# Summary:

Nomic is a unique game proposed by Peter Suber in 1982: «Nomic is a game in which changing the rules is a move. In that respect it differs from almost every other game. The primary activity of Nomic is proposing changes in the rules, debating the wisdom of changing them in that way, voting on the changes, deciding what can and cannot be done afterwards, and doing it. Even this core of the game, of course, can be changed.… » (see [1], [2])

The objective of this document is to describe how a Nomic game can be handled by a computer, to propose a set of requirements and finally a design and an implementation in Haskell.

The proposed game is online and can be played by several players simultaneously. The computer manages the pool of laws, checks automatically their validity, and apply them.

To win, one must achieve an objective fixed by one of the initial laws. Of course this objective can be changed during the game.

# Game description

## Description of a game turn

The game consists into proposing new rules or modifications for the existing ones and voting for them. The rules are written in a special language.

The engine will then check if the proposed rule is valid against the already active rules. The new rule will be either accepted or rejected. If it is rejected, the player will be able to modify it and re-submit it. If it is accepted, it becomes « active » and is added to the pool of active laws.

Once active, the new rule is executed and can modify the state of the game.

## Examples of rules

1. A rule mustn’t do nothing.
2. The rule number 1 is abrogated.
3. A rule mustn’t suppress another rule.
4. A rule must be voted at the unanimity by the players.
5. A new currency is created: the ECU. Every player has a bank account in ECU.
6. The players wins 1 ECU by rule proposed and 5 ECU by rules accepted.
7. With X ECU the players can buy some powers…
8. The rules must be named. The names have the form RI1, RI2…
9. A player cannot propose more than two rules at once.
10. The first player with 100 Ecu wins the game.
11. A player receives 1 Ecu each day at midnight.
12. The democracy is abolished. God save the King, Player #2 !
13. The laws must be approved by the King.
14. The King can be overthrown if a referendum is thrown and won against him, with an absolute majority of the players.
15. This rule must be executed once and suppressed.
16. The player 1 cannot propose new rules.
17. The rules 1 to 10 cannot be suppressed.
18. The player #1 must sing a song out loud.

## Example of game:

1. Start the game / connect to the web server
2. Register as a player
3. Join an existing game / create a new one
4. Display of the state of the game, including the players and the active rules
5. The player #1 proposes a new rule
6. This rule is scanned by the existing rules. If « voting » rules are present, the players must vote for or against the rule.
7. If the new rule succeeds the test, it is accepted and becomes active. Is then executed.
8. If it fails, it is rejected.
9. Another player can propose a new rule (can be done simultaneously).
10. Etc. until the winning criteria (defined by one rule) is met.

# Analysis

## Analysis of the game

The main object handled by the game is the “rule”. It is so important to give a precise definition to this concept in Nomic. We’ll define a rule as: “A rule is a proposition that allows determining the legality of an object”. The main object manipulated by Nomic being the rule, a rule is then a proposition that allows determining the legality of another rule.

This auto-referential definition is rich and gives a lot of interest to the game. Of course, beside from this, when executed, a rule can also change the state of the game.

## Analysis of the rules

From the analysis of the example rules listed above, we can define several necessary concepts and categories:

### Time :

* Rules with a transient effect: ex. #2
* Rules with a permanent effect: ex. #4
* Rules with a regular effect: ex #11

Indeed rule #2 should be deleted as soon as it is applied: we don’t want to suppress every rule with the number #1, only the current one.

On the other hand, the rule #4 must be permanently added to the rule table.

Some rules are applied on a regular basis, like rule #2.

### Subject:

* The rules that can modify the rule set: ex. #2.
* The rules that gives conditions on the future rules: ex. 4
* The rules that does not have the other rules as subject.

### Perimeter :

Some rules introduce new notions, like rules #5 or #7. New vocabulary is added. Others are just using the existing vocabulary.

# Requirements

## Requirements on the rule engine

1. The engine shall allow the player to compose a new rule in an appropriate language.
2. The engine shall allow a rule to be the composition of two other rules.
3. The engine shall allow a rule to access the state of the game to make decisions.
4. The engine shall allow a rule to change the state of the game.
5. The engine shall allow a rule to read another rule and make decisions on its legality.

