A JIT Translator for Oberon

Peter Nelson
New College

May 20, 2010
1

Introduction

1.1 Overview and Motivation

Just-in-time compilation (JIT) is an increasingly popular technique for improving the runtime performance of interpreted programs. Instead of interpreting the instructions of a program one by one, a JIT translator compiles them dynamically at runtime. This approach allows us to combine the advantages of both interpreters and static compilers; namely portability and speed respectively.

Today, JIT is used in a wide variety of applications. The Microsoft .NET Framework and Java Platform runtimes both make extensive use of the technique. JavaScript (arguably the most widely deployed programming language in existence) was interpreted by all of the major web browsers until very recently. As demand for high performance web applications has grown, most modern web browsers now employ JIT techniques to improve JavaScript performance. The use of JIT for this purpose is the subject of ongoing research and is very competitive.

The Oxford Oberon-2 compiler and runtime platform, written by Mike Spivey, currently uses a bytecode interpreter to run programs, and also includes an experimental scratch-built JIT. The aim of this project is to extend the runtime with a new JIT translator which will instead be based on the open-source LibJIT library. This will allow us to take advantage of LibJIT’s features and portability. We will also investigate the advantages of this approach over a simple interpreter.
This section introduces several concepts that are related to the project. Readers are assumed to have a basic knowledge of how a compiler works and an understanding of low-level computer architecture. A working knowledge of the C programming language and Intel x86 assembly is helpful but not essential. Note that the definitions of several standard compiler implementation data structures, basic blocks and flow graphs in particular, are based on those given in [1] and are not reproduced here.

### 2.1 Just-in-Time Compilation

A just-in-time compiler (JIT) translates program instructions into native machine code at runtime. These are often in the form of bytecode instructions for some virtual machine. In contrast to a bytecode interpreter, instead of re-evaluating segments of code each time they are encountered, JIT translates them once and caches the generated machine code. On subsequent encounters of the same section of code the runtime reuses the cached translation.

JIT also offers a number of advantages over static compilers. The JIT may gather information at runtime to optimise frequently executed code paths, enable the use of machine-specific instructions, or rearrange memory for more effective cache utilization. With these techniques, JIT performance can sometimes match and even exceed the performance of statically-compiled native code.

### 2.2 The Keiko Virtual Machine

This section is paraphrased from Mike Spivey’s Oxford Oberon-2 compiler design documentation at [5].

The Oxford Oberon compiler generates bytecode for the stack-based Keiko VM. The existing Oberon runtime implements this VM with an interpreter. The architecture and programming environment of this machine is discussed in detail in this section.

#### 2.2.1 Machine Architecture

The state of the virtual machine at any point during execution is determined by the stack frame of the running procedure, the contents of the evaluation stack and six specialised registers that provide access to the execution context.

The evaluation stack typically stores temporary values during the evaluation of an expression or statement. This is analogous to the general-purpose registers of a register-based machine. The Oberon compiler translates expressions into postfix form, for example the expression $x + 4$ is
translated to the bytecode sequence \texttt{LDLW \textit{x} / CONST 4 / ADD}, with \textit{x} represented by its offset in the stack frame. The first two instructions push the value of the local variable \textit{x} and the constant value 4 onto the stack, respectively. The arithmetic instruction \texttt{ADD} pops two values off the stack, adds them, and pushes the result.

Bytecode instructions come in packed and unpacked forms. Unpacked instructions provide access to the machine’s basic functions, such as arithmetic and data access. Packed instructions are abbreviations for common sequences of unpacked instructions, provided for compactness and efficiency in an interpreter implementation. For example, the packed instruction \texttt{LDLW \textit{x}} expands to the unpacked instruction sequence \texttt{LOCAL \textit{x} / LOADW} which has the combined effect of pushing a word from offset \textit{x} of the stack frame onto the evaluation stack.

In addition to the evaluation stack, six registers are used by the virtual machine:

- **PC** – Program counter. Points to the next bytecode instruction to be executed.
- **CP** – Context Pointer. Points to the procedure descriptor of the current procedure. Values in the procedure’s constant pool are found at fixed offsets from CP.
- **BP** – Base Pointer. Points to a fixed location on the current stack frame. Parameters and local variables are found at fixed offsets from BP.
- **SP** – Stack Pointer. Points to the top of the evaluation stack. Temporary values are found at fixed offsets from SP.
- **Result** – Used for passing return values back to the calling procedure.
- **Link** – Used by nested procedures to point to the stack frame of the enclosing procedure.

### 2.2.2 Procedure Descriptor

A procedure descriptor contains information required to call the procedure, information about its runtime stack layout and its constant pool. The constant pool stores large constants that are used within the procedure. Procedure descriptors are designed in such a way that both bytecode and native code procedures can be executed by the Keiko VM. This is described in detail in section 2.2.3.

### 2.2.3 Calling Convention

A calling convention is a protocol that specifies how one procedure calls another. The Keiko VM uses an extension of the host machine’s C calling convention, enabling bytecode procedures to call native code procedures and vice-versa. As this project was developed on an Intel x86 machine we will focus on the C calling convention for that architecture (namely cdecl) when the details are important. Hence the Keiko VM actually uses two runtime stacks – its own stack (referred to as the evaluation stack or Oberon stack) and the stack of the host machine (referred to as the C stack).

When calling a bytecode procedure, the calling convention creates a stack frame on the evaluation stack in addition to the usual C stack frame on the C stack. An exception to this is when one bytecode procedure calls another, in which case no frame is created on the C stack. All procedure parameters are passed on the evaluation stack. A single parameter containing the value of the BP register is passed on the C stack. Thus, all native code procedures are required to have the function signature \texttt{void func(value* bp)}.

Every procedure’s descriptor contains an address of such a native code function which the runtime calls to handle the procedure. For native code procedures, this points to the function in memory.
which implements that procedure. For bytecode procedures, it is a special built-in native code function which implements the interpreter. An example of this in a program consisting of two bytecode procedures and a single native procedure is shown in figure 2.2.1.

![Diagram showing the difference between native and bytecode procedure runtime function pointers.](image)

**Figure 2.2.1.** Difference between native and bytecode procedure runtime function pointers.
Building a JIT Translator for Oberon

3.1 Analysis

3.1.1 Introducing LibJIT

LibJIT is part of the DotGNU project, an open-source implementation of the Common Language Infrastructure (CLI). The CLI is a specification describing the virtual machine architecture that forms the core of the Microsoft .NET Framework. LibJIT is used in the DotGNU implementation of the CLI to provide a set of routines that constructs, compiles and executes a machine independent representation of a program at runtime.

The Keiko VM is similar to the CLI in many ways. Both are stack-based machines and provide a similar set of instructions. This makes LibJIT ideal for use within the Oberon JIT translator. It is advantageous to use a library such as LibJIT for this task instead of writing one from scratch as it allows us to concentrate on higher-level details of implementing a JIT, such as virtual machine semantics, without having to worry much about the lower-level data flow analysis, register allocation and native code generation.

3.1.2 System Overview

The existing Oberon runtime, written by Mike Spivey, includes a bytecode interpreter and an experimental JIT, and already performs the task of loading bytecode programs from disk and relocating them in memory. Of the existing JIT translator, we will reuse only the functions that are concerned with decoding and unpacking bytecode, as these tasks are largely independent of the JIT implementation. Therefore what we intend to do is to replace the interpreter and existing JIT with a new JIT translator that constructs a LibJIT representation of the bytecode. Due to the architecture of the Keiko VM, this can be done relatively seamlessly and even allows for the JIT and interpreter to run alongside each other. However, as the focus of the project is on JIT concepts, we will disable the interpreter for all but testing purposes and concentrate on the JIT translator. Thus we will also largely ignore parts of the runtime involved with loading, relocating, decoding and unpacking bytecode, and initialising the Keiko VM.

Therefore, we assume that we have access to the whole virtual machine state from our JIT translator module. This includes the stack, registers and bytecode procedure that is to be translated. Translating a bytecode procedure involves a number of stages:

- Determine branch targets. It is necessary to know the source and location of branches as these form the boundaries of basic blocks.
• Create a basic block graph representation of the procedure. This enables various dataflow analysis techniques to be applied for optimisation purposes and for ensuring program correctness.
• Perform type inference on variables in the procedure. The bytecode used by the Keiko VM does not contain type information for variable loads and stores, so this information must be reconstructed at this stage.
• Construct a LibJIT representation of the procedure. This involves using the information gathered in the previous stages to produce the most optimal representation. When completed, LibJIT can compile the procedure to native machine code and it can be executed.

We begin by replacing the bytecode interpreter function (shown in figure 2.2.1) with a function that translates the bytecode, the implementation of which is the subject of the rest of this chapter. By doing this we aim to implement the “on first call” strategy for translation, as opposed to pre-translating every procedure before starting the program. The advantage of this is that it spreads out the JIT translation overheads and means that we don’t needlessly translate procedures that are never used. It also means we can easily apply more complicated strategies in the future, for example only compiling large procedures that are used frequently and falling back to the interpreter for everything else. The example from chapter 2 now looks like that shown in figure 3.1.1.

![Figure 3.1.1. Difference between native and bytecode procedure runtime function pointers before JIT translation.](image)

### 3.2 Example Procedure

Throughout the course of this chapter, concepts will be illustrated using an example Oberon procedure which returns the factorial of its input. This demonstrates use of local variables, parameters, branching and recursive procedure calls. The Oberon source code is given in listing 3.2.1. The procedure compiles to the (unpacked) bytecode shown in listing 3.2.2.
PROCEDURE Fac(n, f: INTEGER): INTEGER;
BEGIN
  IF n = 0 THEN
    RETURN f
  ELSE
    RETURN Fac(n-1, n*f)
  END
END Fac;

Listing 3.2.1. Factorial procedure written in Oberon.

01. LOCAL 12
02. LOADW
03. CONST 0
04. JNEQ <line 09>
05. LOCAL 16
06. LOADW
07. RESULTW
08. RETURN
09. LOCAL 12
10. LOADW
11. LOCAL 16
12. LOADW
13. TIMES
14. LOCAL 12
15. LOADW
16. CONST 1
17. MINUS
18. LDKW 0
19. JPROC <Fac>
20. SLIDEW
21. RESULTW
22. RETURN

Listing 3.2.2. The factorial procedure compiled to bytecode.

3.3 Determining Branch Targets

Both conditional and unconditional branch instructions specify a target instruction that program control should jump to. The address of the target is given relative to the address of the branch instruction.

To detect these locations, we step through the program bytecode, keeping track of the state of the PC register at each point. When a branch instruction is encountered we calculate the target address relative to the first instruction in the program and add this information to a lookup table. The pseudocode for this is given in listing 3.3.1.
relativePC := 0
FOR EACH instruction i in instructions
    IF i is a branch instruction
        targetaddress := relativePC + i.operand
        mark_label(targetaddress)
        relativePC := relativePC + i.length

Listing 3.3.1. Psuedocode for finding branch targets in the bytecode program.

3.4 Creating Basic Blocks

At this stage we have a list of every branch target in the procedure. This provides enough information to identify block leaders and thus allows us to construct a flow graph of the bytecode. A pseudo-C structure of a basic block is given in listing 3.4.1. Due to the nature of the Keiko VM bytecode, a basic block may have any number of predecessors in the flow graph, but at most two successors. In cases where a block ends with an unconditional branch instruction, the jumptarget field holds a pointer to the next block. Where a block ends with a conditional branch instruction, the conditionaltarget and jumptarget fields hold pointers to the block to branch to when the condition is true and false, respectively.

A pseudocode representation of the algorithm used to create a flow graph of basic blocks, based loosely on that found in [1], is shown in listing 3.4.2. The algorithm performs a single pass over the input bytecode and constructs blocks and flow graph edges as it encounters block leaders and branch instructions. A lookup table is used to find blocks that have already been created for a particular block leader. It is assumed that packed bytecode instructions have already been unpacked.

struct basicblock
{
    basicblock[] predecessors;
    instruction[] instructions;
    basicblock jumptarget;
    basicblock conditionaltarget;
}

Listing 3.4.1. Psuedo-C structure definition for a basic block.

