body handler C
           | Lif (cond, ifSo, ifNot) => compIf depth cond ifSo ifNot C
           | Lseq (e1, e2) => compExpr depth e1 (Jpop :: compExpr depth e2 C)
           | Lwhile (cond, body) => compWhile depth cond body C
           | Landalso (e1, e2)   => compAndAlso depth e1 e2 C
           | Lorelse (e1, e2)    => compOrElse depth e1 e2 C
           | Lunspec => C
           | Lshared (ref e, lblRef) => compShared (compExpr depth) e lblRef C
           | Lassign _ => genCodeError "compExpr: Lassign")

        and compLvar n (Jpop :: C) = C
```

```
  | compLvar n C =
    let val (rt, C') = getLvar n C
    in
        C'
    end

and compConst _ (Jpop :: C) = C
  | compConst scon C =
        (case scon of
             ATOMsc a => genAtom a C
           | BLOCKsc (tag, scons) => compBlockConst tag scons C
           | QUOTEsc _ => genCodeError "compConst: QUOTEsc")

and compBlockConst (CONtag (tag, span)) scons C =
    (case (span, scons) of
         (1, []) => genUnit C
       | (1, _)  => genTuple compConst scons C
       | (_, []) => genConstructor0 tag C
       | (_, _)  => genConstructor compConst tag scons C)
  | compBlockConst (EXNtag uid) scons C =
    (registerExn jvmCode uid;
     case scons of
         [] => genException0 uid C
       | _  => genException compConst uid scons C)

and compApply depth (Lprim (Pclosure (tag, SPEC ts), free)) [] C =
    (* assuming List.length(#argTypes ts) = 0 *)
    let (* val _  = print("apply " ^ specClosToString tag ts ^ "\n") *)
        val {class, ...} = jvmCode
        val mref = specMethodref class (specMethodName tag) ts
        val C'   = Jinvokestatic mref :: C
    in
        registerFreeRts tag free lvarRT closureEnv;
        if free = [] then C'
        else genArray (compExpr depth) free C'
    end
  | compApply depth (body as (Lvar n)) args C =
    compApplyLvar depth n args C
  | compApply depth (body as (Lprim (Pget_global uid, []))) args C =
    compApplyGlobal depth uid args C
  | compApply depth body args C = compApplyClosures depth body args C

and compApplyLvar depth n args C =
    let val n' = varIndex n

        fun loadFree C' =
            let val C''  = Jgetfield closureFree :: C'
                val C''' = if n' >= 0 then C''
                           else Jcheckcast (CLASS Closure) :: C''
                (* NOTE: we needn't use checkcast for a
                 * locally bound Lvar that isn't free; it is
                 * assumed to have runtype RTmethod, and thus
                 * to be bound to a Closure object at
                 * run-time.  The bytecode verifier should
                 * accept this. *)
                val (rt, C'''') = getLvar n C'''
            in
                C''''
            end

        fun applyClosure C' =
            let val (new, C'') = compApplyClosureArgs depth args C'
                val (rt, C''') = getLvar n (dropCast Closure C'')
            in
                new C'''
            end
    in
        case lvarRT n of
            RTmethod (tag, SPEC ts) =>
                let val {freeTypes, argTypes, ...} = ts
                in
                    if List.length args >= List.length argTypes then
                        let val mref = specMethodref (#class jvmCode)
                                           (specMethodName tag) ts
                            val C'   = compApplyMethod depth mref
                                           argTypes args C
                        in
                         (* print("apply Lvar " ^ Int.toString n ^
                                  ": " ^ specClosToString tag ts ^
                                  "\n"); *)
                            if freeTypes = [] then C'
                            else loadFree C'
                        end
                    else applyClosure C
                end
          | RTclosure => applyClosure C
          | _ => let val (new, C') = compApplyClosureArgs depth args C
                     val (rt, C'') = getLvar n C'
                 in
                     new C''
                 end

    end

and compApplyGlobal depth uid args C =
    let val uid' =
        (case Binarymap.peek(exportEnv, uid) of
             NONE      => uid
           | SOME uid' => uid')

        fun applyDefault C' =
            let val (new, C'') = compApplyClosureArgs depth args C'
            in
                Jgetstatic (toFieldref RTobject uid') :: new C''
            end
    in
        case Binarymap.peek(globalEnv, uid') of
            SOME (RTmethod (tag, SPEC ts)) =>
                let val {freeTypes, argTypes, ...} = ts
                in
                    if List.length args >= List.length argTypes then
                        let val mref = globalMethodref ts uid'
                            val C'   = compApplyMethod depth mref
                                           argTypes args C
                        in
                         (* print("apply " ^ uidToString uid' ^ ": " ^
                                  specClosToString tag ts ^ "\n"); *)
                            if freeTypes = [] then C'
                            else genCodeError
                                  ("compApplyGlobal: free vars with " ^
                                  uidToString uid')
                        end
                    else applyDefault C
                end
          | _ => applyDefault C
    end

and compApplyMethod depth mref argTypes args C =
    let fun compArgs [] args' =
            let val (new, C') = compApplyClosureArgs depth args' C
            in
```

```
                (new, Jinvokestatic mref :: C')
            end
          | compArgs (rt::rtr) (a::ar) =
            let val (new,  C')  = compArgs rtr ar
                val (new', C'') = coerce RTobject rt C'
            in
                (new o new', compExpr depth a C')
            end
          | compArgs _ _ = genCodeError "compApplyMethod"

        val (new, C') = compArgs argTypes args
    in
        new C'
    end

and compApplyClosureArgs depth args C =
    let fun apply []      = (identity, C)
          | apply (a::ar) =
        let val (new, C') = apply ar
            val C'' =
                compExpr depth a (Jinvokevirtual closureApply :: C')
            val (new', C''') = coerce RTobject RTclosure C''
        in
            (new o new', C''')
        end
    in
        apply args
    end

and compApplyClosures depth body args C =
    let fun h (arg, (new, C')) =
        let val C'' =
            compExpr depth arg (Jinvokevirtual closureApply :: C')
            val (new', C''') = coerce RTobject RTclosure C''
        in
            (new o new', C''')
        end

        val (new, C') = List.foldr h (identity, C) args
    in
        new(compExpr depth body C')
    end

and compLet depth [arg] Lunspec C =
                (* ^ special case arising from val _ = arg *)
    let val C' =
        (case C of
             Jreturn :: _ => C
           | _            => Jpop :: C)
    in
        compExpr depth arg C'
    end
  | compLet depth args body C =
    let fun h d [] = compExpr d body C
          | h d (a::ar) =
        let val rt =
            (case a of
                 Lprim (Pclosure (tag, spec as SPEC _), _) =>
                     RTmethod (tag, spec)
               | Lprim (Pclosure _, _) => RTclosure
               | _ => RTobject)
            val j  = bindLvar localEnv d rt
            val C' = h (d+1) ar
        in
            compExpr d a (storeInVar j C')
        end
    in
        h depth args
    end

and compLetrec depth args body C =
    let fun bindArgs d []       = ()
          | bindArgs d (a::ar) =
            let val rt =
                (case a of
                     Lprim (Pclosure (tag, spec as SPEC _), _) =>
                          RTmethod (tag, spec)
                   | _ => RTclosure)
            in
                bindLvar localEnv d rt;
                bindArgs (d+1) ar
            end

        fun compArgs d [] cs C' = (cs, C')
          | compArgs d ((Lprim (Pclosure (tag, _), free))::ar) cs C' =
            let val (j, _) = Localenv.find (!localEnv) d
                 val C'' =
                     (case free of
                          [] => C'
                        | _  => Jaload j ::
                                genArray (compExpr d) free
                                    (Jputfield closureFree :: C'))
             in
                 registerFreeRts tag free lvarRT closureEnv;
                 compArgs (d+1) ar ((j, tag) :: cs) C''
             end
          | compArgs _ _ _ _ = genCodeError "compLetrec"

        val {class, usesInit0, ...} = jvmCode

        fun initArg ((j, tag), C') =
            genTarget0 class tag (storeInVar j C')

        val depth'   = depth + List.length args
        val (cs, C') =
            (bindArgs depth args;
             compArgs depth args [] (compExpr depth' body C))
             handle Domain => genCodeError "compLetrec: Domain"
    in
        usesInit0:= true;   (* asumming args <> [] *)
        List.foldl initArg C' cs
    end

and compPrim depth prim args C =
    (case (prim, args) of
         (Pidentity, [e]) => compExpr depth e C
       | (Pget_global uid, [])  => compGetGlobal uid C
       | (Pset_global uid, [e]) => compSetGlobal depth uid e C
       | (Pupdate, _) => genCodeError "compPrim: Pupdate"
       | (Ptest t, es) => compBoolTest depth t es C
       | (Pmakeblock tag, es)  => compMakeBlock depth tag es C
       | (Ptag_of, _) => genCodeError "compPrim: Ptag_of"
       | (Pfield n, [e]) => compGetField depth n e C
       | (Psetfield n, [e1, e2]) => compSetField depth n e1 e2 C
       | (Pccall (name, arity), es) =>
             compCcall depth name arity es C
       | (Praise, [e]) => compRaise depth e C
       | (Pnot, [e])   => compNot depth e C
      (* unsigned integer operations *)
       | (Paddint, es) => compUncheckedIntOper Jiadd depth es C
       | (Psubint, es) => compUncheckedIntOper Jisub depth es C
       | (Pmulint, es) => compUncheckedIntOper Jimul depth es C
```

```
          | (Pdivint, es) => compIntDiv false Jidiv depth es C
          | (Pmodint, es) => compIntDiv false Jirem depth es C
          | (Pandint, es) => compUncheckedIntOper Jiand depth es C
          | (Porint,  es) => compUncheckedIntOper Jior depth es C
          | (Pxorint, es) => compUncheckedIntOper Jixor depth es C
          | (Pshiftleftint, es) => compUncheckedIntOper Jishl depth es C
          | (Pshiftrightintsigned, es) =>
                compUncheckedIntOper Jishr depth es C
          | (Pshiftrightintunsigned, es) =>
                compUncheckedIntOper Jiushr depth es C
          | (Pintoffloat, [e]) => compRealToInt depth e C
          | (Pfloatprim Pfloatofint, [e])  => compIntToReal depth e C
          | (Pfloatprim Psmlnegfloat, es) => compRealOper depth es Jdneg C
          | (Pfloatprim Psmladdfloat, es) => compRealOper depth es Jdadd C
          | (Pfloatprim Psmlsubfloat, es) => compRealOper depth es Jdsub C
          | (Pfloatprim Psmlmulfloat, es) => compRealOper depth es Jdmul C
          | (Pfloatprim Psmldivfloat, es) => compRealOper depth es Jddiv C
          | (Pstringlength, [e]) => compStringLength depth e C
          | (Pgetstringchar, [e1, e2]) =>
                compGetStringChar depth e1 e2 C
          | (Psetstringchar, [e1, e2, e3]) =>
                compSetStringChar depth e1 e2 e3 C
          | (Pmakevector, [e1, e2]) => compMakeVector depth e1 e2 C
          | (Pvectlength, [e]) => compVectLength depth e C
          | (Pgetvectitem, [e1, e2]) => compGetVectItem depth e1 e2 C
          | (Psetvectitem, [e1, e2, e3]) =>
                compSetVectItem depth e1 e2 e3 C
          (* NOTE: smlXXXXint are signed integer operations *)
          | (Psmlnegint, [e]) => compNegInt depth e C
          | (Psmlsuccint, _) => genCodeError "compPrim: Psmlsuccint"
          | (Psmlpredint, _) => genCodeError "compPrim: Psmlpredint"
          | (Psmladdint, es) => compIntOper checkOverflow Jiadd depth es C
          | (Psmlsubint, es) => compIntOper checkOverflow Jisub depth es C
          | (Psmlmulint, es) => compIntOper checkOverflow Jimul depth es C
          | (Psmldivint, es) => compIntDiv checkOverflow Jidiv depth es C
          | (Psmlmodint, es) => compIntDiv false Jirem depth es C
          | (Pmakerefvector, [e1, e2]) => compMakeVector depth e1 e2 C
          | (Patom n, [e]) =>
                compExpr depth e (Jpop :: genConstructor0 n C)
          | (Psmlquotint, es) => compIntDiv checkOverflow Jidiv depth es C
          | (Psmlremint, es)  => compIntDiv false Jirem depth es C
          | (Pclosure info, es) => compClosure depth info es C
          | (Pswap, es) => compExprList depth es (Jswap :: C)
          | _ => genCodeError "compPrim: unimplemented primitive")

and compIntExprs depth es C =
    let fun h (e, C') =
        let val (new, C'') = coerce RTobject RTint C'
        in
            new(compExpr depth e C'')
        end
    in
        List.foldr h C es
    end

(* integer operation without check for overflow or division by zero *)
and compUncheckedIntOper oper depth es C =
    let val (new, C') = coerce RTint RTnumber C
    in
        new(compIntExprs depth es (oper :: C'))
    end

and compIntOper chkOverflow oper depth es C =
    if chkOverflow then
        (* integer operation with check for overflow: converts
         * each operand to a long value, performs a long
```

```
          * operation, checks that the result is within int
          * bounds, converts back to an int result, and builds
          * an Integer object *)
         let val (new, C')  = coerce RTlong RTnumber C
             val lbl  = newLabel()
             val lbl' = newLabel()
             val C'' =
                 longOper oper :: Jdup2 :: Jlconst minInt32As64 ::
                 Jlcmp :: Jiflt lbl :: Jdup2 :: Jlconst maxInt32As64 ::
                 Jlcmp :: Jifle lbl' :: Jlabel lbl :: Jpop2 ::
                 throwOverflow(Jlabel lbl' :: C')
         in
             new(compLongExprs depth es C'')
         end
     else compUncheckedIntOper oper depth es C

 and compLongExprs depth es C =
     let fun h (e, C') =
         let val (new, C'') = coerce RTobject RTlong C'
         in
             new(compExpr depth e C'')
         end
     in
         List.foldr h C es
     end

 and compRealExprs depth es C =
     let fun h (e, C') =
         let val (new, C'') = coerce RTobject RTdouble C'
         in
             new(compExpr depth e C'')
         end
     in
         List.foldr h C es
     end

 and compRealOper depth es oper C =
     let val (new, C') = coerce RTdouble RTnumber C
     in
         new(compRealExprs depth es (oper :: C'))
     end

 and compStringExprs depth es C =
     let fun h (e, C') =
         let val (new, C'') = coerce RTobject RTstring C'
         in
             new(compExpr depth e C'')
         end
     in
         List.foldr h C es
     end

 and compGetGlobal uid C =
     let val uid' =
         (case Binarymap.peek(exportEnv, uid) of
              NONE      => uid
            | SOME uid' => uid')
     in
         Jgetstatic (toFieldref RTobject uid') :: C
     end

 and compSetGlobal depth (uid as ({qual, id}, n)) e C =
     let val C'  =
             (case C of
                  Jpop :: C' => C'
                | _          => C)
```

