

Exam Problem: Galápagos

Synopsis: Implement a simulation of multi-agent and multi-round Prisoner's Dilemma using cellular automata.

Preamble

This document consists of 20 pages including this preamble; make sure you have them all. Your solution is expected to consist of a *short* report in PDF format, as well as a `.zip` or `.tar.gz` archive containing your source code, such that it can be compiled immediately with `cabal test` (i.e., include the non-modified parts of the handout as well).

You are expected to upload *two* files in total as your entire exam submission. The report must have the specific structure outlined in appendix 5.

Make sure to read the entire text before starting your work. There are useful examples at the end.

In addition to the normal assignment rules listed on the course website, the following rules apply to the exam.

- The exam is *strictly individual*. You are not allowed to communicate with others about the exam in any way.
- Do not share or discuss your solution with anyone else until the exam is finished. Note that some students have received a deadline extension. Do not discuss the exam until an announcement has been made on Absalon that the exam is over.
- Posting your solution publicly, including in public Git repositories, is in violation of the rules concerning plagiarism.
- If you believe there is an error or inconsistency in the exam text, *contact one of the teachers*. If you are right, we will announce a correction.

- The specification may have some intentional ambiguities in order for you to demonstrate your problem solving skills. These are not errors. Your report should state how you resolved them.
- Your solution to the exam problem is evaluated holistically. You are graded based on how well you demonstrate your mastery of the learning goals stated in the course description. You are also evaluated based on the elegance and style of your solution, including compiler warnings, inconsistent indentation, whether you include unnecessary code or files in your submission, and so on.
- The exam is not a teaching situation, and teachers will answer only questions regarding ambiguities or errors in the exam text. Do not contact the TAs regarding the exam.

1 Introduction

It is time to touch grass! During this exam you will develop a simulation of a multi-round and multi-agent variant of the *Prisoner's Dilemma*, under the metaphor of Galápagos finches helping clean each others' plumage (or not) of parasitic mites¹. The simulation takes the form of a grid of cells, each of which may be empty or contain a finch. Finches may help their neighbours, may reproduce into neighbouring empty cells, and may die of disease or old age. Different species of finches will be present, each with their own strategy for when to help and when to ignore other finches. A strategy is represented by an APL program extended with various new facilities.

A complete solution to this exam thus involves many diverse components, but the exam text is structured into smaller *tasks*, each of which focus on a single aspect of the whole. Some tasks are independent, while others have dependencies—this is noted explicitly for each task. The background for the exam is presented in full before the tasks are listed, but not all tasks require a full understanding of the complete background.

The design of the system is largely an extension of the concepts you worked with in the week 7 exercises. Some of the code handout already contains complete data type definitions—you should not change these. The code contains explanatory comments in addition to this exam text. In contrast to the assignments, you are expected to put all tests in a single Tests module.

2 Background

2.1 Cellular Automatons

The material in this section is equivalent to that of the week 7 exercises.

A two-dimensional *cellular automaton* is a model of computation where the computation is structured as a grid of *cells*, each of which contain some data. Computation occurs by each cell interacting with its neighbours through some rule. Conway's Game of Life is an example of a

¹This is a remake of the OOPD 2006/2007 exam written by Christian Stefansen, now using a much better programming language.

cellular automaton that you may be familiar with. For this exam, each cell interacts only with its immediate (non-diagonal) neighbours.

A *block cellular automaton* is a variant of cellular automata where the grid is partitioned into non-overlapping blocks and interactions take place only for the cells inside each block. By regularly shifting the partition of the grid into blocks, a cell still gets to eventually interact with all of its neighbours. Block cellular automata are useful when the interactions are complicated. In this exam we use the *Margolus neighbourhood* partitioning scheme, which is demonstrated in fig. 1. We also assume a torus-shaped world, where the neighbours of a cell at the edges are found by wrapping around to the opposite edge.

For each iteration and partitioning of the grid into Margolus neighbourhoods, each neighbourhood is updated as follows:

1. The upper left cell interacts with the upper right cell, and the lower left cell interacts with the lower right cell.
2. The upper left cell interacts with the lower left cell, and the upper right cell interacts with the lower right cell.