## Requirements on the rule language

The language shall allow a rule to add/modify/suppress other rules.

## Requirements on the game

1. The game shall be multiplayer and playable on the internet.
2. The game shall be mostly off-line, which means that the players do not need to be connected simultaneously to play in the same game. One player can connect, read the new events, provide some input, submit a new rule, and disconnect.

# Design

Note: the code given in this section is a pseudo-code based on Haskell. It will not compile.

## Rule flow

Here is the flow process followed when a new rule is submitted:

***Constitution***

* Rule 1
* Rule 2
* Rule 3

…

New Rule

Legal ?

Submission

Legal ?

Legal ?

Yes

Yes

Yes

No

No

No

Rejected

Accepted, add to the constitution

***Constitution***

* Rule 1
* Rule 2
* Rule 3
* New Rule

…

Execute the new rule (may change the constitution)

A rule is executed immediately after being accepted.

Some part of it must also be executed on a regular basis.

Based on this, rules have three use cases:

* A constitutional rule is executed on another rule
* A rule is read by a constitutional rule
* A rule is simply executed (without argument)

## Game State

The game state contains all the necessary information for the players.

This includes:

* The game name,
* The rule list,
* The inputs from the players,
* The list of players,
* The current variables,
* The victory flag.

The language keyword to read and write the game state is:

* Get
* Set <Game state>

## Events

In order to allow the rules to have some part of them executed on a regular basis or on event, a new keyword is added:

OnEvent <Event Name> <Code>

The list of events is:

|  |  |
| --- | --- |
| **Event Name** | **Description** |
| NewPlayer <PlayerNumber> | A new player entered the game |
| PlayerLeave <PlayerNumber> | A player left the game |
| Time <TimeCode> | Event raised based on the time code provided. |
| RuleProposed <RuleNumber> | A new rule is proposed. |
| RuleAccepted <RuleNumber> | A rule is accepted. |
| RuleSuppressed <RuleNumber> | A rule is suppressed. |
| Message <RuleNumber> <Data> | Another rule sent a message with data. |
| UserEvent <PlayerNumber> | The player did an action. |

On every event, the active rule list is scanned and the corresponding events are executed. This design allows reducing the stateness of the engine, because when a rule is suppressed, the corresponding event automatically goes away.

The event system is also used for message passing between the rules. This is a preferred mechanism over memory sharing.

The rules can send a message via the keyword:

SendMessage <Data>

The message can then be received by implementing the event handler:

OnEvent (Message <RuleNumber from>) <Code>

Some rules can also define user events with:

OnEvent (UserEvent <PlayerNumber pn>) <Code>

If an active rule implements this event handler, the corresponding player will have an input displayed on their HCI (like a button). If the player triggers the input (i.e. presses the button), the event is called.

This functionality is useful for rules that describe a user-triggered event, like a referendum. In the case of a rule describing a referendum, the user will have to press on the “Referendum” button to launch one.

Note:

The fact that we have to generate signals for rules and players management pleads in favour of not letting the rules accessing the game state directly and adding extra language keywords to access them.

On the other hand, this will add a lot of language keywords.

Problem:

If the event handlers stay in the rules, shall the rules be executed each time a signal is raised? This seems impossible.

## Variables

In order to store values like the amount of Ecu for each player, the game engine shall allow the players to create and manage variables.

The variables are stored inside an array (variable name, value) inside the game state.

There is no need for a new language keyword because variables can be read and write through the game state.

One concern with this design is that it is very stateful. A rule that creates a variable, when it is suppressed, will leave behind the variable, which may not be desirable.

On other possible design is to not store the variables in the state, but have a stack of events?

This way, each time you want to know the state, you unwind all the events and repeat all the creations and updates.

Can we reuse the events of Happstack?

Another possibility is to delete all the variables created by a rule when it is deleted. The rules can be considered as little programs and the variables are their memory. So it makes sense to free the memory when the program is terminated.

## Inputs and votes

The players can vote, according to some of the rules. The act of voting is treated by the system as a simple input. As the system is asynchronous (the player can connect anytime, they are not required to play all at the same time), we have to store the inputs in a table. This table can be read latter, when all the input is present (for example when all players have voted).

