Scope

This analysis only considers simple values and a few different instructions.

Values

Currently, we consider two types of values: characters (32 bit values) and strings. Characters are 32 bit values which are either NULL (0) or some other value. A string is a pointer to a contiguous sequence of characters, terminated by a NULL-valued character.

These values cannot be unrestricted, however. For example, we do not want to allow a *String* to point past its terminator or de-reference a $NULL$ pointer. Haskell notation gives us convenient notation for expressing allowed values:

$$String = NULL \mid EOS \mid Ptr\ Char$$

$$Char = NULL \mid 1 \mid 2 \mid ...$$

In other words, a *String* is a NULL pointer, points to a NULL-character ("end-of-string" – EOS), or points to a valid character. A *Char* is either NULL (0) or itself.

At the machine level many of these values have the same representation. We hope to use types to distinguish each value.

Incrementing Pointers

Consider this instruction, where *r1* is a register:

```
addl    $4, r1
```

If *r1* holds a character, the instruction is not very interesting. If *r1* is a string, though, then this instruction will move the pointer to the next character in the string. Incrementing the pointer is only safe if two conditions holds: the pointer is not NULL and it does not already point to the end of the string.

If we only talk about the type held in $r1$, we can't guarantee it is safe at all. $r1$ could be NULL or point to the end of the string. Incrementing it in those cases is not safe! Types alone are not enough – we need to mention the value held as well. To that end, we write the type of $r1$ as a **case** expression. The rule expresses that if $r1$ matches one of the arms, then r1's type is determined by that arm of the case:

$$\begin{aligned}
&\mathit{r1} = \textbf{case } \mathit{r1} \textbf{ of}\\
&\quad \mathit{Ptr\ Char} \rightarrow \mathit{String}\\
&\quad \mathit{Char} \rightarrow \mathit{Char}
\end{aligned}$$

This says that if we know $r1$ is a pointer to a character ("$Ptr\ \ Char$"), we can stay it is a $String$ afterwards[1]. If $r1$ is a character, it is still a character afterwards.

What is more interesting is the cases that are NOT allowed. Imagine these cases:

$$\begin{aligned}
&\mathit{r1} = \textbf{case } \mathit{r1} \textbf{ of}\\
&\quad \ldots\\
&\quad \mathit{String} \rightarrow \mathit{String}\\
&\quad \mathit{NULL} \rightarrow \ldots\\
&\quad \mathit{EOS} \rightarrow \ldots
\end{aligned}$$

Each would open the door to arbitrary manipulation of the string pointer.

Comparing Values & Conditional Branches

Consider this instruction:

```
cmpl    $0, r1
```

cmpl will set the Zero Flag (ZF) if r1 equals 0. Otherwise, ZF will be 1. If r1 is a character, this doesn't tell us much. If it is a $String$, we know a little more. If $r1$ is 0, we know it holds NULL.

[1] We cannot say it is a Ptr or EOS because we haven't tested the value pointed to yet.

Otherwise, it points to a character or the end of a string. Using the case notation from above, we add these two conditions to the test for *r1*:

$$
\begin{aligned}
&\mathit{r1} = \textbf{case}\ \mathit{r1}\ \textbf{of}\\
&\quad \mathit{String} \rightarrow \mathit{ZF} = 0 \rightarrow \mathit{NULL}\\
&\quad \mathit{String} \rightarrow \mathit{ZF} \not\equiv 0 \rightarrow \{\mathit{EOS} \mid \mathit{Ptr\ Char}\}
\end{aligned}
$$

The second branch indicates that we don't know what *r1* points to, but we know it is not NULL.

Now consider what happens when we branch after a comparison:

```
test:
  cmpl    $0, r1
  jnz     loop
...
loop:
```

If control passed to *loop*, we know that ZF was not 0 and therefore *r1* is is a *Ptr Char* rather than *EOS*. If control falls through the branch, then we know *r1* is *EOS* and therefore we have reached the end of the string.

Conditional branches, therefore, allow us to determine which "branch" should be taken on a match.