This is shown in fig. 2. At the end of each iteration, the partitioning shifts by one. Note that this allows some concurrency within each Margolus neighbourhood, and more importantly, it also allows each neighbourhood to be updated independently.

2.2 Galápagos Finches

A finch has a number of *health points* (HP). When two finches *meet*, they each simultaneously choose to either *groom* or *ignore* the other finch. The HP of each finch is then adjusted based on the decisions made, as determined by a *payoff matrix*. An example of such a matrix might be the following:

Finch <i>a</i> / Finch <i>b</i>	<i>Groom</i>	<i>Ignore</i>
<i>Groom</i>	1 / 1	-4 / 3
<i>Ignore</i>	3 / -4	-2 / -2

From a self-centered perspective, the optimal outcome is to be groomed but ignore the other finch. However, in our simulation, finches have

2	2	4	4	6	6	8	8
2	2	4	4	6	6	8	8
1	1	3	3	5	5	7	7
1	1	3	3	5	5	7	7

(a) The division into Margolus neighbourhoods on even iterations.

8	2	2	4	4	6	6	8
7	1	1	3	3	5	5	7
7	1	1	3	3	5	5	7
8	2	2	4	4	6	6	8

(b) The division into Margolus neighbourhoods on odd iterations.

Figure 1: Splitting a grid with even edge size into Margolus neighbourhoods. The numbers serve only to identify which cells belong to the same neighbourhood—this is significant only in the odd case, where the neighbourhoods wrap around the edges of the grid.

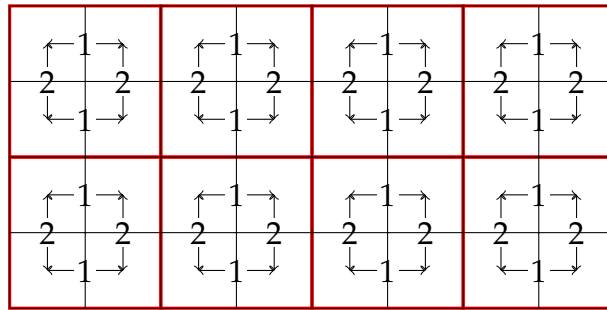


Figure 2: The order of interactions inside each neighbourhood: first the upper left cells interact with the upper right cells and the the lower left cells with the lower right, and then afterwards the upper left cell interacts with the lower left and the upper right with the lower right.

memory—they interact multiple times, and may base future behaviour on past experiences with another finch. This makes the optimal strategy more subtle — indeed, the point of such simulations is to study the benefit of altruism (although our simulation will be too crude to allow meaningful conclusions). Designing good strategies is not a part of this exam, and examples of strategies can be found in appendix 1.

Each finch has a *finch identifier* (FID), which is a randomly generated integer used to recognise finches. For simplicity we do not guarantee that a FID is unique, but the chance of two different finches sharing a FID is vanishingly small.

Each finch also has a *lifespan*, indicating how many interactions it has left before it dies of old age.

The finches are located on a grid, and interact only with their immediate (non-diagonal) neighbours. A grid cell may be empty or contain a finch. Two cells interact with the following procedure:

1. If both cells are empty, nothing happens.
2. If one cell contains a finch and the other is empty, there is a chance of *reproduction*, which causes a new child finch to appear (details in section 2.2.2). Interaction continues, meaning the parent will immediately meet its child.
3. If both cells contain finches, then they *meet*—each finch decides whether to groom or ignore, and HPs of the finches are updated accordingly (up to a certain maximum).
4. Each finch (if any) is aged, meaning its lifespan is decremented by one.
5. If the lifespan or HP of a finch is zero or less, the finch is removed.

The overall parameters of the simulation, such as the contents of the payoff matrix and the maximum HP, are represented by the datatype `Params` in the `Galapagos.Rules` modules.

2.2.1 Initialisation

Initialisation of the simulation involves creating a grid of h rows by w columns, given a list of n potential strategies. Each cell, whether empty

or containing a finch, contains a random number generator (RNG) of type `StdGen`. For cell (i, j) , the RNG is initialised using `mkStdGen` with the seed $s + i \times 10000 + j$, where s is a parameter passed in by the user.

