Frame Analysis and Intermediate Representation Documentation

Anurag Kashyap, Mia de Leon, Austin Kao

March 24, 2021

After type checking the abstract syntax tree for our compiler of the Tiger programming language, we translate the tree into an intermediate representation tree. Since the specific ISA of the machine affects the intermediate representation tree, we also conduct an appropriate frame analysis and build the tree accordingly. We will assume that our compiler runs on a machine using the MIPS architecture.

Frame Analysis

For our frame analysis, we implemented a module for a frame in MIPS. This frame stores a frame pointer, a list of the access variables, and the number of access variables. It can also allocate new space for a new variable and allows for external calls to be made.

Intermediate Representation

For our intermediate representation tree, we use a Translate module written in SML to translate each node of our abstract syntax tree into the appropriate intermediate representation nodes. We return a list of the fragments generated from our tree. Each fragment represents a string literal or a function frame.

Simple Variables

For simple variables, we return its memory location within the appropriate frame, which is the frame's frame pointer plus some offset. We chase the static link to get the right frame pointer if the variable escapes.

Arrays and Records

When creating arrays and records, we make external calls to create the variables outside of the stack, but we store the memory location of these variables within the appropriate frame. When we encounter an array variable, we make sure

that the subscript for that variables falls within the correct bounds. Much like simple variables, we will return the appropriate memory location of an array or record when we encounter it.

Arithmetic Operations

Arithmetic operations translate to an operator with two nodes representing the expression to the left of the operator and the expression to the right of the operator.

Conditional Statements

Conditional statements translate to a relational operator with two nodes representing the left side of the operator and the right side of the operator as well as two labels that indicate where to go depending on whether the statement is true or false. If strings are part of the statement, we make an external call to evaluate the statement instead.

If Statements

We make a differentiation between if-then and if-then-else statements, which affects our translation of each. While an if-then-else statement requires a label that refers to nodes when the statement is true and a label that refers to nodes when the statement is false, an if-then statement only requires a label that refers to nodes when the statement is true.

Strings

String literals are added to the list of fragments.

Function Calls

If the caller and callee frames are located on different levels, we check the different levels to find the right static link and make the appropriate translation. If the callee frame is making an external function call, we do not pass a static link.

While and For Loops

Both of these loops will create a sequence of expressions that have some branching conditions to exit the loop. We account for any breaks in the loop.

Let Expressions

In handling let expressions, we also change how we handle variable and function declarations. For variable declarations, we allocate space in the frame for a new variable and add the frame access to the variable environment. To initialize

the variable properly, we also make a list of the expressions needed for the initialization that are run before the let expression. For function declarations, we create a new level for each function as well as a new frame for each function. Any function arguments are added as access variables in the frame. Functions are added to the variable environment along with the level of the function. We make sure to move the result of the function's body into a register.

For the body of the let expression, we create nodes that will execute the sequence of expressions that comprise the body, returning the result of the final one.