Leveraging Familiar Game Concepts to Create a Game for Teaching Functional Programming

Study Walkthrough

Jesse Linossier
Faculty of Information Technology
Monash University
Clayton, Australia
jlin0076@student.monash.edu
ORCID:0000-0001-6782-7019

Tim Dwyer
Faculty of Information Technology
Monash University
Clayton, Australia
tim.dwyer@monash.edu

Shuai Fu
Faculty of Information Technology
Monash University
Clayton, Australia
shuai.fu@monash.edu

1 PURPOSE OF THIS DOCUMENT

This document serves as a complete level-by-level, phase-by-phase walkthrough of the study and the most noteworthy reactions from participants. Originally planned to be part of the main thesis as the Results section, it has instead been moved into this separate appendix document as while this document is important for repeatability, proof of study, etc. It is a much too verbose and lengthy way of presenting results.

2 STUDY WALKTHROUGH

Figure 1: The Level Selection Menu for switching between levels

The Study consisted of 6 levels which 6 participants worked through in order while the researcher took notes on both on what they were doing in the game as well as relevant things the participants voiced aloud. All data about these participants has been anonymized and as such shall be referred to by the pseudonyms Participant Alice, Participant Bob, Participant Charlie, Participant Dave, Participant Erin, and Participant Frank.

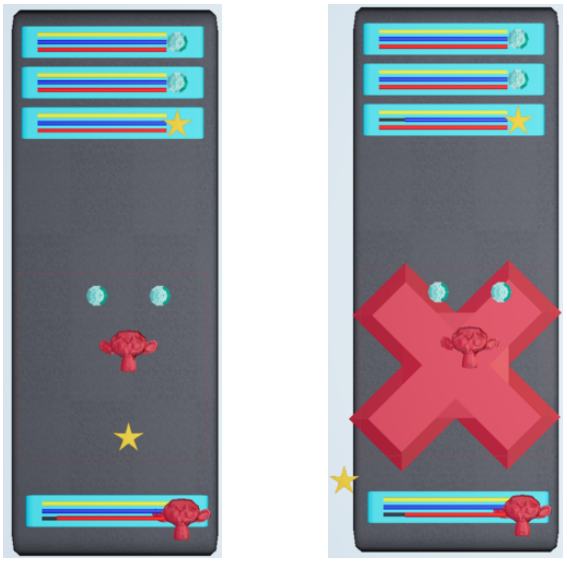
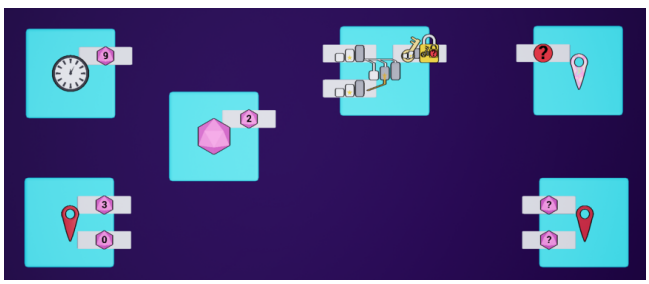
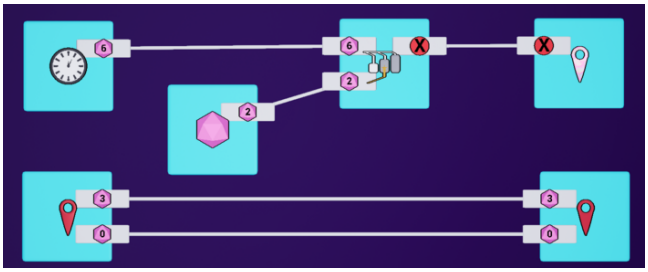


Figure 2: Level 1 - Board

This is the first level that participants saw, a solution was first demonstrated to the participants before they were given control to ensure that all participants were aware of how the game controlled and what was expected of them. The goal of this level is to simply have the Thief (star icon) avoid the periodic attacks from the boss, by using a function to jump off the board when the attack is about to happen. Every tick spent off the board cost 10% of the Thief's mana, so permanently staying off the board is not a solution.



