atoi Proof of Correctness Report – Team 2

Matthew Sheldon
matthew.sheldon@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

Isabella Pereira
isabella.pereira@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

Jarrod Rogers
jarrod.rogers@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

Naja-Lee Habboush
naja-lee.babboush@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

Brandon Wang
brandon.wang@utdallas.edu
The University of Texas at Dallas
Richardson, Texas, USA

I. Introduction

We have been working on proving the formal correctness of the `atoi` function using Picinae on the ARMv8 architecture. This will help to advance the state-of-the-art and improve the Picinae system.

II. Overview of atoi

The `atoi` (ASCII-to-integer) function takes, as input, a string, and converts the "initial portion" of said string into a corresponding 32-bit integer. It does this by converting each numeric ASCII character into it's associated numeric value one character at a time, and adds each converted value up to produce the final number.

The initial portion of a string starts from the beginning of the string, consists of potentially leading whitespace, an optional sign character, and ends after the first non-numeric sequence in the string. At the start of the numeric sequence, a single "+" or "−" will determine the sign of the number output by `atoi` (providing neither will cause `atoi` to treat it as a positive number). Excepting the sign, the only characters allowed between the start of the string and the start of the numeric sequence in the initial portion is whitespace, which is discarded when `atoi` is processing the number. If this rule is violated, for example by having an alphabetic character occur before the first numeric sequence, we call the input "ill-formed", as `atoi` will return **0**. Figure 1 shows a control flow graph for the disassembly of `atoi`.

III. Challenges

Throughout the course of our work on this project, we encountered several challenges that made proving the correctness of `atoi` more complicated than originally anticipated. Here we detail these challenges.

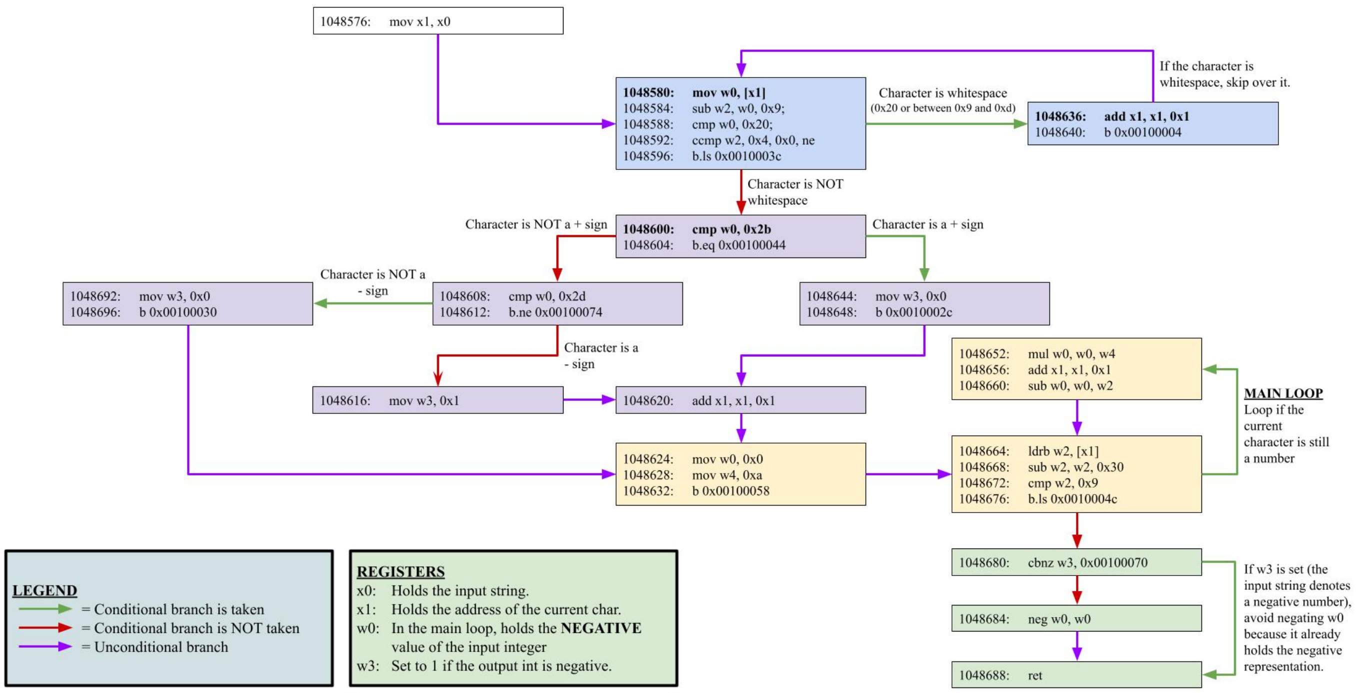
A. Overflow/Underflow

One of the first problems we ran into was determining how to handle integer overflow or underflow scenarios. If the string representation a number greater (or less) than what can be held in a 32-bit signed integer is used as input to `atoi`, it will result in an overflow/underflow and produce a value that doesn't properly correspond to the full input string. `atoi` explicitly does not detect overflow or underflow errors, and will *not* return `0` in the case of such an error. We observed that our specific implementation of `atoi` will simply return the lower 32-bits of what the n-bit representation of the number would have been.

Our initial approach involved using regular expressions on the string to verify that the numeric sequence within would not result in an over/underflow. The principle idea is guided by knowing the bounds of a 32-bit signed integer are static – -2^{31} (-2,147,483,648) on the low end, and $2^{31}-1$ (2,147,483,647) on the high end. We also figured that since we had been working with regular expressions in the homeworks leading up until this, we would likely be able to come up with something that worked with minimal effort. Because of this, we thought we could craft a regular expression to detect numbers that are outside these bounds in order to anticipate an over/underflowed result. However, as we quickly discovered, this approach was fundamentally flawed.