3.4.1 Example

Running this algorithm on the (unpacked) bytecode of the factorial procedure introduced in section 3.2, we obtain the flow graph shown in figure 3.4.3.
FUNCTION Create_Blocks(instruction[] instructions)
    B := NEW basicblock
    B.predecessors := NULL
    B.jumptarget := NULL
    B.conditionaltarget := NULL
    Blocks.add(B)
    FOR EACH instruction i IN instructions
        B.instruction.add(i)
        IF i is a block leader
            IF a block for this leader already exists
                B' := LOOKUP(i)
            ELSE
                B' := NEW basicblock
                B'.predecessors.add(B)
                IF previous instruction is not a RETURN or unconditional JUMP
                    B.jumptarget := B'
                    Blocks.add(B')
                B := B'
            END
        END
        IF i is a conditional branch instruction
            IF a block for the branch target already exists
                C := LOOKUP(TARGET(i))
            ELSE
                C := NEW basicblock
            END
            IF a block for the next instruction already exists
                N := LOOKUP(NEXT(i))
            ELSE
                N := NEW basicblock
            END
            C.predecessors.add(B)
            N.predecessors.add(B)
            B.jumptarget := N
            B.conditionaltarget := C
            Blocks.add(C, N)
            B := N
        END
    END
    FOR EACH basicblock B where B.jump = NULL
        B.jump = END
    END

Listing 3.4.2. Psuedocode algorithm for creating a flow graph of basic blocks.
3.4.2 Bytecode Optimisation

Flow graphs provide a great deal of information that can be used to optimise a program. In a static compiler, sophisticated dataflow analysis techniques are used to produce optimal code. In a JIT translator, a compromise must be made between the time taken to translate a section of code and the quality of code generated. It is therefore common in JIT implementations to optimise a program quickly on the first pass, and delay additional optimisation until heavily used sections of code have been determined. In this implementation we will perform some peephole optimisation and local common sub-expression elimination (CSE). This section will concentrate on the former, as CSE is performed on LibJIT expression trees at a later stage.

Peephole Optimisation

Peephole optimisation works by looking at a small set of instructions in isolation, recognising redundant or useless patterns and replacing them with a smaller set of instructions. For example, the bytecode instruction sequence SLIDEW / RESULTW, which pushes the result register onto the stack
and immediately writes it back to result has no cumulative effect. Similarly the sequence \texttt{LOAD x / STORE x} is also useless. The peephole optimiser repeatedly passes over the bytecode in each basic block, removing these instruction sequences until the code can be reduced no further. After performing this on the flow graph of the factorial program, block 3 is transformed as shown in figure 3.4.4.

![Figure 3.4.4. Section of the factorial procedure before and after peephole optimisation.](image)

### 3.5 Performing Type Inference

In LibJIT, stores and loads of values to and from memory require that the data type of the value is known. This is partly due to the way many platforms handle floating point and integer values differently. For example, the Intel x86 architecture uses a completely different set of hardware registers and instructions for manipulating floating point values. When LibJIT issues an instruction to load a value from memory it needs to know whether to store it in a general purpose or floating point register.

Therefore we must infer local and global symbol (including variable and constant) types from the bytecode before starting to work with LibJIT. For the moment we only consider integers and single precision floating point types. Bytecode in each basic block is pseudo-interpreted to observe the state of the evaluation stack at every program point. If an instruction loads a local or global symbol, that symbol is added to a lookup table if it has not already been encountered. Symbols start with no type information and progressively accumulate it as the algorithm proceeds. If an instruction pops values from the stack of a particular type (for example, \texttt{FPLUS} expects two single-precision floating point values), then the symbols associated with those values, if any, have their type information updated. When no type information can be inferred for a particular symbol it is treated as an integer by default. The algorithm used is given in listing 3.5.1. It is assumed that the evaluation stack is empty at both the start and end of every basic block.
FUNCTION Infer_Types(basicblock[] blocks)
    changed := TRUE
    WHILE changed DO
        changed := FALSE
        FOR EACH basicblock b in blocks
            FOR EACH instruction i in b
                IF i loads the value of a local or global symbol
                    Symbols.add(VAR(i))
                IF i does not modify the stack
                    CONTINUE
                IF i pops values from the stack
                    FOR EACH value v popped from the stack
                        IF v is the value of a symbol sym
                            sym.type := TYPE(v)
                            changed := TRUE
                    Stack.pop(NUMBEROFARGS(i))
                IF i pushes values onto the stack
                    FOR EACH value v pushed onto the stack
                        Stack.push(v, TYPE(v))
    END WHILE
Listing 3.5.1. Pseudocode for inferring symbol types in a bytecode program.

3.5.1 Example

We will illustrate the type inference algorithm with a procedure that increments a single-precision floating point variable. Note that the procedure would have no actual effect on the state of any program as it only modifies local variables and returns nothing, so a good compiler would optimise it out entirely. However it will suffice for illustrative purposes. The code snippet is shown in listing 3.5.2.

FUNCTION TypeTest(f: REAL);
BEGIN
    f := f + 1.0;
END TypeTest;
Listing 3.5.2. Oberon procedure demonstrating single-precision floating point arithmetic.

This compiles to the flow graph consisting of a single basic block shown in figure 3.5.3.

Instructions 1 and 2 together push the contents of variable \( f \) onto the stack. At this point, the variable has not been encountered yet so it is added to the symbol lookup table. Note that we do not yet know what type it is, so it is simply labelled as unknown. The states of the program counter, evaluation
Figure 3.5.4. State of the type inference algorithm after instruction 2.

Instruction 3 loads a constant from the procedure’s constant pool and pushes it onto the stack. Although we know by looking at the Oberon source code that the constant 1.0 is obviously a floating point value, this information is not present at runtime – all we see is the bit pattern with hexadecimal value 0x3F800000. This could correspond to either an integer or floating point (or some other data type). The program state is shown in 3.5.5.

Figure 3.5.5. State of the type inference algorithm after instruction 3.

Instruction 4 is a single-precision floating point arithmetic instruction, expecting two floating point values on the stack and pushing a floating point result. Hence we can infer that both \( f \) and the hexadecimal constant 0x3F800000 are of type single-precision floating point, and so we update their entries in the symbol lookup table. This is shown in figure 3.5.6.

Figure 3.5.6. State of the type inference algorithm after instruction 4.
The remainder of the program simply writes the result back to the variable \( f \) and returns.

### 3.6 Constructing a LibJIT Representation

LibJIT uses a number of data structures to represent program objects. In this project, these objects are created and manipulated through LibJIT’s C API. Brief descriptions of the most important of these are given below.

#### Function

A function object represents a distinct section of code, taking a variable number of parameters as input and (optionally) returning a single value. It contains a sequence of instructions representing the function body. LibJIT provides ways to create functions, add instructions to them, and compile them to native code. We will use exactly one function object to represent each bytecode procedure.

#### Value

Value objects represent variables that are assigned to exactly once. They can be thought of as representing nodes in an expression tree. LibJIT provides functions for creating and manipulating values in the context of a function object. When the function object is compiled, LibJIT generates code to compute the results of the values in the function. We use a stack of pointers to value objects to simulate the Keiko VM evaluation stack.

#### Type

A type object represents either the type of a value object or the signature of a function object. LibJIT provides a number of built-in primitive types (such as integers of varying sizes and floating point numbers) and has functions for creating custom record types. We use these to indicate the type of program symbols and for defining function signatures.

#### Label

Label objects represent branch instruction targets. LibJIT provides functions for explicitly creating labels. We use labels to mark block leaders when generating code for each basic block, and use these to generate branch instructions.

Constructing a LibJIT representation of a procedure generally involves a number of steps. Firstly a function object is created by specifying a particular function signature. Instructions are then added to the function body – most take value objects as arguments and return new value objects. LibJIT then compiles the function and returns a pointer to the native code in memory. In C, this pointer can be used to invoke the function directly like any other function pointer.

As an example, consider the pseudo-C code snippet in listing 3.6.1 that constructs a LibJIT function that adds two integers and returns the result.
jit_type_t param_types[3];
jit_type_t func_signature;

/* specify function parameter and return types */
jit_type_t return_type = jit_type_int;
param_types[0] = param_types[1] = jit_type_int;

/* create a type object representing the function signature */
jit_type_t func_signature = jit_create_signature(param_types, return_type);

/* create the function object */
jit_function_t func = jit_create_function(func_signature);

/* build the function body */
jit_value_t param1 = jit_insn_parameter(func, 1);
jit_value_t param2 = jit_insn_parameter(func, 2);
jit_value_t returnval = jit_insn_add(func, param1, param2);
jit_insn_return(func, returnval);

Listing 3.6.1. C LibJIT example.

The function can be represented as the expression tree in figure 3.6.2.

![Figure 3.6.2. Expression tree for the LibJIT example.](image)

In C, the function could then be compiled and called with the code shown in listing 3.6.3.
// Compile the function
jit_function_compile(func);

// Create a function pointer so we can call it directly.
// Function signature is int (*)(int, int)
int (*add_func)(int, int) = jit_function_to_closure(func);

// Call the function
int result = add_func(21, 21);

Listing 3.6.3. Compiling and calling the LibJIT example function in C.

3.6.1 Initialisation

We initialise LibJIT by firstly creating a function object to represent the procedure to be translated. As was mentioned in chapter 2, all native code procedures must have the function signature `void func(value* bp)`. As in the example above, we then store the argument BP into a global value object so it can be accessed throughout the whole program. The pseudocode for this initialisation is shown in listing 3.6.4.

```
parameter_types := [value *]
return_type := void
func := jit_create_function(parameter_types, return_type)
bp := jit_insn_parameter(func, 1)
```

Listing 3.6.4. Pseudocode LibJIT initialisation.

3.6.2 Code Generation

For each basic block in the flow graph, bytecode instructions are individually translated to an appropriate LibJIT instruction sequence. Some classes of instructions require a greater amount of work than others. At each point, a stack of LibJIT values is used to simulate the virtual machine stack.

Arithmetic Instructions

Arithmetic instructions are straightforward to generate. A typical binary instruction such as `PLUS` or `TIMES` pops two values from the stack, performs the appropriate operation on them and pushes the result. A pseudocode representation of the translation for `PLUS` is shown in listing 3.6.5. The case for unary operations is very similar.

```
operand1 := stack[sp-1]
operand2 := stack[sp-2]
result := jit_insn_add(operand1, operand2)
stack[sp-2] := result
sp := sp - 1
```

Listing 3.6.5. Pseudocode for translating binary arithmetic operations.
LibJIT functions generally take values objects as parameters and return a new value object with the result of the operation. Representing values as expression trees, the operation creates a new expression tree with the result at the root and the operands as subtrees. An expression tree representing a binary operation is shown in figure 3.6.6.

![Expression tree for a binary arithmetic operation.](image)

Figure 3.6.6. Expression tree for a binary arithmetic operation.

**Control Flow Instructions**

Both conditional and unconditional branch instructions make use of the information stored in the flow graph generated in section 3.4. Each basic block is associated with a LibJIT label object that marks the block leader. Branch instructions issue a LibJIT branch instruction to the label of the block pointed to by either `jumptarget` or `conditionaltarget`. The pseudocode for an unconditional branch is shown in listing 3.6.7. Note that a `RETURN` instruction is treated as an unconditional jump to an empty block at the end of the function. Neither of these instructions interacts with the evaluation stack in any way.

```
targetblock := currentblock.jumptarget
jit_insn_branch(targetblock.label)
```

**Listing 3.6.7.** Pseudocode for translating an unconditional branch.

Conditional branches require a slightly different approach. A typical conditional branch instruction such as `JEQ` (Jump-if-Equal) pops two values from the stack, branching if some condition between them is true (equality in the case of `JEQ`). Checking the condition involves performing the appropriate operation on the two values. The method for this is similar to that used for arithmetic operations. LibJIT provides a function `jit_insn_branch_if` which takes a boolean value and branches if it is true. The pseudocode for an example conditional branch, testing for equality, is given in listing 3.6.8.
Memory Access Instructions

Memory access instructions come in the form of either loads (reading from memory) or stores (writing to memory). The method for accessing global symbols (variables and constants) is very similar to that for accessing local variables, so we will focus on the latter in this section. Similarly we will concentrate on dealing with 4-byte (word) integer values and only mention the case for dealing with other types when the distinction is important.

