Inlining in Cairo

Inlining is a common code optimization technique supported by most compilers. It involves replacing a function call at the call site with the actual code of the called function, eliminating the overhead associated with the function call itself. This can improve performance by reducing the number of instructions executed, but may increase the total size of the program. When you're thinking about whether to inline a function, take into account things like how big it is, what parameters it has, how often it gets called, and how it might affect the size of your compiled code.

The `inline` Attribute

In Cairo, the `inline` attribute suggests whether or not the Sierra code corresponding to the attributed function should be directly injected in the caller function's context, rather than using a `function_call` libfunc to execute that code.

There are three variants of the `inline` attribute that one can use:

- `#[inline]` suggests performing an inline expansion.
- `#[inline(always)]` suggests that an inline expansion should always be performed.
- `#[inline(never)]` suggests that an inline expansion should never be performed.

> Note: the `inline` attribute in every form is a hint, with no requirements on the language to place a copy of the attributed function in the caller. This means that the attribute may be ignored by the compiler. In practice, `#[inline(always)]` will cause inlining in all but the most exceptional cases.

Many of the Cairo corelib functions are inlined. User-defined functions may also be annotated with the `inline` attribute. Annoting functions with the `#[inline(always)]` attribute reduces the total number of steps required when calling these attributed functions. Indeed, injecting the Sierra code at the caller site avoids the step-cost involved in calling functions and obtaining their arguments.

However, inlining can also lead to increased code size. Whenever a function is inlined, the call site contains a copy of the function's Sierra code, potentially leading to duplication of code across the compiled code.

Therefore, inlining should be applied with caution. Using `#[inline]` or `#[inline(always)]` indiscriminately will lead to increased compile time. It is particularly useful to inline small functions, ideally with many arguments. This is because inlining large functions will increase the code length of the program, and handling many arguments will increase the number of steps to execute these functions.

The more frequently a function is called, the more beneficial inlining becomes in terms of performance. By doing so, the number of steps for the execution will be lower, while the code length will not grow that much or might even decrease in terms of total number of instructions.

> Inlining is often a tradeoff between number of steps and code length. Use the `inline` attribute cautiously where it is appropriate.

Inlining Example

Let's introduce a short example to illustrate the mechanisms of inlining in Cairo. Listing {{#ref inlining}} shows a basic program allowing comparison between inlined and non-inlined functions.

```cairo
{{#rustdoc_include ../listings/ch11-advanced-features/listing_03_inlining_example/src/lib.cairo}}
```

```
{{#label inlining}}
<span class="caption">Listing {{#ref inlining}}: A small Cairo program that adds the return value of 2 functions, with one of them being inlined</span>
Let's take a look at the corresponding Sierra code to see how inlining works under the hood:
```cairo,noplayground
{{#rustdoc_include ../listings/ch11-advanced-features/listing_03_inlining_example/src/inlining.sierra}}
```
The Sierra file is structured in three parts:
- Type and libfunc declarations.
- Statements that constitute the program.
- Declaration of the functions of the program.
The Sierra code statements always match the order of function declarations in the Cairo program. Indeed, the declaration of the functions of the program tells us that:
- `main` function starts at line 0, and returns a `felt252` on line 5.
- `inlined` function starts at line 6, and returns a `felt252` on line 8.
- `not_inlined` function starts at line 9, and returns a `felt252` on line 11.
All statements corresponding to the `main` function are located between lines 0 and 5:
```cairo,noplayground
00 function_call<user@main::main::not_inlined>() -> ([0])
01 felt252_const<1>() -> ([1])
02 store_temp<felt252>([1]) -> ([1])
03 felt252_add([1], [0]) -> ([2])
04 store_temp<felt252>([2]) -> ([2])
05 return([2])
```
The `function_call` libfunc is called on line 0 to execute the `not_inlined` function. This will execute the code from lines 9 to 10 and store the return value in the variable with id `0`.
```cairo,noplayground
09    felt252_const<2>() -> ([0])
10    store_temp<felt252>([0]) -> ([0])
```
This code uses a single data type, `felt252`. It uses two library functions - `felt252_const<2>`, which returns the constant `felt252` 2, and `store_temp<felt252>`, which pushes a constant value to memory. The first line calls the `felt252_const<2>` libfunc to create a variable with id `0`. Then, the second line pushes this variable to memory for later use.
After that, Sierra statements from line 1 to 2 are the actual body of the `inlined` function:
```cairo,noplayground
06    felt252_const<1>() -> ([0])
07    store_temp<felt252>([0]) -> ([0])
```
The only difference is that the inlined code will store the `felt252_const` value in a
```

variable with id `1`, because `[0]` refers to a variable previously assigned:
```cairo,noplayground
01    felt252_const<1>() -> ([1])
02    store_temp<felt252>([1]) -> ([1])
```

> Note: in both cases (inlined or not), the `return` instruction of the function being called is not executed, as this would lead to prematurely end the execution of the `main` function. Instead, return values of `inlined` and `not_inlined` will be added and the result will be returned.

Lines 3 to 5 contain the Sierra statements that will add the values contained in variables with ids `0` and `1`, store the result in memory and return it:
```cairo,noplayground
03    felt252_add([1], [0]) -> ([2])
04    store_temp<felt252>([2]) -> ([2])
05    return([2])
```

Now, let's take a look at the Casm code corresponding to this program to really understand the benefits of inlining.

