6.818 Project Proposal

Team RockETH

November 2021

Compilation

The MITScript code will be first parsed by the same parser used for the previous two phases. Then, the AST tree will be translated to a new intermediate code, having the following characteristics:

- It will be in static single assignment form. That is, each variable will be assigned exactly once, and every variable be defined before it is used.
- The code will be structured in a control flow graph.

The instruction set includes (so far) the following instructions: `ADD`, `ADD_INT`, `SUB`, `MUL`, `DIV`, `EQ`, `GT`, `GEQ`, `AND`, `OR`, `NOT`, `LOAD_ARG`, `LOAD_FREE_REF`, `MOV`, `REF_STORE`, `REC_LOAD_NAME`, `REC_LOAD_INDX`, `REC_STORE_NAME`, `REC_STORE_INDX`, `ALLOC_REF`, `ALLOC_REC`, `ALLOC_CLOSURE`, `SET_CAPTURE`, `SET_ARG`, `CALL`, `RETURN`, `REF_LOAD`, `LOAD_GLOBAL`, `STORE_GLOBAL`, `ASSERT_BOOL`, `ASSERT_INT`, `ASSERT_STRING`, `ASSERT_RECORD`, `ASSERT_CLOSURE`, `ASSERT_NONZERO`, `PRINT`, `INPUT`, `INTCAST` and `SWAP`. The arguments of a function can be of the following form: `NONE`, `VIRT_REG`, `IMMEDIATE`, `LOGICAL`. Note, the outputs of the instructions (if there is any) can be only `VIRT_REG`.

Optimization

Beside being necessary for register allocation, the intermediate representation being in SSA allows us also to perform optimizations more easily than on the previous bytecode. Our optimizations will include:

- **Dead code elimination**: We will eliminate instructions that assign values to not used virtual registers. For instance, consider the following code appearing in the last block of a function:

  Compiled Code

  ```
  ADD $4 $5 1
  ```

  ```
  MOV $6 2
  ADD $7 $6 1
  RETURN $4
  ```

Optimized Compiled Code

```
ADD $4 $5 1
RETURN $4
```

The need for this optimization is also highly motivated by the fact that we generate unnecessary move instructions when a function is invoked (we set all assigned variables to none according to the programming language semantics). In almost all cases these instructions can be eliminated using dead code elimination.

- **Constant propagation**: We will propagate constant values in instructions where all arguments are constant values. Furthermore, this will allow us to eliminate many unnecessary type asserts that are generated for all arguments of certain arithmetic and boolean operations:

Compiled Code

```
ASSERT_INT 1
ASSERT_INT $3
SUB $4 $3 1
RETURN $4
```

Optimized Compiled Code

```
ASSERT_INT $3
SUB $4 $3 1
RETURN $4
```

- **Type inference**: This optimization should also allow us to eliminate many asserts. Moreover, when two operand are both integer values we will use the `ADD_INT` instruction instead of the usual `ADD` instruction.

Compiled Code

```
ASSERT_INT $3
ASSERT_INT $4
SUB $5 $4 $3
ASSERT_INT $3
ASSERT_INT $5
MUL $6 $5 $3
```

Optimized Compiled Code

```
ASSERT_INT $3
ASSERT_INT $4
SUB $5 $4 $3
MUL $6 $5 $3
```

- **Instruction reordering**: SSA code will also help us to reorder more easily code. The idea is to try to reduce the lifetime of virtual registers as much as possible to avoid unnecessary stores on the stack.

The following optimization will be implemented if further speed up is needed:

- **Record shape analysis**: If we have the guarantees that certain records only contain a pre-determined number of fields, then we can speed up the access to the fields by using a common array (we will assign to each field at compile time an index).

Note, these optimization also concern the intermediate code. Other design choices will allow us also to speed up the code and will be explained later (e.g. the value representation).

Register Allocation

The register allocation algorithm is based on the standard second-chance bin packing algorithm used in many JIT-compiled languages.

First, it is interesting to consider why the simplest linear scan algorithm is not suitable for compiling MITScript. The language is designed around very frequent function calls. The issue with these function calls is that they are mostly dynamic: the value of the object being called depends on runtime variables. Therefore, whenever a function is called, it is necessary to save *all* live registers, as it is not known which registers the called function will clobber (note that in the internal calling convention all registers are volatile). For this reason, it would in most cases be unreasonable to try and allocate a single register to a variable for its entire lifetime, as the linear scan algorithm does. Hence, it is necessary to employ strategies based on interval splitting.
Additional complications stem from architectural restrictions. For example, in x86-64, multiplications have to be performed using the RAX and RDX registers, which further restricts live ranges and introduces the need for spilling registers. The register allocator accommodates all such cases by allowing physical registers to be specified as "reserved" for certain intervals.

As input, the algorithm requires the set of live intervals for all virtual registers. Thanks to the usage of SSA form in the previous compilation steps, the task of finding these intervals can be completed in linear time, without any kind of data flow analysis. While allocating the physical registers, the algorithm also naturally decomposes all phi nodes introduced by the SSA form. The output is a control flow graph which can relatively easily be translated to machine code.

Value Representation

On 64 bit architectures, pointers returned by malloc are guaranteed to be 16-byte aligned. This leaves the lowest 4 bits to store a tag representing what kind of value is stored.
We use this to store none, booleans, integers and strings of seven bytes or less directly in the pointer. This carries the obvious advantage that memory indirection is limited to types which require it, like long strings, records, closures and references. A more subtle advantage comes from short string inlining: computing the hash of a short string is a no-op, as it is already uniquely mapped to a 64 bit value. This means that accesses to the internal record hash map are greatly accelerated for short field names.

JIT Compilation

Translating from the immediate representation to machine code is relatively straightforward. The register allocator guarantees that registers are kept free for certain instructions, such as multiplications. The JIT compiler therefore simply has to translate between IR instructions and machine code.

The major intricacy that the JIT compiler has to handle is the assignment of stack slots to values. Because the garbage collector must be able to trace the stack at any point, all values contained on the stack must be valid. It is therefore not possible to simply decrement the stack pointer at the beginning of the function. Instead, the stack must be grown with push instructions, ensuring that all values on the stack are valid.