In the Keiko virtual machine, both loading and storing the contents of a local variable involves two unpacked instructions. The `LOCAL x` instruction calculates an address by adding the operand `x` to the contents of the BP register, and pushes it onto the stack. This is then followed by either a `LOADW` or `STOREW` instruction. The instruction sequences `LOCAL x / LOADW` and `LOCAL x / STOREW` are common enough that the packed instructions `LDLW x` and `STLW x` are provided as equivalents.

`LOCAL`, `LOADW` and `STOREW` are also used in different contexts for manipulating pointers. The pseudocode for `LOCAL x` is shown in listing 3.6.9.

```plaintext
valx := jit_value_create_constant(x)
address := jit_insn_add(bp, valx)
stack[sp] := address
sp := sp + 1
```

Listing 3.6.9. Pseudocode for translating `LOCAL x`.

`LOADW` pops an address from the stack, loads the word value in memory located at that address and pushes it onto the stack. Loading data from memory in LibJIT requires the `jit_insn_load_relative` function which takes an address, offset, and type parameters. It is important that the correct data type is specified as LibJIT treats different data types in different ways. For example, on the Intel x86 architecture, an integer value will be loaded into one of the general-purpose registers whereas a floating point value will be loaded into one of the x87 FPU registers. We use the type information gathered in the type inference stage, described in section 3.5. The pseudocode for the `LOADW` instruction is shown in listing 3.6.10.

```plaintext
address := stack[sp-1]
type := lookup_type(variable_at(address))
value := jit_insn_load_relative(address, 0, type)
stack[sp-1] := value
```

Listing 3.6.10. Pseudocode for translating `LOADW`.

`STOREW` pops an address and a value from the stack and stores the value into memory at that address. Storing data into memory with LibJIT requires the `jit_insn_store_relative` function which takes an address, offset and value parameters. As LibJIT remembers the data type of each value, it is
not necessary to explicitly state the type in this case. The pseudocode for the `STOREW` instruction is given in listing 3.6.11.

```
address := stack[sp-1]
value := stack[sp-2]
jit_insn_store_relative(address, 0, value)
sp := sp – 2
```

Listing 3.6.11. Pseudocode for translating `STOREW`.

**Procedure Calls**

As has been previously discussed in chapter 2, calling a procedure in the Keiko virtual machine environment requires creating a frame on the Oberon stack. In addition, since JIT translated procedures are actually native, a frame must also be created on the host machine’s C stack. Thankfully LibJIT takes care of this part once we specify the procedure type signature and parameters.

Listing from the bottom of the evaluation stack to the top, a procedure’s frame consists of its parameters (in reverse order), the saved CP, PC and BP registers of the caller, local variables, and temporaries. This information is needed in order to interoperate with bytecode procedures and also for a garbage collector if one were to be added in the future. We will illustrate the layout of the stack using the factorial program given in section 3.2.

Consider an example execution of the factorial program where the instance `fac(4, 5)` has just been activated by the caller instance `fac(5, 1)`. The evaluation stack looks like that shown in figure 3.6.12. Although the factorial procedure does not have any local variables, the area for locals is still indicated on the diagram. The extent of each activation instance’s frame and the parts for which they are responsible for initialising are also shown. Note that it is the responsibility of the calling procedure to push parameters and saved state information onto the stack before passing control to the callee. Also note that the value of PC is set to 0 (this would be a “magic value” if we cared about garbage collection); this is because a native procedure uses the PC of the host machine (on Intel x86, the EIP register) and saves it on the host machine stack.

The `JPROC x` instruction expects the CP value for the procedure at the top of the stack, followed by `x` number of arguments to be passed to the callee. It pops the address, pushes the values of its CP, PC and BP registers and passes control to the native procedure found at the address. When control returns to the caller, the callee’s entire frame is popped from the stack.

When calling a bytecode procedure, the native procedure address expected by `JPROC` points either to a function that interprets the bytecode, or in the case of this project, JIT translates it.

A slight complication arises due to the fact that in the JIT translator we don’t actually store temporaries on the evaluation stack – LibJIT simulates this by using registers instead. Therefore, before calling a procedure, we must copy the arguments held in registers to the evaluation stack. This is done in a similar way to the `STOREW` instruction, except we write to offsets from BP. The pseudocode for `JPROC x` is shown in listing 3.6.13.
offset := -((number_oflocals * 4) + 4)
for each parameter \( p \), in reverse order
  \( \text{jit_insn_store_relative}(bp, \text{offset}, p) \)
  \( \text{offset} := \text{offset} - 4 \)
\( \text{jit_store_insn_relative}(bp, \text{offset}, cp) \)
\( \text{jit_store_insn_relative}(bp, \text{offset} - 4, \text{pc}) \)
\( \text{jit_store_insn_relative}(bp, \text{offset} - 8, bp) \)
newbp := \( \text{jit_value_create_constant}(\text{offset} - 8) \)
address := jstack[sp-1]
parameter_types := [value *]
parameters := [newbp]
\( \text{jit_insn_call_indirect}(\text{address}, \text{parameter_types}, \text{parameters}) \)
sp := sp - (x + 1)

**Listing 3.6.13.** Pseudocode for translating JPROC \( x \).
3.6.3 Finalizing

At this point, we have completed the translation of Keiko virtual machine bytecode into a LibJIT representation using expression trees. In order to compile and run the translation, we use `jit_function_compile` and `jit_function_to_closure` similarly to the example in listing 3.6.3. If LibJIT successfully compiles the function, it returns a function pointer which replaces the bytecode procedure’s native function pointer. The example from section 3.1.2 now looks like that in figure 3.6.14, supposing that bytecode procedure #1 has just been translated and compiled, but #2 hasn’t yet.

![Diagram](image)

**Figure 3.6.14.** Difference between native and bytecode procedure runtime function pointers after JIT translating bytecode procedure #1.

3.7 Summary

By now, we have implemented the bulk of the JIT translator and are now able to compile and execute a bytecode procedure. Although certain features such as double-precision floating point values and interaction with the garbage collector were beyond the scope of the project, we can fully JIT a large number of useful Oberon programs. In the following chapters we will investigate ways to improve the code generation process and compare the runtime against several existing systems.
4

Improving Performance

4.1 Indentifying Areas for Improvement

Although stack machine bytecode is very simple, it can also be inefficient if translated naively, often because there are many redundant memory accesses. Access to main memory (and even the CPU memory cache) is generally significantly slower than accessing CPU registers, so our goal here is to minimise the number of redundant reads by caching recently accessed variables. Using the same methods we can also cache the results of expressions that have already been evaluated. This form of program optimisation is called Common Subexpression Elimination (CSE). As an example of this, consider the program fragment shown in listing 4.1.1. Currently, this is translated into the expression trees shown in figure 4.1.2.

\[
x := a + b \\
y := a + b \\
z := x + y
\]

Listing 4.1.1. Pseudocode program with common subexpressions.

Figure 4.1.2. Expression tree representation of the program in listing 4.1.1 when translated naively.
Notice that the variables \(a\) and \(b\) are loaded twice when it should only be necessary to load them once. This is because the values of \(a\) and \(b\) do not change between execution of the first two program statements. Additionally, the expression \(a + b\) is evaluated twice. We want to reduce the expression tree to something resembling that shown in figure 4.1.3.

![Expression tree representation of the program in listing 4.1.1 with CSE.](image)

**Figure 4.1.3.** Expression tree representation of the program in listing 4.1.1 with CSE.

### 4.2 Implementing CSE

CSE can be applied either on a local or global scope. Local CSE looks at each basic block in isolation whereas global CSE looks at whole flow graph, taking into account the possible flow of data between basic blocks. Global CSE yields the best results in terms of optimisation, but uses sophisticated dataflow analysis techniques which take a relatively long time to perform. Thus it is commonly used in static compilers but not as often in JIT compilers as it increases start-up delay. We decided to only implement local CSE as it would still yield a performance increase in many cases, at less cost than global CSE.

#### 4.2.1 Caching Subexpression Results

As should be clear from figures 4.1.2 and 4.1.3, the problem is that new temporary values are created for every instruction even though an existing temporary is guaranteed to hold the same data at that point. To remedy this, we use a variation of the “value numbering” technique as described in [1]. When a new temporary value is created we store it in a hash table with a key consisting of the instruction and its operands (child subtrees in the expression tree). The operands are stored in the hash table as pointers to their corresponding LibJIT value objects. Thus, addresses of value objects are used in place of arbitrary numbers in the value numbering technique. As we generate code for each expression, the hash table is checked for an existing value that matches it; if one does not exist then it is created and added. An example of how this works for the PLUS instruction is shown in pseudocode in listing 4.2.1, superseding that shown in listing 3.6.5. Note that because addition is commutative, the expression is checked both ways around when querying the hash table.
operand1 := stack[sp-1]
operand2 := stack[sp-2]
key := {PLUS, addr(operand1), addr(operand2)}
result := lookup_table(key)
if no result found
    key' := {PLUS, addr(operand2), addr(operand1)}
    result := lookup_table(key')
    if no result found
        result := jit_insn_add(operand1, operand2)
        add_to_table(key, result)
    stack[sp-2] := result
    sp := sp – 2

Listing 4.2.1. Pseudocode for translating PLUS with CSE.

4.2.2 Invalidating Subexpressions

A problem occurs when the value of a variable changes. For example, if we assign the variable \( a \) to something new between the assignments of \( x \) and \( y \) as shown in listing 4.2.2, it is no longer safe to reuse the value of \( a \). The expression tree that we want in this case is depicted in figure 4.2.3. Note that at the assignment of \( y \) we can still use the previously loaded value of \( b \) as it has not changed since then. We invalidate an expression by simply removing its corresponding value from the hash table.

\[
x := a + b \\
a := 0 \\
y := a + b
\]

Listing 4.2.2. Modified pseudocode example demonstrating the need for CSE invalidation.

Figure 4.2.3. Expression tree representation of the program in listing 4.2.2 with CSE and showing invalidated values.
Therefore we identify possible situations in which the assignments of variables may change, and which parts of the expression trees must be invalidated in each case. This is a conservative approximation to a technique known as alias analysis.

1) Writing to a local or global variable with the `LOCAL / STORE` or `LDKW / STORE` instruction sequences. In this situation, a single variable changes, so we simply invalidate the value of the variable itself.

2) Writing to the target of a pointer. This will normally appear in the form of a `STORE` instruction, preceded by `PLUSA` (for calculating an array offset) or `LOAD` (for loading the contents of a pointer variable), but could be different in rare circumstances. It is impossible in this situation to determine exactly what was written to. Therefore, if we encounter a `STORE` instruction that does not appear in one of the patterns in 1), we invalidate all local and global values.

3) After a procedure call. All procedures have access to the same global variables, so we invalidate all values in the expression tree that depend on globals. A further complication arises when we allow nested procedures, as these have access to the inner procedure’s local variables. It is tricky to identify nested procedures at runtime in the current Keiko VM environment, so we must invalidate all local and global variables.

4) Writing to the result or link registers using `RESULTW` or `SAVELINK` respectively. In this case we simply invalidate all values in the expression tree that depend on them. In practice however, these registers are very rarely reused in the same procedure once they have been written to.

5) At the end of a basic block. All expressions are invalidated upon entering a new basic block. Unlike in global CSE, local CSE does not preserve any information across block boundaries. Without performing a detailed analysis of the flow graph, we cannot tell if an existing value is safe to use in another basic block.

4.3 Can We Do Better?

There are a wide variety of optimisation techniques that can be applied to program code. However, in a JIT compiler there is a trade-off to be made between runtime performance and start-up overhead and so not all optimisation techniques are suitable. In addition, it is often perceived to be the responsibility of the static compiler to perform the majority of optimisations, with the JIT simply applying tweaks at runtime.

