LIGO

WULFMAN CORPORATION

Audit SMAK Farming

November 2021

Tezos

WULFMAN
CORPORATION

Contents

WULFMAN CORPORATION

Disclaimer

This report does not provide any warranty or guarantee regarding the absolute bug-freenature of the technology analyzed.

This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Smart-chain's position is that each company and individual are responsible for their own due diligence and continuous security. Smart-chain's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

Introduction

This audit was commanded to Wulfman Corporation, in quality of main contributor and expert of LigoLANG, by SmartChain, in quality of developer and publisher of the SMAK farm Contract

The object of the audit is the analysis of the SMAK farm Contractin order to identify vulnerabilities and contract optimizations in the source code.

The contract targets the Tezos blockchain and was developed in LigoLANG. The auditing methods consist in manual review along with automated testing of the smart-contract using the ligo test framework and a Florencenet sandbox

The auditing process paid special attention to ensuring that the contract logic is coherent and implements the specification and the best testing schemes.

Overview

Project summary

Project Name	SMAK farm Contract
Publisher	SmartChain
Platform	Tezos
Language	LigoLANG(cameligo flavor)
Codebase	`https://github.com/MattSmartChain/auditFarm`
Original commit	b19c4eb1fa3b02deec2bd71a73f24ef2884fd055
Contract adress	KT1FtoNPPTQUuFFmzF6Ee7mStSZg6wPUYjUF
Contract url	`https://better-call.dev/mainnet/KT1FtoNPPTQUuFFmzF6Ee7mStSZg6wPUYjUF/storage`

Audit summary

Auditer	Wulfman Corporation
Delivery date	November 2021
Scope	Farm contract
Methodology	Manual review, unit testing
Tezos version	GRANADA (PtGRANADsDU8R9daYKAgWn...)
Tezos client version	10.2
LigoLANGversion	0.29.0

Vulnerability summary

Total issues	5
Critical	1
Major	1
Medium	1
Minor	1
Informational	1

Code Quality summary

Total improvements	19
Maintenance	3
Scalability	2
Readability	4
Origination cost	2
Gas cost	8

Vulnerability

Contents

V1. FA1.2 contract

Category	Severity	Location	Status
Potential Vulnerability	Informational	contract/main/fa12.mligo	

Description

The FA 1.2 specification is the standard specification for fungible token on tezos, and at the core of any smart-contract that will emit such non-fungible token. A bug or a vulnerability in the implementation of the standard will render the contract unusable. For this reason, the implementation should not be realised or modified in house but instead just originate from a trusted source and be audited.

The current FA 1.2 implementation was done by ocamlCase and publish within the dexter DApp. Using this implementation is questionable as it imply trusting ocamlCase to not intentional or non-intentionally introduce vulnerability in the contract.

The dexter DApp has be the object of an audit performed by Nomadic Lab on February 2021 which found a vulnerability. The vulnerability does not concern the FA1.2 implementation which increase confidence in this implementation.

Solution

I would be preferable to obtain the FA1.2 contract directly from the Tezos Fundation through tq : `https://assets.tqtezos.com/docs/token-contracts/fa12/2-fa12-ligo/`. This work has been finance by the Tezos Fundation and imlemented by the Ligo team in order to provide a secure, bug-free, trustworthy contract to the community.
This contract has not yet been audited by a third-party. This should be done

V2. Storage scope in code

Category	Severity	Location	Status
Potential Vulnerability	Major	contract/partials/FarmMethods.mligo	

Description

In the contract the storage is use as an environment to store global variable, with many utility function reading from it and writing to it, which is a very bad design pattern.

The storage is critical data that people pay in exchange for the correctness and integrity of such data. This is the value proposal of a smart-contract and should be treated with care

pattern which modify part of the storage just to give a few parameter to a function not only is expensive because it requires copying the whole storage on the stack, it is at risk of return this "temporary" storage instead of the proper one at the end of execution. For this reason, reading and writing to the storage should be only allowed in "entrypoint" function, i.e the function directly called by main after matching on the parameter type.