Since the bounds are static, we knew that any number with between zero and nine digits will not over/underflow. Input numbers with ten or more digits, however, is where this solution starts to get tricky. A number with ten or more digits *could* be a full number that will over/underflow, but *could* also be a small number with many leading zeros, which would not overflow. A solution based on regular expression detection would need to account for these details, and be able to handle cases with a higher precision than just the number of digits (for example, "2147483647" would not result in an overflow, but "214748364**8**" would). For these reasons, we decided to abandon the regular expression approach and instead turn our attention elsewhere.

A more intuitive solution is to, seperate from the *atoi* implementation being validated, first convert the string to it's integer representation and then check to see if the result over/underflowed during the conversion. This would allow us to detect an over/underflow and adjust the expected result so that such behavior can be verified. While neat and intuitive, we initially dismissed this apprach as it would require re-solving the exact problem of implementing `atoi` and would mean verifying the correctness of two different `atoi` implementations. However, after some thought and discussion with the



```
1048576:    mov x1, x0
```

```
1048580:    mov w0, [x1]
1048584:    sub w2, w0, 0x9;
1048588:    cmp w0, 0x20;
1048592:    ccmp w2, 0x4, 0x0, ne
1048596:    b.ls 0x0010003c
```

```
1048636:    add x1, x1, 0x1
1048640:    b 0x00100004
```

```
1048600:    cmp w0, 0x2b
1048604:    b.eq 0x00100044
```

```
1048692:    mov w3, 0x0
1048696:    b 0x00100030
```

```
1048608:    cmp w0, 0x2d
1048612:    b.ne 0x00100074
```

```
1048644:    mov w3, 0x0
1048648:    b 0x0010002c
```

```
1048652:    mul w0, w0, w4
1048656:    add x1, x1, 0x1
1048660:    sub w0, w0, w2
```

```
1048616:    mov w3, 0x1
```

```
1048620:    add x1, x1, 0x1
```

```
1048624:    mov w0, 0x0
1048628:    mov w4, 0xa
1048632:    b 0x00100058
```

```
1048664:    ldrb w2, [x1]
1048668:    sub w2, w2, 0x30
1048672:    cmp w2, 0x9
1048676:    b.ls 0x0010004c
```

```
1048680:    cbnz w3, 0x00100070
```

```
1048684:    neg w0, w0
```

```
1048688:    ret
```

LEGEND
- 🟩 = Conditional branch is taken
- 🟥 = Conditional branch is NOT taken
- 🟪 = Unconditional branch

REGISTERS

x0:	Holds the input string.
x1:	Holds the address of the current char.
w0:	In the main loop, holds the **NEGATIVE** value of the input integer
w3:	Set to 1 if the output int is negative.

If the character is whitespace, skip over it.
Character is whitespace (0x20 or between 0x9 and 0xd)
Character is NOT whitespace
Character is NOT a + sign
Character is a + sign
Character is NOT a - sign
Character is a - sign
MAIN LOOP Loop if the current character is still a number
If w3 is set (the input string denotes a negative number), avoid negating w0 because it already holds the negative representation.

Fig. 1: Control Flow Graph (CFG) for `atoi`

CS 6335 staff, we realized that this approach would be the most straightforward and efficient way to solve the problem. Namely, since the Gallina implementation of `atoi` would not be burdoned with the bit-width restrictions of the ARMv8 arhitecture – using the built-in and structurally defined Integer definition, it would be able to handle the full range of input values that we would need to test. Then, all that would be left is to determine how many bits of memory would be required to store the integer representation of the number. This would allow us to verify that the input to `atoi` did not result in an over/underflow during the conversion.

B. *"Proper" vs. "Improper" Inputs*

Similar to the over/underflow issue is defining what a "proper" input to `atoi` looks like (i.e., an input that will not result in an error). While over/underflowing inputs result in undesired behavior, they are not detected by `atoi` as errors. Handling "ill-formed" strings that are detected as errors by `atoi` has also proven to be a challenge itself. The `atoi` function returns `0` when it detects such an erroneous string. Unfortunately, `0` can also be returned by valid conversions from strings with a numeric sequence equivalent to zero, which means that given an arbitrary string, there is no immediate way to tell if it is an "improper" input based on the output from `atoi` alone, which makes proving any property about the correctness of the input very difficult. We worked on trying to solve this issue for a while, but due to time constraints we eventually decided to compromise. The solution we ultimately settled on for this problem is to narrow the scope of our proof to only prove properties about the behavior of `atoi` on proper inputs, leaving validation of the input as a responsibility of the caller.

IV. Implementation Strategy

A. *Major theorems*

B. *Preconditions*

V. Breakdown of Contributions

A. *Matthew Sheldon*

- TODO

B. *Isabella Pereira*

- TODO

C. *Jarrod Rogers*

- TODO

D. *Naja-lee Habboush*

- TODO

E. *Brandon Wang*

- TODO

VI. Future Work

- TODO

VII. Evaluation

VIII. Conclusion

As a team we have learned a lot from working on the correctness for `atoi`. Even though there is still more work to be done for a complete `atoi` partial proof of correctness, we are proud of what we have accomplished, the challenges that we were not only able to identify, but also begin to overcome, and the work we have contributed towards advancing the state-of-the-art.