Given that, it is certainly possible to do better. The CSE implementation described previously can be thought of as a write-through cache, where every write to a variable causes an immediate write to the underlying memory. We experimented briefly with the idea of extending this to implement a write-back cache, where value objects corresponding to variables are reused for writes as well as reads. This further complicates the invalidation process as cached variables that have been written to (dirty variables) need to be flushed back to memory under certain circumstances, such as before dereferencing a pointer or before leaving a block. Because of this, although nice results were produced in some cases, we would often end up with several needless writes back to memory at the end of every basic block. In addition, due to the limited number of general-purpose registers on the Intel x86 architecture, LibJIT would ‘spill’ variables out of registers so they would have to be accessed in memory anyway (on the C stack instead of the Oberon stack). These effects could be reduced or prevented by applying global CSE and dead code elimination techniques.
5

Testing and Evaluation

5.1 Testing for Correctness

The JIT runtime was tested incrementally as new sections were added. During the early stages it was configured to translate only a single test procedure, falling back to the interpreter for most of the program. By gradually increasing the complexity of the code in this test procedure we could test new instructions as they were added to the runtime. This also allowed us to ensure that the runtime handled interoperability between bytecode and native procedures correctly. During the later stages of implementation the interpreter was disabled completely.

The implementation of CSE required the most demanding tests for correctness. A number of small Oberon programs were written to test the scenarios described in section 4.2.2. Their output was compared with some expected output and the generated LibJIT intermediate code was inspected to find potential errors.

The existing Oberon runtime source code includes a suite of programs which tests a myriad of features of the Keiko VM. After completing the implementation as described in chapters 3 and 4, the JIT runtime was tested using this suite and a number of bugs were found. After fixing these, the runtime managed to successfully run 56 of the 107 test cases. This is more than was expected for the project, and included several non-trivial programs such as a Sudoku solver, an N-Queens solver and Don Knuth’s “Man or Boy” test! The vast majority of test cases that failed did so because of lack of support for double-precision floating point values (many of these pass if they are rewritten to use single-precision instead), passing dynamically sized arrays on the stack, and garbage collection. These features would have been time consuming to implement without illuminating anything additionally interesting for this project. However the runtime was implemented with these future additions in mind, so for example, adding a new LONGREAL type to the type inference algorithm should be trivial.

5.2 Quality of Code Generated

For a number of the test programs, the code generated by the JIT runtime was inspected and analysed to get a measure of its quality. This was also carried out as part of the implementation stage to find areas of code that could be improved with optimisations. In addition, the generated code for the example factorial program (listing 5.2.1) was compared with code produced by Mono (listing 5.2.2) and GCC (listing 5.2.3) for similar programs.
Listing 5.2.1. Assembly code listing for the factorial example, generated by the JIT runtime.

Listing 5.2.2. Assembly code listing for the factorial example, generated by Mono.
Listing 5.2.3. Assembly code listing for the factorial example, generated by GCC.

5.2.1 Analysis

It is immediately obvious that for this particular function the code output by Mono and GCC, at 19 and 12 instructions long respectively, is much smaller than that output by the JIT runtime, at 34 instructions long. As would be expected, the static GCC compiler produces the most heavily optimised code. It is interesting to note that it has transformed the program into a simple loop, eliminating the overhead associated with making a procedure call. This kind of optimisation (called tail recursion optimisation) is possible because the factorial program is written in tail recursive form. If tail recursion optimisation is disabled in GCC, the output code looks very similar to that produced by Mono.

We will concentrate on analysing the differences between the code generated by our JIT runtime and Mono (a JIT implementation of the CLI VM). A few key observations were made:

- The first three instructions in both deal with initialising the C stack frame. The next three instructions (lines 4 to 6) in our runtime output then load the base pointer for the Oberon stack (this is passed as an argument on the C stack, recall section 2.3.3) and with that, load the argument \( n \) from the Oberon stack. The Mono generated code omits this and simply reads its argument directly from the C stack.

- In our JIT runtime output, it takes 14 instructions (lines 16 to 29) to set up the stack frames and call a procedure. In contrast, the Mono output takes only 4 instructions (lines 13 to 16). This is again due to our use of the Oberon stack in addition to the C stack.

- Our JIT runtime output makes an additional write to memory when returning a value from a procedure (line 10), compared to the Mono output which just stores it in the EAX register (line 6). This is because we must return the procedure result in the Keiko VM result register (which turns out to be a location in memory in this case), whereas in the cdecl calling convention, procedures return values in the EAX register.

Overall, the JIT runtime output accesses memory in 15 separate instructions, compared with 4 for the Mono output. The majority of these are for accessing data on the Oberon stack. Due to the relatively long time it takes for memory accesses, this is likely to have a detrimental effect on performance. In order to improve this in some future implementation it would be necessary to omit the Oberon stack from native code, although this would make interoperability between interpreted bytecode and native procedures more difficult. In contrast, Mono does not need to worry about such interoperability. Also, the CLI VM is high-level enough that it does not rely on any specific stack layout so Mono is free to use the host machine stack entirely.
5.3 Comparison of Performance

5.3.1 Benchmark Method

The JIT runtime was informally benchmarked against the existing Oberon interpreter, the Mono runtime and the GCC C compiler. The benchmark program was written in Oberon, C# and C for each, respectively, and full optimisation options were enabled on all compilers. A pseudocode representation of the program is given in listing 5.3.1.

```plaintext
FUNCTION Factorise(INTEGER n)
    i <- 2;
    WHILE n > 1
        IF n MOD i = 0 THEN
            n <- n DIV i
        IF i * i >= n THEN
            i <- n;
        ELSE
            i <- i + 1;
        ENDIF
    ENDWHILE
FUNCTION Benchmark()
    FOR i <- 0 TO 1000000
        Factorise(LARGE_PRIME_NUMBER)
    ENDFOR
```

Listing 5.3.1. Pseudocode for a simple benchmark program.

5.3.2 Results

The benchmark was performed on a laptop with a 2GHz AMD Turion 64 CPU and 1GB RAM, running Fedora Core 11. Results were obtained using the Unix `time` command. A table of those obtained is given in figure 5.3.2.

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Time (s) (Mean of 10 observations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBC Interpreter</td>
<td>41.80</td>
</tr>
<tr>
<td>OBC JIT</td>
<td>4.740</td>
</tr>
<tr>
<td>Mono</td>
<td>4.191</td>
</tr>
<tr>
<td>GCC</td>
<td>4.183</td>
</tr>
</tbody>
</table>

Figure 5.3.2. Benchmark results.

5.3.3 Analysis

The results show a performance increase of close to a factor of 10 over interpretation when using JIT. Mono and GCC both show a further small improvement, most likely due to better code optimisation techniques and the differences discussed in section 5.2. However, the benchmarking method used here just gives us a rough idea and is not nearly rigorous enough to draw exact conclusions about how the Oberon JIT compares to Mono and GCC. A more thorough benchmarking experiment would consider many more details such as CPU performance, cache utilisation and effects on the OS memory system.
Conclusion

The results we see from the Oberon JIT translator are very encouraging and it is clear that the JIT approach has significant advantages over the interpreter. The runtime implements a large enough subset of the features of the Keiko VM to be useful even in non-trivial programs, and is designed in such a way that adding new features is a relatively easy task. In addition, although it was only tested on the x86 architecture, it should be straightforward to port to any 32-bit architecture that LibJIT supports. This means that we can take advantage of the best aspects of both interpreted and native code, portability and speed, which is the ultimate aim of JIT.

We also find that there is some room for improvement, specifically with regards to program optimisation in the Oberon compiler, the JIT runtime and even in LibJIT itself. It would certainly be worth conducting some experiments to determine which optimisations yield the greatest performance gains, whether they should be applied at compile time or runtime, and whether any benefit outweighs the potential start-up delay. We have also illuminated limitations in the Keiko VM, indicating that the lack of type information at runtime, combined with a relatively low-level instruction set and a “hard-wired” stack layout makes it difficult to produce the most optimal code.

In conclusion, we have presented a compelling case for the use of JIT translation techniques and we have seen that it has the potential to match the performance of native code. Given more research, faster hardware and greater demand for high-performance applications, we can expect to see even greater results in the future as these techniques evolve and mature.
Bibliography

5. MIKE SPIVEY: Oxford Oberon-2 compiler design documents, [http://spivey.oriel.ox.ac.uk/corner/Design_documents_for_OBC](http://spivey.oriel.ox.ac.uk/corner/Design_documents_for_OBC)

Acknowledgements

I would like to thank my project supervisor Mike Spivey for his help and guidance and for providing the Oberon source code on which this project was built.

Thanks also go to the guys in the freenode #libjit IRC channel for their help and support with using their excellent library.
9

Code Listing

9.1 analysis.c

```c
#include "obx.h"
#include "analysis.h"
#include "jit.h"
#include "basicblock.h"
#include "keiko.h"

#define HASH 32
#define MAX_STACK 256
#define V_TEMP 0

/* Module global variables */
static var symtab[HASH];
static stackitem vstack[MAX_STACK];
static int sp;
static int changed;
static int nret;

/* add_to_table -- adds a variable to the hashtable */
static void add_to_table(var v)
{
    unsigned index = ((HASH-v->id) + v->scope) % HASH;
    v->next = symtab[index];
    symtab[index] = v;
    changed = TRUE;
}

/* lookup_var -- looks up a variable in the hashtable */
var lookup_var(int id, int scope)
{
    unsigned index = ((HASH-id) + scope) % HASH;
    var v = symtab[index];
    while (v != NULL)
    {
        if (v->id == id && v->scope == scope)
            return v;
        v = v->next;
    }
    return NULL;
}

/* constrain_type -- updates a variables type information */
static int constrain_type(var v, int type)
{
    if (type != T_UNKNOWN && !(v->type & type))
    {
        v->type |= type;
        changed = TRUE;
    }
    return v->type;
}

/* analyse_instr -- infers variable information from
* a single bytecode instruction */
static void analyse_instr(instruction inst)
{
    var v;
}
```
if (dflag > 3)
    printf("analyse_instr - inst = %i\n", inst->inst);

// assumed preconditions and postconditions of the stack layout
// before and after each instruction are shown
switch (inst->inst)
{
    // pre: -
    // post: stack = [addr(local), ...]
    case I_LOCAL:
        v = lookup_var(inst->arg1, S_LOCAL);
        if (v == NULL)
        {
            v = zalloc(sizeof(struct var));
            v->id = inst->arg1;
            v->scope = S_LOCAL;
            v->is_param = inst->arg1 < 0 ? 0 : 1;
            v->val = NULL;
            add_to_table(v);
        }
        vstack[sp].address_of = 1;
        vstack[sp].v = v;
        vstack[sp].type = v->type;
        ++sp;
        break;

    // pre: -
    // post: stack = [v <- {addr(proc), addr(global), constant}, ...]
    case I_LDKW:
        v = lookup_var(inst->arg1, S_GLOBAL);
        if (v == NULL)
        {
            v = zalloc(sizeof(struct var));
            v->id = inst->arg1;
            v->scope = S_GLOBAL;
            v->type = T_UNKNOWN;
            add_to_table(v);
        }
        vstack[sp].v = v;
        vstack[sp].address_of = 0;
        vstack[sp].type = v->type;
        inst->v = v;
        ++sp;
        break;

    // pre: stack = [addr(variable), ...]
    // post: stack = [value(variable), ...]
    //       type(value(variable)) = type(variable)
    case I_LOADW: case I_LOADC: case I_LOADS:
        if (vstack[sp-1].address_of)
            vstack[sp-1].address_of = 0;
        else
            vstack[sp-1].type = constrain_type(vstack[sp-1].v, T_PTR);
        inst->v = vstack[sp-1].v;
        break;