```
        val (field, name, nameInfo) =
            (case Binarymap.peek(exportEnv, uid) of
                 NONE => (toFieldref RTobject uid,
                          toFieldName id n,
                          {runType = RTobject,
                           export  = false})
               | SOME (uid' as ({qual, id}, n)) =>
                     (toFieldref RTobject uid',
                      toFieldName id n,
                      {runType = RTobject,
                       export  = true}))
        val {names, ...} = jvmCode
    in
        names:= Binarymap.insert(!names, name, nameInfo);
        compExpr depth e (putStatic field C')
    end

and compBoolTest depth t es C =
    let val (new, C') = coerce RTbool RTconstructor C
        val (t', lbl, C'') =
            (case C' of
                 Jifne lbl :: C'' => (t, lbl, C'')
               | Jifeq lbl :: C'' => (invertTest t, lbl, C'')
               | _ =>
                     let val (lbl,  C'')  = labelCode C'
                         val lbl' = newLabel()
                         val C''' = intConst 0 :: Jgoto lbl ::
                                    Jlabel lbl' :: intConst 1 :: C''
                     in
                         (t, lbl', C''')
                     end)
        val C''' =
            (case t' of
                 Peq_test =>
                     compExprList depth es (Jif_acmpeq lbl :: C'')
               | Pnoteq_test =>
                     compExprList depth es (Jif_acmpne lbl :: C'')
               | Pint_test t'' =>
                     let val test = (intTest t'') lbl
                     in
                         compIntExprs depth es (test :: C'')
                     end
               | Pfloat_test t'' =>
                     let val test = (zeroTest t'') lbl
                     in
                         compRealExprs depth es (Jdcmpg :: test :: C'')
                     end
               | Pstring_test t'' =>
                     let val test = (zeroTest t'') lbl
                     in
                         compStringExprs depth es
                             (Jinvokevirtual stringCompareTo ::
                              test :: C'')
                     end
               | Pword_test t'' =>
                     let val test = (intTest t'') lbl
                     in
                         compIntExprs depth es (test :: C'')
                     end
               | Pnoteqtag_test tag =>
                     genCodeError "compBoolTest: Pnoteqtag_test")
    in
        new C'''
    end
```

```
and compMakeBlock depth (CONtag (tag, span)) es C =
    let val compExpr' = compExpr depth
    in
        if tag = Config.refTag then
            (case es of
                 [e] => genRef compExpr' e C
               | _   => genCodeError "compMakeBlock: malformed ref")
        else if span = 1 then genTuple compExpr' es C
             else genConstructor compExpr' tag es C
    end
  (* Note: there's no need to treat the case es = []
   * specially here.  Empty tuples, aka () : unit,
   * null-constructors, and null-exceptions are represented as
   * Lconsts. *)
  | compMakeBlock depth (EXNtag uid) es C =
    (registerExn jvmCode uid;
     genException (compExpr depth) uid es C)

and compGetField depth n e C =
    let val (new, C') =
        coerce RTobject RTblock (Jgetfield blockArgs ::
                                 intConst n :: Jaaload :: C)
    in
        new(compExpr depth e C')
    end

and compSetField depth n e1 e2 C =
    let val C' = compExpr depth e2 (Jaastore :: genUnit C)
        val (new, C'') =
            coerce RTobject RTblock (Jgetfield blockArgs ::
                                     intConst n :: C')
    in
        new(compExpr depth e1 C'')
    end

and compCcall depth name arity es C =
    (case (name, arity) of
          ("sml_equal", 2) =>
              let val (new, C') = coerce RTbool RTconstructor C
                  val C'' = Jinvokevirtual objectEquals :: C'
              in
                  new(compExprList depth es C'')
              end
        | ("sml_not_equal", 2) =>
          let val (new, C') = coerce RTbool RTconstructor C
              val C'' =
                  (case C' of
                       Jifne lbl :: C'' => Jifeq lbl :: C''
                     | Jifeq lbl :: C'' => Jifne lbl :: C''
                     | C'' => intConst 1 :: Jixor :: C'')
              val C''' = Jinvokevirtual objectEquals :: C''
          in
              new(compExprList depth es C''')
          end
        | _ => genCodeError "compCcall: unimplemented call")

and compRaise depth e C =
    let val rt = (case e of
                      Lvar n => lvarRT n
                    | _      => RTobject)
        val (new, C') = coerce rt RTexception (Jathrow :: dropDead C)
    in
        new(compExpr depth e C')
    end

and compNot depth e C =
    let val (new, C') = coerce RTbool RTconstructor C
        val C'' =
            (case C' of
                 Jifne lbl :: C'' => Jifeq lbl :: C''
               | Jifeq lbl :: C'' => Jifne lbl :: C''
               | C'' => intConst 1 :: Jixor :: C'')
        val (new', C''') = coerce RTobject RTbool C''
    in
        new(new'(compExpr depth e C'''))
    end

and compIntDiv chkOverflow oper depth (es as [e1,Lconst(ATOMsc k)]) C =
    (case (intOfAtom k, checkDiv, chkOverflow) of
         (0, true, _) =>   (* division by zero *)
             let val uid = ({qual = "General", id = "Div"}, 0)
                 val C' = genException0 uid (Jathrow :: dropDead C)
             in
                 compExpr depth e1 (Jpop :: C')
             end
       | (~1, _, true) =>   (* check for overflow *)
             let val (new, C')  = coerce RTint RTnumber C
                 val (lbl, C'') = labelCode(intConst ~1 :: oper :: C')
                 val (new', C''') =
                     coerce RTobject RTint (Jdup :: Jiconst minInt32 ::
                                            Jif_icmpne lbl :: Jpop ::
                                            throwOverflow C'')
             in
                 new(new'(compExpr depth e1 C'''))
             end
       | _ => (* no need to check for division by zero or overflow *)
              compUncheckedIntOper oper depth es C)
    | compIntDiv chkOverflow oper depth es C =
    if checkDiv then
        let val (new, C') = coerce RTint RTnumber C
            val (branch, C'') = makeBranch C'
            val lbl   = newLabel()
            val lbl'  = newLabel()
            val lbl'' = newLabel()
            val C'''  = Jlabel lbl :: oper :: Jlabel lbl' :: branch ::
                        Jlabel lbl'' :: Jpop :: throwDiv C''
            val exnHdl = {start = lbl,
                          stop  = lbl',
                          entry = lbl'',
                          catch = SOME ArithmeticException}
            val C'''' =
                if chkOverflow then
                    Jdup :: intConst ~1 :: Jif_icmpne lbl :: Jswap ::
                    Jdup_x1 :: Jiconst minInt32 :: Jif_icmpne lbl ::
                    Jpop2 :: throwOverflow C'''
                else C'''
        in
            hdls:= exnHdl :: (!hdls);
            new(compIntExprs depth es C'''')
        end
    else compIntOper chkOverflow oper depth es C

and compRealToInt depth e C =
    let val (new, C') = coerce RTint RTnumber C
        val (new', C'') = coerce RTobject RTdouble (Jd2i :: C')
    in
        new(new'(compExpr depth e C''))
    end

and compIntToReal depth e C =
    let val (new, C') = coerce RTdouble RTnumber C
        val (new', C'') = coerce RTobject RTint (Ji2d :: C')
    in
        new(new'(compExpr depth e C''))
    end

and compStringLength depth e C =
    let val (new, C') = coerce RTint RTnumber C
        val (new', C'') =
            coerce RTobject RTstring (Jinvokevirtual stringLength ::
                                      C')
    in
        new(new'(compExpr depth e C''))
    end

and compGetStringChar depth e1 e2 C =
    let val (new, C') = coerce RTint RTnumber C
        val (new', C'') =
            coerce RTobject RTint (Jinvokevirtual stringCharAt :: C')
        val (new'', C''') =
            coerce RTobject RTstring (compExpr depth e2 C'')
    in
        new(new'(new''(compExpr depth e1 C''')))
    end

and compSetStringChar depth e1 e2 e3 C =
    genCodeError "compSetStringChar"
 (* let val C'   = Jcastore :: genUnit C
        val C''  = compExpr depth e3 (unwrapInteger C')
        val C''' = compExpr depth e2 (unwrapInteger C'')
    in
        compExpr depth e1 (unwrapCharVector C''')
    end
  *)

and compMakeVector depth e1 e2 C =
    let val C' = compExpr depth e2 (Jinvokestatic utilMakeArray :: C)
        val (new, C'') = coerce RTobject RTint C'
    in
        new(compExpr depth e1 C'')
    end

and compVectLength depth e C =
    let val (new, C') = coerce RTint RTnumber C
    in
        new(compExpr depth e (Jarraylength :: C'))
    end

and compGetVectItem depth e1 e2 C =
    let val (new, C') = coerce RTobject RTint (Jaaload :: C)
    in
        new(compExpr depth e1 (compExpr depth e2 C'))
    end

and compSetVectItem depth e1 e2 e3 C =
    let val C' = compExpr depth e3 (Jaastore :: genUnit C)
        val (new, C'') = coerce RTobject RTint C'
    in
        new(compExpr depth e1 (compExpr depth e2 C''))
    end

and compNegInt depth e C =
    let val (new, C') = coerce RTint RTnumber C
        val C'' =
            if checkOverflow then
                let val lbl = newLabel()
                in
```

```
                      Jdup :: Jiconst minInt32 ::
                      Jif_icmpne lbl :: Jpop ::
                      throwOverflow(Jlabel lbl :: Jineg :: C')
                  end
              else Jineg :: C'
          val (new', C''') = coerce RTobject RTint C''
      in
          new(new'(compExpr depth e C'''))
      end

  and compClosure depth (tag, _) free C =
      let val {class, usesInit0, usesInit, ...} = jvmCode
      in
          registerFreeRts tag free lvarRT closureEnv;
          case free of
              [] => (usesInit0:= true;
                     genTarget0 class tag C)
            | _  => (usesInit:= true;
                     genTarget class (compExpr depth) tag free C)
      end

  and compCase _ _ [] _ = genCodeError "compCase"
    | compCase depth arg (cs as ((scon,_)::_)) C =
      let val (new, C') =
          (case scon of
               REALscon _ =>
                   coerce RTobject RTdouble (compTests depth cs C)
             | STRINGscon _ =>
                   coerce RTobject RTstring (compTests depth cs C)
             | _ => let val C' = compClauses (compExpr depth) sfOpt
                                             intOfAtom cs C
                    in
                        coerce RTobject RTint C'
                    end)
      in
          new(compExpr depth arg C')
      end

  and compTests depth clauses C =
      let val sfLbl =
              (case sfOpt of
                   NONE => genCodeError "compTests (sfOpt)"
                 | SOME lbl => lbl)
          val (branch, C') = makeBranch C

          fun compClause ((scon, e), (lbl, C'')) =
              let val C''' = compExpr depth e (branch :: C'')
                  val lbl' = newLabel()
                  val C'''' =
                      (case scon of
                           REALscon r =>
                               Jlabel lbl' :: Jdup2 :: realConst r ::
                               Jdcmpg :: Jifne lbl :: Jpop2 :: C'''
                         | STRINGscon s =>
                               Jlabel lbl' :: Jdup :: Jsconst s ::
                               Jinvokevirtual stringCompareTo ::
                               Jifne lbl :: Jpop :: C'''
                         | _ => genCodeError "compTests")
              in
                  (lbl', C'''')
              end

          val lbl = newLabel()
      in
          case rev clauses of
              (REALscon r, e)::rest =>
```

```
                let val C'' = Jlabel lbl :: realConst r :: Jdcmpg ::
                              Jifne sfLbl :: compExpr depth e C'
                in
                    #2(List.foldl compClause (lbl, C'') rest)
                end
          | (STRINGscon s, e)::rest =>
                let val C'' = Jlabel lbl :: Jsconst s ::
                              Jinvokevirtual stringCompareTo ::
                              Jifne sfLbl :: compExpr depth e C'
                in
                    #2(List.foldl compClause (lbl, C'') rest)
                end
          | _ => genCodeError "compTests"
    end

and compSwitch depth size arg clauses C =
    (* We assume the argument to be safe (not producing
     * side-effects, and always terminating), because switches
     * are generated only by the match compiler *)
    (case (size, clauses) of
         (1, [(CONtag(_,_), e)]) => compExpr depth e C
       | (2, [(CONtag(0,_), e)]) =>
             compIf depth arg Lstaticfail e C
       | (2, [(CONtag(1,_), e)]) =>
             compIf depth arg e Lstaticfail C
       | (2, [(CONtag(0,_), e0), (CONtag(1,_), e1)]) =>
             compIf depth arg e1 e0 C
       | (2, [(CONtag(1,_), e1), (CONtag(0,_), e0)]) =>
             compIf depth arg e1 e0 C
       | _ => let val rt = (case arg of
                                Lvar n => lvarRT n
                              | _      => RTobject)
                  val (rt', C') =
                      if containsExnTags clauses then
                          (RTexception,
                           Jgetfield exceptionTag ::
                           compExnTests depth clauses C)
                      else let val C' =
                                   compClauses (compExpr depth) sfOpt
                                       intOfAbsoluteTag clauses C
                           in
                               (RTconstructor,
                                Jgetfield constructorTag :: C')
                           end
                  val (new, C'') = coerce rt rt' C'
              in
                  new(compExpr depth arg C'')
              end)

and compExnTests depth clauses C =
    let val sfLbl =
            (case sfOpt of
                 NONE => genCodeError "compExnTests (sfOpt)"
               | SOME lbl => lbl)
        val (branch, C') = makeBranch C

        fun compClause ((tag, e), (lbl, C'')) =
            let val C''' = compExpr depth e (branch :: C'')
                val lbl' = newLabel()
                val C'''' =
                    (case tag of
                         EXNtag uid =>
                         let val _ = registerExn jvmCode uid
                             val f = exnToFieldref RTobject uid
                         in
                             Jlabel lbl' :: Jdup :: Jgetstatic f ::
```