The (inverse) probability of a cell initially containing a finch is given by the `invDensity` field of `Params`. If a cell is determined to contain a finch, then the finch must follow the strategy of element $(i + j) \bmod n$ of the given list of possible strategies. The colour of the finch is taken from the chosen strategy. The FID of the finch is randomly generated. The initial HP and lifespan is given by the appropriate fields of `Params`.

2.2.2 Reproduction

When a finch reproduces, an empty adjacent cell is filled with a copy of the parent finch, except:

- The strategy is restarted, implying that memory is not inherited.
- The lifespan of the new finch is given by the `lifespan` field in `Params`.
- Its starting HP is given by the `startHP` field of `Params`.
- A new FID for the child is randomly generated.

Reproduction is not guaranteed even when an adjacent cell is empty, but determined randomly, with the (inverse) probability given by the `invReproductionChance` field of `Params`.

The random numbers must be generated using the RNG belonging to the initially empty cell.

2.3 Extensions to APL

For this exam you have to extend the APL language, familiar from your assignments, with new features to support the writing of finch strategies. Your starting point is a simplified version of the A4 handout code. Most significantly, the key-value store and error recovery have been removed, and the environment is passed explicitly in the `eval` function (as in A1), rather than monadically. The two features you will add are a new compound data type of *sets*, as well as three new effects meaningful for

Operators	Associativity
* /	Left
+ -	Left
union, without elem	Right
==	Left

Table 1: APL table of operators in decreasing order of priority.

writing strategies: `meet`, `ignore`, and `groom`. The effects have already been implemented, but you will have to interpret them in Task D. The full grammar of the extended APL is shown in fig. 3.

2.4 Visualisation

Once sufficient parts of the exam have been implemented (at least parts C, D, and F), you can run a visualisation of the simulation with the following command:

```
$ cabal run -- viewer example.rules
```

Here `example.rules` must be an appropriate configuration file as described in Task E - one is also included with the handout.

```

Atom ::= var
      | int
      | bool
      | "(" Exp ")"
      | "meet"
      | "ignore"
      | "groom"
      | "{" "{"
      | "{" Exp Exps "}"

Exps ::=
      | "," Exp Exps
FExp ::= FExp FExp
      | Atom
LExp ::= "if" Exp "then" Exp "else" Exp
      | "\" var "->" Exp
      | "let" var "=" Exp "in" Exp
      | "loop" var "=" Exp "for" var "<" Exp "do" Exp
      | FExp
Exp  ::= LExp
      | Exp "==" Exp
      | Exp "+" Exp
      | Exp "-" Exp
      | Exp "*" Exp
      | Exp "/" Exp
      | Exp "without" Exp
      | Exp "union" Exp
      | Exp "elem" Exp

```

Figure 3: APL syntax in EBNF. This grammar is ambiguous and contains left recursion, which you will be asked to address. See table 1 for operator priority and associativity. The new language constructs are described in the corresponding tasks. Whitespace is permitted between all terminals. The keywords are if, then, else, true, false, let, in, loop, for, do, meet, ignore, groom, elem, union, and without.

3 Tasks

3.1 Task A: Extend APL Evaluator with Sets

This task depends on no other tasks.

For this task you must implement *sets* in APL. A set is a collection of zero or more values. They are defined as follows in `APL.Eval`:

```
data Val
= ...
| ValSet [Val]
```

Sets are manipulated using expressions with the following AST constructors:

```
data Exp
= ...
| Set [Exp]
| Union Exp Exp
| Without Exp Exp
| Elem Exp Exp
```

The semantics of set expressions are conventional, but stated below for completeness.

A set expression `Set [e0, ..., eN-1]` is evaluated by evaluating the components from left to right, then constructing a `ValSet` with the resulting values. If the evaluation of any expression fails, evaluation of the set expression also fails and the first error encountered should be reported.

An expression `Union e1 e2` first evaluates its operands to sets, reporting an error if they are not sets, then constructs their union.

An expression `Without e1 e2` first evaluates its operands to sets, then produces a set where all the elements of the set produced by `e2` have been removed from the set produced by `e1`.

An expression `Elem e1 e2` first evaluates `e1` to any value v and `e2` to a set s , then returns a true value if $v \in s$ and otherwise false.

