CS 444, Winter Term 2018

A5 Documentation

By Alexandru Gatea, Carl Zhou

Due on April 4

Labels
We have a helper class called CodeGenHelper that generates commonly used code, including label names. In particular, CodeGenHelper provides general-purpose labels (these have a counter at the end to make sure they are distinct), class/interface/field labels (of the form _FileName_CLASS/INTERFACE/FIELD_name), vtable labels (of the form _VTABLE_typeName), array labels (of the form _ARRAY_typeName), array vtable labels (of the form _ARRAY_VTABLE_typeName), constructor and method labels (of the form _fileName_METHOD/CONSTRUCTOR_name#argtype1#argtype2...)

Layout
Each Object is assigned an address from the heap that contains: the vtable address (for its declared type) and values for all nonstatic fields (either inherited or declared)
Each Class is assigned an address from the heap that contains: the vtable address (for the class type) and values for all static fields declared in the class
Each Array (of any basic type, class type or interface type) is assigned an address from the heap that contains: the vtable address (for the array type), a length field, and values for all of array entries
Each Vtable (for a class type or array type) is assigned an address from the heap that contains: the address of the Selector-Index (SI) column corresponding to the vtable's type, the address of the Subtype column corresponding to the vtable's type, the index of the vtable's type in the Subtype column, and all nonstatic method labels (for declared or inherited (and not overriden) methods for the vtable's type)
Each SI column is assigned an address from the heap that contains: method labels for all methods that implement an interface method of a superinterface of the column's type, and 0 (null) for all unimplemented interface methods
Each Subtype column is assigned an address from the heap that contains: a bit (stored in 4 bytes) for each available type (class types, interface types, array types) indicating whether or not they are supertypes of the column's initial type (this is determined using the SUPER set computed before)
THIS: An address is assigned from the heap that contains the current object's address at all times (this is NULL in static contexts), and this address is referenced using a label __THIS

Method calls (for static/nonstatic methods, constructors and field initializers) use a caller saves-convention. In particular, the caller saves __THIS, ebx,ecx,edx,esi,ebp (but not eax, since this will be overwritten with the return value) then overwrites __THIS to accomodate the context in which the method is called (for static methods and static field initializers __THIS is set to NULL, and for all other calls it is set to the object that calls the method or who owns the field being initialized) and finally evaluates and pushes the arguments to the stack in left-to-right order, before calling the method. The callee begins by setting ebp to esp, and before each return it resets esp to ebp. Upon returning from the call, the caller pops the arguments (by adding 4 times the number of arguments to esp) and restores the registers and __THIS.

Local variables are always stored on the stack. An issue the arises because of this is that code blocks (from block statements, if/if-else statements, for/while statements but not method blocks) can create local variables that then need to get popped off the stack once the code block ends. In order to achieve this, we used another register esi to store the value of esp before a code block starts, so that esp can be reset to esi at the end of the code block. More precisely, immediately before each code block we push esi and set esi to esp and store the value of the local variable offset-counter from CodeGenHelper, and immediately after each code block we reset esp to esi and pop the top of the stack into esi and reset the value of the offset-counter.

Initialization
Before generating code for classes (that go in separate .s files) by using the visitor methods in CodePrinterVisitor, initialization is performed by calling the method startInitialization (also written in CodePrinterVisitor). All of the initialization code goes in a separate file _Initialization.s. At the top of this file is the declaration of the start label. startInitialization first declares all of the global labels in the section .data (the start label, all of the class labels, all of the array labels, all of the static field labels), then creates space for the static fields, and then determines and stores offsets for formalParameters (in methods and constructors) and for nonstatic fields. We also keep a copy in CodePrinterVisitor of the Object, String, Boolean, Character, Integer classes and Cloneable, Serializable interfaces for later use, and we obtain these as startInitialization goes through all of the ASTTrees in the list of ASTTrees (determined by the previous assignments' tree traversals).

At this point, we change context to section .text. Next, we do the first phase of initialization of classes by calling initializeClass for each class with isPreliminary set to true. This stores the offsets for all declared static fields, then allocates memory for the class and stores it at the class label, and then calls initializeTables to create the vtable, SI column and Subtype column (this is described below). Once the preliminary initialization of classes is done, startInitialization initializes all arrays of basic types (by calling initializeTables), arrays of class types (by calling initializeClass with isArray set to true, which just calls initializeTables) and arrays of interface types (by calling initializeInterfaceArray, which just calls initializeTables). (NOTE: although some of this seems like code duplication, the code is somewhat different because a ClassDeclNode and an InterfaceDeclNode are different classes, for example) At this point, we finalize the initialization of all classes by calling initializeClass again for each one, with isPreliminary set to false, which runs the initializers of all static fields. It was necessary to split the initialization of classes into two phases because static field initializers might contain method calls or arrays and so we needed to have the offsets and initialization of these done first. Finally, startInitialization calls the starting method test present in the first input file, and upon return calls sys_exit with the return value of test as the exit code.

initializeTables first stores the method labels in the vtable; the list of methods is determined by calling computeVtable, which recursively determines the vtable for the superclass (if any) and then goes through the list of methods declared in the current class, adding any new methods and performing any replacements (using the REPLACE set computed before). Then it allocates memory for the SI column and constructs it by going through a predetermined list of all interface methods and using the REPLACE set, and finally allocates memory for the Subtype column by going through two predetermined lists of all basic array types and of all class/interface array types and using the SUPER set computed before.