We need to define a keyword in the language that allows the rule to ask for input from a player. Let it be:

Input <inputReason> <Player> <Choice1, Choice2, ...>

To display the various input requests from the active rules on the player’s screen, the engine have to search the active rules for the input keyword. If it’s present, the input request is displayed on the players screen when logged in. The player has to make a choice on the GUI between the available choices. Once this choice is made, it is recorded in the game state.

When a rule is executed, and if it depends on player’s input, that input is searched in the input table in the game state. If all the inputs requested by the executed rule are present in the input table, the execution can proceed. Otherwise it is interrupted, as we must wait for all the players to provide their inputs.

In fact, the evaluation function of a rule over the other has two possible outcomes: either the Boolean output, or the list of remaining inputs necessary. This list is displayed on the HCI to ask for the user input.

Note:

Can we replace this input mechanism with UserEvent + variable management?

In this hypothesis:

RuleVote r =

newVariable “Votes” votes

vs <- getVariable “Votes”

ps <- getPlayers

let results = forM pn in ps do

if pn not included in vs then

onEvent (InputEvent pn choice) do

setVariable “Votes” vs:(pn:choice)

cumulate false

else

cumulate true

return and results

## Output

Can the rules make outputs when executed? The problem is that they can be executed at anytime, maybe several times, and when the player is not here...

It’s necessary to have a buffer of outputs time stamped that the player can consult.

## Rule

A rule comes with some decoration. It is composed of:

* a unique rule number,
* a name,
* a description,
* a “rule function”,
* and it’s plain code in the form of a string.
* a state

A “rule function” is the core of the rule. It is a piece of code that allows the rule to judge whereas a new rule is legal or not, based on the code of this rule, and the current game state.

The rule function is simply the interpretation (by a Haskell interpreter) of the plain text rule.

The rule function, the most important part, is indeed a function with one argument: this argument can be a rule, or nothing. This function, when applied to a rule, returns a Boolean representing the legality of the rule in parameter. It is also able to read and write the state of the game.

After being accepted, the rule is executed with the parameter ‘Nothing’. It can then change the game state.

A rule can be:

|  |  |
| --- | --- |
| **State** | **Description** |
| Active | The rule is active, new rules will be subject to its analysis. |
| Pending | The rule has been submitted by a player, but is not yet active, maybe because some input are needed. |
| Suppressed | A rule once active but the suppressed by another rule. |
| Rejected | A rule never active and rejected by another rule on submission. |

## Display

On the user interface, we display:

* The list of players
* The list of games
* A particular game the player is in
* The list of active rules and pending rules
* Optionally the list of suppressed rules
* The pending inputs
* The variables
* The submission fields for a new rule
* A “Amend Constitution” button

In a separate panel, we can display the code available in the rule space. This code with helper functions and examples can help the player to understand and compose their rules.

## Load/save

It is important to be able to load in the game some external code, in order to avoid having to retype it in each game, and to factorize between several rules.

The software proposes a load function where you can submit a file to be loaded in the rule space. This file can contain subprograms that the rules can call. However, the rules must still be proposed individually by the player.

The save function saves to a file the complete content of the rule space. This code will be available for loading latter.

Open question

Can the rules define sub-programs? For themselves it should be yes, and for other rules?

It is not clear whereas this load function should allow the rule behaviour to be modified. A rule should be voted and pass through the acceptance mechanism so probably not.

Redefining an already existing subprogram should be forbidden.

We shall provide a library mechanism to store and load easily different rules and utility.

## Language

In recollection with the needs expressed in the previous chapters, we need in the language:

|  |  |
| --- | --- |
| **Name** | **Description** |
| NewVar |  |
| DelVar |  |
| GetVar |  |
| SetVar |  |
| OnEvent | Sets a handler on an event (see list) |
| SendMessage | Triggers a “Message” event |
| Input |  |
| Output |  |
| AddRule |  |
| DeleteRule |  |
| ModifyRule |  |
| SetVictory |  |

## Interpreter

The player will submit the rules written in this language, in the form of a string. They will be read by an Haskell interpreter to be transformed in an AST described above.