Solution

There is no automate tool to check this property to my knowledge. The good design pattern to adopt is :

- extra the necessary value at the beginning of the entrypoint (`let <value> = storage.<value>`)
- write the parameters of utility function explicitly but limit it to the necessary
- return tuple of value from utility function if necessary
- update the storage once at the end of the entrypoint and immediately return it

V3. Sequence computation and cost of exponentiation

Category	Severity	Location	Status
Specification limitation	Critical	contract/partials/FarmMethods.mligo	

Description

The contract use a geometric sequence to calculate the weekly rewards. To calculate each term, the contract use the classical index definition of a geometric function, which makes intensive use of the power function. In computation, a power function is very costly as i^n requires at best $O(log(n))$ multiplication function but in this code $O(n)$ multiplication. Since we computing for each week, the total number of multiplication is $O(n^2)$ which may be the cause of the limitation on the period number. Furthermore, the computation of each term is quite far-fetched. It seems that due to the absence of floating, we need to use floating point for the ratio, which leads to inaccuracy during division. To limit the imprecision, the formula has been adapt and use 5 exponentiation instead of the classical 2. Even with just a 100 periods, you will reach 50000 instruction. The gas limit for operation is 1040000 This is about 21 gas for the multiplication, the subtraction, the absolute value and the function calls. This should be enough, be we are getting close to the limit.

Solution

- As a general rule, always compute the constant value outside of the iterated code. This will avoid unnecessary re-computation of the same value
- Use the recursive definition of the sequence. Compute the seed term outside of the iteration and then use the previous value to generate the next one. This reduce the complexity to $O(n)$ computation
- Pay less attention to the division accuracy. It is important but the trade-off between correct value and efficient code should be in favour of efficiency in general
- Compute the last week reward as the difference between the total reward and the sum of the previous ones, instead of the recursive compute. This will fix some accuracy issues

V4. Precision of point ratio

Category	Severity	Location	Status
Unnecessary Problem	Minor	contract/partials/FarmMethods.mligo	

Description

Due to fix-point computation, the computation of a ration needs to first multiply the numerator by a precision factor to make it several order of magnitude greater than the denominator, otherwise the result would be 0. The number of order of magnitude gives the precision of the computation.

In order to choose a value for the precision p we need to estimate to magnitude of

$$\frac{dem}{num}$$

When computing the point ration for a week the value for the user i is

$$\frac{\sum_n t_n \cdot lp_n}{t_i \cdot lp_i}$$

. with t_i the time spend staking in the week and lp_i the number of coin staked. (we simplified with the assumption,that they didn't stake and unstake during the week otherwise we have to add extra sum on each different amount for each users, but it doesn't changes the analysis)

In the best case, all users staked about the average amount of token $< lp >$ during the whole week and the fraction simplifies to (n).

In the general case, the will be

$$\frac{n \cdot t_{week} \cdot < lp >}{t_i \cdot lp_i}$$

assuming that a significant number of user staked their token for the whole .

In the worst case, assuming the user staked its token within the last hour of the week, the fraction became

$$\frac{200 \cdot n \cdot < lp >}{lp_i}$$

. Considering a number of user of 10000 we need precision to be over 2000000000 to have three digit of precision for the fraction. But this could be the expected behaviour.

An other way to see it is that an user will never get that one unit of the reward currency (here milliSMAK), hence we only need the precision to be equal to the maximum possible recompense, which should be reasonable.

