Under the hood of Cairo 1.0: Exploring Sierra

Part 2: Anatomy of Sierra programs

Mathieu

Follow

Nethermind.eth

--

Listen

Share

Introduction

In our previous blog post, we introduced Sierra and discussed how it simplifies the development process of Starknet contracts. This post will dive deeper into Sierra code and explore its various features, including data types, library functions, program statements, and user-defined functions. We aim to comprehensively understand Sierra code and what makes it a safe intermediate representation of Cairo code. Familiarizing oneself with Sierra is optional to writing Cairo programs, and it helps to understand how programs and proofs work together.

Analyzing a simple Sierra program

To better understand Sierra's structure, we will examine a simple Sierra program before progressing to more complex code. The following code, written in Cairo 1.0, is compiled to Sierra using the command cairo-compile program.cairo program.sierra -r

. It is a straightforward function that returns a variable of type felt252

with the value 1.

A Sierra program consists of four distinct parts, separated by empty lines.

The first part of a Sierra program involves declaring the types used in the program. During the compilation of a Cairo program to Sierra, a unique ID is assigned to each type used. This ID is subsequently reused in the program to identify the type of variables expected in function declarations. Type declarations are written with the syntax type my_id = concrete_type

, where concrete_type

is a type that is defined in the Sierra Core. A common example is the declaration of the felt252

type, with type felt252 = felt252

The next part of a Sierra program involves listing the libfuncs used. Libfuncs are built-in functions defined in the Sierra compiler that can be compiled into CASM code. In this step, each libfunc used in the program must be defined along with the input type it expects. The syntax used for libfunc declaration is libfunc my_id = function

. For instance, the libfunc drop

can be used with the felt252

type, and the NonZero

type, resulting in the declaration of both libfunc drop_felt = drop

and libfunc drop_nz_felt = drop>

.

The third part of a Sierra program involves declaring the statements that make up the program's code and specifying its intended behavior. These statements are executed sequentially and can either invoke a previously declared libfunc or return a variable. The syntax for declaring these statements is as follows: ( ) -> ()

or return()

Finally, the last part involves declaring user-defined functions used within the program. These functions are assigned a unique ID and associated with an index corresponding to the statement where their execution begins. The signature of each function specifies the input parameters, their types, and the types of the returned variable. The format is as follows: function_id@statement_index() -> ();

.

For example, consider the declaration of a function named fib

, which starts at index 0, takes three inputs of type felt252

, and returns a single output of type felt252

: fib@0(a: felt252, b: felt252, n: felt252)->felt252

.

Compiling the Cairo code from earlier outputs the following Sierra code:

The given code can be interpreted as follows: “The program uses a single data type, felt252

. It uses two library functions - felt_const<1>

, which returns the constant felt252 1

, and store_temp

, which pushes a constant value to memory. The program has a main function

that starts at statement 0 and returns a variable of type felt252

. During the execution of my program, we call the felt_const<1>

libfunc to create a variable with id [0]

. We push this variable to memory and retrieve another variable of id [1]

, which I return at the end of the function.”

From failing code to branching code

As discussed in the previous post, if we remove all failing operations from the Sierra semantics, we can prove the execution of every transaction, regardless of the outcome of the execution. This enables the inclusion of failed transactions in blocks, allowing sequencers to receive payment for their work. Smart contracts frequently require verifying whether a user is authorized to perform a specific action like minting or transferring tokens. This verification is done by using assert

statements to check boolean conditions. The Cairo 1 core library implements assertions as a function that can panic if the condition is not met.

For simplicity, we'll look at a Cairo program that can panic before ending its execution and see how it compiles to Sierra.

Let's highlight the most interesting concepts Sierra demonstrated in this program:

- The absence of a panic

libfunc, as the concept of panics as proper runtime errors, doesn't exist in Sierra programs.

- The felt_is_zero

library function used in statement #3 can continue to multiple branches after execution. In this particular case, If the variable with ID [2]

is zero, the code will continue with the fallthrough

case at the following statement. However, if the variable is non-zero, the program will move to statement #10.

- The branch_align

libfunc is used to equalize gas costs and ap changes across merging paths of branching code.

- A core::PanicResult

type is declared in our Sierra program, indicating that our Cairo 1 function, initially returning felt252

, now returns a PanicResult

- The array_new

libfunc is used to instantiate a new array.

- Enums are initialized using the enum_init

libfunc.

- Structs are constructed/deconstructed using the struct_construct

and struct_deconstruct

libfunc. To access a struct member, the struct is first deconstructed into multiple variables and then reconstructed when necessary.