If an expression that expected to produce a set does not produce a set, then an error must be reported.

Your task: Implement evaluation of set operations in `APL.Eval` and add some tests to `Tests.taskATests`.

3.2 Task B: Extend APL Parser with Sets

This task depends on no other tasks.

For this task you must extend the APL parser to recognise set notation and operations, as specified in fig. 3 and table 1. Sets are constructed by

$$\{\}$$

corresponding to an empty set, and by

$$\{ \text{Exp Exps} \}$$

corresponding to a set with one or more elements. Both forms are represented with the Set constructor.

The set operations are the operators without, union, and elem

	Exp	“without”	Exp
	Exp	“union”	Exp
	Exp	“elem”	Exp

and represented by the corresponding AST constructors.

Your task: Implement parsing of sets and set operations in `APL.Parser`.

3.3 Task C: Concurrent Block-Cellular Automata

This task depends on no other tasks.

In this task you must implement a concurrent server for executing block-cellular automata. The execution semantics are the same as `BlockAutomaton.Simulation`. You must implement a highly concurrent simulation, where the number of threads is proportional (not necessarily identical) to the number of cells, and threads communicate with their neighbours through message passing. `BlockAutomaton.Server` specifies the external interface that you must implement. You must design and implement any internal protocol(s), including the forms of the messages. Interactions *within* each Margolus neighbourhood need not be executed concurrently, but it is important that the Margolus neighbourhoods within a step do not depend on each other.

Your task: Implement the functions specified in `BlockAutomaton.Server`.

3.3.1 Hints

1. If you find it difficult to solve this task in the intended fully concurrent manner, then simply implement `BlockAutomaton.Server` as a server-based frontend to the logic in `BlockAutomaton.Simulation`, although still with the prescribed types. This will let you demonstrate some familiarity with concurrent programming. Note that the intended concurrent implementation of `BlockAutomaton.Server` should not use `BlockAutomaton.Simulation` in any way.
2. For a concurrent implementation, one approach (but not the only correct one) is to model each cell as a process, with five channels: one for receiving requests from a supervisor server, and four for receiving requests from its neighbours. The challenge here is then to determine when a process should read from which channel.

3.4 Task D: Finch-Finch Interaction

This task depends on no other tasks.

In this task you must implement the logic for two finches interacting, including evaluating their strategies to make decisions on how to act. A finch is represented by the datatype `Galapagos.Rules.Finch`. Your task is to implement the function

```
meet :: Params -> (Finch, Finch)
      -> (Maybe Finch, Maybe Finch)
```

that implements two finches meeting, deciding how they interact by evaluating their strategy program (see below), then returning the two finches with their state modified accordingly. `Nothing` is returned in case a finch behaves illegally, which results in the finch being removed from the world (in Task F).

The strategy of a finch is given by

```
type Strategy = EvalM ()
```

that is, a strategy is a suspended execution within the `EvalM` monad. To decide how the finches a, b involved in a meeting should behave:

1. Evaluate the strategy of the finch a up to the next effect, which must be a `MeetOp`.

2. Invoke the continuation of the `MeetOp` with the `finchID` of finch *b*.
3. Evaluate the strategy up to next effect, which must be `GroomOp` or `IgnoreOp`. This decides the behaviour of finch *a* during this meeting.
4. Similarly, determine the behaviour of finch *b*, using the `finchID` of finch *a*.
5. For each finch, invoke the continuation of the `GroomOp` or `IgnoreOp` with the behaviour of the other finch (True for grooming, False for ignoring), yielding two new `Strategys`.
6. The resulting `Strategys` are then stored in the `Finch` objects for the next meeting of the finches.

If at any step a strategy does not produce one of the expected effects (or no effect at all), then the finch has behaved illegally. For example, if in step 1 the first effect is a `GroomOp`, then this is illegal.

If a finch *a* interacts with a finch *b* that behaves illegally, then finch *a* must not be modified - from finch *a*'s perspective, the interaction has not taken place (except if *a* also behaves illegally, in which case it is replaced by `Nothing` as usual).

If no finch behaves illegally, then the HP of each finch must be updated based on the result of the meeting and the payoff values in `Params`.

Your task: In `Galapagos.Rules`, implement the `meet` function.