Solution

The best is to avoid introducing ratios and doing the final computation in one step. In the current code, there is a percentage vector which is computed to then be then use twice. The first time to compute the rewards from the percentages, and the second time just to get the week indices. The second usage is not required and by merge the first one with the computation of the percentage, you will not need to introduce the `precision` by simplifying

$$\frac{points \cdot precision}{farm\_points} \cdot \frac{weekly\_reward}{precision} = \frac{points \cdot weekly\_reward}{farm\_points}$$

V5. Remove double computation

Category	Severity	Location	Status
Potential inaccuracy	Medium	contract/partials/FarmMethods.mligo	

Description

In the processing of Stake and Unstake, you compute the increase of user points and the increase of farms points separately while they are equal, as indicate that you use wrote the same function twice. First of all, when you have similar function like those, you could consider factorizing them (this may require adding extra parameter to the function to remove the difference). Second of all by computing it twice you remove the guaranty that the two quantity are equal. Either due to one of the computation being erroneous or some computing. (Very unlikely, but an hardware malfunction could cause the result of a computation to be faulty). Not having this guaranty is an issues for your application.

Solution

Compute the delta and add it to both the farm total points and the user points. And it would be better to do that in one loop instead of first computing the vector of deltas, and then the vector of the farm and user (see Q11.).

WULFMAN CORPORATION

Code Quality

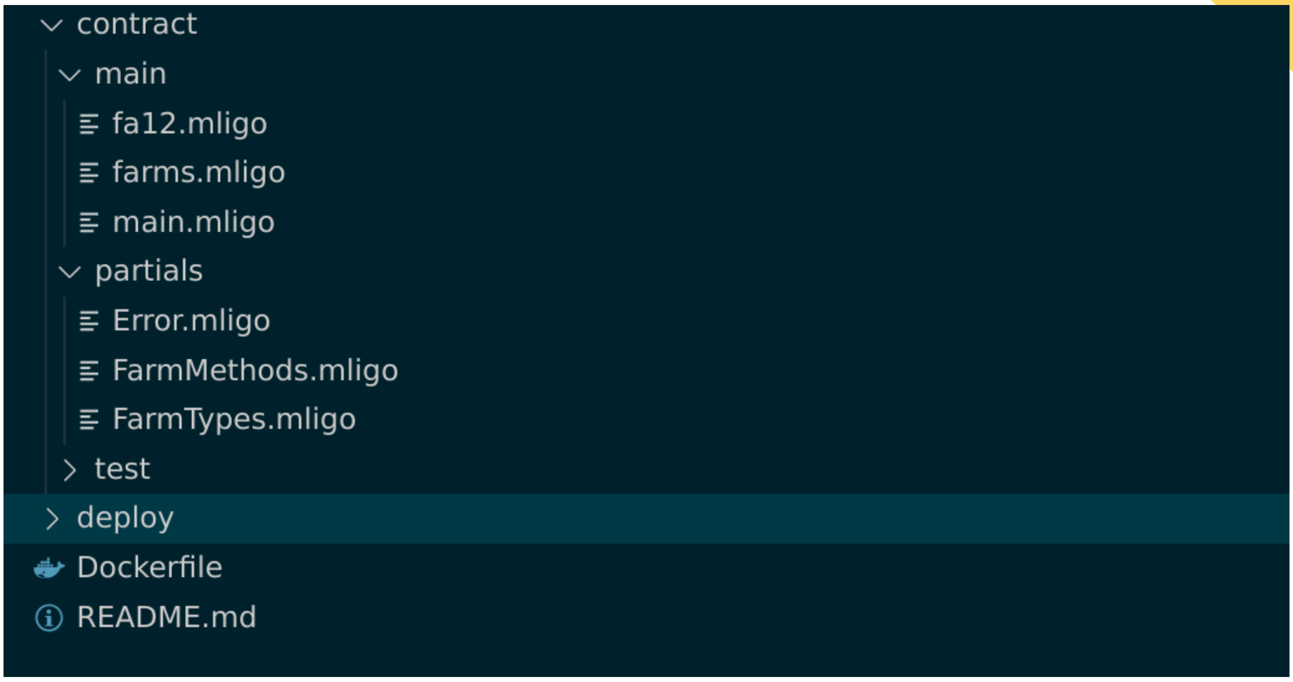
Contents

Q1. Source files organisation

Category	Impact	Location	Status
Organisation	Maintenance & Scalability	root folder	