## Further developments

The main idea of Nomic is to experiment with the self amendment paradox. We want to explore a bit about programs analysing and modifying programs.

In this respect, the proposed design may not be very good: it may not be easy to analyse the rule functions themselves because they are written in native Haskell. The solution may be to extend the existing GATD used (named Exp) to be more descriptive?

# Use cases

## Use case 1: voting a new rule

Let’s suppose we have the following rules active in the constitution:

RuleVoteUnanimity: Every new rule must be voted at unanimity.

The code of this rule will:

* + Get all the players,
  + Issue an “Input” command on all the players, with the choices “For” or “Against”,
  + Return whereas everyone voted “for”.

This rule enforces that every new rule get voted.

A player proposes a new rule:

RuleProposed: Output “Hello World”

Once proposed, the new rule is in the state “Pending”.

If the button “Amend Constitution” is pressed, all pending rules will be evaluated against all active rules. In our case, the rule “RuleProposed” is evaluated against “RuleVoteUnanimity”.

As the “RuleVoteUnanimity” cannot complete without user’s input, the process should not terminate and the proposed rule will remain pending.

As a parallel process, the rule pool will be examined (with the same evaluation function) to determine the inputs needed from the user. In our case, as a deduction from the active and pending rules, the following input will be displayed on the player’s screen:

Please enter your input for RuleVoteUnanimity: “For” “Against”

Let’s assume the “For” link is clicked. The corresponding input will be recorded. Then, on the next click on “Amend Constitution” (by whatever player), the evaluation process will be able to terminate. The new rule is then “Active”. It will be executed, which will result in displaying “Hello World” in the output buffer.

# Examples

In this chapter we’ll try to implement in a pseudo language the rules examples defined in chapter 2.2.

1. A rule mustn’t do nothing.

That’s impossible to do in the general case (unless we implement theorem proving?).

In practical we can check if the rule modifies the current state or not, and if it installs some event handlers.

1. The rule number 1 is abrogated.

We can modify directly the game state:

rule2 \_ = do

rs <- gets activeRules

store activeRules (tail rs)

1. A rule mustn’t suppress another rule.

rule3 r = do

rs <- gets activeRules

return and (map r rs)

1. A rule must be voted at the unanimity by the players.

rule4 r = do

results <- MapM $ Input (“Vote for rule number “ ++ ruleNumber r) [“Yes”, “No”] playerList

return and results

1. A new currency is created: the ECU. Every player as a bank account in ECU.

Rule5 \_ = do

ps = get Players

let accounts = [(playerNumber p, 0) | p <- ps]

newVariable “Accounts” accounts

OnEvent (NewPlayer pn) do

ModifyVariable “Accounts” (\as -> as:(pn,0))

OnEvent (PlayerLeave pn) do

ModifyVariable “Accounts” (\as -> filter (\a -> fst a /= pn))

1. The players wins 1 ECU by rule proposed and 5 ECU by rules accepted.

Rule6 \_ = do

OnEvent (RuleProposed pn) do

as <- GetVariable “Accounts”

SetVariable “Accounts” $ modify (\a -> fst a == pn) (\(a,b) ->(a,b+1)) as

OnEvent (RuleAccepted pn) do

as <- GetVariable “Accounts”

SetVariable “Accounts” $ modify (\a -> fst a == pn) (\(a,b) ->(a,b+5)) as

1. With X ECU the players can buy some powers…

Rule7 \_ = do

OnEvent (Time Hourly) do

as <- GetVariable “Accounts”

1. The rules must be named. The names have the form RI1, RI2…

We use a regular expression library to evaluate the name:

Rule8 r = do

Return (ruleName r) ~= “RI[1..]”

1. A player cannot propose more than two rules at once.

We simply check that the two preceding rules are not proposed by the player.