    // pre: stack = [addr(variable), value, ...]
    // post: stack = [...]
    //       value(variable) = value
    case I_STOREW:
    case I_STOREC:
    case I_STORES:
        if (vstack[sp-2].v)
            constrain_type(vstack[sp-2].v, vstack[sp-1].type);
        if (vstack[sp-1].v)
            constrain_type(vstack[sp-1].v, vstack[sp-2].type | T_PTR);
        sp -= 2;
        break;

    // pre: -
    // post: stack = [int, ...]
    case I_CONST:
        vstack[sp].type = T_INT;
        vstack[sp].v = V_TEMP;
        vstack[sp].address_of = 0;

+++sp;
break;

// pre: stack = [offset, addr(variable), ...]
// post: stack = [ptr, ...]
case I_PLUSA:
    constrain_type(vstack[sp-2].v, vstack[sp-2].type | T_PTR);
    v = lookup_var(vstack[sp-2].v->id, S_ARRAY);
    if (v == NULL)
    {
        v = zalloc(sizeof(struct var));
        v->id = vstack[sp-2].v->id;
        v->scope = S_ARRAY;
        v->is_param = 0;
        v->val = NULL;
        add_to_table(v);
    }
    vstack[sp-1].address_of = 1;
    vstack[sp-1].v = v;
    vstack[sp-1].type = T_PTR;
    --sp;
    break;

// pre: stack = [float, ...]
// post: stack = [float, ...]
case I_FUMINUS:
    type = T_FLOAT;
    // fall through!
    // pre: stack = [int, ...]
    // post: stack = [int, ...]
case I_UMINUS:
    case I_BITNOT:
    case I_NOT:
        if (vstack[sp-1].v)
            constrain_type(vstack[sp-1].v, type);
        vstack[sp-1].type = type;
        vstack[sp-1].v = V_TEMP;
        vstack[sp-1].address_of = 0;
        break;

    // pre: stack = [float, float, ...]
    // post: stack = [float, ...]
case I_FPLUS: case I_FDIV: case I_FTIMES: case I_FMINUS: case I_FCMP:
    type = T_FLOAT;
    // fall through!
    // pre: stack = [int, int, ...]
    // post: stack = [int, int, ...]
case I_AND: case I_OR: case I_LEQ: case I_LT: case I_GEQ: case I_GT: case I_EQ:
    case I_NEQ: case I_PLUS: case I_TIMES: case I_MINUS: case I_DIV:
    case I_MOD: case I_BITAND: case I_BITOR: case I_BITXOR: case I_BITSUB: case I_LSL:
    case I_LSR:
        if (vstack[sp-1].v)
            constrain_type(vstack[sp-1].v, type);
        if (vstack[sp-2].v)
            constrain_type(vstack[sp-2].v, type);
        vstack[sp-2].type = type;
        vstack[sp-2].v = V_TEMP;
        vstack[sp-2].address_of = 0;
        --sp;
        break;

    // pre: stack = [int, ...]
    // post: stack = [float, ...]
case I_CONVNF:
        if (vstack[sp-1].v)
            constrain_type(vstack[sp-1].v, T_INT);
        vstack[sp-1].v = V_TEMP;
        vstack[sp-1].type = T_FLOAT;
        break;

    // pre: stack = [int, ...]
    // post: stack = [float, ...]
case I_CONVNC:
        if (vstack[sp-1].v)
            constrain_type(vstack[sp-1].v, T_INT);
        vstack[sp-1].v = V_TEMP;
        vstack[sp-1].type = T_FLOAT;
vstack[sp-1].address_of = 0;
vstack[sp-1].type = T_INT;
break;

// pre: stack = [uint, ...]
// post: stack = [uint, ...]
case I_CONVNS:
    if (vstack[sp-1].v)
        constrain_type(vstack[sp-1].v, T_INT);
vstack[sp-1].v = V_TEMP;
vstack[sp-1].address_of = 0;
vstack[sp-1].type = T_INT;
break;

// procedure call stuff
case I_JPROC:
    sp -= (inst->arg1 + 1);
break;

case I_SLIDEW:
    v = lookup_var(nret, S_RETURN);
    if (v == NULL)
        {
            v = zalloc(sizeof(struct var));
            v->scope = S_RETURN;
            v->id = nret;
            v->type = T_UNKNOWN;
            add_to_table(v);
        }
    inst->v = v;
vstack[sp].v = v;
vstack[sp].type = v->type;
vstack[sp].address_of = 0;
++sp;
++nret;
break;

case I_BOUND:
    --sp;
break;

// stack manipulation instructions
case I_DUP:
    memcpy(&vstack[sp], &vstack[sp-1], sizeof(stackitem));
++sp;
break;

case I_SWAP:
    memcpy(&s, &vstack[sp-1], sizeof(stackitem));
    memcpy(&vstack[sp-1], &vstack[sp-2], sizeof(stackitem));
    memcpy(&vstack[sp-2], &s, sizeof(stackitem));
break;

case I_POP:
    sp -= inst->arg1;
break;

// no type information can be inferred from these
case I_JLEQ: case I_JEQ: case I_JGT: case I_JLT: case I_JGEQ: case I_JNEQ:
    sp -= 2;
break;

case I_RESULTW:
    --sp;
break;

case I_RETURN: case I_ALIGNC: case I_ALIGNS: case I_LNUM:
break;

case I_ERROR:
break;

case I_TYPETEST:
break;

case I_BIT:
break;

case I_FIXCOPY:
    sp -= 3;
break;

case I_LINK:
    --sp;
break;
case I_SAVELINK:
    break;

default:
    panic("(analysis) unimplemented or illegal instruction \%s", instrs[inst->inst].i_name);
}
/* analyse_types -- infers the types of variables found in the function. *
 * Basically pseudo-executes the code to inspect the stack layout after each instruction. */
static void analyse_types()
{
    changed = TRUE;
    if (dflag > 2)
        printf("Begin analyzing types...\n");
// loop until no more type data can be inferred
while (changed)
{
    listnode l = blocks;
    basicblock b;
    sp = 0;
    nret = 0;
    changed = FALSE;
    while (l != NULL)
    {
        b = (basicblock)l->contents;
        // forward pass to identify variables
        listnode il = b->instrs;
        while (il != NULL)
        {
            analyse_instr((instruction)il->contents);
            il = il->next;
        }
        l = l->next;
    }

    // print debugging information
    if (dflag > 2)
    {
        printf("End analyzing types...\n");
        int i;
        for (i = 0; i < HASH; ++i)
        {
            var v = symtab[i];
            while (v != NULL)
            {
                printf("%s symbol id = %i, type = %i, is_param = %i\n", 
                        v->scope == S_LOCAL ? "local" : (v->scope == S_RETURN ? "return" : "global"), v->id, 
                        v->type, v->is_param);
                v = v->next;
            }
        }
    }
/* analyse_init -- initialises analysis data */
static void analyse_init()
{
    memset(symtab, 0, sizeof(var) * HASH);
}
/* peephole -- performs various peephole optimisations */
static void peephole()
{
    listnode l = blocks;
    basicblock b;
    while (l != NULL)
    {
        b = (basicblock) l->contents;
    }
listnode i = b->instrs;
listnode o = NULL;
while (i != NULL)
{
  instruction j = (instruction) i->contents;

  // OPTIMIZATION:
  // [SLIDEW, RESULTW]
  // => []
  if (j->inst == I_SLIDEW && i->next != NULL &&
      ((instruction)i->next->contents)->inst == I_RESULTW)
  {
    if (o != NULL)
      o->next = i->next->next;
    else
      b->instrs = i->next->next;
    if (i->next->next != NULL)
      i->next->next->prev = o;
  }
  o = i;
  i = i->next;
}

/* optimise -- perform optimisation and analysis on basic blocks */
void optimise()
{
  analyse_init();
  analyse_types();
  peephole();
}

9.2 analysis.h

#ifndef _ANALYSIS_H
#define _ANALYSIS_H
#include <jit/jit.h>

// data definitions
enum SCOPE {S_LOCAL, S_GLOBAL, S_RETURN, S_ARRAY};
enum TYPE {T_UNKNOWN = 0, T_INT = 1, T_FLOAT = 2, T_PTR = 4, T_PROC = 8, T_DOUBLE = 16};
typedef struct var
{
  int id;
  int scope;
  int type;
  unsigned is_param : 1;
  struct var * next;
  jit_value_t val;
} *var;
typedef struct stackitem
{
  var v;
  int type;
  unsigned address_of : 1;
} stackitem;

// function prototypes
void optimise();
var lookup_var(int id, int scope);
#endif /* _ANALYSIS_H */

39
#include "obx.h"
#include "basicblock.h"
#include "keiko.h"
#include "jit.h"

// size of codepoint hashtable
#define HASH 128

/* Global variables */
uchar * codelen;     // length of code, used to get end label
basicblock endblock;    // empty end basic block
basicblock entryblock;    // empty start basic block
listnode blocks = NULL;    // list of blocks in order that instructions are added

/* Module global variables */
static basicblock cblock;   // current block that instructions are added to
static int follow;     // used in block_add_instr
static codepoint hashtab[HASH];  // hashtable mapping addr -> codepoint

/* list_add -- add data to a doubly linked list */
void list_add(listnode * list, void * data)
{
    listnode n = *list;
    listnode p = n;
    if (*list == NULL)
    {
        *list = (listnode) zalloc(sizeof(struct listnode));
        (*list)->contents = data;
    }
    else
    {
        while ((n = p->next) != NULL)
        {
            p = n;
        }
        n = zalloc(sizeof(struct listnode));
        p->next = n;
        n->prev = p;
        n->contents = data;
    }
}

/* instruction_create -- creates an instruction with the specified data */
static instruction instruction_create(int inst, int arg1, int arg2)
{
    instruction i = zalloc(sizeof(struct instruction));
    i->inst = inst;
    i->arg1 = arg1;
    i->arg2 = arg2;
    return i;
}

/* block_visited -- sets the visited value of every block */
void block_visit_all(int visited)
{
    basicblock b;
    listnode l = blocks;
    while (l != NULL)
    {
        b = (basicblock) l->contents;
        b->visited = visited;
        l = l->next;
    }
}

/* block_create -- creates a new empty basic block */
basicblock block_create()
{
    basicblock b = (basicblock) zalloc(sizeof(struct basicblock));
    b->instrs = b->preds = NULL;
    b->cjump = b->jump = NULL;
    b->label = jit_label_undefined;
    b->visited = FALSE;
    return b;
}
/* block_init -- Initialises basic block creation data */
void block_init()
{
    entryblock = NULL;
    endblock = NULL;
    cblock = NULL;
    blocks = NULL;
}

/* block_create_entry -- Creates the entry block */
void block_create_entry()
{
    entryblock = block_create();
    cblock = block_create();
    entryblock->jump = cblock;
    list_add(&cblock->preds, entryblock);
}

/* block_create_end -- Create the end block */
void block_create_end()
{
    basicblock b;
    listnode l = blocks;
    if (endblock == NULL)
    {
        endblock = block_create();
        (lookup(codelen, FALSE))->b = endblock;
    }

    // point blocks that 'fall off the edge'
    // to jump to our end block
    while (l != NULL)
    {
        b = (basicblock) l->contents;
        if (!b->jump)
        {
            b->jump = endblock;
            list_add(&endblock->preds, b);
        }
        l = l->next;
    }
}

/* block_add_instr -- Adds a single instruction to the current block */
void block_add_instr(uchar * pc, int inst, int arg1, int arg2)
{
    basicblock n;

    // if instruction is target of a branch, create new block
    codepoint p = lookup(pc, FALSE);
    if (p != NULL)
    {
        // if current block is empty, use it, otherwise new block
        if (p->b == NULL)
            p->b = cblock->instrs == NULL ? cblock : block_create();

        // if the previous block is a predecessor,
        // link this new one to it
        if (!follow)
        {
            cblock->jump = p->b;
            list_add(&((p->b)->preds), cblock);
        }
    }

    // empty block, add it anyway, redundant
    // branching should be optimized out later
    if (!cblock->visited)
    {
        list_add(&blocks, cblock);
        cblock->visited = TRUE;
    }
    cblock = p->b;
}

