Homework 8 Notes

Robert Steele

Exercise 5.8:

For this problem I simply modified the make-elementary-expression definition to not check for and handle label expressions, thereby throwing an error if one is encountered.

Exercise 5.9:

I implemented this by slightly modifying extract-labels. In the inner lambda that accumulates the results there is a check to see if the next instruction is a label or an instruction. In the case that it is a label I simply checked to see if the labels list already holds that label. If it does already exist then throw an error, otherwise we can add the label to the list.

Exercise 5.19:

Labels:

I saved all a machines labels in a local variable during assembly. I wrote a setter for a machine's labels called *install-labels* which is called inside the receive function in assemble. It is called potentially many times but the last time it will be called is after all the labels have been processed and aggregated. This saves the local variable in assemble called *labels* to the relevant machine's local variable, also named *labels*.

Breakpoints:

I decided to represent breakpoints in a similar way to labels. *Labels* is a list of lists with the first element of each list being the label and the second element being the list of instructions that label is associated with. Each breakpoint is a list of lists with the first element being the set of instructions remaining in the program at that breakpoint, the second element being the label the breakpoint is associated with, and the third element being the offset the breakpoint is from its label. With all this information I can identify and report breakpoints quickly and efficiently.

trim-instructions:

My implementation hinged on this procedure. It takes as arguments the instructions, the offset, and a label (for reporting purposes). If the offset is out of bounds, either less than 1 or larger than the length of the instruction set, we signal an error. Otherwise, we initiate a loop which trims the instruction set "from the top" until we reach an offset of 1. At this point the trimmed instruction set is returned.

set-breakpoint:

I created an external function *set-breakpoint* which takes a machine, label, and offset (n) and dispatches the label and offset to the machine with 'create-breakpoint. Internally the machine looks up the instructions associated with the label (if none exist it throws an error), trims the instructions given the offset, checks to see if a breakpoint with those trimmed instructions already exists (if it does simply it simply signals a breakpoint already exists at that location), and finally if the label exists and the trimmed

instructions are not already associated with a breakpoint it creates a new breakpoint and prepends it to the list of breakpoints.

cancel-breakpoint:

I created an external function *cancel-breakpoint* which takes a machine, label, and offset (n) and dispatches the label and offset to the machine with 'create-breakpoint. Internally the machine looks up the instructions associated with the label (if the label does not exist it throws an error), trims the instructions given the offset, and initiates a loop to remove the relevant breakpoint, the result of which overrides *breakpoints*. For each breakpoint in *breakpoints* it compares the trimmed instructions with the instructions of the breakpoint. If they do not match, it creates a pair with the breakpoint in the pair's car and a recursive call to loop in the pair's cdr. If the loop finds a match it simply returns the rest of the breakpoints list (minus the current breakpoint). If the loop reaches a null value it signals an error indicating the breakpoint does not exist.

cancel-all-breakpoints:

This was very simple. I created an external function *cancel-all-breakpoints* which takes a machine and dispatches to the machine with 'cancel-all-breakpoints. Internally the machine sets *breakpoints* to the empty set.

break:

This was a simple procedure that takes a single argument (a breakpoint) and prints a simple message indicating we have stopped at a breakpoint. Because it does not resume execution the program halts and we can view or change register values at our leisure and proceed when we are ready.

execute:

My changes to execute were simple. To check for a breakpoint, I simply assoc *breakpoints* with the current pc register value to determine if we should stop. If assoc returns something, then there is a breakpoint at this location, and I call break with the breakpoint as the argument. Otherwise continue execution as normal.

proceed:

This was also very simple. I created the external function *proceed* which takes a machine and dispatches to the machine with 'proceed. Internally the machine does exactly what it used to do in execute before I made changes to it and finishes by calling execute. This causes any breakpoint located at that pc value to be ignored.

print:

This was a helper procedure I wrote to display an indeterminate number of arguments and call newline.

Initial Approach:

Initially some of the set/cancel-breakpoint calculations (like trimming) were done outside the machine. This required me to write a getter for the labels and was just too convoluted and unnecessary. Moving everything inside the machine's local expressions made the code a little more compact and easier to follow.

Potential Improvements:

I think it would be nice to have a GDB like debugger here which could be implemented with a read loop inside break that takes short arguments like ‘r’ to access all registers or ‘r’ *register-name* to access a particular register. This could make debugging a particular machine far easier and an escape keyword could be added to allow us to exit the debugger prompt and still allow us to resume processing later. It could even be an optional flag to enter this GDB like prompt or just stop and exit. There is lots that could still be done here.