Rule9 r = do

Let pn = proposedBy r

Let rn = ruleNumber r

Return proposedBy (getRuleByNumber $ rn - 1) /= pn ||

proposedBy (getRuleByNumber $ rn - 2) /= pn

1. The first player with 100 Ecu wins the game.

We scan the accounts and declare victories (there can be ex-aequo)

Rule7 \_ = do

as <- GetVariable “Accounts”

let winners = map fst $ filter (\(pn, e) -> e >=100) as

Victory winners

1. A player receives 1 Ecu each day at midnight.

We implement a hook on a time event and increment the accounts (same as above).

1. The democracy is abolished. God save the King, Player #2 !

We implement a rule that suppresses all democratic rules (by changing the state).

The name of player 2 may be also changed and his account gratified! :)

The playerNumber of the king can be hold by a variable for flexibility.

1. The laws must be approved by the King.

We add a new voting rule only on the king.

1. The King can be overthrown if a referendum is thrown and won against him, with an absolute majority of the players.

This one is delicate. The referendum law must be already in the law pool and non modifiable by the king? What prevents the king from suppressing it?

How to trigger the referendum? Is there a signal that the players themselves can launch?

1. This rule must be executed once and suppressed.

The rules can access the rule pool and suppress themselves. The language must provide a way to know the self number of a rule.

1. The player 1 cannot propose new rules.

Rule16 r = do

Return (proposedBy r != 1)

1. The rules 1 to 10 cannot be suppressed.

Rule16 r = do

getRule

ruleFunc r

Return (proposedBy r != 1)

1. The player #1 must sing a song out loud.

Obviously this one will be hard to check automatically. Therefore, it must be checked manually.

The idea is that the players must validate that player 1 sang a song, and maybe vote about it. Actions must be foreseen on the outcome of the vote (for ex. A money prize or a sanction).

# Implementation

## Rule

A rule comes with some decoration. Included is a unique rule number, a name, a description, a “rule function” and its plain code in the form of a string.

Type Rule = Rule { ruleNumber :: Int,

ruleName :: String,

ruleDesc :: String,

ruleFunc :: RuleFunc,

ruleCode :: String}

A rule “function” (“ruleFunc”) is the core of the rule. It is a piece of code that allows the rule to judge whereas a new rule is legal or not, based on the code of this rule, and the current game state. This leads to the following functional definition of a rule function:

Type RuleFunc = Maybe Rule -> StateT GameState Exp Bool

The rule function is simply the interpretation (by a Haskell interpreter) of the plain text rule (“ruleCode”).

The “ruleFunc”, the most important part, is indeed a function with one argument: it can be a rule, or nothing. This function, when applied to a rule, returns a Boolean and is able to read and write the state of the game, due to the monad state.

After being accepted, the rule is executed with the parameter ‘Nothing’. It can then change the game state.

On a style note, the drawback of this implementation is that when executed after being accepted, the rule is passed with a Nothing argument, and the returned Boolean is dismissed, which is not very elegant.

Alternative implementation:

In order to clearly separate the meta rules (rules that judges rules) from the normal rules (rules that just change the game state), we can do:

Type NormalRule = State GameState Exp ()

Type MetaRule = Maybe Rule -> State GameState Bool

Data RuleFunc = NR NormalRule | MR MetaRule

The meta rules will be managed with the NewRule event.

If a new rule comes, then the corresponding event will be raised, and the sub-rule will be called. It can then suppress the proposed rule from the list.

Problems: the constitution mechanism is less visible. How to deal with pending inputs?

This implementation has the advantage to get rid of the parameter if the rule is not a meta rule. Thus, it’s a little more elegant for the execution. However, the type is more complex and redundant.

Problem:

We need to abstract on the subject. Indeed, the subject of a rule (what we have to assess the legality) can vary.

## Evaluation

The rules are evaluated as described in chapter 5. This evaluation mechanism acts like a virtual machine, which reads and executes the rules written in their specific language.

Remark: since it’s a complete event processing system, it could suffice on its own, without the need to have an hard coded mechanism of evaluation of the legality of the new rule as described in the diagram of §5. The new rule, upon coming, would trigger the NewRule event, and thus be potentially suppressed. Then it would be executed.

# References

[1] <http://www.earlham.edu/~peters/nomic.htm>

[2] <http://en.wikipedia.org/wiki/Nomic>

* How to mock a mocking bird
* Atoms
* Financial Haskell by S. P. Jones