Haskell Rapport

Simon Holm
DM580: Funktionel programmering
Teacher: Casper Bach

Syddansk Universitet

Contents

1. Introduction

This report serves as a brief documentation of the implementation of a domain-specific language for a Snake-like game, as described in the project description [1].

This project first and foremost focuses on implementing a few simple yet critical functions of the domain-specific language (DSL), including the `combine` function and the `runProg` interpreter for the `SProg` datatype. The report also explains the `SProg` datatype constructors and how they use proper Haskell syntax to implement their intended behavior in the interpreter.

Finally, the Snake game has been designed to meet all requirements and implemented using the DSL syntax, with controls, movement, and collision handling explained in detail. The game has been tested throughout development, which is also documented.

Lastly, possible areas for improvement are discussed along with the the use of generative AI throughout the project

2. Implementation of combine and runProg

2.1. combine

The `combine` function in `app/Scratchy/Syntax.hs` is used to compose two `SProg` programmes. This means that running ``m1 `combine` m2`` will have the same effect as running `m1` first, then `m2` immediately after. Since the given code deal with functions by constructing them in `Syntax.hs` and later defining their bahavior in a interpreter, a data constructor `Seq` has been added to the `SProg` datatype.

The `combine` function is just an alias for the `Seq` constructor, making it easier and clearer to compose programs in sequence.

```
-- Sequential composition
| Seq (SProg ()) (SProg a)

-- more code

combine :: SProg () -> SProg () -> SProg ()
combine = Seq
```

2.2. runProg

The function `runProg` is the interpreter for the `SProg` datatype, which represents a sequence of actions or events in a game or application. It updates the world state by executing some actions step by step. The `runProg` function is then using the recursive helper function `go`, which pattern matches on each constructor of `SProg` and applies the corresponding effect to the world.

The `go` function handles the actions from the `Syntax.hs` file. The intended behavior of each constructor is documented as comments in the `Syntax.hs` file directly underneath each constructor. This design allows the interpreter to process complex game logic by chaining and combining simple actions, while always returning a new, updated world state.

Although `runProg` might look pretty complicated, it rougly follows some pretty consistent patterns across all cases. By explaining some representative examples and their syntax, understanding how the interpreter uses the given syntax (from the code given in then assignment [1]) and handles different types of actions.

2.2.1. rest and next

The interpreter mainy uses two patterns to continue execution after an action: `rest` and `cont`.

rest is used for actions that perform an action on the world and then continues with the remaining program (`runProg`). It represents the “rest of the program”. For example, in event handlers:

```
go (OnTargetReached ptr handler rest) world =
      go rest world { trHdlrs = (ptr, handler) : trHdlrs world }
```

Note that, if gamelogic wants `rest` to do nothingit can pass `pure()` as `rest`.

cont is a continuation function [2] (Section 17.4: Adding a continuation) that takes a function from `SProg` to `SProg` as an additional argument. For example, in `InspectCell`:

```
go (InspectCell cell cont) world =
      let result = ... -- compute HasBarrier, HasSprite, or IsFree
      in go (cont result) world
```

Here, `cont` is called with the inspection result to obtain the next program. This allows the game code to decide what to do based on the computed value. This pattern appears throughout the DSL in actions like `GetTarget`, `NewSprite`, and `InspectCell`, where computed values (target cells, sprite indices, or inspection results) must be passed to continuation code.

2.2.2. OnBarrierHit

```
go (OnBarrierHit ptr handler rest) world =
      go rest world { bhHdlrs = (ptr, handler) : bhHdlrs world }
```

This case registers a barrier hit handler for a specific sprite. The pattern match extracts the sprite `ptr`, the handlerfunction `handler`, and the continuation program (`rest`).

The implementation uses record update syntax [3] (Section 3.15.3: Updates Using Field Labels) to create a new `World` with an updated `bhHdlrs` field. The expression `(ptr, handler) : bhHdlrs world` uses the cons operator (`:`) to prepend the new handler tuple to the front of the existing handler list. This tuple pairs the sprite pointer with its corresponding handler function.

This creates a new updated world state without modifying the original state.

2.2.3. InspectCell

The `InspectCell` case demonstrates the use of the `let` expressions [3] (Section 3.12: Let Expressions), which are used throughout the interpreter. A `let` expression allows defining local variables that are only visible within its scope (the part after `in`).

The syntax is: “`let bindings in expression`”, where the bindings define variables that can be used in the expression.

```
go (InspectCell cell next) world =
      let hasSprite = spriteExistsAt cell (sprites world) /= Nothing
          hasBarrier = not (inBounds cell)
          result
            | hasBarrier = HasBarrier
            | hasSprite  = HasSprite
            | otherwise  = IsFree
      in go (next result) world
```

In this case, the `let` block defines three local bindings:

- `hasSprite`: Checks if a sprite exists at the cell using the `spriteExistsAt` helper function
- `hasBarrier`: Checks if the cell is in the barriers list using the `elem` operator
- `result`: Uses guards (`|`) to determine the cell state based on the previous checks

The guards still evaluate in order: if the cell is `inBound`, return `HasBarrier`, if it has a sprite, return `HasSprite`, otherwise return `IsFree`. It then passes the result with `cont`.