    // add instruction to block, unless it is an unconditional jump
    if (inst != I_JUMP)
    {
        // add block to list - we never return to
// a block after leaving it, so this list
// will have the blocks completed in order
if (!cblock->visited)
{
    list_add(&blocks, cblock);
    cblock->visited = TRUE;
}
list_add(&cblock->instrs), instruction_create(inst, arg1, arg2));

switch (inst)
{
    // conditional jumps, can either
    // branch or pass through
    case I_JLEQ: case I_JEQ: case I_JGT: case I_JLT: case I_JGEQ: case I_JNEQ:
    case I_JLTZ: case I_JGTZ: case I_JLEQZ: case I_JGEQZ: case I_JUMPF:
    case I_JUMPT: case I_JEQZ: case I_JNEQZ:
        n = block_create();
        p = lookup(arg1, FALSE);
        if (p->b == NULL)
            p->b = block_create();
        cblock->cjump = p->b;
        cblock->jump = n;
        list_add(&n->preds, cblock);
        list_add(&(p->b)->preds), cblock);
        cblock = n;
        follow = TRUE;
        break;
    // returning from function, branch
    // to end of function label
    case I_RETURN:
        p = lookup(codelen, FALSE);
        if (p->b == NULL)
            endblock = block_create();
        p->b = endblock;
        follow = FALSE;
        break;
    // unconditional jump
    case I_JUMP:
        p = lookup(arg1, FALSE);
        if (p->b == NULL)
            p->b = block_create();
        cblock->jump = p->b;
        list_add(&(p->b)->preds), cblock);
        follow = FALSE;
        break;
    default:
        follow = TRUE;
        break;
}

/* write_block -- Prints block information for debugging purposes */
void write_block(basicblock b)
{
    printf("*** BASIC BLOCK START ***\n");
    printf("addr = 0x%X, jump = 0x%X, cjump = 0x%X\n", (int)b,
            b->jump != NULL ? b->jump : -1, b->cjump != NULL ? b->cjump : -1);
    printf("*** INSTRUCTIONS ***\n");
    listnode l = b->instrs;
    while (l != NULL)
    {
        instruction i = (instruction) l->contents;
        printf("%s\n", instrs[i->inst].i_name);
        l = l->next;
    }
    printf("*** PREDECESSORS ***\n");
    l = b->preds;
    while (l != NULL)
{  
  basicblock i = (basicblock) l->contents;
  l = l->next;
  
  printf("*** BASIC BLOCK END ***\n");
}

/* lookup -- looks up a codepoint in the hashtable for a * particular address, or creates one if it does not exist */
codepoint lookup(int addr, bool create)
{
  unsigned int h = addr % HASH;
  codepoint p;
  
  for (p = hashtab[h]; p != NULL; p = p->l_hlink)
    if (p->l_lab == addr) return p;

  if (create)
    {
      p = (codepoint) zalloc(sizeof(struct codepoint));
      p->l_lab = addr;
      p->l_hlink = hashtab[h];
      p->b = NULL;
      hashtab[h] = p;
    }
  return p;
}

/* mark_lab -- create a label at the specified address */
void mark_lab(int addr)
{
  if (dflag > 1)
    printf("Mark %d\n", addr);
  lookup(addr, TRUE);
}

/* map_labels -- determine branch targets in a bytecode routine */
void map_labels(uchar *pc0, uchar *end)
{
  uchar *pc; int i; char *s;

  // initialise hashtable
  memset(hashtab, 0, HASH * sizeof(codepoint));

  for (pc = pc0; pc < end; )
    {
      int op = *pc;
      uchar *pc1 = pc+1;
      struct decode *d = &decode[op];

      for (s = d->d_patt; *s != '\0'; s++)
        {
          switch (*s)
            {
            case '1':
              pc1++;
              break;
            case '2':
              pc1 += 2;
              break;
            case 'R':
              mark_lab(get2(pc1)+(pc-pc0));
              pc1 += 2;
              break;
            case 'S':
              mark_lab(get1(pc1)+(pc-pc0));
              pc1 += 1;
              break;
            case 'N':
              break;
            default:
              panic("*bad pattern char %c", *s);
            }
        }
      pc += d->d_len;
    }

  if (op == K_JCASE_1)
int n = pc[-1];
for (i = 0; i < n; i++)
{
    mark_lab(get2(pc)+(pc-pc0));
    pc += 2;
}

// label for end of function:
codelen = end-pc0;
mark_lab(codelen);

9.4 basicblock.h

#ifndef _BASICBLOCK_H
#define _BASICBLOCK_H
#include <jit/jit.h>
#include "analysis.h"

typedef struct basicblock *basicblock;
typedef struct listnode *listnode;
typedef struct instruction *instruction;
typedef struct codepoint *codepoint;
typedef struct threeaddr *threeaddr;

struct codepoint
{
    int l_lab; /* Bytecode address */
    codepoint l_hlink; /* Next label in hash chain */
    basicblock b; // basicblock starting at label
};

struct basicblock
{
    int visited; // used for data-flow analysis
    listnode preds; // list of predecessor blocks
    listnode instrs; // list of instructions
    jit_label_t label; // libjit label corresponding to block
    basicblock jump; // next block for unconditional jump or if condition is false
    basicblock cjump; // block to branch to if condition is true
    void * data; // used for data-flow analysis
};

struct instruction
{
    int inst; // instruction opcode
    int arg1; // first argument
    int arg2; // second argument
    var v; // symbol referred to by instruction (if any)
};

struct listnode
{
    void *contents; // containing data
    listnode next; // previous in list
    listnode prev; // next in list
};

// block functions
void block_init();
void block_visit_all(int visited);
basicblock block_create();
void block_create_entry();
void block_create_end();
void block_add_instr(uchar * pc, int inst, int arg1, int arg2);
void write_block(basicblock b);

// linked list functions
void list_add(listnode * list, void * data);

// labels functions
void map_labels(uchar *pc0, uchar *end);
void mark_lab(int addr);
codepoint lookup(int addr, bool create);
9.5  jit.c

#include "obx.h"
#include "jit.h"
#include "keiko.h"
#include "basicblock.h"
#include "analysis.h"
#include <jit/jit.h>

// for debugging
FILE * dfile = NULL;

/* Macros for generating common instruction sequences. */
#define lcond(op)                     
    temp = jit_insn_##op(func, jstack[sp-2].v, jstack[sp-1].v);         
    goto do_branch_insn
#define ibinop(op, commute)                  
    jstack[sp-2].e = lookup_instr(inst, 0, jstack[sp-2].e, jstack[sp-1].e, TRUE, (commute));  
    if (jstack[sp-2].e->val == NULL)                
    {                        
        jstack[sp-2].e->val = jit_insn_##op(func, jstack[sp-2].v, jstack[sp-1].v);     
    }                        
    jstack[sp-2].v = jstack[sp-2].e->val;               
    --sp
#define iboolop(op, commute) ibinop(op, commute)
#define iunop(op)                     
    jstack[sp-1].e = lookup_instr(inst, 0, jstack[sp-1].e, NULL, TRUE, FALSE);      
    if (jstack[sp-1].e->val == NULL)                
    {                       
        jstack[sp-1].e->val = jit_insn_##op(func, jstack[sp-1].v);        
    }                        
    jstack[sp-1].v = jstack[sp-1].e->val
#define iuboolop(op) iunop(op)
#define iload(type)                    
    jstack[sp-1].e = lookup_instr(inst, 0, jstack[sp-1].e, NULL, TRUE, FALSE);      
    if (jstack[sp-1].e->val == NULL)                
    {                       
        jstack[sp-1].e->val = jit_insn_load_relative(func, jstack[sp-1].v, 0, jit_type_##type); 
    }                        
    jstack[sp-1].v = jstack[sp-1].e->val

/* Definitions for compile-time data types */

struct inst_entry;

typedef struct inst_entry
{
    int op;
    int p1;
    struct inst_entry * c1;
    struct inst_entry * c2;
    jit_value_t val;
    struct inst_entry * next;
} *inst_entry;

#define MAX_STACK 256 // maximum size of the compile-time stack
#define IHASH 256   // maximum number of CSEs in hash table

// contains information about a stack entry
typedef struct stack_entry
{
    jit_value_t v;  // the value in the stack entry
inst_entry e; // instruction this entry is a result of
int type;  // the type of the value
}

/* Various module global variables */
static int sp;      // stack pointer for compile time stack
static value * context;    // cp for the current procedure
static jit_context_t jit_context; // libJIT context
static jit_function_t func;   // current libJIT function being built
static jit_value_t bp;    // function base pointer
static jit_value_t kp;    // constant pool base pointer
static jit_value_t result;   // function return value
static jit_value_t staticlink;  // static link for a frame
static jit_value_t bits;   // base of bit array (for BIT instruction)

// signature for a native procedure
static jit_type_t signature = NULL;

// params for a native procedure
static jit_type_t params[1];

static stack_entry jstack[MAX_STACK]; // compile time stack used for the JIT
static inst_entry inst_hashtab[IHASH];

/* Memory pool variables and functions */

// memory for constructing basic blocks
// (used in analysis.c and basicblock.c)
#define MAPSIZE (16384*4)
static uchar *zpool, *zfree, *zlimit;
void * zalloc(int size) {
    void *p = (void *) zfree;
    zfree += size;
    if (zfree > zlimit) panic("out of patch memory");
    return p;
}

// memory for CSE hash table in this module
#define HTSIZE (256*4*sizeof(struct inst_entry))
static uchar *hpool, *hfree, *hlimit;
static void * halloc(int size) {
    void *p = (void *) hfree;
    hfree += size;
    if (hfree > hlimit) panic("out of hash table memory");
    return p;
}

/* lookup_instr -- looks up a CSE in the hash table, or creates it if not found */
static inst_entry lookup_instr(int op, int p1, inst_entry c1, inst_entry c2, bool create, bool commute) {
    // compute hashes for instruction operands
    unsigned h = ((unsigned)(op + p1 * 2 + ((int)c1) * 4 + ((int)c2) * 8)) % IHASH;
    unsigned h2 = ((unsigned)(op + p1 * 2 + ((int)c1) * 8 + ((int)c2) * 4)) % IHASH;
    inst_entry p;
    for (p = inst_hashtab[h]; p != NULL; p = p->next)
        if (p->op == op && p->p1 == p1 && p->c1 == c1 && p->c2 == c2 && p->val != NULL)
            return p;
    // commutative instruction, so order of operands does not matter
    if (commute)
        for (p = inst_hashtab[h2]; p != NULL; p = p->next)
            if (p->op == op && p->p1 == p1 && p->c2 == c2 && p->c1 == c1 && p->val != NULL)
                return p;
    // create a new CSE entry
    if (create)
        p = (inst_entry) halloc(sizeof(struct inst_entry));
        p->op = op;
        p->p1 = p1;
        p->c1 = c1;
        p->c2 = c2;
        p->next = inst_hashtab[h];
inst_hashtab[h] = p;
} return p;
}

/* invalidate_cses -- invalidates CSEs depending on the instruction that was just executed */
static void invalidate_cses(basicblock b, instruction insn, int inst, int arg1, int arg2)
{
    stack_entry dest = jstack[sp-1];
    int i;
    inst_entry p;
    bool changed;

    if (dest.e->op == I_LOCAL || dest.e->op == I_LDKW)
    {
        // invalidate CSEs that depend on the local
        // or global that was just written to
        for (i = 0; i < IHASH; ++i)
        {
            p = inst_hashtab[i];
            while (p != NULL)
            {
                if ((p->op == dest.e->op && p->p1 == dest.e->p1 && p->val != NULL) ||
                    (p->c1 != NULL && p->c1->val == NULL && p->val != NULL) ||
                    (p->c2 != NULL && p->c2->val == NULL && p->val != NULL))
                {
                    p->val = NULL;
                }
                p = p->next;
            }
        }
    }
    else // we can’t be sure what has been written to, so invalidate everything
    {
        for (i = 0; i < IHASH; ++i)
        {
            p = inst_hashtab[i];
            while (p != NULL)
            {
                p->val = NULL;
                p = p->next;
            }
        }
    }
}

