Scoring Games Calculator

João Pedro Neto

Dez 19 draft

Contents

Introduction

Haskell is a functional language with lazy evaluation and static typing.

The Scoring Game Calculator (SGC) is implemented in Haskell, and it's run on the Haskell interpreter, provided the adequate module is imported.

This means the user can mix Haskell code and SGC functions to perform computations.

Warning: The program and this doc are in version 0.01 beta. As usual, bugs and typos are expected.

1 Haskell Nanotutorial

This section provides a very short tutorial on Haskell, so that the user can take advantage of the language potential.

The basic Haskell data structure is the list. Lists are sequences of same type data. The next are egs of lists of integers:

```
> [1..5]
 [1,2,3,4,5]
> [1,2,5] ++ [3,4]
 [1,2,5,3,4]    -- operator ++ concatenates lists
```

The main operation in Haskell is function application, like `f x` (no parenthesis):

```
> length [1,-2,3]
 3    -- -- length returns list size
> reverse [1,-2,3]
 [3,-2,1]
> [10,20,30]!!1
 20   -- get (i+1)-th element, same as (!!)[10,20,30] 1
```

Haskell has lambda-expressions:

```
> let positive = \x -> x>0
> positive 3
 True
> let succ = \x -> x+1
> succ 3
 4
> let add3 = \x y z -> x+y+z
> add3 1 2 3
 6
> let make = \n -> LE n [Nu (n+1)]
> make 0
 <^0|1>    -- see next section about Nu & LE
```

It's also possible to define new operators:

```
> let (-->) a b = not a || b -- define implies
> False --> True
 True
```

More on lambda-expressions: `http://www.cs.bham.ac.uk/~vxs/teaching/Haskell/handouts/lambda.pdf`

Functions can be high-order, ie, they can receive functions as arguments:

```
> filter positive [-1,2,-3,4]
 [2,4]
> map (\x -> 2*x) [10,20,30]
 [20,40,60]
> map make [1,2,3]
 [<^1|2>,<^2|3>,<^3|4>]
```

Also, we can apply composition of functions using operator '.':

```
> map ((\x -> 2*x).succ) [10,20,30]
 [22,42,62]
```

The operator `$` is the same has function application but with lower precedence. It is useful to save the writing of parenthesis:

```
> length (map (\x -> 2*x) (filter positive [1,2,3]))
 3
> length $ map (\x -> 2*x) $ filter positive [1,2,3]
 3
```

2 Instructions

2.1 Scoring Module

This module includes the apparatus to create and manipulate scoring games.

To access this module please load module `Scoring.hs`.

To list all available functions insert:

```
> commands
  -- big list of functions
```

To get individual help about one of those functions, say rightOp:

```
> help "rightOp"
  -- function description
```

What follows is a description of the module's available functions.

2.1.1 Constructors

There are four constructors of scoring games:

- `Nu n` endgame $\equiv \{\emptyset^n|\emptyset^n\}$
- `BE n m` endgame $\equiv \{\emptyset^n|\emptyset^m\}, n \leq m$
- `LE n g` left endgame $\{\emptyset^n|$[list of games]$\}$
- `RE g n` right endgame $\{$[list of games]$|\emptyset^n\}$
- `Op gL gR` game with left and right options $\{$[list of games]$|$[list of games]$\}$

Some egs:

```
> Nu 4
> Nu (−2)
> let g = LE 3 [Nu 4]
> g
 <^3|4>
> RE [g] 0
 <<^3|4>|^0>
> Op [Nu 1, g] [Nu (−1)]
 <1,<^3|4>|−1>
```

Notice how lists intermix with game constructors. This is necessary since the left/right options of a scoring game can contain several different games. They are thus organized in lists of games.

After an assignment, a game is presented in the format `<...|...>`. If the user wishes to see the true game representation, function `showRaw` shows it:

```
> let g = LE 3 [Nu 4]
> g
 <^3|4>
> showRaw g
 "LE 3 [Nu 4]"
```

If `n` is an integer, it's also possible to represent `Nu n` just by `n`, so:

```
> let g = LE 3 [4]
> g
 <^3|4>
> showRaw g
 "LE 3 [Nu 4]"
```

2.1.2 Game options manipulation

- `leftOp game` returns a list with the left options of a given game
- `rightOp game` returns a list with the right options of a given game

```
> let g = Op [Nu 1, LE 2 [Nu 3]] [Nu 4]
> g
 <1,<^2|3>|4>
> leftOp g
 [1,<^2|3>]
> rightOp g
 [4]
```

There are two other functions useful to extract a specific game from the original game structure.