3. Implementation of my Snakey game

3.1. How to run

All of the snakey-game's game logic is located in the `MyGame.hs` file. To run the game, do the following (i do it in vscode):

1. Open the terminal
2. Navigate to the folder `/scratchy/`
3. Type in the terminal
   - `cabal run`

3.2. The snakes creation and structure

```
    NewSprite (10,15) (Color green (circleSolid (cellSize * 0.6))) (\headG ->
    NewSprite (9,15)  (Color green (rectangleSolid cellSize cellSize)) (\tailG1 ->
    NewSprite (8,15)  (Color green (rectangleSolid cellSize cellSize)) (\tailG2 ->
    NewSprite (7,15)  (Color green (rectangleSolid cellSize cellSize)) (\tailG3 ->
    NewSprite (6,15)  (Color green (rectangleSolid cellSize cellSize)) (\tailG4 ->
    NewSprite (5,15)  (Color green (rectangleSolid cellSize cellSize)) (\tailG5 ->
    NewSprite (4,15)  (Color green (rectangleSolid cellSize cellSize)) (\tailG6 ->
    NewSprite (3,15)  (Color green (rectangleSolid cellSize cellSize)) (\tailG7 ->

    -- Blue snake (head at (20,15), tail extending left to (14,15))
    NewSprite (20,15) (Color blue (circleSolid (cellSize * 0.6))) (\headB ->
    NewSprite (21,15) (Color blue (rectangleSolid cellSize cellSize)) (\tailB1 ->
    NewSprite (22,15) (Color blue (rectangleSolid cellSize cellSize)) (\tailB2 ->
    NewSprite (23,15) (Color blue (rectangleSolid cellSize cellSize)) (\tailB3 ->
    NewSprite (24,15) (Color blue (rectangleSolid cellSize cellSize)) (\tailB4 ->
    NewSprite (25,15) (Color blue (rectangleSolid cellSize cellSize)) (\tailB5 ->
    NewSprite (26,15) (Color blue (rectangleSolid cellSize cellSize)) (\tailB6 ->
    NewSprite (26,15) (Color blue (rectangleSolid cellSize cellSize)) (\tailB7 ->

    -- controls
    ))))))))))))))
```

The code block above creates two snakes, each with 1 head and 7 links (tails)[4] using nested continuation functions. The constructor `NewSprite` takes a cell `Cell`, a picture `Picture`, and a continuation function that receives the newly created sprite's pointer `SpritePtr`. Each `NewSprite` call follows this pattern:

```
NewSprite position picture (\spritePointer -> nextSnakeSegment)
```

The lambda function `\spritePointer ->` is a continuation that receives the sprite's index and defines what happens next. The nesting serves a crucial purpose: it keeps all sprite pointers in scope so they can all be used at the deepest level when setting up the game controls. When the interpreter processes these nested calls, it will then assigns sequential indices: `headG = 0`, `tailG1 = 1`, through `tailB6 = 15`. The nested structure is purely for keeping all 16 sprite pointers accessible when calling for controls:

```
    -- Controls - these will set up the head movement and tail following
  player1SnakeControls headG [tailG1, tailG2, tailG3, tailG4, tailG5, tailG6, tailG7]
     `combine`
    player2SnakeControls headB [tailB1, tailB2, tailB3, tailB4, tailB5, tailB6,
  tailB7]))))))))))))))))
```

3.3. Controls

The game uses `combine` to register controls for both players, ensuring responsive input handling. Player 1 uses WASD keys (W for up, A for left, S for down, D for right), while Player 2 uses IJKL keys (I for up, J for left, K for down, L for right).

Each control is registered using the `OnKeyEvent` constructor, which takes a key, an action to perform, and the rest of the program to continue with:

```
OnKeyEvent key (moveInDirection head tails dir) (nextOnKeyEvent)
```

This pattern ensures that when we detect one key (e.g., 'w' for up), we still continue checking for the remaining keys. After key detection, the function calls `moveInDirection` with the appropriate head sprite, tail sprites list, and direction according to the pressed key.

3.4. Snake movement

The snake's movement is controlled by `moveInDirection`. It takes the snake head, the tail segments, and a direction. The function combines setting up a tail chain with the `OnTargetReached` handler, which then calls `moveInDirection` for the head once again in order to keep the snake moving continuously.

This is how the movement works:
1. Set up the tail chain so each segment follows the one in front of it.
2. When the head reaches its target cell, inspect the next cell in the movement direction.
3. Based on the inspection result (barrier, sprite, or free), either block the movement or allow it and continue moving.