```
                            Jif_acmpne lbl :: Jpop :: C'''
                        end
                      | _ => genCodeError "compExnTests")
              in
                  (lbl', C'''')
              end

          val lbl = newLabel()
      in
          case rev clauses of
              (EXNtag uid, e)::rest =>
                  let val _ = registerExn jvmCode uid
                      val f = exnToFieldref RTobject uid
                      val C'' = Jlabel lbl :: Jgetstatic f ::
                                  Jif_acmpne sfLbl :: compExpr depth e C'
                  in
                      #2(List.foldl compClause (lbl, C'') rest)
                  end
            | _ => genCodeError "compExnTests"
      end

  and compStaticHandle depth body Lstaticfail C = compExpr depth body C
    | compStaticHandle depth body handler     C =
      let val (branch, C')  = makeBranch C
          val (sfBody, C'') = labelCode(compExpr depth handler C')
          val C''' = branch :: dropDead C''
      in
          compileExpr globalEnv exportEnv closureEnv jvmCode hdls
              varIndex getLvar lvarRT localEnv (SOME sfBody) compShared
              depth body C'''
      end

  and compHandle depth body handler C =
      let val (branch, C') = makeBranch C
          val lbl  = newLabel()
          val lbl' = newLabel()
          val j    = bindLvar localEnv depth RTexception
          val C''  = Jlabel lbl :: branch :: Jlabel lbl' ::
                     storeInVar j (compExpr (depth+1) handler C')
          val (lbl'', C''') = labelCode(compExpr depth body C'')
      in
          hdls:= {start = lbl'',
                  stop  = lbl,
                  entry = lbl',
                  catch = SOME Exception} :: (!hdls);
          C'''
      end

  and compIf depth cond Lstaticfail ifNot C =
      let val sfLbl =
          (case sfOpt of
               NONE     => genCodeError "compIf (sfOpt)"
             | SOME lbl => lbl)
          val (new, C') =
              coerce RTobject RTbool (Jifne sfLbl ::
                                      compExpr depth ifNot C)
      in
          new(compExpr depth cond C')
      end
    | compIf depth cond ifSo Lstaticfail C =
      let val sfLbl =
          (case sfOpt of
               NONE     => genCodeError "compIf (sfOpt)"
             | SOME lbl => lbl)
          val (new, C') =
              coerce RTobject RTbool (Jifeq sfLbl ::
```

```
                                  compExpr depth ifSo C)
    in
        new(compExpr depth cond C')
    end
  | compIf depth cond ifSo ifNot C =
    let val (branch, C') = makeBranch C
        val (lbl, C'')   = labelCode(compExpr depth ifNot C')
        val C''' = compExpr depth ifSo (branch :: dropDead C'')
        val (new, C'''') = coerce RTobject RTbool (Jifeq lbl :: C''')
    in
        new(compExpr depth cond C'''')
    end

and compWhile depth cond body C =
    let val lbl1 = newLabel()
        val (new, C') =
            coerce RTobject RTbool (Jifne lbl1 :: genUnit C)
        val (lbl2, C'') = labelCode(new(compExpr depth cond C'))
    in
        Jgoto lbl2 :: Jlabel lbl1 :: compExpr depth body (Jpop :: C'')
    end

and compAndAlso depth e1 e2 C =
    let val (new, C') =
        (case C of
             Jcheckcast (CLASS c) :: Jgetfield f :: C' =>
                 if c = Constructor andalso f = constructorTag then
                     let val C'' =
                         (case C' of
                              Jifeq lbl :: _ =>
                                  Jifeq lbl :: compExpr depth e2 C
                            | Jifne lbl :: C'' =>
                                  let val (lbl', C''') = labelCode C''
                                      val (new, C'''') =
                                          coerce RTobject RTconstructor
                                          (Jgetfield constructorTag ::
                                          Jifne lbl :: C''')
                                  in
                                      Jifeq lbl' ::
                                      new(compExpr depth e2 C'''')
                                   end
                            | _ => genCodeError "compAndAlso")
                     in
                         coerce RTobject RTbool C''
                     end
                 else genCodeError "compAndAlso"
           | _ => let val (lbl, C') = labelCode C
                      val (new, C'') =
                          coerce RTobject RTbool (Jifeq lbl :: Jpop ::
                                                  compExpr depth e2 C')
                  in
                      (new, Jdup :: C'')
                  end)
    in
        new(compExpr depth e1 C')
    end

and compOrElse depth e1 e2 (Jcheckcast (CLASS c) :: Jgetfield f :: C) =
    if c = Constructor andalso f = constructorTag then
        let val C' =
            (case C of
                 Jifne lbl :: _ =>
                     Jifne lbl :: compExpr depth e2 C
               | Jifeq lbl :: C' =>
                     let val (lbl', C'') = labelCode C'
                         val (new, C''') =
                                   coerce RTobject RTconstructor
                                          (Jgetfield constructorTag ::
                                           Jifeq lbl :: C'')
                               in
                                   Jifne lbl' :: new(compExpr depth e2 C''')
                               end
                         |_ => genCodeError "compOrElse")
                      val (new, C'') = coerce RTobject RTbool C'
                  in
                      new(compExpr depth e1 C'')
                  end
              else genCodeError "compOrElse"
            | compOrElse depth e1 e2 C =
              let val (lbl, C') = labelCode C
                  val C'' = Jifne lbl :: Jpop :: compExpr depth e2 C'
                  val (new, C''') = coerce RTobject RTbool C''
              in
                  new(compExpr depth e1 (Jdup :: C'''))
              end

        and compExprList depth es C =
                List.foldr (fn (e, C') => compExpr depth e C') C es
    in
        compExpr
    end  (* compileExpr *)

(* General compilation of an Lshared expression.  Assumption: all
 Lshared expressions are flagged either as LIFTED or as COMPILED. *)

fun compSharedGeneral compExpr expr lblRef C =
    (case !lblRef of
         LIFTED =>
             let val (lbl, C') = labelCode(compExpr expr C)
             in
                 lblRef:= COMPILED lbl;
                 C'
             end
       | COMPILED lbl => Jgoto lbl :: dropDead C
       | _            => genCodeError "compSharedGeneral")

(* Compilation of an Lshared expression for a specialized closure
 * Assumption: all Lshared expressions are flagged either as COMPILED
 * or SPECIALIZED. *)
fun compSharedSpecial compExpr expr lblRef C =
    (case !lblRef of
         COMPILED _ =>
             let val (lbl, C') = labelCode(compExpr expr C)
             in
                 lblRef:= SPECIALIZED lbl;
                 C'
             end
       | SPECIALIZED lbl => Jgoto lbl :: dropDead C
       | _               => genCodeError "compSharedSpecial")


(* Compile time model of runtime environment, mapping Lvar id to
 * closure args index or Localvar index; free is the environment of
 * free Lvars; the position of a negative id in free is the closure
 * args index; localEnv is a reference to a map from Lvar id to Localvar
 * index and runtype. *)

fun getTopLvar localEnv n C =
    if n < 0 then genCodeError("getTopLvar: free Lvar " ^ Int.toString n)
    else
        let val (j, rt) = Localenv.find (!localEnv) n
            handle Domain =>
```

```
                genCodeError("getTopLvar: Lvar " ^ Int.toString n)
        in
            (rt, Jaload j :: C)
        end

fun topLvarRT localEnv n =
    if n < 0 then genCodeError("topLvarRT: free Lvar " ^ Int.toString n)
    else
        let val (_, rt) = Localenv.find (!localEnv) n
            handle Domain =>
                genCodeError("topLvarRT: Lvar " ^ Int.toString n)
        in
            rt
        end

fun getClosureLvar freeRuntype free localEnv n C =
    if n < 0 then
        let val i = Freeenv.find free n
            handle Domain =>
                genCodeError("getClosureLvar: free Lvar " ^ Int.toString n)
        in
            (freeRuntype i,
             Jaload Localvar.this :: Jgetfield closureFree ::
             intConst i :: Jaaload :: C)
        end
    else let val (j, rt) = Localenv.find (!localEnv) n
             handle Domain =>
                 genCodeError("getClosureLvar: Lvar " ^ Int.toString n)
         in
             (rt, Jaload j :: C)
         end

fun closureLvarRT freeRuntype free localEnv n =
    if n < 0 then
        let val i = Freeenv.find free n
            handle Domain =>
                genCodeError("closureLvarRT: free Lvar " ^ Int.toString n)
        in
            freeRuntype i
        end
    else let val (_, rt) = Localenv.find (!localEnv) n
                 handle Domain =>
                     genCodeError("closureLvarRT: Lvar " ^ Int.toString n)
         in
             rt
         end

fun getMethodLvar freeRuntype free ofs localEnv n C =
    let val n' = n + ofs
    in
        if n' < 0 then
            let val i = Freeenv.find free n'
                handle Domain =>
                    genCodeError("getMethodLvar: free Lvar " ^ Int.toString n')
            in
                (freeRuntype i,
                 Jaload (Localvar.fromInt 0) :: intConst i :: Jaaload :: C)
            end
        else let val (j, rt) = Localenv.find (!localEnv) n'
                     handle Domain =>
                         genCodeError("getMethodLvar: Lvar " ^ Int.toString n')
             in
                 (rt, Jaload j :: C)
             end
    end
```

```
fun methodLvarRT freeRuntype free ofs localEnv n =
    let val n' = n + ofs
    in
        if n' < 0 then
            let val i = Freeenv.find free n'
                handle Domain =>
                    genCodeError("methodLvarRT: free Lvar " ^ Int.toString n')
            in
                freeRuntype i
            end
        else let val (_, rt) = Localenv.find (!localEnv) n'
                      handle Domain =>
                          genCodeError("methodLvarRT: Lvar " ^ Int.toString n')
             in
                 rt
             end
    end


fun compileTopLevel globalEnv exportEnv exprs closureEnv
    (jvmCode: jvm_code as {clinit, ...}) =
    let val {code, env, hdls} = !clinit
        val env'     = ref env   (* side-effected in compileExpr *)
        val hdls'    = ref hdls  (* side-effected in compileExpr *)
        val varIndex = identity
        val getLvar  = getTopLvar env'
        val lvarRT   = topLvarRT env'
        val sfOpt    = NONE
        val depth    = 0
        val compileExpr' =
            compileExpr globalEnv exportEnv closureEnv jvmCode hdls' varIndex
                getLvar lvarRT env' sfOpt compSharedGeneral depth

        fun h (expr, code') = compileExpr' expr code'
    in
        clinit := {code = List.foldl h code exprs,
                   env  = !env',
                   hdls = !hdls'}
    end  (* compileLambdas *)


fun compileClosures globalEnv exportEnv pendingClos closureEnv
    (jvmCode: jvm_code as {apply, ...}) =
    let val {code, env, entries, hdls} = !apply
        val env'     = ref env    (* side-effected in compileExpr *)
        val hdls'    = ref hdls   (* side-effected in compileExpr *)
        val varIndex = identity
        val sfOpt    = NONE

        fun compileClos ({tag, free, depth, body}, (C, lbls)) =
            let val argTypes = copy RTobject depth
                val freeRT   = freeRuntype closureEnv tag
                val getLvar  = getClosureLvar freeRT free env'
                val lvarRT   = closureLvarRT freeRT free env'
                val C' =
                    (env':= Localenv.bindRange (!env') 0 argTypes;
                     compileExpr globalEnv exportEnv closureEnv jvmCode hdls'
                         varIndex getLvar lvarRT env' sfOpt compSharedGeneral
                         depth body (Jreturn :: dropDead C))
                val (lbl, C'') = labelCode C'
            in
                (C'', (tag, lbl)::lbls)
            end

        val (code', entries') =
            List.foldl compileClos (code, entries) pendingClos
    in
        apply := {code    = code',
                  env     = !env',
                  entries = entries',
                  hdls    = !hdls'}
    end  (* compileClosures *)


fun compileMethods globalEnv methodEnv exportEnv pendingMeth closureEnv
    (jvmCode: jvm_code as {methods, ...}) =
    let fun compileMethod ({tag, free, nargs, body, depth}, ms) =
            let val argTypes = copy RTobject nargs
                val anyFree  = Freeenv.numItems free > 0
                val env      = Localenv.freshEnv (if anyFree then 1 else 0)
                val env'     = ref (Localenv.bindRange env 0 argTypes)
                               (* side-effected in compileExpr *)
                val hdls     = ref []
                val varOfs   = nargs - depth
                val varIndex = fn i => i + varOfs
                val freeRT   = freeRuntype closureEnv tag
                val getLvar  = getMethodLvar freeRT free varOfs env'
                val lvarRT   = methodLvarRT freeRT free varOfs env'
                val sfOpt    = NONE
                val depth'   = nargs
                val code =
                    (compileExpr globalEnv exportEnv closureEnv jvmCode hdls
                         varIndex getLvar lvarRT env' sfOpt compSharedSpecial
                         depth' body [Jreturn])
                val (name, export) =
                    (case Binarymap.peek(methodEnv, tag) of
                         NONE => (specMethodName tag, false)
                       | SOME res => res)
            in
                {name     = name,
                 argTypes = if anyFree then RTarray::argTypes
                            else argTypes,
                 resType  = RTobject,
                 export   = export,   (* only specialized closures declared
                                       * in the same module are invoked *)
                 code     = code,
                 env      = !env',
                 hdls     = !hdls} :: ms
            end
    in
         methods := List.foldl compileMethod (!methods) pendingMeth
    end  (* compileMethods *)
```

```
(* Instantiate.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

local
    open Const Jvmtype Bytecode Smlclasses Codeutil Runtype Coercion

    val dropTupleCast       = dropCast Tuple
    val dropConstructorCast = dropCast Constructor
    val dropClosureCast     = dropCast Closure

    (* Generate a new instance of the specified class, using compArgs
     * to generate code for the arguments, and invoking initialization
     * method init *)
    fun genInstance class =
        let val dropCast' = dropCast class
        in
            fn compArgs => fn init => fn C =>
                Jnew class :: Jdup ::
                compArgs (Jinvokespecial init :: dropCast' C)
        end

    fun genTarget' class compArgs init C =
        Jnew class :: Jdup ::
        compArgs (Jinvokespecial init :: dropClosureCast C)

    val genTuple'       = genInstance Tuple
    val genRef'         = genInstance Ref
    val genConstructor' = genInstance Constructor
    val genException'   = genInstance Exception

    val tagConst = intConst o Tag.toInt
in
    (* Generate a new Object instance *)
    fun genObject C =
        Jnew Object :: Jdup :: Jinvokespecial objectInit :: C

    (* Generate a new Integer object containing the specified value *)
    fun genIntAtom n C =
        let val (new, C') = coerce RTint RTnumber C
            val C'' =
                (case C' of
                     Ji2l :: C'' => Jlconst (Int64.fromInt n) :: C''
                   | _ => intConst n :: C')
        in
            new C''
        end