3.5 Task E: Parsing Galápagos Configurations

This task depends on no other tasks.

A *configuration* of a Galápagos simulation is given by the datatype `Config` in `Galapagos.Rules`. It comprises the height and width of the simulation, its parameters, and a list of strategies. In this task you will implement a parser for a configuration file format, based on the grammar in fig. 4. Note that the height and width can be passed in any order. Note also that parameter settings may be passed in any order, each parameter may occur multiple times, or even not at all. If a parameter does not exist in a configuration file, its value is taken from `defaultParams` in `Galapagos.Rules`. The intention is that all parameters are initially as

specified in `defaultParams`, and are then overridden if they occur in the configuration file.

Your task: In `Galapagos.Parser`, implement `parseGalapagos`. Remember to remove the existing dummy definition first. Add tests to `Tests.hs` that demonstrate the ability to handle arbitrary orders (if implemented).

3.5.1 Hints

Do not spend much time implementing support for arbitrary ordering if you find it too difficult; just use the order they are listed in the grammar. You can also skip the parsing of parameters entirely if you are running short on time, as it is not required for most of the examples - if so, `Config` must always contain `defaultParams`.

3.6 Task F: Galápagos Finch Simulation

This task depends on tasks C and D, and partially on A and B, and E.

In this task you combine the previous parts to produce a fully operational Galápagos finch simulator. Concretely, you must implement the functions needed for the `galapagos` definition in `Galapagos.Rules`. These are the `initial` function for setting up the initial state (details to follow), as well as the sub-parts of the `interact` function, namely the functions `reproduce`, `groom`, `age`, and `kill`, which implement the different parts of the cell interaction as described in section 2.2.

The `initial` function must initialise a cell as described in section 2.2.1.

The `reproduce` function must implement the reproduction logic described in section 2.2.2.

The `groom` function determines if both cells contain finches. If so, it must carry out a meeting with the `meet` function implemented in task D. If a finch behaves illegally (`Nothing` is returned from `meet`), then it is removed here.

The `age` function decreases the `finchRoundsLeft` of a finch by one.

The `kill` function kills a finch whose `finchRoundsLeft` or `finchHP` has reached zero.

Your task: In `Galapagos.Rules`, implement the functions `initial`, `reproduce`, `groom`, `age`, and `kill`.

```

Size      ::=  "height" "=" int "width" "=" int
             |  "width" "=" int "height" "=" int

Param     ::=  "startHP" "=" int
             |  "invReproductionChance" "=" int
             |  "maxHP" "=" int
             |  "lifespan" "=" int
             |  "invDensity" "=" int
             |  "payoffs" "=" "(" int "," int "," int "," int ")"

Params    ::=  Param Params
             |

Strategy   ::=  "strategy" name colour "{" Exp "}"

Strategies ::=  Strategy Strategies
              |

Config     ::=  Size Params Strategies

```

Figure 4: The grammar for Galápagos configurations. The grammar is unambiguous and has no left-recursion, so it does not need to be transformed. A name is one or more alphanumerics, dashes, underscores, and dots. A colour is a # followed by a six-digit hexadecimal number (case-insensitive), encoding a 24-bit RGB colour, represented by the type `Galapagos.Rules.Colour`. An int is a decimal number with an optional sign, with no intervening whitespace. An Exp is an APL expression as given by fig. 3. All word-like terminals (`height`, `width`, `startHP`, etc) are keywords. Note that the APL keywords are *only* keywords in those parts of Galápagos configurations that correspond to APL expressions (e.g., “for” is a legal strategy name).

4 Code handout

The code handout consists of the following nontrivial files.

