Phase 3 Documentation

Robert Durfee, Philip Murzynowski, Noah Pauls

Design

The phase 3 design can be split into three basic modules: high-level intermediate, low-level intermediate, and x86 assembly generation. The first is located in the `hl` subdirectory and the rest are located in the `ll` subdirectory.

High-Level Intermediate

The high-level intermediate representation takes the abstract syntax tree and reduces it to a tree with simpler operations (e.g. loads and stores at the leaves) and incorporates links directly to declarations using the symbol table. The representation is as shown below, using the following format:

- Inheritance is represented by indentation.
- Interfaces are surrounded with `<>`.
- Fields of a class are given after `:`.
- Optionals are appended with `?`.
- Arrays are surrounded with `[]`.
- Sets are surrounded with `{}`.
- Identifiers are Strings

(More detailed representation overview can be found here.)

Declarations

```
Program: [import], [scalar], [array], [string], [method]
Import: identifier
<Scalar>
  GlobalScalar: identifier
  LocalScalar: index
  Argument: index
<Array>
  GlobalArray: identifier, length
  LocalArray: index, length
String: index, value
Method: type, identifier, body
Block: [argument], [scalar], [array], [statement]
```

The program was split up into arrays following the AST representation and allowing for simple access and organization of global identifiers. The decision we made was to have GlobalScalars hold string identifiers as global variables can later be accessed by unique labels in assembly, while LocalScalars and Arguments contain integers indexes as they are either obtained by a relative index with respect to the stack if a localScalar or a particular register or stack location if an Argument. We thought to keep the identifier for locals as well but determined that doing so would be unnecessary as each local will have a unique index into the stack or list of arguments. Arrays follow the same ideas additionally keeping length information. The block holds the different forms of declarations in separate lists for organization, including arguments if the block is a method body, as each block has a new scope and arguments for a method are passed into that scope. Finally, one thing that we are considering is reinstating type information for these classes for optimization, especially for boolean arrays.

Statements

```
<Statement>
  <Store>
    StoreScalar: scalar, expression
    StoreArray: array, index, expression
    StoreArrayCompound: array, index, type, expression
  Call: call
  If: condition, body, other
  For: initial, condition, update, body
  While: condition, body
  Return: expression?
  Break
  Continue
```

The statements are laid out nearly identically to the AST, with much of the more complicated machinery relegated to the high level expressions. However, our initial specification did not include the StoreArrayCompound statement, as we figured that the left hand side of a compound statement could be decomposed into a load and a store. This proved to be incorrect in our framework for arrays as this would result in the index expression being evaluated twice, which more often than not was completely fine, however, in test case x-32-assign-order.dcf in which the index expression is a function that can modify global state, it led to global state being modified twice, and thus needed a special statement.

Expressions

```
<Argument>
  <Expression>
    Binary: left, type, right
    Unary: type, expression
    LoadScalar: scalar
    LoadArray: array, index
    <Call>
      InternalCall: method, [argument]
      ExternalCall: import, [argument]
    Length: array
```

The expressions follow from the AST almost exactly, with the greatest thing of note likely being the distinction between the internal and external calls, as in this project the arguments of external calls need not be checked against the import and therefore we can have a simplified construction of the high level representation depending on whether the method call is external.

Literals

```
<Argument>
  <Expression>
    Integer: value
  String: string
```

Literals were split into integers and strings as chars, ints, and booleans can all be represented as integers, while strings would need special treatment later in the code generation process as they must be global, allocated in the .data section of our assembly, and loaded with a leaq instruction.

Symbol Table

```
SymbolTable: parent?, {import}, {scalar}, {array}, {string}, {method}
```

The symbol table is constructed during the build process as described below. All contained declarations can be looked up by their symbol and the process recursively continues by traversing parent symbol tables up to the global symbol table.

Builder

`HLBuilder` consumes a syntactically valid AST and outputs the high-level intermediate representation tree. The entry point for the builder is `buildProgram`, and from there the builder traverses the AST by calling builder functions for the various AST nodes encountered. Rather than use a visitor to traverse the AST, `HLBuilder` makes use of Java 16's pattern matching abilities, greatly simplifying the process of "visiting" nodes in the tree and giving a more granular set of static checks.