/* invalidate_allGlobals -- invalidates all CSEs which depend on
* global values. Called after a procedure call, as globals may
* have been modified in the called procedure. */
static void invalidate_allGlobals()
{
    bool changed = TRUE;
    int i;
    inst_entry p;
    for (i = 0; i < IHASH; ++i)
    {
        p = inst_hashtab[i];
        while (p != NULL)
        {
            if ((p->op == I_LDKW && p->val != NULL) ||
                (p->c1 != NULL && p->c1->val == NULL && p->val != NULL) ||
                (p->c2 != NULL && p->c2->val == NULL && p->val != NULL))
            {
                p->val = NULL;
            }
            p = p->next;
        }
    }
}

#ifdef JIT_EXCEPTIONS
value * runtime_bp;    // function base pointer
jit_value_t jit_runtime_bp;  // stores address of runtime_bp
int line;      // current line number
jit_value_t jit_line;   // stores address of line

/* obc_jit_exception_handler -- called when an exception is thrown,
* or when a runtime error occurs. */
void * obc_jit_exception_handler(int ec)
switch (ec) {
    case -2: // libJIT division-by-zero
        runtime_error(E_DIV, line, runtime_bp, 0);
        break;
    default: // runtime error
        runtime_error(ec, line, runtime_bp, 0);
        break;
}

// should never reach here, but anyway
return NULL;
}
#endif

/* instr -- translates a single bytecode instruction into
* the appropriate libJIT representation. */
static void instr(basicblock b, instruction insn, int inst, int arg1, int arg2)
{
    jit_value_t temp;

    switch (inst) {
        /* Instructions for loading values from memory */
        case I_LOCAL:
            jstack[sp].e = lookup_instr(inst, arg1, NULL, NULL, TRUE, FALSE);
            if (jstack[sp].e->val == NULL)
                jstack[sp].e->val = jit_insn_add_relative(func, bp, arg1);
            jstack[sp].v = jstack[sp].e->val;
            jstack[sp].type = T_PTR;
            ++sp;
            break;
        case I_LDKW:
            jstack[sp].e = lookup_instr(inst, arg1, NULL, NULL, TRUE, FALSE);
            if (jstack[sp].e->val == NULL)
            {
                if (insn->v->type & T_FLOAT && !(insn->v->type & T_PTR))
                {
                    jstack[sp].e->val = jit_insn_load_relative(func, kp, arg1 * sizeof(int), jit_type_float32);
                    jstack[sp].type = T_FLOAT;
                }
                else
                {
                    jstack[sp].e->val = jit_insn_load_relative(func, kp, arg1 * sizeof(int), jit_type_int);
                    jstack[sp].type = T_INT;
                }
            }
            jstack[sp].v = jstack[sp].e->val;
            ++sp;
            break;
        case I_LOADW:
            // load either as a floating point value
            // or integer, depending on what we’ve
            // managed to infer the type as.
            if (insn->v->type & T_FLOAT)
            {
                iload(float32);
                jstack[sp-1].type = T_FLOAT;
            }
            else
            {
                iload(int);
                jstack[sp-1].type = T_INT;
            }
            break;
        case I_LOADC:
            iload(ubyte);
            jstack[sp-1].type = T_INT;
            break;
        case I_LOADS:
            iload(sizeof(int));
            jstack[sp-1].type = T_INT;
            break;
        /* Instructions for storing values into memory. */
        case I_STOREW:
            break;
    }
}

/* instr -- translates a single bytecode instruction into
* the appropriate libJIT representation. */
static void instr(basicblock b, instruction insn, int inst, int arg1, int arg2)
{
    jit_value_t temp;

    switch (inst) {
        /* Instructions for loading values from memory */
        case I_LOCAL:
            jstack[sp].e = lookup_instr(inst, arg1, NULL, NULL, TRUE, FALSE);
            if (jstack[sp].e->val == NULL)
                jstack[sp].e->val = jit_insn_add_relative(func, bp, arg1);
            jstack[sp].v = jstack[sp].e->val;
            jstack[sp].type = T_PTR;
            ++sp;
            break;
        case I_LDKW:
            jstack[sp].e = lookup_instr(inst, arg1, NULL, NULL, TRUE, FALSE);
            if (jstack[sp].e->val == NULL)
            {
                if (insn->v->type & T_FLOAT && !(insn->v->type & T_PTR))
                {
                    jstack[sp].e->val = jit_insn_load_relative(func, kp, arg1 * sizeof(int), jit_type_float32);
                    jstack[sp].type = T_FLOAT;
                }
                else
                {
                    jstack[sp].e->val = jit_insn_load_relative(func, kp, arg1 * sizeof(int), jit_type_int);
                    jstack[sp].type = T_INT;
                }
            }
            jstack[sp].v = jstack[sp].e->val;
            ++sp;
            break;
        case I_LOADW:
            // load either as a floating point value
            // or integer, depending on what we’ve
            // managed to infer the type as.
            if (insn->v->type & T_FLOAT)
            {
                iload(float32);
                jstack[sp-1].type = T_FLOAT;
            }
            else
            {
                iload(int);
                jstack[sp-1].type = T_INT;
            }
            break;
        case I_LOADC:
            iload(ubyte);
            jstack[sp-1].type = T_INT;
            break;
        case I_LOADS:
            iload(sizeof(int));
            jstack[sp-1].type = T_INT;
            break;
        /* Instructions for storing values into memory. */
    }
    jit_insn_store_relative(func, jstack[sp-1].v, 0, jstack[sp-2].v);
    goto do_store_insn;

    case I_STORES:
        temp = jit_insn_convert(func, jstack[sp-2].v, jit_type_short, 0);
        jit_insn_store_relative(func, jstack[sp-1].v, 0, temp);
        goto do_store_insn;

    case I_STOREC:
        temp = jit_insn_convert(func, jstack[sp-2].v, jit_type_ubyte, 0);
        jit_insn_store_relative(func, jstack[sp-1].v, 0, temp);
        // stores fall through to here!
    do_store_insn:
        // invalidate common subexpressions that
        // depend on the value we just stored to.
        invalidate_cses(b, insn, inst, arg1, arg2);
        sp -= 2;
        break;

    case I_RESULTW:
        jit_insn_store_relative(func, result, 0, jstack[sp-1].v);
        --sp;
        break;

    case I_SLIDEW:
        jstack[sp].e = NULL;
        // store value as either a float or integer, depending
        // on what we’ve inferred the result type as.
        if (insn->v->type & T_FLOAT)
            jstack[sp++].v = jit_insn_load_relative(func, result, 0, jit_type_float32);
        else
            jstack[sp++].v = jit_insn_load_relative(func, result, 0, jit_type_int);
        break;

    /* Arithmetic and data manipulation instructions. */
    case ICONST:
        jstack[sp].e = lookup_instr(inst, arg1, NULL, NULL, TRUE, FALSE);
        if (jstack[sp].e->val == NULL)
            jstack[sp].e->val = jit_value_create_nint_constant(func, jit_type_int, arg1);
        jstack[sp].v = jstack[sp].e->val;
        ++sp;
        break;

    case IPLUS:
    case IPLUSA:
        ibinop(add, TRUE);
        jstack[sp-2].type = T_INT;
        break;

    case IFPLUS:
        ibinop(add, TRUE);
        jstack[sp-2].type = T_FLOAT;
        break;

    case ITIMES:
        ibinop(mul, TRUE);
        jstack[sp-2].type = T_INT;
        break;

    case IFTIMES:
        ibinop(mul, TRUE);
        jstack[sp-2].type = T_FLOAT;
        break;

    case I_MINUS:
        ibinop(sub, FALSE);
        jstack[sp-2].type = T_INT;
        break;

    case IFMINUS:
        ibinop(sub, FALSE);
        jstack[sp-2].type = T_FLOAT;
        break;

    case IDIV:
        ibinop(div, FALSE);
        jstack[sp-2].type = T_INT;
        break;

    case IFDIV:
        ibinop(div, FALSE);
        jstack[sp-2].type = T_FLOAT;
        break;

    case I_MOD:
        ibinop(rem, FALSE);

49
jstack[sp-2].type = T_INT;
break;
case I_UMINUS:
lunop(neg);
jstack[sp-1].type = T_INT;
break;
case I_FUMINUS:
lunop(neg);
jstack[sp-1].type = T_FLOAT;
break;
case I_LSL:
ibinop(shl, FALSE);
jstack[sp-2].type = T_INT;
break;
case I_LSR:
ibinop(shr, FALSE);
jstack[sp-2].type = T_INT;
break;

// bitwise ops
case I_BITAND:
ibinop(and, TRUE);
jstack[sp-2].type = T_INT;
break;
case I_BITOR:
ibinop(or, TRUE);
jstack[sp-2].type = T_INT;
break;
case I_BITXOR:
ibinop(xor, TRUE);
jstack[sp-2].type = T_INT;
break;
case I_BITSUB:
ibinop(sub, FALSE);
jstack[sp-2].type = T_INT;
break;
case I_BITNOT:
lunop(not);
jstack[sp-2].type = T_INT;
break;

// boolean ops
// Pretty much identical to bitwise ops, the runtime
// does not distinguish between ints and bools at the
// present.
case I_AND:
iboolop(and, TRUE); break;
case I_OR:
iboolop(or, TRUE); break;
case I_NOT:
iboolop(not); break;
case I_LEQ:
iboolop(le, FALSE); break;
case I_LT:
iboolop(lt, FALSE); break;
case I_GEQ:
iboolop(ge, FALSE); break;
case I_GT:
iboolop(gt, FALSE); break;
case I_EQ:
iboolop(eq, FALSE); break;
case I_NEQ:
iboolop(ne, FALSE); break;

/* Instructions for branching. */
* I_JUMP does not feature here as it is eliminated
* in the basic block phase. Explicit jumps are added
* between linked basic blocks which are then (usually)
* optimised out by libJIT. */
case I_JLEQ: icondj(le); break;
case I_JEQ: icondj(eq); break;
case I_JGT: icondj(gt); break;
case I_JLT: icondj(lt); break;
case I_JGEQ: icondj(ge); break;
case I_JNEQ: icondj(ne); break;

// no fall through, we goto here!
do_branch_insn:
jit_insn_branch_if(func, temp, &b->cjump->label);
sp -= 2;
break;

case I_JUMP:
    // unconditional jump, should never get here
    // as these are eliminated when constructing
    // basic blocks
    break;

/* Instructions for manipulating the stack. */
case I_DUP:
    jstack[sp] = jstack[sp-arg1-1];
    ++sp;
    break;

case I_SWAP:
    jstack[sp] = jstack[sp-1];
    jstack[sp-1] = jstack[sp-2];
    jstack[sp-2] = jstack[sp];
    break;

case I_POP:
    sp -= arg1;
    break;

// floating point stuff

case I_FCMP:
    jstack[sp-2].v = jit_insn_cmpl(func, jstack[sp-2].v, jstack[sp-1].v);
    --sp;
    break;

case I_CONVNF:
    jstack[sp-1].v = jit_insn_convert(func, jstack[sp-1].v, jit_type_float32, FALSE);
    break;

// other conversion instructions

case I_CONVNC:
    jstack[sp-1].v = jit_insn_convert(func, jstack[sp-1].v, jit_type_ubyte, FALSE);
    break;

case I_CONVNS:
    jstack[sp-1].v = jit_insn_convert(func, jstack[sp-1].v, jit_type_short, FALSE);
    break;

/* Procedure call instructions. */
case I_JPROC:
    temp = jit_insn_dup(func, bp);
    int offset = -((context[CP_FRAME].i + 1) * sizeof(int)); // jump over locals

    // copy arguments from our compile-time stack to runtime oberon stack, in reverse order
    int i = arg1;
    for (; i > 0; --i, offset -= sizeof(int))
    {
        if (jstack[sp-i-1].type == T_FLOAT)
            jit_insn_store_relative(func, temp, offset, jit_insn_convert(func, jstack[sp-i-1].v, jit_type_float32, 0));
        else
            jit_insn_store_relative(func, temp, offset, jit_insn_convert(func, jstack[sp-i-1].v, jit_type_nint, 0));
    }