- `exam.cabal`: Cabal build file. **Do not modify this file.**
- `runtests.hs`: Test runner. **Do not modify this file.**
- `viewer.hs`: Visualisation program. No tasks require you to modify this file, but you may.
- `example.rules`: A sample set of simulation rules. No tasks require you to modify this file, but you may.
- `src/GenServer.hs`: The GenServer library. **Do not modify this file.**
- `src/APL/AST.hs`: The APL AST definition. **Do not modify this file.**
- `src/APL/Eval.hs`: The APL evaluator, which you will modify in task A.
- `src/APL/Monad.hs`: Definition of APL effects. **Do not modify this file.**
- `src/APL/Parser.hs`: The APL parser, which you will modify in task B.
- `src/BlockAutomaton/Rules.hs`: The block automaton rules abstraction. **Do not modify this file.**
- `src/BlockAutomaton/Server.hs`: Skeleton for the server-based implementation of block automata, which you will modify in task C.
- `src/BlockAutomaton/Simulation.hs`: A sequential simulator for block automata. **Do not modify this file.**
- `src/Galapagos/Parser.hs`: Skeleton for the parser of Galápagos configurations, which you will modify in Task E.
- `src/Galapagos/Rules.hs`: Skeleton for the Galápagos evaluation rules, which you will modify in Tasks D and F.
- `src/Tests.hs`: Skeleton for the complete test suite, including several useful definitions.

5 Your Report

Your report should be no more than ten pages in length and must be structured exactly as follows:

Introduction: Briefly mention very general concerns, any ambiguities in the exam text and how you resolved them, and your own estimation of the quality of your solution. Briefly mention whether your solution is functional, which test cases cover its functionality, which test cases it fails for (if any), and what you think might be wrong.

A section answering the following numbered questions:

1. Show the final APL grammar with all ambiguities and left recursion removed, and explain to which extent it resembles your parser implementation.
2. Suppose an APL program runs a loop with a large number of iterations where it repeatedly unions a set with itself ($x \cup x$). Does this cause memory consumption to increase?
3. In your implementation, what happens if a finch strategy goes into an infinite loop, and how could the design of the system be modified to handle infinite loops with a timeout, such that they are treated as the finch behaving illegally?
4. How confident are you that your implementation of Task C does not have race conditions, and why?
5. Roughly how many messages does your implementation of `stepAutomaton` send, and are they sent sequentially or concurrently? Does this have any impact on the concurrency of your implementation?
6. Suppose we wanted the offspring of a finch to have the memory of its parent, i.e., it would be more of a clone. What would be the simplest way to implement this, and what would be the downside of the simple way?
7. How could one test that the Galápagos configuration file parser correctly supports arbitrary parameter ordering, and have you done so?

Your answers must be as precise and concrete as possible, with reference to specific programming techniques and/or code snippets when applicable. All else being equal, **a short report is a good report.**

A Examples

Note that the Tests module also contains a few definitions that demonstrate usage of the specified modules.

The text file `example.rules` included in the handout also demonstrates a large configuration file.

1 Strategies

The following APL programs (plus Task E notation) express Galápagos strategies. Because APL lacks while loops, the strategies are written as for loops running for 100 iterations. If the lifespan of a finch exceeds 100 (not possible with the default parameters), then these strategies would result in illegal behaviour (as defined in Task D).

More strategies can be found in the `example.rules` file. You do not need to understand what the strategies do.

Samaritan: This finch always helps.

```
strategy samaritan #ff0000 {  
  loop x = 0 for i < 100 do let unused = meet in groom  
}
```

Cheater: This finch never helps.

```
strategy cheater #0000ff {  
  loop x = 0 for i < 100 do let unused = meet in ignore  
}
```

Student: This finch initially helps, then repeats the response it received forever.

```
strategy student #a68b3d {  
  let unused = meet in  
  if groom  
    then loop x = 0 for i < 100 do let unused = meet in groom  
    else loop x = 0 for i < 100 do let unused = meet in ignore  
}
```

Flip-flop: This finch flips between helping and ignoring.

```
strategy flipflop #a0a0a0 {
  loop x = 0 for i < 100 do
    let unused = meet in
    if i / 2 * 2 == i then ignore else groom
  }
```

Grudger: This finch remembers the actions of other finches, and refuses to help a finch that has ignored it in the past.

```
strategy grudger #992288 {
  loop jerks = {} for i < 100 do
    let other = meet in
    let outcome = if other elem jerks then ignore else groom
    in if outcome then jerks else jerks union {other}
  }
```

2 Colours

The following demonstrates how the colour terminal from fig. 4 should be parsed into Colour values.

#000000: RGB 0 0 0

#ff0000: RGB 255 0 0

#00ff00: RGB 0 255 0

#0000ff: RGB 0 0 255

#123456: RGB 18 52 86