Description

The project is organise using the following tree :



```
contract
  main
    fa12.mligo
    farms.mligo
    main.mligo
  partials
    Error.mligo
    FarmMethods.mligo
    FarmTypes.mligo
  test
deploy
Dockerfile
README.md
```

There is several remark to make on the organisation that could hinder scalability in the futur:

1. not very important but the sources parents directory is name *contract* instead *src* and the *test* folder should be outside this folder. Those are just convention that can be of course modify, but the importance of following convention is to improve performance of developer (not having to think about where to look at things) and tools (script breaking for folder being at the wrong place).

2. The code is split between ‘main‘ and ‘partials‘. ‘partials‘ is a bad name as it doesn’t give extra information. A better name would be *farm* as it regroup the implementation of the Farm logic for the project.

3. The file in the *partilas* folder have inconsistent naming, some are prefix with *Farm* others are not. In the case where the folder is named *farm*, both options are valid but shouldn’t be mixed. In the other case, the prefix should be used. As it is, it give the impression that *Error.mligo* is for all project while it is only for the Farm.

4. The *fa12.mligo* file should not be your implementation (see). It should be place in a *lib* or *vendors* folder next to the *src* folder

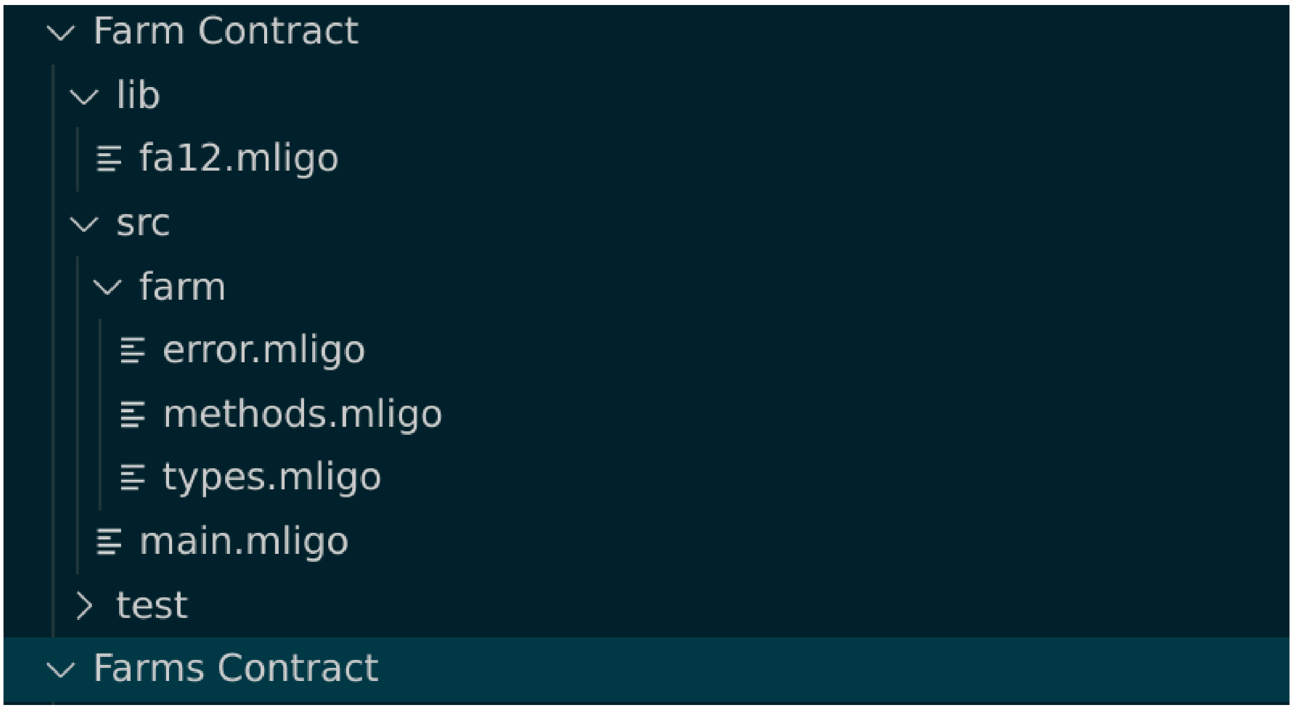
5. The *main* folder contains a *main* and a *farms*. This is already non standard but it turns out that *farms* is a standalone contract that does something else entirely and should be outside of this folder tree.