    (* Generate an atomic constant *)
    fun genAtom a C =
        (case a of
             INTscon n    => genIntAtom n C
           | WORDscon w   => let val (new, C') = coerce RTint RTnumber C
                             in
                                 new(wordConst w :: C')
                             end
           | CHARscon c   => genIntAtom (ord c) C
           | REALscon r   => let val (new, C') = coerce RTdouble RTnumber C
                             in
                                 new(realConst r :: C')
                             end
           | STRINGscon s => Jsconst s :: C)
```

```
(* Generate an Object array from a list of expressions *)
fun genArray _        [] C = Jaconst_null :: C
  | genArray compExpr es C =
    let fun h _ []      C' = C'
          | h k (e::er) C' =
            Jdup :: intConst k :: compExpr e (Jaastore :: h (k+1) er C')
    in
        intConst (List.length es) ::
        Jnewarray {elem = Tclass Object, dim = 1} :: h 0 es C
    end

(* Generate an empty Tuple object, a.k.a. () : unit *)
fun genUnit (Jpop :: C) = C
  | genUnit C = Jgetstatic tupleUnit :: dropTupleCast C

(* Generate a Tuple object with the specified arguments *)
fun genTuple compExpr args =
    genTuple' (fn C => genArray compExpr args C) tupleInit

(* Generate a Ref object with the specifed argument *)
fun genRef compExpr arg = genRef' (fn C => compExpr arg C) refInit

(* Generate a Constructor object with the specified tag, and with no
 * arguments *)
fun genConstructor0 tag C =
    let val C' = dropConstructorCast C
    in
        case tag of
            0 => Jgetstatic constructorZero :: C'
          | 1 => Jgetstatic constructorOne :: C'
          | 2 => Jgetstatic constructorTwo :: C'
          | 3 => Jgetstatic constructorThree :: C'
          | 4 => Jgetstatic constructorFour :: C'
          | 5 => Jgetstatic constructorFive :: C'
          | 6 => Jgetstatic constructorSix :: C'
          | 7 => Jgetstatic constructorSeven :: C'
          | 8 => Jgetstatic constructorEight :: C'
          | 9 => Jgetstatic constructorNine :: C'
          | _ => genConstructor' (fn C'' => intConst tag :: C'')
                                 constructorInit0 C
    end

fun genConstructor compExpr tag args =
    let fun compArgs C = intConst tag :: genArray compExpr args C
    in
        genConstructor' compArgs constructorInit
    end

(* Generate a new instance of the specified class (assumed to be a
 * subclass of Closure) with the specified tag, and with no free
 * variables *)
fun genTarget0 class =
    let val init : method_ref =
            {class = class,
             name  = initName,
             msig  = ([Tint], NONE)}
    in
        fn tag => genTarget' class (fn C => tagConst tag :: C) init
    end

(* Generate a new instance of the specified class (assumed to be a
 * subclass of Closure) with the specified tag and free variables *)
fun genTarget class =
    let val init : method_ref =
            {class = class,
             name  = initName,
                msig  = ([Tint, Tarray (Tclass Object)], NONE)}
        in
            fn compExpr => fn tag => fn free =>
                let fun compArgs C = tagConst tag :: genArray compExpr free C
                in
                    genTarget' class compArgs init
                end
        end

    (* Generate a new Exception object with the tag built by `tag'
     * code, and with no arguments *)
    fun genException0 uid C =
        let fun compArgs C = Jgetstatic (exnToFieldref RTobject uid) :: C
        in
            genException' compArgs exceptionInit0 C
        end

    (* Generate a new Exception object with the specified tag and
     * arguments *)
    fun genException compExpr uid args =
        let fun compArgs C = Jgetstatic (exnToFieldref RTobject uid) ::
                             genArray compExpr args C
        in
            genException' compArgs exceptionInit
        end

    fun throwGeneralExn id C =
        genException0 ({qual = "General", id = id}, 0) (Jathrow :: dropDead C)

    val throwDiv      = throwGeneralExn "Div"
    val throwOverflow = throwGeneralExn "Overflow"
end
```

```
(* Jvmcode.sml -- intermediate representation of target JVM code and class info
 *
 * Peter Bertelsen
 * December 1997
 *)

(* open Jvmtype Bytecode Label Localenv Classdecl Smlclasses Codeutil
 Instantiate Runtype *)


type name_info =
    {runType : Runtype.runtype,
     export  : bool}

type exn_info = Const.unique_id list Binaryset.set

type method_info =
    {name     : string,
     argTypes : Runtype.runtype list,
     resType  : Runtype.runtype,
     export   : bool,
     code     : Bytecode.jvm_instr list,
     env      : Localenv.local_env,
     hdls     : Classdecl.exn_hdl list}

type clinit_info =
    {code : Bytecode.jvm_instr list,  (* code for method clinit *)
     env  : Localenv.local_env,       (* local variable env *)
     hdls : Classdecl.exn_hdl list}   (* exception handlers *)

type apply_info =
    {code    : Bytecode.jvm_instr list,      (* code for method apply *)
     env     : Localenv.local_env,           (* local variable env *)
     entries : (Tag.tag * Label.label) list, (* closure tags and body labels *)
     hdls    : Classdecl.exn_hdl list}       (* exception handlers *)

type jvm_code =
    {class     : Jvmtype.jclass,            (* target class id *)
     names     : (string, name_info) Binarymap.dict ref,  (* top-level names *)
     exns      : Const.unique_id Binaryset.set ref,
                 (* static exception names *)
     methods   : method_info list ref,   (* specialized closures *)
     usesInit0 : bool ref,        (* is <init>(int) ever invoked? *)
     usesInit  : bool ref,        (* is <init>(int, Object[]) ever invoked? *)
     clinit    : clinit_info ref,
     apply     : apply_info ref
     }


(* Generate trailing code for a target class *)
fun codeTrailer class =
    {class     = class,
     names     = ref (Binarymap.mkDict String.compare),
     exns      = ref (Binaryset.empty Const.compareUids),
     methods   = ref [],
     usesInit0 = ref false,
     usesInit  = ref false,
     clinit    = ref {code = [Bytecode.Jreturn],
                      env  = Localenv.envStatic,
                      hdls = []},
     apply     = ref {code    = [],
                      env     = Localenv.envInstance,
                      entries = [],
                      hdls    = []}
     } : jvm_code
```

```
(* Lift.sig
 *
 * Peter Bertelsen
 * January 1998
 *)

type pending_closure =
    {tag   : Tag.tag,
     free  : Freeenv.free_env,
     depth : int,
     body  : Lambda.Lambda
     }

type pending_method =
    {tag    : Tag.tag,
     free   : Freeenv.free_env,
     nargs  : int,   (* number of curried arguments (method arguments) *)
     depth  : int,
     body   : Lambda.Lambda
     }

val liftLambdas : pending_closure list ref -> pending_method list ref ->
```

```
                  (bool * Lambda.Lambda) list * Lambda.Lambda list ->
                  Lambda.Lambda list
```

```
(* Lift.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

(* Translation of lambda-deBruijn to lambda-merged-stackdepth.
 * This requires giving all the Lvars unique ids within the enclosing
 * function (unique within their scope).  To this end we maintain an
 * rstack depth model, and the translation is simply:
 *
 *     deBruijn -> depth - deBruijn - 1
 *
 * All functions are lifted to top level simultaneously; as a result
 * of this, all free variables have negative ids.  A closure map is
 * built for use by compileFn, and each Lfn term is replaced with a
 * corresponding Pclosure primitive.
 *)

open Lambda Const Prim Codeutil Runtype Error

type pending_closure =
    {tag   : Tag.tag,
     free  : Freeenv.free_env,
     depth : int,
     body  : Lambda.Lambda
     }

type pending_method =
    {tag   : Tag.tag,
     free  : Freeenv.free_env,
     nargs : int,
     depth : int,
     body  : Lambda.Lambda
     }

fun liftLambdas pendingClos pendingMeth (lams, exprs) =
    let fun liftExpr free inCtx depth =

        (* NOTE: inCtx indicates whether the expression ‘appears in a
         * context’, that is, whether the run-time operand stack will
         * be non-empty prior to the evaluation of the expression.
         * This is significant in connection with exception handling:
         * throwing a JVM exception clears the operand stack of the
         * current method. *)

        let fun lift expr =
            (case expr of
                 Lvar n =>
                     let val id = depth - n - 1
                         val _  = if id < 0 then free:= Intset.add(!free, id)
                                  else ()
                     in
                         Lvar id
                     end
               | Lassign _ => fatalError "Lift.liftExpr: Lassign"
               | Lconst cst => expr
               | Lapply (body, args) =>
                     let fun h (Lfn body) = liftBody free depth true 1 body
                           | h a = liftExpr free true depth a
                     in
                         Lapply (lift body, List.map h args)
                     end
               (* NOTE: Lapply with k curried arguments is preserved,
                * rather than being translated to k single-argument
                * applications. *)
               | Lfn body => liftBody free depth false 1 body
               (* Optimize special case arising from #lab arg: *)
               | Llet ([arg], Lprim (p, [Lvar 0])) => Lprim (p, [lift arg])
               | Llet (args, body) =>
                     let fun h ea [] acc =
                             Llet (rev acc, liftExpr free inCtx ea body)
                           | h ea (expr' :: rest) acc =
                             let val acc' =
                                 ((liftExpr free inCtx ea expr') :: acc)
                             in
                                 h (ea + 1) rest acc'
                             end
                     in
                         h depth args []
                     end
               | Lletrec (args, body) =>
                     let val ea = depth + List.length args
                     in
                         Lletrec (List.map (liftExpr free inCtx ea) args,
                                  liftExpr free inCtx ea body)
                     end
               | Lprim (p, []) => expr
               | Lprim (p, e::er) =>
                     Lprim (p,
                           lift e :: List.map (liftExpr free true depth) er)
               | Lstatichandle (body, handler) =>
                     Lstatichandle (lift body, lift handler)
               | Lstaticfail => Lstaticfail
               | Lhandle (body, handler) =>
                     if inCtx then
                         Lapply (liftBody free depth false 0 expr, [])
                     else
                         Lhandle (lift body,
                                  liftExpr free inCtx (depth + 1) handler)
               | Lif (cond, ifSo, ifNot) =>
                     Lif (lift cond, lift ifSo, lift ifNot)
               | Lseq (expr1, expr2)     => Lseq (lift expr1, lift expr2)
               | Lwhile (cond, body)     => Lwhile (lift cond, lift body)
               | Landalso (expr1, expr2) => Landalso (lift expr1, lift expr2)
               | Lorelse (expr1, expr2)  => Lorelse (lift expr1, lift expr2)
               | Lcase (arg, clauses) =>
                     let fun h (tag, act) = (tag, lift act)
                     in
                         Lcase (lift arg, List.map h clauses)
                     end
               | Lswitch (size, arg, clauses) =>
                     let fun h (tag, act) = (tag, lift act)
                     in
                         Lswitch (size, lift arg, List.map h clauses)
                     end
               | Lunspec => Lunspec
               | Lshared (exprRef, lblRef as (ref TODO)) =>
                     (exprRef:= lift(!exprRef);
                      lblRef := LIFTED;
                      expr)
               | Lshared _ => expr)
        in
            lift
```

```
end

and liftBody free depth inArg nargs body =
    let val (tag, free', spec) = liftBody' nargs nargs body
        val spec' =
            if inArg then GENERAL
            else (registerMethod tag free' spec;
                  SPEC {freeTypes = copy RTobject (Freeenv.numItems
                                                     free'),
                        argTypes  = copy RTobject (#nargs spec),
                        resType   = RTobject})

        fun h (id, _, l) =
            let val n = depth + id
                val _ = if n < 0 then free:= Intset.add(!free, n)
                        else ()
            in
                Lvar n :: l
            end
    in
        Lprim (Pclosure (tag, spec'),
               Freeenv.foldr h [] free')
    end

and registerMethod tag free {nargs, depth, body} =
    let val method =
        {tag   = tag,
         free  = free,
         nargs = nargs,
         depth = depth,
         body  = body}
    in
        pendingMeth:= method :: (!pendingMeth)
    end

and liftBody' nargs depth body =
    let val tag = nextTag()
        val (free, body', method) =
            (case body of
                 Lfn body' =>
                     let val (tag', free, method) =
                             liftBody' (nargs+1) 1 body'

                         fun h (id, _, l) = Lvar (id+1) :: l
                     in
                         (Freeenv.shift free,
                          Lprim (Pclosure (tag', GENERAL),
                                 Freeenv.foldr h [] free),
                          method)
                     end
               | _ => let val free = ref Intset.empty
                                    (* side-effected in liftExpr *)
                          val body' = liftExpr free false depth body
                          val method =
                              {nargs = nargs,
                               depth = depth,
                               body  = body'}
                      in
                          (Freeenv.fromSet(!free), body', method)
                      end)
        val _ = if nargs > 0 then registerClosure tag free depth body'
                else ()
    in
        (tag, free, method)
    end
```

```
        and registerClosure tag free depth body =
            let val closure =
                {tag   = tag,
                 free  = free,
                 depth = depth,
                 body  = body}
            in
                pendingClos:= closure :: (!pendingClos)
            end

        fun liftLambda ((pure, expr), exprs) =   (* NOTE: pure is ignored *)
            let val free  = ref Intset.empty   (* side-effected in liftExpr *)
                val expr' = liftExpr free false 0 expr
            in
                if Intset.numItems(!free) = 0 then expr' :: exprs
                else fatalError "Lift.liftLambda: free variables at top-level"
        end

    in
        List.foldl liftLambda exprs lams
    end  (* liftLambdas *)
```

```
(* Localenv.sig -- applicative map from Lvar id to Localvar index and runtype
 *
 * Peter Bertelsen
 * December 1997
 *)

type local_env

val freshEnv    : int -> local_env   (* create local_env with specified number
                                      * of unreachable Localvar bindings *)
val envStatic   : local_env  (* fresh local_env for static method *)
val envInstance : local_env  (* fresh local_env for instance method *)
val bind        : local_env -> int -> Runtype.runtype ->
                  local_env * Localvar.index
val bindRange   : local_env -> int -> Runtype.runtype list -> local_env
val lookup      : local_env -> int -> (Localvar.index * Runtype.runtype) option
val find        : local_env -> int -> (Localvar.index * Runtype.runtype)
val numItems    : local_env -> int   (* number of bindings *)
val maxLocals   : local_env -> int   (* highest Localvar index used *)

(* NOTE: find raises Domain if the requested lvar # is not found in
 * the local_env.
 *)
```