- `lop n game` returns the n-th left option
- `rop n game` returns the n-th right option

```
> g
 <<1|1,<1|1>>,<2|3>|<<<^4|4>|<4|^4>>|-5>>
> lop 1 $ g
 <1|1,<1|1>>
> rop 2 . lop 1 $ g
 <1|1>
> rop 1 . lop 2 $ g
 3
```

The `goto` function travels down the game tree following a list of indexes. In this list, a positive number means go to the left options, a negative to the

right options (zero is not allowed). If the index does not exist, an error is produced.

```
> g
 <<^0|<-1|-2>>|<<-1|-2>,<-1|^2>|<-2|^1>>>
> goto [1] g
 <^0|<-1|-2>>   -- same as lop 1 g
> goto [1,-1] g
 <-1|-2>   -- same as rop 1 .  lop 1 $ g
> goto [1,-1,1] g
 -1   -- same as lop 1 .  rop 1 .  lop 1 $ g
```

Functions `addLop` and `addRop` add a new game as the n-th game option:

```
> g
 <<2|2,<2|2>>,<1|1>|<<<^4|4>|<4|^4>>|-5>>
> addRop 1 g $ 7
 <<2|2,<2|2>>,<1|1>|7,<<<^4|4>|<4|^4>>|-5>>
> addLop 2 g 7
 <<2|2,<2|2>>,7,<1|1>|<<<^4|4>|<4|^4>>|-5>>
```

Functions `remLop` and `remRop` delete the n-th game option:

```
> g
 <<2|2,<2|2>>,<1|1>|<<<^4|4>|<4|^4>>|-5>>
> remLop 2 g
 <<2|2,<2|2>>|<<<^4|4>|<4|^4>>|-5>>
> remRop 1 g
 *** Exception: Cannot delete only game option
```

2.1.3 stable

Function `stable` checks if a given game description is stable, ie, $stable(G) \iff ls(G) \leq rs(G)$:

```
> stable $ BE 2 1
 False
> stable $ RE [Op [Nu 4] [Nu (-3)]] (-2)
 True
```

2.1.4 guaranteed

Function `guaranteed` checks if a given game G description is guaranteed, ie $G \in \mathbb{SG}$:

```
> guaranteed $ RE [Op [Nu 4] [Nu (−3)]] (−2)
 False
> guaranteed $ RE [Op [Nu (−4)] [Nu (−3)]] (−2)
 True
```

There are also functions `hot`, `tepid` and `zugzwang` that verify if the respective condition holds for a scoring game.

2.1.5 rank

Function `rank` returns the game's birthday:

```
> rank 412
 0
> rank $ Op [LE 1 [2], 2] [Op [1] [1]]
 2
```

2.1.6 conjugate

The conjugate of a game $G = \{G^{\mathcal{L}}|G^{\mathcal{R}}\}$ is $-G = \{-G^{\mathcal{R}}| - G^{\mathcal{L}}\}$:

```
> let g = RE [Nu 4] (−1)
> conjugate g
 <^1|−4>
> −g
 <^1|−4>   -- same thing
```

2.1.7 stops

- `ls` game left stop of game
- `rs` game right stop of game
- `ls_` game left stop pass-allowed of game
- `rs_` game right stop pass-allowed of game

```
> let g = LE 5 [Op [Op [Nu 10] [Nu 1]] [Nu (−3)]]
> rs g
 1.0
> ls g
 5.0
```

2.1.8 r-protected

- `lrp game` checks if a game is left-r-protected

```
> lrp 2 (Nu 3)
 True
```

2.1.9 relational operators

There are three sets of relational operators.

The first set compares a game to a number:

- `>=.` , means $G \succeq n$
- `<=.` , $G \preceq n$
- `>.` , $G \succ n$
- `<.` , $G \prec n$
- `==.` , $G \succeq n \wedge G \preceq n$
- `/=.` , the negation of `==.`

A game $g \succeq n \iff$ `lrp n g` is true.

```
> Nu 3 >=. 2
 True
> Nu 3 <=. 2
 False
```

The second set compares two games, returning true if the condition is satisfied, or false otherwise (this includes the possibility that the two games are not comparable).