During the compilation lowering phase, the Cairo function's return type containing a panic

statement is converted to the PanicResult

type. This new type is passed on to all its parent functions until it reaches the program's entry point. The program run is considered a failure if an error is propagated back to the entry point.

Writing our own Sierra program

In this part, we'll write our own Sierra program to get familiar with its linear type system, how to handle branches in Sierra code, and get a better overall understanding of how the Cairo stack works.

We will develop a straightforward program that computes the factorial of a given value n

, where n

is a felt252. The program will take n

as input, which will be a hardcoded value in the code, and return n!

without taking gas costs into account for simplicity. Before writing the Sierra statements, we will declare all the necessary types, libfuncs, and user-defined functions.

Defining types

Our program will need felts

. Since we will evaluate whether n equals 0, we also need to declare the NonZero

type, a wrapper type returned by the felt_is_zero

libfunc.

Defining libfuncs

Because we're writing a program with a recursive function, the state of ap

at the end of the execution of the program is not known at compile time. To do so, we will need to use disable_ap_tracking

. We will need store_temp

to move our values to ap before returning from a function, branch_align

to equalize ap changes across conditional branches, function_call

to call our user-defined recursive function, felt_const

to instantiate a constant felt252, felt_sub

and felt_add

for our felt252 operations, dup

, drop

and drop

to duplicate and drop variables where needed, and finally felt_is_zero

to evaluate which value to return. Here, we declare the felt_const

libfunc to return the value 24

, which is the “input” of our program. Finally, we also need the rename

libfunc to align identities for flow control merge.

Declaring user-defined functions

Before writing the core part of the Sierra code, we can start by declaring our functions. We will need the main

function, which returns a felt252

, and the multiply_rec

function, which returns a felt252

. At this point, we don't know the statement index where multiply_rec

starts - I filled this value after writing the rest.

Writing Sierra statements

We now need to write the statements that dictate how our program behaves. We will start with the main function, which begins at statement #0. Our main function is simple: We need to call the multiply_rec

function with an input value of 24

and return the result. Before calling a user-defined function, we need to push the value to memory using the store_temp

libfunc.

We can then proceed with the recursive function, which is more complex, so we will divide it into three parts.

First, we need to evaluate if n

is equal to zero with felt_is_zero

. Because n

needs to be used exactly once, we must duplicate it to make it available later. It's worth noting that dup

and drop

libfuncs don't generate code when compiling a Sierra program to CASM. They're only used at the Sierra level to comply with the linear type system constraints.

If the condition is true, the program executes the next statement through the fallthrough

branch. In this case, we must drop all the pending variables and return the value 1

.

If the condition is false, we need to calculate the value of n-1

, call multiply_rec

, multiply the result of the function call by n

, and return it. The second branch of felt_is_zero

starts at statement 14 and declares a variable n_not_zero that we need to drop because it is not used.

The corresponding Cairo code for the program we just wrote in pure Sierra would be:

Conclusion

In this post, we explored the anatomy of a Sierra program, analyzing both a simple Sierra program and a more complex one that compiles Cairo 1 panics into branching code. We also wrote our own Sierra program to better understand the Cairo stack. By the end of the post, we learned about the types, libfuncs, and user-defined functions used in Sierra code, providing the basic principles to read and understand Sierra programs. You can refer to this Sierra documentation to learn more about the existing libfuncs and their utility.

About Us

Nethermind is a team of world-class builders and researchers. We empower enterprises and developers worldwide to access and build upon the decentralized web. Our work touches every part of the web3 ecosystem, from our Nethermind node to fundamental cryptography research and infrastructure for the Starknet ecosystem. Discover our suite of tools for Starknet:

[Warp, the Solidity to Cairo compiler

](https://nethermind.page.link/8a9f);

Voyager, a StarkNet block explorer; [Horus, the open-source formal verification tool

](https://nethermind.io/horus/)for Starknet smart contracts;

[Juno

](https://github.com/NethermindEth/juno), Starknet client implementation

, and

[Cairo Smart Contracts Security Auditing

](https://nethermind.page.link/25ee)services.

If you're interested in solving some of blockchain's most difficult problems, visit our

[job board

](https://nethermind.page.link/careers)!

Disclaimer

This article has been prepared for the general information and understanding of the readers, and it does not indicate Nethermind's endorsement of any particular asset, project or team, nor guarantee its security. No representation or warranty, express or implied, is given by Nethermind as to the accuracy or completeness of the information or opinions contained in the above article. No third party should rely on this article in any way, including without limitation as financial, investment, tax, regulatory, legal or other advice, or interpret this article as any form of recommendation.