The builder makes use of a symbol table to keep track of symbols and symbol metadata during construction, but this table is thrown away after construction. Rather than keep a symbol table object, the HLIR uses the table to construct object references between variable/method references and their declaration objects. When a symbol is declared, the symbol-to-declaration mapping is stored in the symbol table; when that same symbol is referenced, the builder retrieves the declaration object stored in the symbol table and stores it in the new `HLLoad` / `HLStore` object. See `buildLocationExpression` for an example of this in action. Strings get special treatment; instead of being limited to the scope where they are used, they are hoisted to the top level `HLProgram`, since they will be aggregated in the assembly's `.data` section.

Low-Level Intermediate

The low-level intermediate representation takes the high-level intermediate tree representation and converts it to a control flow graph with basic blocks as vertices, each with a list of instructions. These instructions operate on `Alias` declarations which store intermediate expression results. The representation is as follows:

(More detailed representation overview can be found here, including assembly templates.)

Declarations

```
Program: [import], [scalar], [array], [string], [method]
<Declaration>
  Import: identifier
  <Scalar>
    GlobalScalar: identifier
    LocalScalar: index
    Argument: index
  <Array>
    GlobalArray: identifier, length
    LocalArray: index, length
  String: index, value
  Method: type, identifier, [argument], [scalar], [array], [alias], body
  Alias: index
```

Control Flow Graph

```
ControlFlowGraph: entry, exit
```

The control flow graph must have exactly one entry basic block and one basic block with no true or false target: the exit basic block.

Basic Block

```
BasicBlock: index, [instruction], true?, false?
```

If a basic block has a false target, its last instruction must be a `Compare`. If the comparison is equal, execution jumps to the false target, otherwise execution jumps to the true target. `Compare` instructions must not appear anywhere else within the list of instructions.

Instructions

```
<Instruction>
  StoreScalar: scalar, expression
  StoreArray: array, index, expression
  Return: expression?
  Exception: type
  Compare: left, right
  Binary: left, type, right, result
  Unary: type, expression, result
  LoadScalar: scalar, result
  LoadArray: array, index, result
  InternalCall: method, [argument], result
  ExternalCall: import, [argument], result
  Length: array, result
  Integer: value, result
  String: string, result
```

Builder

The `LLBuilder` is designed similarly to the `HLBuilder`. However, instead of producing a tree, the `LLBuilder` produces either declarations or control flow graphs. For declarations, an HLIR declaration is converted to an LLIR declaration and a forwarding pointer is created for accessing the declaration from other LLIR instructions later in the build process.

For all non-declaration HLIR, a control flow graph is produced instead. The simplest HLIR involving only straight-line code (such as loading scalars or binary expressions) returns a linear sequence of basic blocks. To handle composed expressions, aliases (temporaries) are created to store intermediate results. All aliases are hoisted to the method declaration and are local to that method.

More complicated HLIR (like if or while statements) return more complicated CFGs. To aid in the construction of conditional statements, `LLShortCircuit` is used for connecting true/false targets. In the end, all CFGs have a single entry and exit basic block to allow composability.

Break and continue statements require special attention. All statements are provided a break and continue target, if composed within a loop. When a break or continue statement is reached, the respective target is returned as the CFG. Because the true/false targets of a break or continue CFG should not be set, any time a break or continue target could be returned, it is first checked before setting true/false targets (notably in block, if, for, and while builders).

Furthermore, whenever an HLIR block is created, several additional tasks are required. First, all variables within the block must be initialized to zero. These initializations use regular, existing LLIR instructions. For scalars, it is as simple as creating an `LLStoreScalar` with a zero `LLIntegerLiteral`. The array initialization is similar, but a for loop CFG is created to iterate over the array. This was chosen instead of unrolling otherwise large arrays would flood the instruction cache.

Moreover declarations are also hoisted to the method declaration scope. This allows each function to have a single stack frame and all local variables can be allocated at once at function entry. The alternative would require creating (and destroying) stack frames at each block which seemed quite unnecessary.