- `>==`
- `<==`
- `===`

```
> Nu 3 >== Nu 2
 True
> Nu 5 >== LE (-4) [Nu 6]
 False   -- only checks if >==, not if <==
```

The third set compares two games, as before, but checks if they are comparable. If they are comparable it returns `Just <result>`, if not it returns `Nothing`.

- >=?
- <=?
- ==?

```
> Nu 3 >=? Nu 2
 Just True
> Nu 5 >=? LE (-4) [Nu 6]
 Nothing
```

Another eg:

```
> let g1 = Op [1] [Op [1][1]]
> let g2 = Op [2] [2]
> g1 >== g2
 False
> g1 >== g2
 False
> g1 >=? g2
 Nothing
> g2 >=? g1
 Nothing
```

At this moment, the functions include comparing games with the same game tree, using the monotone principle and comparing with atomic substitution.

2.1.10 disjunctive sum of games

There are three operators available `#`, `+` and `-`.

Operator `#` performs the disjunctive sum, just like `+`, but without simplifications

```
> let g1 = LE (-3) [Op [2] [1]]
> let g2 = RE [-3] 0
> g1#g2
 <<^-6|<-1|-2>>|<<-1|^2>,<-1|-2>|<-2|^1>>>
> g1+g2
 <<^-6|^-1>|<<-1|^2>|<-2|^1>>>
> g1-g2
 <^-3|<^0|^5>,<<^2|5>|<^1|4>>>
```

2.1.11 invertible

A game G is invertible if $G - G = 0$.

```
> let g = Op [2] [2]
> g-g
 0
> invertible g
 True
```

2.1.12 canonize

When a game is made using the constructors, it might not be in its canonical form.

Function `canonize` performs that operation:

```
> g
 <<2|2,<2|2>>,<1|1>|<<<^4|4>|<4|^4>>|-5>>
> canonize g
 <^2|^4>   -- a quick alternative is to evaluate g+0
```

2.1.13 Conway games

The next functions embed Conway games into scoring games.

SGC stands for Short (ie, finite) Conway Game.

- `scgInt n` or `hat n` transform a integer Conway's game into a scoring game
- `scgDiatic (num,den)` transform a diatic Conway's game into a scoring game
- `zeta num den` order preserving embedding for numbers (integers and diatics) in the format `num/den`
- `scgStar n` star game `*n` into a scoring game
- `up` ie $\{0|*\}$
- `down` ie $\{*|0\}$
- `star` ie $\{0|0\}$ (there's also `star2` and `star3` available)

Notice that a diatic is a fraction like $a/2^b$, egs: 1/4, 5/16, -7/32, 1/1.

```
> scgInt (−5)
 <^0|<^0|<^0|<^0|<^0|0>>>>>
> hat (−5)
 <^0|<^0|<^0|<^0|<^0|0>>>>>
> scgDiatic (2,8)
 <<0|<0|<0|<0|^0>>>>|<<0|<0|<0|^0>>>|<0|<0|^0>>>>
> zeta (−1) 32
 <<<<<<^0|0>|0>|0>|0>|0>   -- scgDiatic (-1,32)
> scgStar 2
 <<0|0>,<<0|0>|<0|0>>|<0|0>,<<0|0>|<0|0>>>
> star2
 <<0|0>,<<0|0>|<0|0>>|<0|0>,<<0|0>|<0|0>>>
> let g = Op [star, hat 1][−5, LE 5 [down]]
> g
 <<<0|0>|<0|0>>,<0|^0>|−5,<^5|<<<0|0>|<0|0>>|0>>>
> canonize g
 <<0|^0>|−5>
```

2.1.14 converting to latex

This command creates a latex description of a game

```
> latex (RE [Op [Nu 4] [Nu (−3)]] (−2))
 <<4|−3>|\emptyset^{−2} >
```

This can be copy-pasted to a latex file and shown as $<< 4| - 3 > |\emptyset^{-2} >$.