```
(* Localenv.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

open Localvar Runtype Error

datatype local_env = ENV of (index * runtype) Intmap.intmap * locals

fun freshEnv n =
    let fun h 0 l = l
          | h k l =
            let val (l', _) = nextVar1 l
            in
                h (k-1) l'
            end
    in
        if n >= 0 then ENV (Intmap.empty(), h n freshLocals)
        else raise Domain
    end

val envStatic   = freshEnv 0
val envInstance = freshEnv 1

fun bind' (m, l) id rt =
    (case Intmap.peek(m, id) of
         NONE => let val (l', j) = nextVar1 l
                     val m' = Intmap.insert(m, id, (j, rt))
                 in
                     ((m', l'), j)
                 end
       | SOME (j, _) =>
             let val m' = Intmap.insert(m, id, (j, rt))
             in
                 ((m', l), j)
             end)

fun bind (ENV env) id rt =
    let val (env', j) = bind' env id rt
    in
        (ENV env', j)
    end

fun bindRange (ENV env) firstId rts =
    let fun h id [] env' = ENV env'
          | h id (rt::rest) env' =
            let val (env'', _) = bind' env' id rt
            in
                h (id+1) rest env''
            end
    in
        h firstId rts env
    end

fun lookup (ENV (m, _)) n = Intmap.peek(m, n)

fun find (ENV (m, _)) n = Intmap.retrieve(m, n)
    handle Intmap.NotFound => raise Domain

fun numItems (ENV (m, _)) = Intmap.numItems m

fun maxLocals (ENV (_, l)) = count l
```

```
(* Runtype.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

(* Run-time representation of SML values *)

datatype runtype =
    RTnumber | RTint | RTlong | RTdouble
  | RTbool
  | RTstring | RTchararray
  | RTvector | RTarray
  | RTblock  | RTtuple | RTconstructor | RTref | RTexception
  | RTclosure
  | RTmethod of Tag.tag * spec_clos
  | RTobject
and spec_clos =
    GENERAL
  | SPEC of {freeTypes : runtype list,  (* types of free variables *)
             argTypes  : runtype list,  (* types of arguments *)
             resType   : runtype}       (* result type *)

(* NOTE: GENERAL should not be used with RTmethod *)

fun specClosToString tag {freeTypes, argTypes, resType} =
    "RTmethod " ^ Tag.toString tag ^ " (" ^
    Int.toString(List.length freeTypes) ^ " free, " ^
    Int.toString(List.length argTypes) ^ " args)"


(* NOTE: the arity of a RTmethod-typed closure is List.length args *)

local
    open Jvmtype Smlclasses Error
in
    val toJvmType =
        (fn RTnumber      => Tclass Number
          | RTint         => Tint
          | RTlong        => Tlong
          | RTdouble      => Tdouble
          | RTbool        => Tboolean
          | RTstring      => Tclass String
          | RTchararray   => Tarray Tchar
          | RTvector      => Tclass Vector
          | RTarray       => Tarray (Tclass Object)
          | RTblock       => Tclass Block
          | RTtuple       => Tclass Tuple
          | RTconstructor => Tclass Constructor
          | RTref         => Tclass Ref
          | RTexception   => Tclass Exception
          | RTclosure     => Tclass Closure
          | RTmethod _    => Tclass Closure
          | RTobject      => Tclass Object)

    fun toMethodSig argTypes resType =
        (List.map toJvmType argTypes, SOME (toJvmType resType)) : method_sig
end
```

```
(* Smlclasses.sig
 *
 * Peter Bertelsen
 * December 1997
 *)

val clinitName : string
val initName   : string

(* from package java.lang *)

val Object       : Jvmtype.jclass
val objectInit   : Bytecode.method_ref
val objectEquals : Bytecode.method_ref

val Number      : Jvmtype.jclass
val intValue    : Bytecode.method_ref
val longValue   : Bytecode.method_ref
val doubleValue : Bytecode.method_ref

val Integer     : Jvmtype.jclass
val integerInit : Bytecode.method_ref

val Long     : Jvmtype.jclass
val longInit : Bytecode.method_ref

val Double     : Jvmtype.jclass
val doubleInit : Bytecode.method_ref

val String            : Jvmtype.jclass
val stringInit        : Bytecode.method_ref
val stringLength      : Bytecode.method_ref
val stringCharAt      : Bytecode.method_ref
val stringCompareTo   : Bytecode.method_ref
val stringToCharArray : Bytecode.method_ref

val ArithmeticException : Jvmtype.jclass


(* from package sml.lang *)

val Block     : Jvmtype.jclass
val blockArgs : Bytecode.field_ref

val Tuple     : Jvmtype.jclass
val tupleUnit : Bytecode.field_ref
val tupleInit : Bytecode.method_ref

val Ref     : Jvmtype.jclass
val refInit : Bytecode.method_ref

val Constructor      : Jvmtype.jclass
val constructorZero  : Bytecode.field_ref
val constructorOne   : Bytecode.field_ref
val constructorTwo   : Bytecode.field_ref
val constructorThree : Bytecode.field_ref
val constructorFour  : Bytecode.field_ref
val constructorFive  : Bytecode.field_ref
val constructorSix   : Bytecode.field_ref
val constructorSeven : Bytecode.field_ref
val constructorEight : Bytecode.field_ref
val constructorNine  : Bytecode.field_ref
val constructorTag   : Bytecode.field_ref
val constructorInit0 : Bytecode.method_ref
val constructorInit  : Bytecode.method_ref
val Closure      : Jvmtype.jclass
val closureTag   : Bytecode.field_ref
val closureFree  : Bytecode.field_ref
val closureInit  : Bytecode.method_ref
val closureApply : Bytecode.method_ref

val Exception      : Jvmtype.jclass
val exceptionTag   : Bytecode.field_ref
val exceptionInit0 : Bytecode.method_ref
val exceptionInit  : Bytecode.method_ref

val General          : Jvmtype.jclass
val generalException : Bytecode.field_ref

val Vector      : Jvmtype.jclass
val vectorElems : Bytecode.field_ref
val vectorInit  : Bytecode.method_ref

val SmlError     : Jvmtype.jclass
val smlErrorInit : Bytecode.method_ref

val Util          : Jvmtype.jclass
val utilMakeArray : Bytecode.method_ref
```

```
(* Tag.sig
 *
 * Peter Bertelsen
 * December 1997
 *)

eqtype tag
type tags

val firstTag : tag
val fromInt  : int -> tag
val toInt    : tag -> int
val toString : tag -> string
val compare  : tag * tag -> order

val freshTags : tags
val nextTag   : tags -> tags * tag
val lastTag   : tags -> tag
```

```
(* Tag.sml
 *
 * Peter Bertelsen
 * December 1997
 *)

datatype tag  = TAG of int
datatype tags = TAGS of int

val firstTag = TAG 0

fun fromInt n = TAG n

fun toInt (TAG n) = n
fun toString (TAG n) = Int.toString n

fun compare (TAG m, TAG n) = Int.compare(m, n)

val freshTags = TAGS 0

fun nextTag (TAGS n) = (TAGS (n+1), TAG n)

fun lastTag (TAGS n) = TAG n
```

```
(* Compiler.sml  --  adopted from Moscow ML version 1.42
 *
 * Modified by Peter Bertelsen
 * December 1997
 *)

open Obj Mixture Const Lambda Prim Globals Location Units Types Smlperv Asynt
     Parser Ovlres Infixres Elab Sigmtch Tr_env Front Jvmcode Jvmtype Bytecode
     Smlclasses Codeutil Runtype Gencode Genclass Error


val showLambda = true   (* flag: show Lambda code before and after lifting? *)


(* Lexer of stream *)

fun createLexerStream (is : BasicIO.instream) =
  Lexing.createLexer (fn buff => fn n => Nonstdio.buff_input is buff 0 n)


(* Parsing functions *)

fun parsePhrase parsingFun lexingFun lexbuf =
    let fun skip () =
        (case lexingFun lexbuf of
             EOF       => ()
           | SEMICOLON => ()
           | _         => skip())
             handle LexicalError (_,_,_) => skip()
    in
        (parsingFun lexingFun lexbuf)
        handle Parsing.ParseError f =>
                   let val pos1 = Lexing.getLexemeStart lexbuf
                       val pos2 = Lexing.getLexemeEnd lexbuf
                   in
                       Lexer.resetLexerState();
                       if f(Obj.repr EOF) orelse f(Obj.repr SEMICOLON) then ()
                       else skip();
                       msgIBlock 0;
                       errLocation(Loc(pos1, pos2));
                       errPrompt "Syntax error.";
                       msgEOL();
                       msgEBlock();
                       raise Toplevel
                   end
              | LexicalError(msg, pos1, pos2) =>
                   (msgIBlock 0;
                    if pos1 >= 0 andalso pos2 >= 0 then
                        errLocation (Loc(pos1, pos2))
                    else ();
                    errPrompt "Lexical error: "; msgString msg;
                    msgString "."; msgEOL();
                    msgEBlock();
                    skip();
                    raise Toplevel)
             | Toplevel =>
                   (skip ();
                    raise Toplevel)
    end

fun parsePhraseAndClear parsingFun lexingFun lexbuf =
    let val phr = (parsePhrase parsingFun lexingFun lexbuf)
                  handle x => (Lexer.resetLexerState();
                               Parsing.clearParser();
                               raise x)
    in
        Lexer.resetLexerState();
        Parsing.clearParser();
        phr
    end

val parseToplevelPhrase = parsePhraseAndClear Parser.ToplevelPhrase Lexer.Token
val parseStructFile     = parsePhraseAndClear Parser.StructFile Lexer.Token
val parseSigFile        = parsePhraseAndClear Parser.SigFile Lexer.Token

fun cleanEnvAcc [] acc = acc
  | cleanEnvAcc ((k, v) :: rest) acc =
        if List.exists (fn (k', _) => k = k') acc then
            cleanEnvAcc rest acc
        else
            cleanEnvAcc rest ((k, v) :: acc)

fun cleanEnv env =
        cleanEnvAcc (foldEnv (fn a => fn x => fn acc => (a,x)::acc) [] env) []

(* Reporting the results of compiling a phrase *)

val verbose = ref false

fun reportFixityResult (id, status) =
        ((case status of
              NONFIXst   => msgString "nonfix "
            | INFIXst i  => (msgString "infix ";
                             msgInt i; msgString " ")
            | INFIXRst i => (msgString "infixr ";
                             msgInt i; msgString " "));
         msgString id)

fun reportEquOfType equ =
        msgString (case equ of
                       FALSEequ => ""
                     | TRUEequ  => "eq"
                     | REFequ   => "prim_EQ")

fun reportLhsOfTypeResult (tyname : TyName) =
    let val vs  = newTypeVars (#tnArity (!(#info tyname)))
        val lhs = type_con (map TypeOfTypeVar vs) tyname
    in
        printType lhs
    end

fun reportTypeResult (tyname : TyName) =
    let val {qualid, info} = tyname
        val {tnEqu, tnStr, ...} = !info
    in
        case tnStr of
```

```
            NILts =>
                (reportEquOfType tnEqu;
                 msgString "type ";
                 reportLhsOfTypeResult tyname)
          | TYPEts(vs, tau) =>
                let val lhs = type_con (map TypeOfTypeVar vs) tyname
                in
                    msgString "type ";
                    resetTypePrinter();
                    collectExplicitVars lhs;
                    collectExplicitVars tau;
                    printNextType lhs; msgString " =";
                    msgBreak(1, 2);
                    printNextType tau;
                    resetTypePrinter()
                end
          | DATATYPEts dt =>
                let val uname = #qual qualid
                    val sign = if uname = currentUnitName() then (!currentSig)
                               else findSig Location.nilLocation uname
                    val CE = findConstructors sign dt
                in
                    if null CE then
                        (msgString "abstype ";
                         reportLhsOfTypeResult tyname)
                    else
                        (msgString "datatype ";
                         reportLhsOfTypeResult tyname)
                end
          | REAts _ => fatalError "reportTypeResult"
    end

fun lookupNewCBas cBas id =
        (lookupEnv cBas id : ConStatus)
        handle Subscript => fatalError "lookupNewCBas"

fun reportCompResults iBas cBas static_VE static_TE =
        (app (fn x =>
              (msgIBlock 0; reportFixityResult x; msgEOL(); msgEBlock()))
             (cleanEnv iBas);
         app (fn (id, tn) =>
              (msgIBlock 0; reportTypeResult tn; msgEOL(); msgEBlock()))
             (cleanEnv static_TE);
         app (fn (id, sch) =>
              let val status = lookupNewCBas cBas id
              in
                  msgIBlock 0;
                  msgCBlock 0;
                  msgString (case #info status of
                                 VARname  _ => "val "
                               | PRIMname _ => "val "
                               | CONname  _ => "con "
                               | EXNname  _ => "exn "
                               | REFname    => "con ");
                  msgString id;
                  msgString " :";
                  msgBreak(1, 2);
                  printScheme sch;
                  msgEBlock();
                  msgEOL();
                  msgEBlock()
              end)
         (cleanEnv static_VE);
         msgFlush())
```

```
(* To write the signature of the unit currently compiled *)
(* The same value has to be written twice, because it's unclear *)
(* how to ‘open‘ a file in "read/write" mode in a Caml Light program. *)

fun writeCompiledSignature filename_ui =
    let val sigStamp = ref dummySigStamp
        val sigLen   = ref 0
    in
        let val os = BasicIO.open_out_bin filename_ui
        in
            (Nonstdio.output_value os (!currentSig);
             sigLen:= Nonstdio.pos_out os;
             BasicIO.close_out os)
            handle x =>
                (BasicIO.close_out os;
                 FileSys.remove filename_ui;
                 raise x)
        end;
        let val is = BasicIO.open_in_bin filename_ui
        in
            let val sigImage = BasicIO.input(is, !sigLen)
                prim_val md5sum_ : string -> string = 1 "md5sum"
            in
                if size sigImage < !sigLen then raise Size
                else ();
                BasicIO.close_in is;
                FileSys.remove filename_ui;
                sigStamp := md5sum_ sigImage
            end
            handle x => (BasicIO.close_in is;
                         FileSys.remove filename_ui;
                         raise x)
        end;
        let val os = BasicIO.open_out_bin filename_ui
        in
            (BasicIO.output(os, !sigStamp);
             Nonstdio.output_value os (!currentSig);
             BasicIO.close_out os)
            handle x => (BasicIO.close_out os;
                         FileSys.remove filename_ui;
                         raise x)
        end;
        !sigStamp
    end


