CMSC 430 Multiple Return Values Iniquity Report
Jonathan Camberos
May 18, 2023

1 Introduction

1.1 Project Selection and Implementation

My compiler project was to add Racket's let-values and values forms in order to allow for multiple return values from a single function. This implementation modifies the run-time system to mimic receiving an 'array', a vector pointer in our case, in the return register rax instead of a single value. Via returning a pointer to a vector currently on the heap, we are able to mimic the functionality of returning multiple values, while still only returning a single value in the rax register.

2 Implementation

2.1 Run-Time Additions and Issues

I began by modifying the ast.rkt and parser.rkt file to account for the new forms "Let-Values" and "Values". None of the other files needed to be modified to get the parsing working so this step was fairly simple. In order to modify the runtime system, I followed the provided instructions to modify main.c, unload/free, and the compiler section. Immediately after however, segfaults on simple tests that returned a unary vector began to occur. After debugging for a few hours and starting from scratch, I combed through the Cmsc430 class forum to see if anyone had run into similar issues and found a thread that mentioned the project description was incomplete and that the compiler code should before the (Pop rbx) to restore the callee-save register instead of just the (ret). This was indeed my problem, and I was able to begin with the Values compiler implementation.

2.2 Values Implementation

The concept of returning a vector was still opaque to me. I was unsure how the runtime system was able to read a "vector" from rax so my next step was to read line by line, the provided main.c, unload/free, and other modifications to better understand how to strategize my Values implementation. In the main.c I recognized that the system had replaced the previous "val_t result = entry(heap);" with a "val_vect_t *result = entry(heap);" For my vector modification to work, in cases where I would be returning multiple values, I would actually just be returning a tagged pointer in the rax register. Immediately I anticipated multiple issues with this. If main.c only takes pointers now, how would I return a single result ex: (add1 4). Would it be possible to compile the already provided vector type and simply return that pointer? If not, could I utilize the compile vector code as a blueprint to create my own vector type? I put a pin on the first question and looked into the second as reusing provided code seemed like the easiest route, but quickly noticed that this would not be possible as vectors can only be created with (make-vector 1 0)), to instantiate multiple occurrences of the same single value as oppose to (values 2 5 6) which would require different values in the vector. My next option was to review the vector code and lecture video to re-learn how vectors were created. Our abstraction of vectors entailed holding the length of the vector on the heap, offsetting the resulting values by a multiple distance of 8 bytes, and returning a pointer to the memory location in the heap that held the vector length integer. This abstraction was precisely what I needed, and a simple modification of manually placing the different values on the heap ex: (the 2 5 6 in the previous "values" example) would fix the issue of make-vector only allowing a single value.

2.3 Testing Issues and New "type-ID-Vect" Pointer

Due to an unrealized error on my end (that I also unknowingly fixed later on), my raco test compile.rkt threw an segfault on every values multiple return values test case, which led me to utilizing the racket repl manually for Values testing as shown in the image below. In order to debug, I would manually print out and read the assembly until my Values compiler implementation was functional and returned a pointer. In addition, to distinguish between regular vectors and these new vectors (which we will now refer to as ID-Vect), I added a tag type "type-ID-vect" with #b110.

```
> (unload/free (asm-interp (compile (parse '[ (values 3 5 6)  ]  ) ) ) )
3
5
6
```

2.4 Let-Values Implementation
I proceeded with adding the Let-Values functionality of receiving multiple return-values and adding them to the environment. As opposed to the Values implementation which took into account the runtime system, understanding Let-Values conceptually and in code was much more straightforward. After compiling a Value clause, rax would be currently holding a pointer to the heap where we could access: 1) the length of the ID-Vect and 2) the values in the ID-Vect via offsetting the pointer in rax. This concept also reminded me of the Let-Star method from a previous Cmsc430 assignment where we matched multiple values to multiple variables, added them to the given environment then compiled a single clause. This mimicked our Let-Values goal with the exception of having to grab the values from a Vector on the heap instead of from a list of compiled values. Having a blueprint, I followed my old Let-Start implementation and was able to get the Let-Values working alongside the Values implementation.

```
> (unload/free (asm-interp (compile (parse '[ (let-values ([(x y z) (values 3 5 6) ]) y) ] ) ) ) )
5
```

2.5 Updating Manual Testing Process to Raco Test
Having my Let-Values and Values methods now functional, I decided to try and decipher the segfault error in my raco test compile.rkt to allow for more robust testing for my project. However, to my surprise, the segfault error was inexplicably gone. Having the raco tester available, I loaded up the previous tests from the provided Loot code and ran into a rather simple issue. For programs such as (add1 4) that were returning a single value, in this case the integer 5 which would be left in rax, our updated main.c runtime system was expecting a vector pointer (this was an issue I previously predicted, pinned question in section 2.2). For this issue, I simply added a helper function (ID-Vector-Check) before the (pop rbx) that would check if what is inside rax before the (ret) is of type ID-Vector pointer. If it is simply return the pointer and if not it moves what is in rax, in this case the integer 5, and puts it in a unary ID-vector while moving the ID-Vect pointer to rax. Now all single return values or multiple return result in a ID-Vect pointer in rax being returned to the main.c program.

2.6 Arity and Final Testing
After having the barebones functionally working properly, I added arity checking for my Let-Values methods ensuring that Values would be providing the same length of arguments as the number of variables in the Let-Values clause. The final issue arose from a provided test-case I had missed, specifically (add1 (values 5)). This implied, with my current Values implementation, that clauses such as (add1) needed to now check if the compiled expression in rax was an ID-Vector of length 1. To fix this last bug, I simply added a helper function (single-Return-ID-Vec-Check) that would check for ID-Vectors of length 1 and move the value into rax. Ultimately, I took the provided Iniquity tests and modified them to return multiple values, alongside my own tests I wrote while creating Values and Let-Values.

2.7 Shifting to Iniquity Due to Final Testing
In the later days of testing, I ran into a lambda test case: (values ((lambda () (add1 4)))). As this was a few days before the final project was due I spent the rest of the week attempting to figure out how to implement lambda with the newly added Let-Value and Value features. I was not able to get a working version in time and as such I decided to abort my partially working Loot final project attempt and instead get a final version of a completely functional Iniquity final project with the new Let-Value and Value features. This was not a challenge as the logic that was in the Loot project transitioned easily and worked in the Iniquity version without any issues.

3.0 Conclusion
Overall, like many of the Cmsc430 compiler projects the Let-Values and Values functionality seemed daunting to add to my program. However, as I chose to write out this report at the same time as I implemented the project, it pushed me to plan out my steps and essentially narrate my approach to problems. Having to explain my strategies towards tackling the issue of implementing Values, for

example, forced me to break the larger issue into smaller steps of first understanding the main.c implementation, discerning then what was expected in rax, and finally choosing how to represent Values inside my program. In terms of compilers, this project and course have opened my perspective and made me grateful for what occurs under the hood of my everyday projects, and I hope to apply my compiler experiences to my professional career in the future.