2.1.15 testing properties

It is possible to verify if some game properties hold for a large set of random generated games. The program might be able to find a counter-example (which would negate the property). If not, while not proving the property, the user may become more confident and try to demonstrate it mathematically.

There are two testing functions:

- `test` for unary propositions, it receives a proposition (using a lambda-expression) and a number determining the number of required tests
- `test2` works the same way but for binary propositions

Notice that only guaranteed games will be tested.
An eg using `test`:

```
> test (\g -> ls g == ls_ g) 100
 Found a counter example:
 Game: <<-1|^3>,<^-4|^4>|-1,<4|-2>>
> let g = Op [RE [-1] 3, BE (-4) 4][-1, Op [4][-2]]
> ls g
 4
> ls_ g
 -1
```

Let's see an eg where the function does not find a counter-example:

```
> test (\g -> ls g >= ls_ g) 10
 ..........Tests finished!
```

Function `test2` receives a lambda expression with two games:

```
> test2 (\g1 g2 -> g2 <=. ls_ (g1+g1)) 20
 Found a counter example:
 Game 1: -3
 Game 2: -2
```

Another example: say we have the following conjecture (it is a theorem, but let's suppose) saying:

If G_1, G_2 are guaranteed scoring games then,

$$\underline{Ls}(G_1 + G_2) \leq \underline{LS}(G_1) + \underline{Ls}(G_2)$$

we can try to find a counter-example like this:

```
> let prop g1 g2 = ls_ (g1+g2) <= ls_ g1 + ls_ g2
> test2 prop 35
 ....................................Tests finished!
```

It's possible that the generator produces large games, which might take a while to sum or canonize. On those cases, the user might interrupt the interpreter using the `CTRL+C` key combination.

As said, `test` and `test2` generate guaranteed and canonized games.

Functions `testn` and `test2n` work the same way but include a random number, which is useful to test propositions using a number.

```
> prop g n = let h=LE n [Nu$n+1] in ls_(g+h)==ls_ g+n
> testn prop 30
 ..............................Tests finished!
```

It's possible to return one guaranteed and canonized random game:

```
> getCanonize
 <<4|−3>|<^−5|−2>,<^−4|^3>>
> getCanonize
 <4|^6>
> liftM showRaw getCanonize
 "RE [Nu 4] 8"
> liftM showRaw getCanonize
 "Op [Nu 0,BE (−9) 7] [BE (−1) 5]"
```

Or even to return a list of them:

```
> getCanonizeList 4
 [5,<^−10|^−9>,<1|−3>,0]   -- list of four games
> liftM (map showRaw) $ getCanonizeList 4
 ["BE 4 10","Nu 3","Nu 0","Nu 8"]
```

2.1.16 converting strings into games

To translate a raw description string into a game:

```
> let description = "BE (−3) 7"
> description
 "BE (−3) 7"
> (read description)::Game
 <^−3|^7>
> let gs = ["BE (−3) 7","Nu (−6)","LE 5 [Nu 8]"]
> (map read gs)::[Game]
 [<^−3|^7>,−6,<^5|8>]
```

There is no functionality to translate the standard output string into the corresponding game.

2.2 Position Module

Module `Position` is an abstraction for game rules, like Dots'n'Boxes or Diskonnect.

It defines a new type called `Position` with a certain number of operations that each game must implement.

The operations are:

- `points` what is the current scoring
- `boards` what is the current board
- `moves` what are the legal next moves?

- `toText` converts position into a textual description
- `fromData` given a number and a textual description, creates a position

In this context, a Position value is just the current board position and the current scoring.

To define a new game, the user must implement the previous five functions. However, only function `moves` has some complexity. The others are quite straightforward. To make a new game, please check the games already implemented, copy its code and adapt them to your purposes.

To access the module's functions, please import the module (it will also import the Scoring and Position modules). To check what are the available functions, type `commands_pos`.

These functions only work given concrete game rules. The next egs use Kobber so the user must import module `Kobber` which implements Kobber gamerules.

2.2.1 preparing a file with a game position

A practical way to input game positions is to write it on a text file (place it in the same folder as the modules) and import it directly.

Say we have written file `"kobber1.txt"` like this:

```
1
lr.
rl.
l.r
```

The first line is a number with the current score.

The next lines are used to write the board. The representation might vary depending on the specific characteristics of the game.

