Stage 4: Intermediate Representation

Due May 19, 2021 by 11:59pm **Points** 100 **Available** until Jun 5, 2021 at 11:59pm

This assignment was locked Jun 5, 2021 at 11:59pm.

In this project, we are going one step further from AST to build intermediate representation (IR) for the input programs.

Getting Started

Step 1: Understanding the IR Structure

The IR we are using is organized in a three-level hierarchy: `Program`, `Function`, and `Instruction`, listing from top to low level.

The Program is equivalent to the `program` directive in your grammar. It contains global variable declarations and function definitions; A Function is a function definition that contains arguments and a list of Instructions.

Instructions within a Function are organized as a graph representing the control flow: A node in this graph is an Instruction, and an (directed) edge goes from node **A** to node **B** if **A** is executed before **B**.

Instructions use variables to store or read values. There are two kinds of variables: `AddressVar` and `LocalVar`. `AddressVar`s are used to locate global variables and (global) arrays. An `AddressVar` always starts with a `%` sign in the textual IR representation (e.g., `%x`).

`LocalVar`s are used to store values within a function. These variables might be local variables in a Crux program or temporary variables we created specifically for IR to store temporary values. A `LocalVar` always starts with a `$` sign in the textual IR representation (e.g., `$a`).

We also have constants representing any constant values you have during the IR constructions. Variables (`AddressVar` and `LocalVar`) and Constants are all values.

As mentioned earlier, instructions are organized as a graph representing the control flow, where each edge implies the execution ordering. Branching instructions (i.e., `IfElseBranch` and `Loop`) have more than one successor nodes. `IfElseBranch` for example, has `thenBlock` and `elseBlock` as its successors (strictly speaking, the edge is pointing from the `JumpInst` to the *first instruction* in the `thenBlock` and `elseBlock`). For these situations, we have special two special kinds of labels attached on these edges: `Jump` and `Continue`. Take `IfElseBranch` as an example again, the edge goes toward the `thenBlock` has the `Jump` label attached on it, and the other edge goes toward the `elseBlock` has the `Continue` label attached. `Continue` is also the default label attached on edges other than branching edges.

The full specification of each IR instructions would be put in the IR Spec section at the very bottom of this page.

Step 2: Implementing the ASTLower.java

We've already provided most of the IR instruction classes. The only thing you're gonna do is implement the `ASTLower.java` that converts AST to IR. We again use `ASTVisitor` to achieve this task, so what you need to do is implementing all the `visit(<AST Node>)` methods in `ASTLower` to generate the corresponding IR for an AST node.

There are several important APIs in ASTLower that you must use:

```
Program ASTLower::lower()
```

This method returns the object that represents the current Program.

```
Instruction Function::getStart()
```

This API returns the first instruction in a Function. Each `Instruction` object can have outgoing edges that indicated the flow of control. `JumpInst` objects can have two outgoing edges. The second edge is taken if the conditional jump instruction is taken. Accessing and setting edges is via the following three API functions:

```
Instruction::setNext(int i, Instruction inst)
```

sets the ith outgoing edge of the receiver object `Instruction` to be the `Instruction` inst.

```
Instruction Instruction::getNext(int i)
```

gets the ith outgoing edge of the receivever object `Instruction`.

```
int InstructionnumNext()
```

returns the number of outgoing edges an Instruction has.

Step 3: Run Public Test Cases

In this project we're going to evaluate your project with an emulator (check the emulator flag in Compiler.java). So that the IR generated from your project does not need to be identical to ours. Regardless, we will still provide the textual output of each test case generating from our own implementation, as a reference answer for you.

IR Spec

The IR is composed from the following classes:

- `Function`: A function which is the lowered version of a `FunctionDefinition`. The difference to the AST version is, that the body of the function does not consist of a list of statements, but instead it is a graph, in which instructions are nodes and the control flow are the edges.
- `Instruction`: The base class for all instructions. Every instruction consists of a destination variable and a list of operands. Note that not every instruction needs a destination variable (for example a jump instruction that takes a target address as operand). Further, the list operands can be empty as well (e.g., a nop instruction that does nothing.)
  - `AddressAt`: Calculates the address given a base symbol and an offset.
  - `GlobalDecl`: Global variable or array declaration.
  - `BinaryOperator`: Any binary expression operator.
  - `CallInst`: Calls a function with the provided arguments.
  - `CompareInst`: Compares two values with each other. The result of the operation is a boolean.
  - `CopyInst`: Copies the source in to the destination.
  - `JumpInst`: A conditional jump, which jumps to its true-successor if the condition is true, otherwise it continues with the next instruction. Note that the jump instruction does not contain the destination. The two possible destinations are stored within the instruction graph of the function.
  - `LoadInst`: Loads the value located at the source address into the destination variable.
  - `NopInst`: Does nothing. Can be useful during lowering from AST to IR when a dummy instruction is needed.
  - `ReturnInst`: Store the return value and leave the current function, returning the control to the caller.
  - `StoreInst`: Stores the value of the source at the destination address.
  - `UnaryNotInst`: Inverts a boolean.
- `Value`: A value is anything, that can be used as an operand to an instruction. The purpose of the type is primarily to verify, that the generated IR is consistent and that no type errors were introduced during lowering.
  - `Constant`: A constant represents any kind of constant value. In our language that is integers and booleans.
    - `BooleanConstant`: A constant boolean (i.e., true or false). This is equivalent to `LiteralBool`.
    - `IntegerConstant`: A constant integer, e.g., an array offset (like the 2 in a[2]). This is equivalent to `LiteralInt`.
  - `Variable`: Any sort of value in the memory which holds a value that can be read or written by instructions. This includes both variables declared in the AST, as well as temporaries.
    - `AddressVar`: An address variable is a variable, that contains an address to a location in memory. This can be used to access an array or a global variable.
    - `LocalVar`: A scalar variable is a variable that holds a scalar value - in our language that is a variable containing either an integer or a boolean.

Submission

Submission Details

Grade: 100% (100 pts possible)
Graded Anonymously: no

Comments:
No Comments