Project 3

Abhi Agarwal

Notes

My first assumption is that there can't be more than 4 arguments. I have manually handled each argument to save time designing a register allocator for that portion. This also makes it simpler for us as designers because there are registers R0-R3 that are specifically as parameters, and so we didn't have to push things on and off the stack.

There are also limitations I want to offer because I couldn't figure them out or ran out of time:

1. The biggest flaws are the ideas that register allocation is not complete. I wasn't able to get this done as I ran out of time for this particular assignment. I managed to get register allocation done for variables and arguments, but sadly not for the expressions (1 * 1). They get placed into a preset register that I've coded into there manually. It's not the most promising solution.

2. The next biggest flaw is that Expressions are not evaluated. I did not get enough time to be able to program them well, and so they are just simulating a pre-fixed condition that just does a branch from them to either True or False depending on how I programmed them. I also did not get time to complete this. There's a scheme (T) that takes in 3 arguments and just returns the branching variable.

3. Something more minor is that the function call parameters have to either be all variables or all integers. This was my mistake, and I took the easy way out. I made this a design decision because I didn't have enough time to complete a good design. However, the better way here would just be to have a better designed scheme that does a recursive call.

4. The operations in Expression (+,-, etc.) work, but they have pre-defined registers as I have mentioned above. This would have been simple to fix, but I just ran out of time (apologies again).

My register allocation was done by a simple NextRegister scheme, which I manually mapped out to provide for the next register. For the Argument scheme this was not too useful because I just set the next register manually (I knew how many registers there were). This was only used when the user declared var because then we could just update the attribute.

I had 3 inherited attributes - **e**, **r**, **nr**. **e** was the environment, **r** was just a mapping from the register and if it was being used or not, and **nr** was the next register we used. The idea behind **r** was just to help me clean up the garbage, and allow me to switch **nr** more intelligently in the future, but I wasn't able to accomplish this in this given program. The idea for **nr** is just a simple way to get the next register we need to allocate. The + as they key because I wanted to use the minus later as a lookahead for the next register if I ever needed to allocate 2 registers at the same time.

I did not have to push anything on the stack, but the initial configurations when a new function was loaded. I wanted to use this particular feature incase we ran out of registers, but I just made an assumption that we would never get to that point. The NextRegister scheme would just throw an error if we the next register was greater than R12.

Moreover, it is vital for my if statements to have {}, and also for my else. This adds for while loops as well. None of the conditional/loops can have inline condition checks.

Overall, I think I'm very proud of this particular assignment as it turned out and taught me much more about HACS than Pr2 did. I wish I had the knowledge I had for this particular assignment for the previous projects. I really enjoyed HACS, and I think it's one of the more incredible tools I've seen (after I understood it well).