6. The close naming of the two contract *farm* and *farms* can leads two confusions, like mixing up the two. *farms* also lack precision, is it a manager ? is it a factory ?

7. The file use different case convention. *Camel* case in *main*, *Pascal* case in *partials*. It is common practice to use different case for differentiate things (i.e. file vs folder) but it's bad practice to mix it for the same things, like source code filenames.

Solution

Following this recommendation will lead to this filetree:



```
∨ Farm Contract
  ∨ lib
    fa12.mligo
  ∨ src
    ∨ farm
      error.mligo
      methods.mligo
      types.mligo
    main.mligo
  > test
∨ Farms Contract
```

Q2. Deprecation: `include`

Category	Impact	Location	Status
Deprecation	Maintenance	all source files	

Description

The `#include` preprocessor directive, was introduce very early in LigoLANGdevelopment to quickly meet the demand for an ability to split the code into multiple file. What it does is literally insert the text of the include file at the location, which can lead to low-level "hack" and other improper design pattern. Moreover it necessitate all source file to have the same syntax.

A year ago, LigoLANGhas released a proper solution for separate compilation with the `#import` primitive. `#import "<path to file name>" "<ModuleName>"` will produce the compilation of the file at `"<path to file name>"` alone, verifying that it typechecks, and package its declarations into a module `"<ModuleName>"` to not interfere with local declarations. It is possible to import file written in other flavors.

Since then, `#include` is deprecated

Solution

Replace all `#include "<file name>"` with `#include "<file name>" "<ModuleName>"` choosing `"<ModuleName>"` wisely. The code in the files has to be adapt replacing `<symbol>` from `"<file name>"` with `<ModuleName>.<symbol>`

Q3. Replace error string by error code

Category	Impact	Location	Status
Suggestions	Origination cost	contract/partials/Error.mligo	

Description

The contract define 14 custom error message, accounting for 600 extra bytes in the final contract which is 10% of the rest of the code. replacing them with integer will reduce the total size of the contract from 7338 bytes to 6581 bytes, and thus reducing the gas cost

This however comes with several drawbacks : you have to provide the user with an interpretation tables and you need to be assure that they are not conflicting with other error codes from libraries

Originating the contracts on the mockup node, we obtain (7564 bytes, 2930.956 gas, tz1.95525) against (6983 bytes, 2923.030 gas, tz1.81)

Solution

Replace all string in *Error.mligo* with non repeated integer and add a signification table in your documentation.

Q4. Code unit organisation

Category	Impact	Location	Status
Organisation	Readability	contract/partials/FarmTypes.mligo	

Description

1. The source file contains constant declaration in the middle of the type declaration. This is a problem for two reason : First, there is no reason to declare not first declare all the type and then all constant. Second, since the filename end with *Types* the assumption is that it contains only types and the constant are define in another file and at usage, it would be counter intuitive to write `FarmTypes.constant_name`

2. The type of the entrypoint may be better located in the main file. The reason is that files in this folder should be focus on describing a farm and the entrypoint type describe the contract entrypoint, but the difference is really thin. It is also an expectation to read the entrypoint type next to the entrypoint. But this it also apply recursively to the type in the entrypoint typedef leading to move all types to the main file which may not be the best solution

Solution

1. Either rename the file and move the constants to the end or move the constant to a new file, *FarmConstants.mligo* using you current convention.