(* Checks and error messages for compiling units *)

fun checkUnitId msg (locid as (loc, id)) uname =
    if id = uname then ()
    else (msgIBlock 0;
          errLocation loc;
          errPrompt "Error: ";
          msgString msg;
          msgString " name and file name are incompatible";
          msgEOL();
          msgEBlock();
          raise Toplevel)

fun fileExists s = FileSys.access(s, [])

fun rmFileIfExists s = if fileExists s then FileSys.remove s
                       else ()

fun checkExists filename_ui filename_sig filename_sml =
    if fileExists filename_ui then ()
    else (msgIBlock 0;
          errPrompt "File ";
          msgString filename_sig;
          msgString " must be compiled before ";
          msgString filename_sml;
          msgEOL();
          msgEBlock();
          raise Toplevel)

fun checkNotExists filename_sig filename_sml =
    if fileExists filename_sig then
        (msgIBlock 0;
         errPrompt "File ";
         msgString filename_sig;
         msgString " exists, but there is no signature constraint in ";
         msgString filename_sml;
         msgEOL();
         msgEBlock();
         raise Toplevel)
    else ()


(* Compiling a signature *)

fun compileSpecPhrase spec =
    let val (iBas, cBas) = resolveToplevelSpec spec
        val (VE, TE)     = elabToplevelSpec spec
    in
        updateCurrentInfixBasis iBas;
        extendCurrentConBasis cBas;
        extendCurrentStaticTE TE;
        updateCurrentStaticVE VE;
        if !verbose then
            (reportCompResults iBas cBas VE TE;
             msgFlush())
        else ()
    end

fun compileSignature uname filename =
    let val filename_sig = filename ^ ".sig"
        val filename_ui  = filename ^ ".ui"
     (*        val _ = (msgIBlock 0;
                 msgString "[compiling file \"";
                 msgString filename_sig;
                 msgString "\"]";
                 msgEOL();
                 msgEBlock();) *)
        val _ = startCompilingUnit uname
        val _ = initInitialEnvironments()
        val _ = rmFileIfExists filename_ui;

        val is     = BasicIO.open_in_bin filename_sig
        val lexBuf = createLexerStream is

        fun compileSig (AnonSig specs) = app compileSpecPhrase specs
          | compileSig (NamedSig{locsigid, specs}) =
                (checkUnitId "signature" locsigid uname;
                 app compileSpecPhrase specs)
    in
        input_name  := filename_sig;
        input_stream:= is;
        input_lexbuf:= lexBuf;
        (compileSig(parseSigFile lexBuf);
         ignore(rectifySignature());
         ignore(writeCompiledSignature filename_ui);
         BasicIO.close_in is)
        handle x => (BasicIO.close_in is; raise x)
    end


(* Compiling an implementation *)

fun updateCurrentCompState ((iBas, cBas, VE, TE), RE) =
        (updateCurrentInfixBasis iBas;
         updateCurrentConBasis cBas;
         updateCurrentStaticTE TE;
         updateCurrentStaticVE VE;
         updateCurrentRenEnv RE;
         if !verbose then (reportCompResults iBas cBas VE TE;
                           msgFlush())
         else ())

fun isInTable key tbl =
    (case Hasht.peek tbl key of
         NONE   => false
       | SOME _ => true)

fun filterExnRenList exnRen cBas =
    List.filter (fn ({qual, id}, _) => isInTable id cBas) exnRen

fun filterValRenList valRen cBas =
    List.filter (fn (id, stamp) => isInTable id cBas) valRen

fun checkSig sigOpt uname filename_ui lamss (jvmCode : jvm_code) =
    let val {class, names, exns, ...} = jvmCode
        val (exnRen, valRen) = rectifySignature()
        val (exnRen', valRen', lamss') =
            (case sigOpt of
                 NONE => (writeCompiledSignature filename_ui;
                          (exnRen, valRen, lamss))
               | SOME (specSig as {uConBasis, uStamp, ...}) =>
                     (filterExnRenList exnRen uConBasis,
                      filterValRenList valRen uConBasis,
                      matchSignature (!currentSig) specSig :: lamss))

        fun registerExn ((qid, uid as ({qual, id}, n)),
                         NE as (names', exns')) =
            if qual = className class then
                let val name = exnToFieldName id n
                    val nameInfo =
                        {runType = RTobject,
                         export  = true}
                in
                    (Binarymap.insert(names', name, nameInfo),
                     Binaryset.add(exns', uid))
                end
            else NE

        fun exportVal ((id, n), exportEnv) =
            let val qid = {qual = uname, id = id}
            in
                Binarymap.insert(exportEnv, (qid, n), (qid, 0))
            end

        val (names', exns') = List.foldl registerExn (!names, !exns) exnRen
        val exportEnv =
            List.foldl exportVal (Binarymap.mkDict compareUids) valRen
    in
        names:= names';
        exns := exns';
        (exportEnv, lamss')
    end

fun buildGlobalEnvs exportEnv lams =
    let val globalEnv = Binarymap.mkDict compareUids   (* uid -> runtype *)
        val methodEnv = Binarymap.mkDict Tag.compare   (* tag -> uid *)

        fun h (Lprim (Pset_global uid, [arg]), (GE, ME)) =
            let val (uid', export) =
                (case Binarymap.peek(exportEnv, uid) of
                     NONE      => (uid, false)
                   | SOME uid' => (uid', true))
                val ({qual, id}, n) = uid'
                val name = toFieldName id n
            in
                case arg of
                    Lprim (Pclosure (tag, spec as SPEC _), _) =>
                        (Binarymap.insert(GE, uid', RTmethod (tag, spec)),
                         Binarymap.insert(ME, tag, (name, export)))
                  | Lprim (Pclosure _, _) =>
                        (Binarymap.insert(GE, uid', RTclosure),
                         ME)
                  | _ => (Binarymap.insert(GE, uid', RTobject),
                          ME)
            end
          | h (_ , GE_ME) = GE_ME

        (* NOTE: it is assumed that all global names in the lifted
         * code are bound in this way: Lprim (Pset_global uid, [arg]).
         *)
    in
        List.foldl h (globalEnv, methodEnv) lams
    end

fun printExportEnv (exportEnv : export_env) =
    let fun h (uid, uid') = print(uidToString uid ^ " as " ^
                                  uidToString uid' ^ "\n")
    in
        if Binarymap.numItems exportEnv = 0 then ()
        else (print "Exporting:\n";
              Binarymap.app h exportEnv)
    end

fun printLamsList lamss =
    let fun h (lams, _) =
        List.app (fn (_, lam) => (Pr_lam.printLam lam; print "\n")) lams
    in
        List.foldr h () lamss
    end

fun printLiftedLams lams =
    let fun h (lam, _) = (Pr_lam.printLam lam; print "\n")
    in
        List.foldr h () lams
    end

fun printFree free =
    Freeenv.app (fn (i, _) => print(" " ^ Int.toString i)) free

fun printClosures pendingClos =
    let fun h {tag, free, depth, body} =
        (print("--- closure " ^ Tag.toString tag ^
               ", depth " ^ Int.toString depth ^ ", [");
         printFree free;
         print("] free ---\n");
         Pr_lam.printLam body;
         print "\n")
    in
        List.app h pendingClos
    end

fun printMethods pendingMeth =
    let fun h {tag, free, nargs, depth, body} =
        (print("--- method " ^ Tag.toString tag ^
               ", depth " ^ Int.toString depth ^ ", [");
         printFree free;
         print("] free, " ^ Int.toString nargs ^ " args ---\n");
         Pr_lam.printLam body;
         print "\n")
    in
        List.app h pendingMeth
    end

fun compileAndEmit uname filename sigOpt decs =
    let val filename_sml   = filename ^ ".sml"
        val filename_ui    = filename ^ ".ui"
        val filename_class = filename ^ ".class"
        val targetClass    = Jvmtype.class{pkgs = [], name = filename}

        val _ = startCompilingUnit uname
        val _ = initInitialEnvironments()
        val _ = resetCodeGen()

        val pendingClos = ref []  (* side-effected in Lift.liftLambda *)
        val pendingMeth = ref []  (* side-effected in Lift.liftLambda *)

        fun compilePhrase (dec, lamss) =
            let val (iBas, cBas, dec') = resolveToplevelDec dec
                val (VE, TE)           = elabToplevelDec dec'
                val _                  = resolveOvlDec dec'
                val (RE, lams)         = translateToplevelDec dec'
            in
                updateCurrentCompState((iBas, cBas, VE, TE), RE);
                lams :: lamss
            end

        val (topLams, exportEnv, jvmCode) =
            (* NOTE: lamsList and lamsList' are not used after
             * lifting; by limiting the scope of these to the
             * let-binding below, we give GC a chance to get rid of
             * the original Lambdas *)
            let (* Build list of lists of Lambdas; each list of Lambdas
                 * corresponds to a top-level declaration *)
                val lamsList = List.foldl compilePhrase [] decs

                (* Create stub JVM code; side-effected below *)
                val jvmCode = codeTrailer targetClass

                (* Prepare ‘exports’ from the module *)
                val (exportEnv : export_env , lamsList') =
                    checkSig sigOpt uname filename_ui lamsList jvmCode

                (* val _ = printExportEnv exportEnv *)

                val _ =
                    if showLambda then
                        (print "===== before lifting =====\n";
                         printLamsList lamsList')
                    else ()

                (* Lift the Lambdas, creating a list of lifted Lambdas in
                 * reversed order (compared to the original sequence of
                 * top-level declarations) *)
                val topLams =
                    List.foldr (Lift.liftLambdas pendingClos pendingMeth)
                               [] lamsList'
            in
                if showLambda then
                    (print "===== after lifting =====\n";
                     printLiftedLams topLams;
                     printClosures(!pendingClos);
                     printMethods(!pendingMeth))
                else ();
                (topLams, exportEnv, jvmCode)
            end

        val (globalEnv : global_env, methodEnv : method_env) =
            buildGlobalEnvs exportEnv topLams

        val closureEnv : closure_env ref = ref (Binarymap.mkDict Tag.compare)

         (* closureEnv is a map from a closure tag to a free variable
          * runtype map; the latter maps a free variable index to its
          * runtype; closureEnv is side-effected in compileTopLevel,
          * compileClosures, and compileMethods *)

    in
        (* Generate code backwards for the top-level Lambdas *)
        compileTopLevel globalEnv exportEnv topLams closureEnv jvmCode;

        (* Generate code for pending closures and methods *)
        compileClosures globalEnv exportEnv (!pendingClos) closureEnv jvmCode;
        compileMethods globalEnv methodEnv exportEnv (!pendingMeth)
            closureEnv jvmCode;

        (* Complete the generated code and build the target class *)
        genClass filename_sml filename_class jvmCode
    end

fun compileUnitBody uname filename =
    let val filename_sig = filename ^ ".sig"
        val filename_ui  = filename ^ ".ui"
        val filename_sml = filename ^ ".sml"
        val is           = BasicIO.open_in_bin filename_sml
        val lexbuf       = createLexerStream is

        fun compileStruct (AnonStruct decs) =
            if file_exists filename_sig then
                (hasSpecifiedSignature:= true;
                 checkExists filename_ui filename_sig filename_sml;
                 compileAndEmit uname filename (SOME (readSig uname)) decs)
            else
                (hasSpecifiedSignature:= false;
                 rmFileIfExists filename_ui;
                 compileAndEmit uname filename NONE decs)
          | compileStruct (NamedStruct{locstrid, locsigid = NONE, decs}) =
                (checkUnitId "structure" locstrid uname;
                 checkNotExists filename_sig filename_sml;
                 hasSpecifiedSignature:= false;
                 rmFileIfExists filename_ui;
                 compileAndEmit uname filename NONE decs)
          | compileStruct (NamedStruct _) = fatalError "compileUnitBody"
          | compileStruct (Abstraction{locstrid, locsigid, decs}) =
                (checkUnitId "structure" locstrid uname;
                 checkUnitId "signature" locsigid uname;
                 checkExists filename_ui filename_sig filename_sml;
                 hasSpecifiedSignature:= true;
                 compileAndEmit uname filename (SOME (readSig uname)) decs)
    in
        input_name  := filename_sml;
        input_stream:= is;
        input_lexbuf:= lexbuf;
        (compileStruct (parseStructFile lexbuf))
        handle x => (BasicIO.close_in is;
                     raise x)
    end
```

```
(* Sigmtch.sml  --  adopted from Moscow ML version 1.42
 *
 * Modified by Peter Bertelsen
 * September 1997
 *)

open List Fnlib Mixture Const Prim Lambda Globals Units Types Front

(* Signature matching *)

fun lookupSig_TyEnv (sign : CSig) id =
  Hasht.find (#uTyEnv sign) id
  handle Subscript =>
    (msgIBlock 0;
     errPrompt "Type "; msgString id;
     msgString
       " is specified in the signature but not defined in the unit body";
     msgEOL();
     msgEBlock();
     raise Toplevel)


fun lookupSig_VarEnv (sign : CSig) id =
  Hasht.find (#uVarEnv sign) id
  handle Subscript =>
    (msgIBlock 0;
     errPrompt "Value "; msgString id;
     msgString
       " is specified in the signature but not defined in the unit body";
     msgEOL();
     msgEBlock();
     raise Toplevel)


fun lookupSig_cBas (sign : CSig) id =
  Hasht.find (#uConBasis sign) id
  handle Subscript =>
    (msgIBlock 0;
     errPrompt "Value "; msgString id;
     msgString
       " is specified in the signature but not defined in the unit body";
     msgEOL();
     msgEBlock();
     raise Toplevel)


fun errorImplMismatch id =
(
  msgIBlock 0;
  errPrompt "Mismatch between the specification of the value ";
  msgString id; msgEOL();
  errPrompt "in the signature and its implementation in the unit body";
  msgEOL();
  msgEBlock();
  raise Toplevel
)

fun errorConImplMismatch id =
(
  msgIBlock 0;
  errPrompt "Mismatch between the specification of the value constructor ";
  msgString id; msgEOL();
  errPrompt "in the signature and its implementation in the unit body";
  msgEOL();
  msgEBlock();
  raise Toplevel
)

fun errorExConImplMismatch id =
(
  msgIBlock 0;
  errPrompt "Mismatch between the specification of the exception constructor ";
  msgString id; msgEOL();
  errPrompt "in the signature and its implementation in the unit body";
  msgEOL();
  msgEBlock();
  raise Toplevel
)

fun mkTypeFcnOfTyName (tn : TyName) =
  let val vs = newTypeVars (#tnArity (!(#info tn))) in
    TYPEts(vs, type_con (map TypeOfTypeVar vs) tn)
  end