Please use char `l` to represent left pieces, and `r` to represent right pieces. The dot is used to represent an empty board cell. Char `'x'` represents a wall.

2.2.2 evaluating a position in a file

To read a file with a game position use function `evalG` where `G` is the specific game. In our case, since we are using Kobber, the function is named `evalKobber`.

```
> evalKobber "kobber1.txt"
 -- Read: "1\nlr.\nrl.\nl.r"
 -- Board
 1
 lr.
 rl.
 l.r
 -- Position Value:
 <<1|<<0|0,<-1|-2>>|^0>,<1|<2|1>>>, ...
```

This presents the game value but it does not get assigned to a variable.

To do that use `fromRaw` and `toGame` with the description that function eval produces in its first line:

```
> let pos = fromRaw "1\nlr.\nrl.\nl.r"::Kobber
> pos
 P pts = 1.0, board = ["lr.","rl.","l.r"]
> let g = toGame pos
> g
 <<1|<<0|0,<-1|-2>>|^0>,<1|<2|1>>>, ...
```

2.2.3 presenting a position

Function `present` shows a position in a textual format:

```
> let pos = fromRaw "1\nlr.\nrl.\nl.r"::Kobber
> present pos
 lr.
 rl.
 l.r
 1 points
```

Function `presents` does the same but for a list of positions:

```
> let pos1 = fromData 0 ["lr","rl",".."]::Kobber
> let pos2 = fromData 2 ["rr",".."]::Kobber
> presents $ [pos1,pos2]
 lr
 rl
 ..
 0 points
 rr
 ..
 2 points
```

2.2.4 getting the next possible positions

Function `moves` returns the next moves given a position and a player.

```
> let pos = fromData 0 ["lr","rl",".."]::Kobber
> presents $ moves pos Left
 .r
 ll
 ..
 0 points
 .l
 rl
 ..
 0 points
 .r
 .l
 l.
 1 points
 ll
 r.
 ..
 0 points
 lr
 l.
 ..
 0 points
```

2.3 Games

2.3.1 Kobber

Module `Kobber` implements the game Kobber.

Rules:

Pieces move orthogonally. A piece can capture by replacement an adjacent enemy stone (earning a point) or capture by jumping over (not earning a point).

2.3.2 Dots'n'Boxes

Module `Dots` implements the game Dots'n'Boxes.

Rules:

Starting with an empty grid of dots, players take turns, adding a single horizontal or vertical line between two unjoined adjacent dots. A player who

completes the fourth side of a 1 by 1 box earns one point and takes another turn.

```
> evalDots "dots1.txt"
 -- Read: "1\nx..xxx\nxx.xxxx\nxxx.xx"
 -- Board
 1
 x . . x x x
x x . x x x x
 x x x . x x
 -- Position Value:
 <<4|0>,<4,<4|0>|0,<4|0>>|<2|-2>,<2,<2|-2>| ...
> let pos = fromRaw "1\nx..xxx\nxx.xxxx\nxxx.xx"::Dots
> present pos
+-+ + +-+-+-+
|#|   | |#|#|
+-+-+-+ +-+-+
 1 points
```

2.3.3 Diskonnect

Module `Diskonnect` implements the game Dots'n'Boxes.

Rules:

Each move must capture stones.

Stones are captured by jumping orthogonally over enemy stones, checkers-like.

Captures can be multiple, but cannot change the initial direction.

When a player cannot move, she takes the final penalty equal to the number of her own dead stones (ie, a dead stone is a stone that can still be captured)

```
> evalDiskonnect "diskonnect.txt"
 -- Read: "0\n...\n..r\nlr.\n"
 -- Board
 0
 ...
 ..r
 lr.
 -- Position Value:
 <<2|^2>|^2>
```

3 Annex: How to start

Goto `https://www.haskell.org/platform/`, download and install *The Haskell Platform* for your Operating System.

After the installation, enter the GHC interpreter. On Windows choose WinGHCi.

Then select the Load option, goto the folder with the current modules and position files, and select the adequate module. If you choose a game module, modules `Scoring` and `Position` will also be imported.

It's also possible to create a standalone executable program to perform a sequence of fixed steps. Check file `run_disk.hs` as an eg of how to do it.