    // push CP, PC, BP onto stack
    jit_insn_store_relative(func, temp, offset, jstack[sp-1].v);
    jit_insn_store_relative(func, temp, offset-sizeof(int), jit_value_create_nint_constant(func, jit_type_nint, 0));
    jit_insn_store_relative(func, temp, offset-sizeof(int)*2, bp);
    temp = jit_insn_add(func, temp, jit_value_create_nint_constant(func, jit_type_nint, offset-sizeof(int)*2));

    // get address of the procedure and call it
    jit_value_t addr = jit_insn_load_relative(func, jstack[sp-1].v, CP_PRIM * sizeof(value*), jit_type_void_ptr);
    jit_insn_call_indirect(func, addr, signature, &temp, 1, 0);

    // pop arguments and callee CP off the stack
    sp -= (arg1 + 1);

    // invalidate CSEs based on globals, as these may have changed in the function call
    invalidate_all_globals();

51
break;

case I_RETURN:
  jit_insn_branch(func, &endblock->label);
  break;

// Static link instructions for nested procedures.
case I_LINK:
  jit_insn_store_relative(func, staticlink, 0, jstack[sp-1].v);
  --sp;
  break;

case I_SAVELINK:
  temp = jit_insn_load_relative(func, staticlink, 0, jit_type_void_ptr);
  jit_insn_store_relative(func, bp, SL * sizeof(int), temp);
  break;

/* Instructions for runtime error checking, *
 * exceptions and diagnostics. */
case I_BOUND:
  --sp;
  break;

case I_LNUM:
  #ifdef JIT_EXCEPTIONS
    temp = jit_value_create_nint_constant(func, jit_type_int, arg1);
    jit_insn_store_relative(func, jit_line, 0, temp);
  #endif
  break;

case I_ERROR:
  #ifdef JIT_EXCEPTIONS
    temp = jit_value_create_nint_constant(func, jit_type_int, arg2);
    jit_insn_store_relative(func, jit_line, 0, temp);
    jit_value_t e_args[1];
    jit_type_t e_types[1];
    e_args[0] = jit_value_create_nint_constant(func, jit_type_int, arg1);
    e_types[0] = jit_type_int;
    jit_type_t e_sig = jit_type_create_signature(jit_abi_cdecl, jit_type_void, e_types, 1, 0);
    jit_insn_call_native(func, "obc_jit_exception_handler",
                         obc_jit_exception_handler, e_sig, e_args, 1, 0);
  #endif
  break;

case I_TYPETEST:
  ; jit_value_t n, d2, t1, t2, t3, t4;
  n = jit_value_create_nint_constant(func, jit_type_nint, arg1);
  d2 = jit_insn_dup(func, jstack[sp-1].v);
  t1 = jit_insn_load_relative(func, jstack[sp-2].v, DESC_DEPTH * sizeof(int), jit_type_int);
  t2 = jit_insn_load_relative(func, jstack[sp-2].v, (DESC_ANCES + arg1) * sizeof(int), jit_type_int);
  t3 = jit_insn_convert(func, jit_insn_to_bool(func, jit_insn_ge(func, t1, n)), jit_type_ubyte, FALSE);
  t4 = jit_insn_convert(func, jit_insn_to_bool(func, jit_insn_eq(func, t2, d2)), jit_type_ubyte, FALSE);
  jstack[sp-2].e = NULL;
  jstack[sp-2].v = jit_insn_and(func, t3, t4);
  --sp;
  break;

/* Miscellaneous instructions. */
// no-ops on i386
case I_ALIGNC:
  case I_ALIGNS:
  case I_NOP:
    break;

case I_BIT:
  jstack[sp-1].e = lookup_instr(inst, 0, jstack[sp-1].e, NULL, TRUE, FALSE);
  if (jstack[sp-1].e->val == NULL)
    { jstack[sp-1].e->val = jit_insn_load_elem(func, bits, jstack[sp-1].v, jit_type_uint); }
  jstack[sp-1].v = jstack[sp-1].e->val;
  break;

case I_FIXCOPY:
  // calls the native jit_memcpy function.
  ; jit_value_t args[3];
  jit_type_t types[3];
  int o;
  for (o = 0; o < 3; ++o)
{ args[_o] = jit_insn_dup(func, jstack[(sp-3)+o].v);
  types[_o] = jit_type_void_ptr;
}

  types[2] = jit_type_int;
  jit_type_t signature = jit_type_create_signature(jit_abi_cdecl, jit_type_int, types, 3, 1);
  jstack[sp-3].v = jit_insn_call_native(func, "jit_memcpy", jit_memcpy, signature, args, 3, 0);
  sp -= 3;
  break;
}
default:
  panic("(jit) unimplemented or illegal instruction %s", instrs[inst].i_name);
}
/* unpack_instr -- expand a packed bytecode instruction. */
static void unpack_instr(uchar * pc, int inst, int arg1, int arg2, int lev)
{
  int *equiv = instrs[inst].i_equiv;
  int i;
  if (equiv[0] == 0)
  {
    // add this instruction to the current basic block
    block_add_instr(pc, inst, arg1, arg2);
  }
  else
  {
    // recursively unpack this instruction
    for (i = 0; equiv[i] != 0; i++)
    {
      int e = equiv[i];
      int arg = 0;
      if (e&BARG)
        arg = arg1;
      else if (e&BICON)
        arg = equiv[++i];
      if (dflag > 1)
      {
        printf("%.*s%s %d
", lev, "++++++++++",
                 instrs[e&IMASK].i_name, arg);
      }
      unpack_instr(pc, e&IMASK, arg, 0, lev+1);
    }
  }
}
/* create_blocks -- constructs a basic block representation of the program. */
static void create_blocks(uchar * pc_start, uchar * pc_end)
{
  uchar * pc;
  char * s;
  // create entry block
  block_init();
  block_create_entry();
  for (pc = pc_start; pc < pc_end; )
  {
    int op = *pc;
    uchar * pc1 = pc + 1;
    struct decode * d = &decode[op];
    int args[2] = {0, 0};
    int nargs = 0;
    for (s = d->d_patt; *s != '\0'; s++)
    {
      switch (*s)
      {
        case '1':
          args[nargs++] = get1(pc1); pc1++;
          break;
        case '2':
          args[nargs++] = get2(pc1); pc1 += 2;
          break;
        case 'R':
          args[nargs++] = get2(pc1)+(pc-pc_start); pc1 += 2;
          break;
      }
      if (dflag > 1)
      {
        printf("%.*s%s %d
", lev, "++++++++++",
                 instrs[e&IMASK].i_name, arg);
      }
    }
  }
}
case 'S':
    args[nargs++] = get1(pc1)+(pc-pc_start); pc1 += 1; break;

case 'N':
    args[nargs++] = d->d_arg; break;

default:
    panic("*bad pattern char %c", *s);
}

if (dflag > 1)
{
    printf("%s ", instrs[d->d_inst].i_name);
    int i;
    for (i = 0; i < nargs; ++i)
        printf("%i ", args[i]);
    printf("\n");
}

unpack_instr(pc - context[CP_CODE].x, d->d_inst, args[0], args[1], 0);
pc += d->d_len;

// create end block
block_create_end();

if (dflag > 2)
{
    printf("\n");
    listnode l = blocks;
    while (l != NULL)
    {
        basicblock b = (basicblock) l->contents;
        write_block(b);
        printf("\n");
        l = l->next;
    }
}

/* translate -- translates the basic block representation into a libJIT representation. */
static primitive * translate()
{
    basicblock b;
    listnode l = blocks;
    int h;

    // load parameters
    int id = 12; // magic number; 12 is id of first param, then +4 for next etc..
    var v = lookup_var(id, S_LOCAL);
    jit_value_t addr;
    while (v != NULL)
    {
        if (v->is_param)
        {
            addr = jit_insn_add_relative(func, bp, id);
            if (v->type & T_FLOAT)
                v->val = jit_insn_load_relative(func, addr, 0, jit_type_float32);
            else
                v->val = jit_insn_load_relative(func, addr, 0, jit_type_int);
        }
        v = lookup_var(id += 4, S_LOCAL);
    }

    // initialise memory for CSE hash table entries
    hpool = (uchar *) alloc_mem(HTSIZE);
    hlimit = hpool + HTSIZE;

    // generate code for basic blocks
    while (l != NULL)
    {
        b = (basicblock) l->contents;

        // label the block
        jit_insn_label(func, &b->label);

        // reset hash table memory
        // (CSEs are not preserved over block boundaries)
hfree = hpool;
for (h = 0; h < IHASH; ++h)
{
    inst_entry e = inst_hashtab[h];
    while (e != NULL)
    {
        e->val = NULL;
        e = e->next;
    }
    inst_hashtab[h] = NULL;
}

listnode i = b->instrs;
while (i != NULL)
{
    instruction j = (instruction) i->contents;
    i = i->next;
    instr(b, j, j->inst, j->arg1, j->arg2);
}

l = l->next;
// add jump to next block, if it's not the next
// we visit anyway...
if (b->jump != NULL & l != NULL & b->jump != l->contents)
jit_insn_branch(func, b->jump->label);

// label for end block
jit_insn_label(func, &endblock->label);

// dump intermediate code
if (dfile)
jit_dump_function(dfile, func, find_proc(context)->p_name);

// compile the constructed function
jit_function_compile(func);

// dump disassembly
if (dfile)
jit_dump_function(dfile, func, find_proc(context)->p_name);

// construct a closure for the function and return its address.
return (primitive *) jit_function_to_closure(func);

/* once_init_libjit -- performs one-time initialisation */
void once_init_libjit()
{
    // construct the signature for a native function call
    params[0] = jit_type_void_ptr;
    signature = jit_type_create_signature(jit_abi_cdecl, jit_type_void, params, 1, 1);
}

/* init_libjit -- initialises libJIT and generates function
 * initialisation instructions. */
void init_libjit()
{
    jit_context = jit_context_create();
    jit_context_build_start(jit_context);

    func = jit_function_create(jit_context, signature);
    // save important values for later use
    bp = jit_value_get_param(func, 0);
    kp = jit_value_create_nint_constant(func, jit_type_void_ptr, (int)&context[CP_CONST]);
    result = jit_value_create_nint_constant(func, jit_type_void_ptr, (int)&result);
    staticlink = jit_value_create_nint_constant(func, jit_type_void_ptr, (int)&statlink);
    bits = jit_value_create_nint_constant(func, jit_type_void_ptr, (int)&bit);

    #ifdef JIT_EXCEPTIONS
    // set up exception handling
    jit_exception_set_handler(obc_jit_exception_handler);
    jit_Runtime_bp = jit_value_create_nint_constant(func, jit_type_void_ptr, (int)&runtime_bp);
    jit_insn_store_relative(func, jit_runtime_bp, 0, bp);
    jit_line = jit_value_create_nint_constant(func, jit_type_void_ptr, (int)&line);
    #endif
}
/* jit_compile -- replace a bytecode routine with native code */

void obc_jit_compile(value * cp)
{
    uchar * pc = cp[CP_CODE].x;
    uchar * end = pc + cp[CP_SIZE].i;

    // initialise memory pool.
    if (zpool == NULL)
    {
        zpool = (uchar *) alloc_mem(MAPSIZE);
        zlimit = zpool + MAPSIZE;
        zfree = zpool;
    }
    context = cp;
    sp = 0;

    map_labels(pc, end);
    init_libjit();
    create_blocks(pc, end);
    optimise();
    cp[CP_PRIM].z = (primitive *) translate();
}

9.6  jit.h

struct inst {
    char *i_name;   /* Name of the instruction */
    int i_equiv[MAX_EQ]; /* Expansion into simpler instructions */
    int stack;
};

#define IMASK 0xffff
#define IARG 0x10000
#define ICON 0x20000

extern struct inst instrs[];

struct decode {
    int d_inst;
    char *d_patt;
    int d_arg;
    int d_len;
};

extern struct decode decode decode[];