fun applyRea (tyname : TyName) ts =
  case #tnStr(!(#info tyname)) of
      NILts => type_con ts tyname
    | TYPEts(pars, body) => fatalError "applyRea"
    | DATATYPEts _ => type_con ts tyname
    | REAts tn =>
        let val arity = List.length ts
            val {info=ref{tnArity, tnStr, ...}, ...} = tn
        in
          if tnArity <> arity then
            fatalError "applyRea"
          else ();
          case tnStr of
              NILts => type_con ts tn
            | TYPEts(pars, body) =>
                type_subst (zip2 pars ts) body
            | DATATYPEts _ => type_con ts tn
            | REAts _ => fatalError "applyRea"
        end


fun expandRea UE tau =
  case normType tau of
    VARt var =>
      (lookup var UE
       handle Subscript => fatalError "expandRea: Unknown variable")
  | ARROWt(t,t') =>
      ARROWt(expandRea UE t, expandRea UE t')
  | CONt(ts, tn) =>
      applyRea tn (map (expandRea UE) ts)
  | RECt rt =>
      let val {fields=fs, rho=rho} = !rt in
        RECt (ref{fields=map_fields (expandRea UE) fs, rho=rho})
      end


fun newParTypeVar () =
  mkTypeVar false false false 0

fun newHardTypeVar () =
  let val tv = mkTypeVar false false false 0 in
    setTvKind tv (Explicit "");
    tv
  end

fun isTypeFcnEqu vs' tau' vs tau =
  let val ts = map (fn _ => TypeOfTypeVar(newHardTypeVar())) vs
      val UE = zip2 vs ts
      val tau0 = expandRea UE tau
      val UE' = zip2 vs' ts
      val tau'0 = type_subst UE' tau'
  in
    (unify tau'0 tau0; true)
    handle Unify _ => false
  end


fun matchDatatype (tyname : TyName) (CE : ConEnv) (CE' : ConEnv) =
  let val domCE  = map (fn gci => #id(#qualid gci)) CE
      val domCE' = map (fn gci => #id(#qualid gci)) CE'
  in
    (* domCE' is non-empty, because 'abstype' is not allowed *)
    (* in signatures, and "primitive" types are represented  *)
    (* as NILts. *)
    if domCE <> domCE' then (
      msgIBlock 0;
      errPrompt "Realization mismatch: variant type constructor ";
      msgString (#id (#qualid tyname)); msgEOL();
      errPrompt "has specification and realization that differ"; msgEOL();
      errPrompt "in the names and/or the order of value constructors";
      msgEOL();
      msgEBlock();
      raise Toplevel)
    else ();
    (* We don't have to compare the types of constructors here, *)
    (* because they will be compared as values. Note that all *)
    (* constructors are visible, for redefining values in signatures *)
    (* is not allowed. *)
    ()
  end

fun refresh0HardTypeVar (var : TypeVar) =
  let val {tvEqu, tvImp, ...} = !var
      val tv = mkTypeVar tvEqu tvImp false 0
  in
    setTvKind tv (Explicit "");
    tv
  end

fun refresh0TypeVar (var : TypeVar) =
  let val {tvEqu, tvImp, ...} = !var
  in
    mkTypeVar tvEqu tvImp false 0
  end

fun matchStamps (inferredSig : CSig) (specSig : CSig) =
  Hasht.apply
    (fn uname => fn stamp =>
      let val stamp' = Hasht.find (#uMentions inferredSig) uname in
        if stamp' <> stamp then (
          msgIBlock 0;
          errPrompt "The signature of "; msgString uname;
          msgString " has changed, while "; msgString (#uName specSig);
          msgString ".sig depends on it."; msgEOL();
          errPrompt "Please, recompile "; msgString (#uName specSig);
          msgString ".sig, before compiling "; msgString (#uName specSig);
          msgString ".sml."; msgEOL();
          msgEBlock();
          raise Toplevel)
        else ()
      end
      handle Subscript => ())
    (#uMentions specSig)

fun realizeTyName (infTyName : TyName) (specTyName : TyName) =
  let val {info=ref infInfo, ...} = infTyName
      val {info=ref specInfo, qualid={id, ...}} = specTyName
  in
    if #tnArity specInfo  <> #tnArity infInfo then (
      msgIBlock 0;
      errPrompt "Arity mismatch: type constructor ";
      msgString id; msgString " is specified as having arity ";
      msgInt (#tnArity specInfo); msgEOL();
      errPrompt "but declared as having arity ";
      msgInt (#tnArity infInfo); msgString " in the unit's body";
      msgEOL();
      msgEBlock();
      raise Toplevel)
    else ();
    case #tnEqu specInfo of
        REFequ =>
          if #tnEqu infInfo <> REFequ then (
            msgIBlock 0;
            errPrompt "Type constructor "; msgString id;
            msgString " is specified as `prim_EQtype',";
            msgEOL();
            errPrompt "but isn't realized as a `prim_EQtype'";
            msgEOL();
            msgEBlock();
            raise Toplevel)
          else ()
      | TRUEequ =>
          if #tnEqu infInfo = FALSEequ then (
            msgIBlock 0;
            errPrompt "Type constructor "; msgString id;
            msgString " is specified as admitting equality,";
            msgEOL();
            errPrompt "but its realization doesn't admit equality";
            msgEOL();
            msgEBlock();
            raise Toplevel)
          else ()
      | FALSEequ =>
          ();
    case #tnStr specInfo of
        NILts => setTnStr (#info specTyName) (REAts infTyName)
      | TYPEts _ => ()
      | DATATYPEts _ => ()
      | REAts _ => fatalError "realizeTyName"
  end

fun checkRealization (inferredSig : CSig) (specSig : CSig)
                     (infTyName : TyName) (specTyName : TyName) =
  let val {info=ref infInfo, ...} = infTyName
      val {info=ref specInfo, qualid={id, ...}} = specTyName
  in
    case #tnStr specInfo of
        NILts => fatalError "checkRealization"
      | TYPEts(vs, tau) =>
          (case #tnStr infInfo of
               NILts =>
                 (msgIBlock 0;
                  errPrompt "Realization mismatch: type constructor ";
                  msgString id; msgString " is specified"; msgEOL();
                  errPrompt "as a type abbreviation,"; msgEOL();
                  errPrompt "but implemented as a primitive type"; msgEOL();
                  msgEBlock();
                  raise Toplevel)
             | TYPEts(vs', tau') =>
                 if not(isTypeFcnEqu vs' tau' vs tau) then (
                   msgIBlock 0;
                   errPrompt "Realization mismatch: type constructor ";
                   msgString id; msgString " is bound"; msgEOL();
                   errPrompt "to non-equivalent type abbreviations"; msgEOL();
                   errPrompt "in the signature and in the unit body"; msgEOL();
                   msgEBlock();
                   raise Toplevel)
                 else ()
             | DATATYPEts _ =>
                 (msgIBlock 0;
                  errPrompt "Realization mismatch: type constructor ";
                  msgString id; msgString " is specified"; msgEOL();
                  errPrompt "as a type abbreviation,"; msgEOL();
                  errPrompt "but implemented as a variant type"; msgEOL();
                  msgEBlock();
                  raise Toplevel)
             | REAts tn' => fatalError "checkRealization")
      | DATATYPEts dt =>
          let val CE = findConstructors specSig dt in
            case #tnStr infInfo of
                NILts =>
                  (msgIBlock 0;
                   errPrompt "Realization mismatch: type constructor ";
                   msgString id;
                   msgString " is specified as a variant type,"; msgEOL();
                   errPrompt "but implemented as a primitive type"; msgEOL();
                   msgEBlock();
                   raise Toplevel)
              | TYPEts(vs', tau') =>
                  (msgIBlock 0;
                   errPrompt "Realization mismatch: type constructor ";
                   msgString id;
                   msgString " is specified as a variant type,"; msgEOL();
                   errPrompt "but implemented as a type abbreviation"; msgEOL();
                   msgEBlock();
                   raise Toplevel)
              | DATATYPEts dt' =>
                  let val CE' = findConstructors inferredSig dt'
                  in matchDatatype specTyName CE CE' end
              | REAts tn' => fatalError "checkRealization"
          end
      | REAts _ => ()
  end


fun matchTypeSchemes id infSc specSc =
  let
    val TypeScheme{tscParameters=vs, tscBody=tau} = specSc
    val ts = map (fn v => TypeOfTypeVar(refresh0HardTypeVar v)) vs
    val UE = zip2 vs ts
    val tau0 = expandRea UE tau
    val TypeScheme{tscParameters=vs', tscBody=tau'} = infSc
    val ts' = map (fn v => TypeOfTypeVar(refresh0TypeVar v)) vs'
    val UE' = zip2 vs' ts'
    val tau'0 = type_subst UE' tau'
  in
    unify tau'0 tau0
    handle Unify _ =>
      (let
         val ts = map TypeOfTypeVar vs
         val UE = zip2 vs ts
         val tau0 = expandRea UE tau
       in
         msgIBlock 0;
         errPrompt "Type mismatch: value identifier "; msgString id;
         msgString " in the signature has type"; msgEOL();
         errPrompt "  "; printType tau0; msgEOL();
         errPrompt "whereas its implementation in the unit's body has type";
         msgEOL();
         errPrompt "  "; printScheme infSc; msgEOL();
         msgEBlock();
         raise Toplevel
       end)
  end

fun checkHomeUnits infQual specQual id thing =
  if specQual <> infQual then (
    msgIBlock 0;
    errPrompt "Specified signature expects the ";
    msgString thing; msgString " ";
    msgString id; msgString " to be defined"; msgEOL();
    errPrompt "in the unit "; msgString specQual;
    msgString " but it is defined in the unit ";
    msgString infQual; msgEOL();
    msgEBlock();
    raise Toplevel)
  else ()

fun exportValAsVal (infStatus: ConStatus) (specStatus: ConStatus) =
    (true, Lprim(Pset_global (#qualid specStatus, 0),
                 [Lprim(Pget_global (#qualid infStatus, 0), [])]))

fun exportPrimAsVal (pi : PrimInfo) (specStatus : ConStatus) =
    (true, Lprim(Pset_global (#qualid specStatus, 0),
                 [trPrimVar (#primOp pi)]))

fun exportConAsVal (ci : ConInfo) (specStatus : ConStatus) =
    (true, Lprim(Pset_global (#qualid specStatus, 0),
                 [trConVar ci]))

fun exportExConAsVal (ei : ExConInfo) (specStatus : ConStatus) =
    (true, Lprim(Pset_global (#qualid specStatus, 0),
                 [trTopExConVar ei]))

fun matchIdStatus infStatus specStatus lams =
    let val {qualid = infQualid,  info = infInfo}  = infStatus
        val {qualid = specQualid, info = specInfo} = specStatus
        val {qual = infQual, ...}      = infQualid
        val {qual = specQual, id = id} = specQualid
    in
        case specInfo of
            VARname ovltype =>
                (* checkHomeUnits infQual specQual id "value"; *)
                (case infInfo of
                  VARname ovltype' =>
                      let val _ = if ovltype = ovltype' then ()
                                  else errorImplMismatch id
                      in
                          if specQual <> infQual then
                              exportValAsVal infStatus specStatus :: lams
                          else lams
                      end
                | PRIMname pi' => exportPrimAsVal pi' specStatus :: lams
                | CONname ci'  => exportConAsVal ci' specStatus :: lams
                | EXNname ei'  => exportExConAsVal ei' specStatus :: lams
                | REFname      => errorImplMismatch id)
         | PRIMname pi =>
         (* checkHomeUnits infQual specQual id "prim_value"; *)
               (case infInfo of
                    VARname ovltype' => errorImplMismatch id
                  | PRIMname pi'=> if pi = pi' then lams
                                   else errorImplMismatch id
                  | CONname ci' => errorImplMismatch id
                  | EXNname ei' => errorImplMismatch id
                  | REFname     => errorImplMismatch id)
         | CONname ci =>
         (* checkHomeUnits infQual specQual id "value constructor"; *)
               (case infInfo of
                    VARname ovltype' => errorImplMismatch id
                  | PRIMname pi' => errorImplMismatch id
                  | CONname ci'  =>
                        if #conArity(!ci) <> #conArity(!ci')
                            orelse #conIsGreedy(!ci) <> #conIsGreedy(!ci')
                            orelse #conTag(!ci) <> #conTag(!ci')
                            orelse #conSpan(!ci) <> #conSpan(!ci')
                            then errorConImplMismatch id
                        else lams
                  | EXNname ei'  => errorImplMismatch id
                  | REFname      => errorImplMismatch id)
         | EXNname ei =>
               (checkHomeUnits infQual specQual id "exception";
                case infInfo of
                    VARname ovltype' => errorImplMismatch id
                  | PRIMname pi' => errorImplMismatch id
                  | CONname ci'  => errorImplMismatch id
                  | EXNname ei'  =>
                        if #exconArity(!ei) <> #exconArity(!ei')
                            orelse #exconIsGreedy(!ei) <> #exconIsGreedy(!ei')
                            then errorExConImplMismatch id
                        else lams
                  | REFname      => errorImplMismatch id)
         | REFname =>
               (case infInfo of
                    VARname ovltype' => errorImplMismatch id
                  | PRIMname pi' => errorImplMismatch id
                  | CONname ci' => errorImplMismatch id
                  | EXNname ei' => errorImplMismatch id
                  | REFname => lams)
    end

fun matchSignature (inferredSig : CSig) (specSig : CSig) =
    let val _ =  (* Matching stamps of mentioned signatures *)
        matchStamps inferredSig specSig

        val _ =  (* Type realization *)
            Hasht.apply (fn id => fn specTyName =>
                  realizeTyName (lookupSig_TyEnv inferredSig id) specTyName)
                  (#uTyEnv specSig)
        val _ =
            Hasht.apply (fn id => fn specTyName =>
                  checkRealization inferredSig specSig
                                   (lookupSig_TyEnv inferredSig id) specTyName)
                 (#uTyEnv specSig);

        val _ = (* Matching value types *)
            Hasht.apply (fn id => fn specSc =>
                  matchTypeSchemes id (lookupSig_VarEnv inferredSig id) specSc)
                 (#uVarEnv specSig);
    in
        (* Status matching; this may cause some code to be generated, in
         * case a primitive function or a value constructor is exported
         * as a value *)
        Hasht.fold (fn id => fn specStatus => fn lams =>
                  matchIdStatus (lookupSig_cBas inferredSig id) specStatus
                  lams) [] (#uConBasis specSig)
    end
```

Appendix G

Test Programs

```
(* andalso.sml *)

val _ = util.println(true andalso false,
                     if true andalso false then 111 else 222)
```

```
(* callextern.sml *)

val _ =
    let val arg = ("hejsa", "dav")
    in
        showarg.f' arg;
        showarg.g'' arg
    end
```