Casm Code Explanations

Here is the Casm code for our previous program example:
```cairo,noplayground
1    call rel 3
2    ret
3    call rel 9
4    [ap + 0] = 1, ap++
5    [ap + 0] = [ap + -1] + [ap + -2], ap++
6    ret
7    [ap + 0] = 1, ap++
8    ret
9    [ap + 0] = 2, ap++
10    ret
11    ret
```

Don't hesitate to use cairovm.codes playground to follow along and see all the execution trace.

Each instruction and each argument for any instruction increment the Program Counter (known as PC) by 1. This means that `ret` on line 2 is actually the instruction at `PC = 3`, as the argument `3` corresponds to `PC = 2`.

The `call` and `ret` instructions allow implementation of a function stack:

- `call` instruction acts like a jump instruction, updating the PC to a given value, whether relatively to the current value using `rel` or absolutely using `abs`.
- `ret` instruction jumps back right after the `call` instruction and continues the execution of the code.

We can now decompose how these instructions are executed to understand what this code does:

- `call rel 3`: this instruction increments the PC by 3 and executes the instruction at this

location, which is `call rel 9` at `PC = 4`.
- `call rel 9` increments the PC by 9 and executes the instruction at `PC = 13`, which is actually line 9.
- `[ap + 0] = 2, ap++`: `ap` stands for Allocation Pointer, which points to the first memory cell that has not been used by the program so far. This means we store the value `2` in the next free memory cell indicated by the current value of `ap`, after which we increment `ap` by 1. Then, we go to the next line which is `ret`.
- `ret`: jumps back to the line after `call rel 9`, so we go to line 4.
- `[ap + 0] = 1, ap++` : we store the value `1` in `[ap]` and we apply `ap++` so that `[ap - 1] = 1`. This means we now have `[ap-1] = 1, [ap-2] = 2` and we go to the next line.
- `[ap + 0] = [ap + -1] + [ap + -2], ap++`: we sum the values `1` and `2` and store the result in `[ap]`, and we apply `ap++` so the result is `[ap-1] = 3, [ap-2] = 1, [ap-3]=2`.
- `ret`: jumps back to the line after `call rel 3`, so we go to line 2.
- `ret`: last instruction executed as there is no more `call` instruction where to jump right after. This is the actual return instruction of the Cairo `main` function.
To summarize:
- `call rel 3` corresponds to the `main` function, which is obviously not inlined.
- `call rel 9` triggers the call the `not_inlined` function, which returns `2` and stores it at the final location `[ap-3]`.
- The line 4 is the inlined code of the `inlined` function, which returns `1` and stores it at the final location `[ap-2]`. We clearly see that there is no `call` instruction in this case, because the body of the function is inserted and directly executed.
- After that, the sum is computed and we ultimately go back to the line 2 which contains the final `ret` instruction that returns the sum, corresponding to the return value of the `main` function.
It is interesting to note that in both Sierra code and Casm code, the `not_inlined` function will be called and executed before the body of the `inlined` function, even though the Cairo program executes `inlined() + not_inlined()`.
> The Casm code of our program clearly shows that there is a function call for the `not_inlined` function, while the `inlined` function is correctly inlined.
Additional Optimizations
Let's study another program that shows other benefits that inlining may sometimes provide. Listing {{#ref code_removal}} shows a Cairo program that calls 2 functions and doesn't return anything:
```cairo
{{#rustdoc_include ../listings/ch11-advanced-features/listing_02_inlining/src/lib.cairo}}
```

{{#label code_removal}}
Listing {{#ref code_removal}}: A small Cairo program that calls `inlined` and `not_inlined` and doesn't return any value.
Here is the corresponding Sierra code:
```cairo,noplayground
{{#rustdoc_include ../listings/ch11-advanced-features/listing_02_inlining/src/
inlining.sierra}}
```
In this specific case, we can observe that the compiler has applied additional

optimizations to the `main` function of our code : the code of the `inlined` function, which is annotated with the `#[inline(always)]` attribute, is actually not copied in the `main` function. Instead, the `main` function starts with the `function_call` libfunc to call the `not_inlined` function, entirely omitting the code of the `inlined` function.
> Because `inlined` return value is never used, the compiler optimizes the `main` function by skipping the `inlined` function code. This will actually reduce the code length while reducing the number of steps required to execute `main`.
In contrast, line 0 uses the `function_call` libfunc to execute the `not_inlined` function normally. This means that all the code from lines 7 to 8 will be executed:
```cairo,noplayground
07 felt252_const<133508164995039583817065828>() -> ([0])
08 store_temp<felt252>([0]) -> ([0])
```
This value stored in the variable with id `0` is then dropped on line 1, as it is not used in the `main` function:
```cairo,noplayground
01 drop<felt252>([0]) -> ()
```
Finally, as the `main` function doesn't return any value, a variable of unit type `()` is created and returned:
```cairo,noplayground
02 struct_construct<Unit>() -> ([1])
03 return([1])
```
Summary
Inlining is a compiler optimization technique that can be very useful in various situations. Inlining a function allows to get rid of the overhead of calling a function with the `function_call` libfunc by injecting the Sierra code directly in the caller function's context, while potentially optimizing the Sierra code executed to reduce the number of steps. If used effectively, inlining can even reduce code length as shown in the previous example.
Nevertheless, applying the `inline` attribute to a function with a lot of code and few parameters might result in an increased code size, especially if the inlined function is used many times in the codebase. Use inlining only where it makes sense, and be aware that the compiler handles inlining by default. Therefore, manually applying inlining is not recommended in most situations, but can help improve and fine-tune your code's behavior.