2. Move entrypoint typedef to *main.mligo*

Q5. Checking assumption

Category	Impact	Location	Status
Language issue	Readability	contract/partials/FarmMethods.mligo	

Description

In the file, assumption are verified using the syntax

```
let _check_if_no_tez : bool =
  if Tezos.amount = 0tez
    then true
    else (failwith(amount_must_be_zero_tez) : bool)
in
```

The good point of this line is the naming of the constant. However, there is not really a reason for it to be a *boolean*. The condition should return a *unit* instead, and would make the cast of the failwith unecessary.

At some location, the syntax deviate to make a constant definition if the assumption is true. This should be avoided as it make the code less readable and the gain on the generated code is uncertain.
There is a function in LigoLANGmade for this purpose : `{assert_with_error}`which can be use to improve clarity.

Solution

```
let _check_if_no_tez : unit =
    asssert_with_error (Tezos.amount = 0tez) amount_must_be_zero_tez in
```

Q6. Computation of remaining rewards

Category	Impact	Location	Status
Code smell	Gas cost	contract/partials/FarmMethods.mligo	

Description

In *increaseReward*, to compute the new amount of reward to distribute, we need to know how much has been distributed already, to know how much is left before increase. This is done by parsing the map of reward per week and summing the values for the past weeks, but checking which are in the set of remaining weeks.

First of all, a similar code can directly sum the values for the remaining weeks, which is what we want in the first place and will avoid an unnecessary natural subtraction followed by a cast. Most importantly, the set of the remaining list is unnecessary for this check. We just have to compare the map key against the current week number. This reduce the algorithmic complexity from $O(n \cdot log(n))$ to $O(n)$

Solution

Replace

```
let folded (acc, elt: nat * (nat * nat) ) : nat =
                if Set.mem elt.0 weeks_set then acc  else acc + elt.1 in
let sum_R : nat = Map.fold folded s.reward_at_week 0n in
let new_r_total : nat = delta + abs(s.total_reward - sum_R) in
```

with

```
let folded (acc, elt: nat * (nat * nat) ) : nat =
                if  elt.0 < current_week then acc  else acc + elt.1 in
let remaining_r : nat = Map.fold folded s.reward_at_week 0n in
let new_r_total : nat = delta + remaining_r in
```

and remove the unused `weeks_set`

Q7. Iteration in functional programing

Category	Impact	Location	Status
Code Smell	Gas cost	contract/partials/FarmMethods.mligo	

Description

When calculating the reward for all the weeks, the program iterate over the week indices. The code generate a list of indices and run a recursive function over them, which reminds of the python style loops. This lead to writing custom code for handling the parsing of the list element and the stopping condition, which may be incorrect or inefficient.

Solution

When iterating over collection, the functional operators `iter`, `map` and `fold` has to be use instead. `iter` to perform test on elements, `map` to apply a function on all elements separately and return the collection of the result, `fold` to accumulate the function result in a single value, or propagate an effect while parsing the collection. In this case, we want a `fold`

replacing

```
let rec modify_rewards_func(resulting_acc, weeks_indices : (nat, nat) map * nat list) : (nat, nat) map
  let week_indice_opt : nat option = List.head_opt weeks_indices in
    match week_indice_opt with
    | None -> resulting_acc
    | Some(week_indice) ->
      let modified : (nat, nat) map =
          update_reward_per_week_func(week_indice, s.rate, s.weeks, s.total_reward, resulting_acc) in
      let remaining_weeks_opt : nat list option = List.tail_opt weeks_indices in
      let remaining_weeks : nat list = match remaining_weeks_opt with
        | None -> ([] : nat list)
        | Some(l) -> l
      in
      modify_rewards_func(modified, remaining_weeks)
 in
 let final_rewards : (nat, nat) map = modify_rewards_func(s.reward_at_week, weeks) in
```