```
(* clos1.sml *)

val f = fn x => x
```

```
(* clos2.sml *)

val f =
    let val x = 42
    in
        fn y => x+y
    end

val _ = util.println(f 117)
```

```
(* clos3.sml *)

val f = fn x =>        fn y => x+y
val g = f 42
val _ = util.println(g 117)
```

```
(* clos4.sml *)

val f = fn x =>        fn y => x+y
val g = f 42
val h = fn z => util.println(g z)

val _ = (h 0;
         h 100;
         h 1000)
```

```
(* clos5.sml *)

val f  = fn x => x+x
val g  = fn y => y*y
val h  = fn f' => fn g' => fn z => f'(g' z)
val fg = h f g

val _ = util.println(f 21, g 117, fg 1)
```

```
(* clos6.sml *)

fun f x =
    let val a = 42
        fun g y = a+y
    in
        g x
    end

val _ = util.println(f 117)
```

```
(* clos7.sml *)

val _ =
    let val f = fn x => x+x
        val g = fn y => y*y
        val h = fn f' => fn g' => fn z => f'(g' z)
    in
        util.println(h f g 21)
    end
```

```
(* clos8.sml *)

val _ =
    let val f   = fn x => x+x
        val g   = fn x => x*x
        val h   = fn x => f(g x)
        val h'  = fn x => fn y => f(x - y)
        val h'' = fn x => fn y => f x - g y
    in
        util.println(h 23,
                     h' 42 117,
                     h'' 37 7)
    end
```

```
(* clos9.sml *)

val _ =
    let fun id f = f
    in
        util.println(id (fn x => x+x) 21)
    end
```

```
(* clos10.sml *)

fun f x = x+1

fun g x = fn y => x+y

val f = fn x => f x

val _ = util.println(f 42, g 7 117)

val g =
    let val a = 42
    in
        fn x => g a x
    end
```

```
(* clos11.sml *)

fun f g h =
    let val x = 117
    in
        h x
    end

val _ = f 42 (fn y => util.println y)
```

```
(* div.sml *)

val a =
    let val a' = 42
    in
        (a' div 0) handle Div => 117
    end

val b = (42 div 3) handle _ => 217

val c = let val c' = 0
        in
            (42 div c') handle Div => 317
        end

val _ = util.println(a, b, c)
```

```
(* exn1.sml *)

exception A and B
exception C = A

local
    exception D and E = B
    val v = (A, B, C, D, E)
in
    exception A and B
    exception F = C and G = D

    fun f () = (raise F) : unit

    fun g () = (raise G) : unit

    val _ = (f() handle F => ();
             g() handle G => ();
             util.println v)
end
```

```
(* exn2.sml *)

exception Slam = Fail

fun f 0 = raise Slam "Kazam!"
  | f k = (util.println k;
           f(k-1))

val _ = (f 3) handle (e as Slam _) => (util.print "Caught exception ";
                                       util.println e)

val a = 42 handle e => raise Slam "Zap!"

val _ = util.println a
```

```
(* exn3.sml *)

val _ =
    let val a = 2
        val b = 42 + ((7 div a) handle _ => 117)
    in
        util.println(a, b)
    end
```

```
(* exn4.sml *)

exception Zap of string

val _ =
    let fun f () = (raise Zap "kazam!") : int
        val a = 117
        val b = 42 + (f() handle Zap _ => a)
    in
        util.println(a, b)
    end
```

```
(* exn5.sml *)

fun f () =
    let exception Slam
    in
        (raise Slam): unit
    end

val _ = f()
```

```
(* exn6.sml *)

fun f n =
    let exception Slam
    in
        if n=0 then raise Slam
        else (f (n-1) handle Slam => 117)
    end

val _ = f 3
```

```
(* exn7.sml *)

val _ =
    let exception Slam

        fun f n = if n=0 then raise Slam
                  else (f (n-1) handle Slam => 117)
    in
        util.println(f 3)
    end
```

```
(* exn8.sml *)

val _ =
    let exception Slam

        val e = Slam

        fun f n =
            if n=0 then
                let exception Slam
                in
                    raise e
                end
            else (f (n-1) handle Slam => 117)
    in
        util.println(f 3)
    end
```

```
(* exn9a.sml *)

fun A x = x+x

val f = A

val _ = util.println f

exception A of string

local
    exception B of int
in
    exception C = B
    exception B of string
end

exception D = Fail

val _ = util.println(A "Slam",
                     B "Bam",
                     C 42,
                     D "Oops!")
```

```
(* exn9b.sml *)

val _ =
    let open exn9a
    in
        util.println f;
        util.println(A "Kazam!",
                     B "Zap!",
                     C 117,
                     D "What?",
                     Fail "Bingo!")
    end
```

```
(* exn9c.sml *)

open exn9a

val _ = (util.println f;
         util.println(A "Kazam!",
                      B "Zap!",
                      C 117,
                      D "What?",
                      Fail "Bingo!"))
```

```
(* exn9d.sml *)

val _ =
    let open exn9c
    in
        util.println f;
        util.println(A "Kazam!",
                     B "Zap!",
                     C 117,
                     D "What?",
                     Fail "Bingo!")
    end
```

```
(* fnpair.sml *)

val (f, g) = (fn x => x+x, fn y => y*y)

val _ = util.println(f 42, g 42)
```

```
(* if.sml *)

val _ = util.println(if true then 111 else 222)
```

```
(* intcase.sml *)

fun app f =
    let fun app' [] = []
          | app' (x::xr) = (f x; app' xr)
    in
        app'
    end

fun f x =
    util.print
        (case x of
             0   => "a "
           | 117 => "b "
           | 43  => "c "
           | _   => "d ")

val _ = (app f [0, 42, 43, 117];
         util.println "")
```

```
(* inteq.sml *)

fun mem v ([] : int list) = false
  | mem v (x::xr)         = v=x orelse mem v xr

val xs = [1, 2, 3, 4]
val ys = [1, 2, 4, 5]

val _ = util.println(mem 3 xs, mem 3 ys)
```

```
(* intexpr.sml *)

val (a, b) =
    let val c = 2+3
        val d = 4-5
    in
        (c mod d, c div d)
    end

val _ = util.println (a, b)

val a = 42
val e = ~a

val _ = util.println (a, b, e)
```

```
(* inttest.sml *)

fun f' (a: int, b) =
    (util.println(a, b);
     util.println(a < b, a <= b, a = b, a <> b, a >= b, a > b))

val _ = (f'(42, 42);
         f'(42, 117);
         f'(117, 42))
```

```
(* let.sml *)

val _ =
    let val a = (util.println "a"; 1)
        val _ = (util.println "_"; 2)
        val b = (util.println "b"; 3)
    in
        util.println(a, b)
    end
```

```
(* letrec1.sml *)

val _ =
    let fun f x = (util.print "f ";
                   util.println x)
        and g y = (util.print "g ";
                   util.println y)
        and h z = (util.print "h ";
                   util.println z)
    in
        f 111;
        g 222;
        h 333
    end
```

```
(* letrec2.sml *)

val _ =
    let fun f x = (util.print "f ";
                   util.println x;
                   if x < 11 then g x
                   else ())
        and g y = (util.print "g ";
                   util.println y;
                   h(y+1))
        and h z = (util.print "h ";
                   util.println z;
                   f(z*2))
    in
        f 1
    end
```

```
(* listswitch.sml *)

fun ints n =
    let fun ints' (0, l) = l
          | ints' (k, l) = ints'(k-1, k::l)
    in
        ints'(n, [])
    end

fun f [] = util.println "nil"
  | f (x::xr) = (util.println x;
                 f xr)

fun g [] = util.println "nil"
  | g (x::xr) = (g xr;
                 util.println x)

val xs = ints 3

val _ = (f xs;
         util.println xs;
         g xs)
```

```
(* not.sml *)

fun h x = not x

val _ = util.println(not false, h true,
                     if not true then 111 else 222,
                     if h false then 333 else 444)
```

```
(* orelse.sml *)

val _ = util.println(false orelse true,
                     if false orelse true then 111 else 222)
```

```
(* overflow.sml *)

val _ =
    let val min = ~1073741824 * 2
        val max = 1073741823 * 2 + 1

        val a = (min - 1)
                handle Div      => 117
                     | Overflow => 127
                     | _        => 137

        val b = (max + 1)
                handle Overflow => 217
                     | Div      => 227
```

```
                  | _         => 237

        val c = (42 div ~1)
                handle Div      => 317
                     | Overflow => 327
                     | _        => 337

        val d = (min div ~1)
                handle Overflow => 417
                     | Div      => 427
                     | _        => 437

        val e = (117 * 3)
                handle Div      => 517
                     | Overflow => 527
                     | _        => 537

        val f = (max * 2)
                handle Div      => 617
                     | Overflow => 627
                     | _        => 637
    in
        util.println(a, b, c, d, e, f)
    end
```

```
(* polyeq.sml *)

fun mem v ([])    = false
  | mem v (x::xr) = v=x orelse mem v xr

val _ =
    let val xs = [1, 2, 3, 4]
        val ys = [1, 2, 4, 5]
        val zs = [1, 2, 3, 4]
    in
        util.println(mem 3 xs, mem 3 ys, xs = ys, xs = zs)
    end

val _ =
    let val xs = [1.0, 2.0, 3.0, 4.0]
        val ys = [1.0, 2.0, 4.0, 5.0]
        val zs = [1.0, 2.0, 3.0, 4.0]
    in
        util.println(mem 3.0 xs, mem 3.0 ys, xs = ys, xs = zs)
    end

val _ =
    let val xs = ["1", "2", "3", "4"]
        val ys = ["1", "2", "4", "5"]
        val zs = ["1", "2", "3", "4"]
    in
        util.println(mem "3" xs, mem "3" ys, xs = ys, xs = zs)
    end

datatype t =
    A
  | B of int
  | C of real
  | D of string

val _ = (util.println(A = B 42, C 117.0 = C 117.0);
         util.println(() = (), (A, B 17) = (C 43.0, D "slam")))
```

```
(* realcase.sml *)

fun app f =
    let fun app' [] = []
          | app' (x::xr) = (f x; app' xr)
    in
        app'
    end

fun f x =
    util.print
        (case x of
             0.0   => "a "
           | 117.0 => "b "
           | 43.0  => "c "
           | _     => "d ")

val _ = (app f [0.0, 42.0, 43.0, 117.0];
         util.println "")
```

```
(* realeq.sml *)

fun mem v ([] : real list) = false
  | mem v (x::xr)          = v=x orelse mem v xr

val xs = [1.0, 2.0, 3.0, 4.0]
val ys = [1.0, 2.0, 4.0, 5.0]

val _ = util.println(mem 3.0 xs, mem 3.0 ys)
```

```
(* realtest.sml *)

fun f' (a: real, b) =
    (util.println(a, b);
     util.println(a < b, a <= b, a = b, a <> b, a >= b, a > b))

val _ = (f'(42.0, 0.0 / 0.0);
         f'(42.0, 42.0);
         f'(42.0, 117.0);
         f'(117.0, 42.0))
```

```
(* refloop.sml *)

datatype t = T of t option ref

val _ =
    let val x = T (ref NONE)
        val T y = x
    in
        y:= SOME x;
        util.println x
    end
```

```
(* showarg.sml *)

fun f' x = util.println x

val g'' : 'a -> unit = util.println
```

```
(* sig.sig *)

type t1
datatype t2 = C | D of t2

val A : int -> t1
val g : 'a -> 'a
```

```
(* sig.sml *)

datatype t1 = A of int | B of real
datatype t2 = C | D of t2

fun f x = x

val g = f
```

```
(* stringcase.sml *)

fun app f =
    let fun app' [] = []
          | app' (x::xr) = (f x; app' xr)
    in
        app'
    end

fun f x =
    util.print
        (case x of
             "0"   => "a "
           | "117" => "b "
           | "43"  => "c "
           | _     => "d ")

val _ = (app f ["0", "42", "43", "117"];
         util.println "")
```

```
(* stringeq.sml *)

fun mem v ([] : string list) = false
  | mem v (x::xr)            = v=x orelse mem v xr

val xs = ["1", "2", "3", "4"]
val ys = ["1", "2", "4", "5"]

val _ = util.println(mem "3" xs, mem "3" ys)
```

```
(* stringprim.sml *)

prim_val size : string -> int  = 1 "string_length"
prim_val sub  : string -> int -> char = 2 "get_nth_char"

val _ = util.println(size "slam", sub "SLAM" 2)
```

```
(* stringtest.sml *)

fun f (a: string, b) =
    (util.println(a, b);
     util.println(a < b, a <= b, a = b, a <> b, a >= b, a > b))

val _ = (f("hej", "hej");
         f("hej", "dav");
         f("dav", "hej");
         f("hej", "hejsa");
         f("hejsa", "davsdu"))
```

```
(* switch.sml *)

datatype t =
    A
  | B of int
  | C of real
  | D of string

fun println A = util.println "A"
  | println (B i) = (util.print "B ";
                     util.println i)
  | println (C r) = (util.print "C ";
                     util.println r)
  | println (D s) = (util.print "D ";
                     util.println s)

val _ = (println A;
         println(B 42);
         println(C 117.0);
         println(D "slam"))
```

```
(* tostring.sml *)

val _ = util.println(42, 42.000, "slam", (), [1, 2, 3], true, SOME 117,
                     ref 42, fn x => x)
```

```
(* while.sml *)

val _ =
    let val x = ref 9
    in
        while !x > 0 do (util.println(!x);
                         x:= !x - 1);
        util.println "Lift-off!"
    end
```

Bibliography

[1] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark, April 1997. Available at `http://www.dina.kvl.dk/~pmb`.

[2] J. Gosling et al. *The Java Language Specification.* Addison-Wesley, 1996. ISBN 0-201-63451-1.

[3] J. Reppy et al. Standard ML Basis Library. Technical report, Bell Labs, Lucent Technologies, 1997. Available at `http://www.cs.bell-labs.com/~jhr/sml/basis`.

[4] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996. ISBN 0-201-63452-X.

[5] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[6] R. Milner, M. Tofte, R. Harper, and D.B. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[7] Martin Odersky and Philip Wadler. Pizza and Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[8] R. Perera and P. Bertelsen. The unofficial Java Spec Report. Available at `http://www.nodule.demon.co.uk/java`. Last issue released November 1997.

[9] S. Romanenko and P. Sestoft. *Moscow ML Language Overview, version 1.42*, July 1997. Available at `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[10] S. Romanenko and P. Sestoft. *Moscow ML Owner's Manual, version 1.42*, July 1997. Available at `http://www.dina.kvl.dk/~sestoft/mosml.html`.