Code Generation
Code generation is done by one tree traversal using the class CodePrinterVisitor. At each node, visits to subnode are interleaved with code being printed. To print code, we have a helper class CodeChunk that prints ; before each comment, a : after each label declaration, and a newline at the end of each line of code. Most of the code generation is just following the JLS rules and the suggestions from class. I will briefly outline some of the more tricky visit methods.
-MethodDeclarationNode: Visits the method body, and then adds an implicit return at the end for void methods.
-ConstructorDeclarationNode: This calls the implicit constructor followed by calling the nonstatic field initializers for all fields declared or inherited in the current class. To get these fields, it calls a method getSuperFieldsDeclared in ASTTree that recursively calls itself on the superclass (if any) before adding the fields declared in the current class to a list being returned. Since nonstatic fields do not use dynamic dispatch, it was important to return all inherited fields, even the ones that are hidden by a field in the current class.
-FieldDeclNode: For nonstatic fields, the code generated starts and ends like the code for methods (because the field declaration code is called each time an object of the corresponding class type is created), and in between it either prints the code for the initializer or it initializes the field to the default value 0. For static fields, the only difference is that instead of the starting and ending methodcall-code, the value of the field (returned by the initializer if any, or 0 otherwise) is stored directly at the field's location (which is statically known because it is the address of the class's label plus the field's offset).
-MethodInvocationNode: For static methods, this computes the value of the prefix (if any) and then calls the appropriate method using its label. For nonstatic methods, this computes the value of the prefix (if any) and stores it in __THIS, and then gets the vtable of the object and gets and calls the method label using the offset stored in the MethodHeader.
-NameNode: This was tricky to implement because we had to deal with prefixes of the name that were previously resolved when dealing with ambiguous names in A3. We stored the computed list of prefixes in the NameNode in resolveAmbiguousNames, and then in visit(NameNode) we have a for-loop going through the prefixes and generating code for the field, array, local variable accesses.
-AssignmentNode: We added a flag isLHS that is set to true while visiting the LHS of an assignment. In NameNode, FieldAccessNode, ArrayAccessNode if isLHS is true then the address is returned rather than the value of that particular entity.

Type Assignability Run-Time Checks
We used the vtable address stored at each object's location to determine if two types are equal, and to get the Subtype column and Subtype index in the Subtype column of various types in order to check if a type is type assignable to another at run-time.

Strings
We have several helper methods for strings.
-createStringLiteral: This creates a String object containing a particular String literal. It creates a String object (using code similar to code in visit(classInstanceCreationNode)), creates a character array and stores the characters of the input string in it (using code similar to code in visit(arrayCreationNode)), and calls the String constructor with the array as its argument.
-createStringObjectFromNull: This creates a String object containing "null", by calling createStringLiteral. The assembly code for this is written in _Initialization.s
-createStringObject: This converts a basic type to a reference type by calling the appropriate constructor (Integer, Character or Boolean).
-callToString: This calls toString, assuming eax has the address of the object.
-concatenate: This calls the concatenate method from String, assuming eax, ebx contain the left and right String objects respectively.
Here is how these methods are used:
-In visit(LiteralNode), createStringLiteral is called for string literals.
-In visit(binOpExpr), to add a String to another expression: If one of the addends is a basic type, we convert it to a reference type by calling createStringObject. Then we check that both the left and right are not null (if either is null, we write assembly code to call createStringObjectFromNull in order to replace null with a String object containing "null") and then call the helper method callToString. Finally, we call concatenate.

General Comments
Even though a lot of the code generation was straightforward, we had several major issues that took a long time to resolve.
- First of all, labels were surprisingly tricky because we need an extern before each label in a different file, but it is difficult to keep track of which file calls a particular helper method in CodeGenHelper. In the end we added checks both for the current file name and also for the current class (these were used in different contexts).
-Offsets were equally troublesome. Initially, we made it so that offsets are stored when by the visit methods, but the problem was that initializers may contain field accesses or method calls and these had no offsets at the initialization phase. We had to compute and store field offsets and method offsets in the initialization methods.
-When we first tried to run the assembly code, we kept getting segmentation faults because of invalid use of labels (we were dereferencing labels inappropriately). Also the assembler sometimes did not accept the generated assembly code because we were storing values directly at addresses which led to "operation size not specified" errors. This took a few days to sort out, and we did it mainly by trial and error.
-The local variable issue discussed above was also difficult to figure out and to resolve. We had to use GDB to determine why the offsets for local variables were wrong; also, we made

several attempts at solving this problem before being successful.
-In our first attempt at implementing Strings we tried to use one helper method to do most of the work, but it led to a lot of bugs so we decided it was easier if we separated the tasks out into several helper methods (even though they had some similar bits of code), and this worked very well.

Testing
Other than the Marmoset tests, we wrote some tests for conditions not checked in the Marmoset tests but that are present either in the JLS or in the Joos page on the CS 444 website. To elaborate on this latter set of tests, we had tests with:
-code blocks and nested if and for loop statements
-adding a String and an object
-more complex class inheritances and overloaded methods
-multiple interfaces
-arrays of interfaces
-more complex instanceof and casting (with interfaces and superclasses)
-abstract classes
-static dispatch for nonstatic fields
-tests for all of the run-time checks required
Wrapper functions for running the nasm, ld, and the compiled binary (main) were created to complete the full integration testing. These classes used the Java Runtime library to execute the binary(nasm, ld, or main) with arguments, read the input and output, and retrieve the exit code from running the process. The exit code was compared against the expected exit code to determine if execution was correct.