with

```
 let modify_rewards_func(resulting_acc, weeks_indices : (nat, nat) map * nat list) : (nat, nat) map =
       update_reward_per_week_func(week_indice, s.rate, s.weeks, s.total_reward, resulting_acc)
 in
 let final_rewards : (nat, nat) map = List.fold modify_rewards_func s.reward_at_week weeks in
```

Furthermore, we don't actually want to iterate over a list here, we want to iterate with a counter from 1 to max_1, un functional programming this is done with the `unfold` function which is not yet implemented in LigoLANG, but can be easily done by using tail-recursive function

```
let rec modify_rewards_func(resulting_acc, weeks_indice : (nat, nat) map * nat) : (nat, nat) map =
  if week_indice > s.weeks
    then resulting_acc
    else
      let modified : (nat, nat) map =
        update_reward_per_week_func(week_indice, s.rate, s.weeks, s.total_reward, resulting_acc) in
      modify_rewards_func(modified, week_indice + 1)
in
let final_rewards : (nat, nat) map = modify_rewards_func(s.reward_at_week, weeks) in
```

which doesn't necessitate the creation of a list.

In the case that you do need to parse a list with a recursive function, because of more complex processing that you can't do with the classical iterators. Or because you need one that has not been implemented yet. Then you will need to extract the head and the tail of the list. But not with `List.head_opt` and `List.tail_opt`, but with the list destructuring `head :: tail` like such

```
let rec function lst res =
  match lst with
    | [] -> res
    | hd :: tl ->
      // do stuff using the current value of the list hd and potential extra argument of the functi
      function tl res
```

Q8. Cost of nat and Week sequence index

Category	Impact	Location	Status
Code Smell	Gas cost	contract/partials/FarmMethods.mligo	

Description

Natural in Michelson and Ligo are a data structure whose purpose is too guaranty that they will never be negative. But this guaranty comes at the cost of extra `abs` operation when making operation with integers and subtraction. These could be avoided by not using naturals when the guaranty is not necessary. Not using abs when the sign of an operation can be know from static analysis and reducing the number of subtraction required.

When indexing element of a sequence, there is either the possibility to start at 1 the index encoding the position, or at 0, the index encoding the distance from the first element. By choosing the first encoding, getting the distance will require subtracting by 1 and an `abs` if the index is a natural. With the later encoding, the former is obtained by adding 1 and no cast are necessary.

Solution

Index your maps starting from 1 and adapt your operation accordingly.

Q9. Map update

Category	Impact	Location	Status
Language issue	Gas cost	contract/partials/FarmMethods.mligo	

this :

```
let value_opt : nat option = Map.find_opt (week_indice+offset) map_accumulator in
let new_map : (nat, nat) map = match value_opt with
| None -> Map.add (week_indice+offset) result map_accumulator
| Some(_v) -> Map.update (week_indice+offset) (Some(result)) map_accumulator
in
new_map
```

is logically equivalent to

```
Map.update (week_indice+offset)(Some(result)) map_accumulator
```

but much more costly in gas

Q10. Save calls to function

Category	Impact	Location	Status
Language issue	Gas cost	contract/partials/FarmMethods.mligo	

Description

The syntax is confusing but `Tezos.sender` is not an access to the variable `sender` that is store somewhere. It is a call to the operation `SENDER` with talk to the protocol to get the sender. Using this value multiple time means calling it multiple time which should be more expensive than a data access.

Solution

Store the result of the call in a variable and use the variable instead. And that applies to other processing as well, similarly to **??**. If the call to `SENDER` is cheaper than a data access. You can easily revert the change by using the `[@inline]` attribute for the variable declaration.