Lastly, the `LLBuilder` is also responsible for checking bounds of array accesses. We elected to represent the bounds checks in our LLIR directly. As a result, the bounds checks are simply CFGs which compare the index to the array length. If the length is not in bounds, an exception instruction is executed. Note: To satisfy the single entry, single exit property of CFGs, the exception instruction is connected to the exit of the CFG, but this will never be traversed.

Short Circuit

`LLShortCircuit` aids in the construction of control flow graphs for conditional checks by short-circuiting boolean expressions. Short circuiting takes in the expression to short and the true/false targets of the condition in which the expression is used, and returns a basic block that marks the entry to the condition evaluation.

`LLShortCircuit` handles the heavy lifting of connecting basic blocks to skip unnecessary conditional statement evaluation. It does not return a control flow graph, as the client of `LLShortCircuit` knows all the possible targets of the complete short circuit graph.

Simplification

The control flow graph generated by the builder has *many* basic blocks. To reduce the jump overhead, and to aid in readability, we simplify the control flow graph.

The simplification takes three traversals over the graph. The first identifies all back edges for each basic block. The second uses back edge information to merge basic blocks with only a true target and a single back edge. The last looks for the new simplified exit basic block. All traversals are done using a depth-first search. To break loops, simplified basic blocks are constructed before recursive calls and their instructions are filled after the recursive call completes.

x86 Assembly Generation

This pass takes the low-level intermediate representation and generates x86 assembly instructions. Declarations are used to allocate appropriate space in the `.data` section (for globals), `.text` section (for methods), and on the stack (for locals and temporaries). Control flow graphs are flattened in this pass.

Stack Indices

Stack indices are calculated for all local and alias (temporary) declarations. After all declarations are hoisted, a method does a prefix sum on the sizes of each of its declarations. First scalars, then arrays, then aliases (all aliases in this phase reside on the stack). The indices are set within the declaration and referred to later in the generation process. The total size is returned to initialize the stack frame.

Generator

The generator converts the low level intermediate representation to a string composed of assembly instructions which is later written to a file. It designates predetermined exception messages labels, metadata, and space for specifying global identifiers such as strings, global variables, and method declarations, generates templated assembly for low level instruction such as a load, store, or a binary operation. Instructions can be easily templated as the inputs and outputs for each instruction are currently temporary locations on the stack. It also flattens the control flow graph by uniquely identifying basic blocks with labels and connecting targets with conditional/non-conditional jumps to labels. Method declarations include the proper prologue callee convention and along with generating internal or external calls make sure to keep stack aligned as a multiple of 16. Lastly, call expressions additionally load in the 6 first arguments to registers according to caller convention with remaining arguments being stack allocated.

Extras

In the previous phase, we traversed the abstract syntax tree using the visitor pattern. However, we found this pattern to be difficult to work with. We realized that having mutable state maintained across different `accept` calls was too messy to implement, especially when dividing work between members. As a result, we chose to construct a new, immutable visitor for each recursive call. However, this introduced a lot of unnecessary information being passed into each method call.

In this phase, we decided to use Java 16's (experimental in Java 15) feature for basic pattern matching. Instead of using a visitor pattern, we pattern match each possible type. This resulted in far more flexibility, without worrying about mutable state. Particularly, this made implementing negative integer literals way easier.

To aid in our debugging, we created special, recursive print statements for all representations above accessed via `debugString(depth)`. Since these were super verbose, they were helpful for identifying obscure issues once the area of interest was narrowed down. However, when exploring the low-level intermediate representation, it was too uncomfortable to work with. Thus, we developed `prettyString(depth)` which provides flattened output very similar to the LLVM IR format.

Difficulties

There are a few problems we identified with our implementation when writing documentation: - A compound assign statement with a unary operator indexing into an array will still evaluate index expression once for a load and once for a store, meaning that if the expression calls a function that modifies global state we will generate incorrect output. - We still allocate extra memory for local arrays as an artifact of once storing the length of the array as the first element of the array. - All boolean expressions should be short circuited, we only short circuit expressions in the condition of if statements or loops.

Contributions

The initial design was formulated in group meetings. The initial skeleton for the high-level, low-level, and assembly modules were created together. All individual method implementations were divided up evenly between group members. Debugging occurred together in group meetings.