The basic structure of `moveInDirection` is:

```
moveInDirection :: SpritePtr -> [SpritePtr] -> Dir -> SProg ()
moveInDirection headPos tails dir =
  case tails of
    [] -> Pure ()
    (firstTail:_) ->
      setupTailChain tails `combine`
      OnTargetReached headPos (\curPos ->
        -- Inspect next cell and handle collision
        (SetTarget firstTail curPos (moveInDirection headPos tails dir))
      ) (Pure ())
```

The `setupTailChain` function ensures each tail segment follows the segment in front of it. The `OnTargetReached` handler checks whether the head can move forward, and if so, updates both the head and the first tail segment to ensure that the first tail follows the head. Both of these functions are explained further in the following sections.

3.4.1. Tail movement

Since `OnTargetReached` already ensures that the first tail segment follows the head, we now need to ensure that all the other tail segments follow the one in front of them. Since all tail segments are stored in a list, we can recursively set up this following behavior for each segment.

The tail movement is controlled by the recursive function `setupTailChain`, shown below:

```haskell
setupTailChain :: [SpritePtr] -> SProg ()
setupTailChain [] = Pure ()
setupTailChain [_] = Pure ()
setupTailChain (leader:follower:rest) =
  followTail leader follower `combine` setupTailChain (follower:rest)
```

This function takes the list of tail segments and does the following:

1. If there are **0 tails** (`[]`), do nothing (`Pure ()`).
2. If there is **only 1 tail** (`[_]`), do nothing—there's no follower to set up.
3. If there are **2 or more tails**, split the list into `(leader:follower:rest)`.
4. Make the `follower` follow the `leader` using `followTail`.
5. Recursively process the rest of the tail chain by calling `setupTailChain (follower:rest)`, where the current `follower` becomes the new `leader` for the next segment.

The helper function `followTail` registers an event handler using `OnTargetReached`. When the `leader` reaches its target cell, the handler sets that cell as the new target for the `follower`:

```haskell
followTail :: SpritePtr -> SpritePtr -> SProg ()
followTail leader follower =
  OnTargetReached leader (\leaderPos ->
    SetTarget follower leaderPos (Pure ())
  ) (Pure ())
```

This ensures that each tail segment moves to the cell that the segment in front of it just left, creating a smooth following chain.

3.4.2. Collision handling

There are several ways to handle collision detection in games like Snake. One common approach is to check for collisions after movement (i.e., detect if you have hit something once you move and go back). Another approach is to check the surroundings before moving, ensuring the path is clear before allowing the move.

From my experience programming a Pacman-like game last semester, handling collisions with boundaries (such as walls) is more simple and more robust when you check for a clear path before movement. This preemptive check prevents invalid moves and makes the game logic easier to manage.

This approach integrates naturally with the DSL's `InspectCell` action. In the `moveInDirection` function, collision detection happens before movement:

```
OnTargetReached head (\curPos ->
  let nextPos = nextCell dir curPos in -- finding the adjacent cell (the next one)
  InspectCell nextPos (\cellResult ->
    case cellResult of
      HasBarrier -> SetBackgroundColor red (Pure ()) -- collision with barrier
      HasSprite  -> SetBackgroundColor black (Pure ()) -- collision with another sprite
      IsFree     -> SetBackgroundColor white -- just keep swimming.. or moving :)
        (SetTarget head nextPos
          (SetTarget firstTail curPos (moveInDirection head tails dir))))
) (Pure ())
```

When the snake head reaches its target, the next cell in the movement direction is inspected before moving. If the cell contains a barrier or another sprite, the movement is prevented and visual feedback is provided by the background ganging color (red for barriers, black for sprite collisions). Only when the cell is free does the actual movement occur via `SetTarget` and the background stays/becomes white.

3.5. Testing

As for testing while implementing the snake game, I mostly used a Haskell printing function (such as `trace`) to output values and debug information to the terminal. This helped me understand what was happening during game execution, especially in the early stages of development. By printing out e.g positions of sprites, and results of key functions, I could more easily identify and fix logic errors as they came up.

4. Areas for Improvement

This section discusses selected areas for improvement that would be interesting to explore given more time and a deeper understanding of Haskell:

1. The snakes could be implemented as a custom data structure instead of just lists. This would probably make the code easier to maintain and extend in the future.
2. Better gameplay features, such as preventing snakes from moving after death, detecting self-collision, and eliminating color flickering caused by background color updates when one snake is dead and the other moves.

5. Use of Generative AI

While generative AI (specifically Claude Sonnet 4.5) has not been used for code autocompletion or generation, it has been used for identifying issues such as syntax errors as well as interpreting compiler error messages. This assistance has been very valuable in areas where I do not yet have a deep enough understanding of Haskell and/or GHCi to confidently know why some issues occur.

Bibliography

[1] S. Casper Bach, “Haskell Project Assignment Description.” 2025.

[2] G. Hutton, *Programming in Haskell*, 2nd ed. Cambridge, UK: Cambridge University Press, 2016.

[3] S. Marlow, Ed., “Haskell 2010 Language Report.” 2010. [Online]. Available: https://www.haskell.org/definition/haskell2010.pdf

[4] S. Casper Bach, “Haskell Project Assignment Description pt.2.” 2025.