Q11. Use coherent data structure

Category	Impact	Location	Status
Language issue	All	contract/partials/FarmMethods.mligo	

Description

After closer inspection to your coding style. It seems to me that your choice of data-structure, more specifically the maps was an attempt to recreate Array in functional programming which doesn't have them. Which then lead you to many code smells for trying to use an imperative style in a functional programming language.

Array doesn't make in functional programming because al variables are immutable. Which means that when you modify a variable you actually create a new variable with the new value. Then modifying and array cost $O(n)$, accessing an immutable array is not different than accessing a variable, and parsing an array is equivalent to parsing a list. That why their is no array collection in functional programming.

The immutability of variables means that we need to separate the processing and the data. In imperative programming, you often iterate over a collection or an value, perform some processing and then store the result into a mutable array before going to the next loop. which correspond to the abstract model of a Turing or a Von Newman machine. You take data from a memory, you process it and then you store it in your writable memory. In functional programming, we define functions the describes how to transform and process a datum, and then we apply these function on our data to get a result (with `iter`,`map` or `fold` or others). It is the abstract model of a data processing pipeline, with the data coming from one source (for instance light) and going through several transformation (for instance a camera) to get the output data (in that case, the numeric video flux). If it helps, you can imagine a product going through an assembly line.

This imply that having an index to get data from a collection and store it to a collection in an iteration doesn't makes a lot of sense in functional programming.

Solution

Instead of using Maps to implement Array, use List. Maps should be reserve for key that are not just index from 1 to n and data that you won't just loop through it. They are look up table, that you use when you need to search if a value exist or get a specific value while List are use for a sequence of data that you will parse and transform. `user_points` and `user_stakes` are a good use of Maps. There is nothing wrong with the use of Maps for weekly reward and weekly points. But not using them will force you to adopt a functional coding style

The reason you try to get an index in you loop is because you want to get a value at the same position in another array. Even in some imperative language like python, this is better done by iterating over both data structure at the same time using `map2` or making a single list with a pair of the two datas, using `zip`. Both are not yet implemented in ligo, so instead you'll have to fall back to recursive function like this :

```
let rec function lst1 lst2 res =
  match lst1, lst2 with
    [], [] -> res
    [], _lst -> failwith "size don't match"
    _lst, [] -> failwith "size don't match"
    hd1::tl1, hd2::tl2 ->
      // do your processing with hd1 and hd2 as your current data
      function tl1 tl2 res
```

This work for any number of data structure.

Q12. Factorize code by user

Category	Impact	Location	Status
Code Smell	Readability	contract/partials/FarmMethods.mligo	

Description

Following the previous point, the code of the `claimAll` function is odd. The code is constantly modifying the `user_point` Map which is inefficient unless you are handling data for several user. This challenges a functional program assumption and makes it harder to read.

Solution

Use the expected and better design.

1. Take the data concerning the user to process
2. Feed them to the processing function for one user.
3. Store all the results at the same time

Q13. Reducing number of operation

Category	Impact	Location	Status
Code Smell	Gas cost	contract/partials/FarmMethods.mligo	

Description

When sending the reward money to the user, the contract send an operation by weeks. This multply the number of transaction for the network to handle and the fee to handle them by the number of the week

Solution

Unless their is a good reason to separate the reward per weeks. Sum the reward and send them in one transaction.

Q14. Don't make NOP entrypoint

Category	Impact	Location	Status
Code Smell	Gas cost	contract/main/farms.mligo	

Description

The farms contract has an entrypoint that does nothing on purpose. I really don't see the point of such an operation. You are just paying money for no doing anything. They may be a use case for calling the entrypoint from another contract but in that case it make more sense to retrieve the current storage before the transaction from `https://better-call.dev/mainnet/KT1FtoNPPTQUuFFmzF6Ee7mStSZg6w`
`storage`.

Solution

Don't do it