(a) Start of Phase



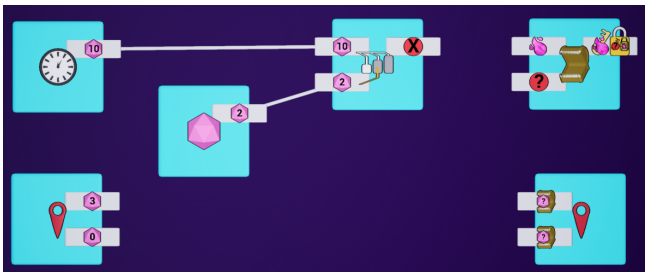
(b) Solution to Phase

Figure 3: Level 1 Phase 1

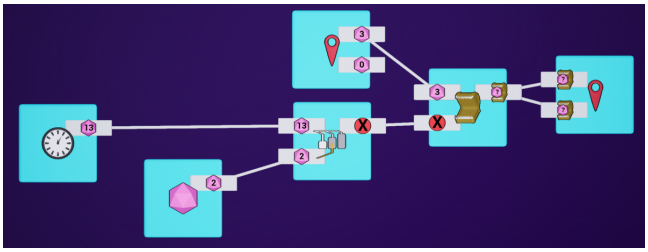
Level-1-1's solution is equivalent to the haskell code:

```haskell
type UntilAttack = Int
type Position = (Int, Int)
type Dodge = Bool
gameTick :: UntilAttack -> Position -> (Dodge, Position)
gameTick t (px, py) = (dodge, position) where
  dodge = t < 2
  position = (px, py)
```

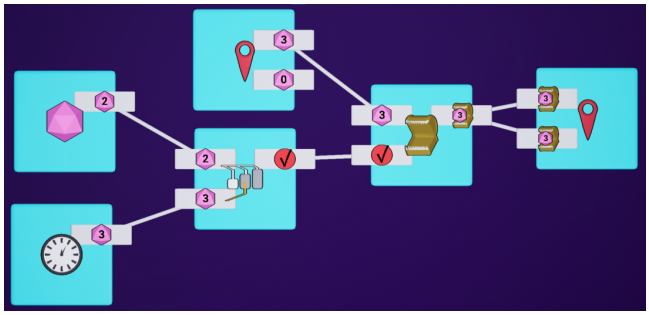
Level-1-1 can simply be solved by connecting up the function blocks as their placement would suggest. Charlie, Dave and Erin all walked through the level function by function and made guesses at what each block was doing and how it contributed to the level's solution, however Charlie, as the only one of the three inexperienced in functional programming, had trouble reasoning about the level due to thinking in the imperative programming style of "If player needs to dodge is true, then dodge".



(a) Start of Phase



(b) Almost a Solution



(c) Solution to Phase

Figure 4: Level 1 Phase 2

Level-1-2's solution is equivalent to the haskell code:

```haskell
type DodgePosition = (Maybe Int, Maybe Int)
gameTick :: UntilAttack -> Position -> DodgePosition
gameTick t (px, _) = (mPos, mPos) where
  mPos = if' (t < 2) (Just px) Nothing
```

Level-1-2 introduced the Maybe Dataclass which is represented using the icon of a 'Mimic', it also replaced the skill to dodge with turning the position outputs into Maybe Int values. All participants with functional programming experience were able to identify the Mimic as a Maybe from a combination of its iconography and its values, but rather interestingly Charlie who didn't know what a Maybe was able to describe what the Mimic was doing and how it was working to replace the dodge skill.

A small trick in this phase was that the inputs to the Less Than block had to be switched around, only Frank made this switch before looking at the board. The rest of the participants as Dave put "[went] through the control flow [to] see what is happening" rather than copying the demonstrated solution.

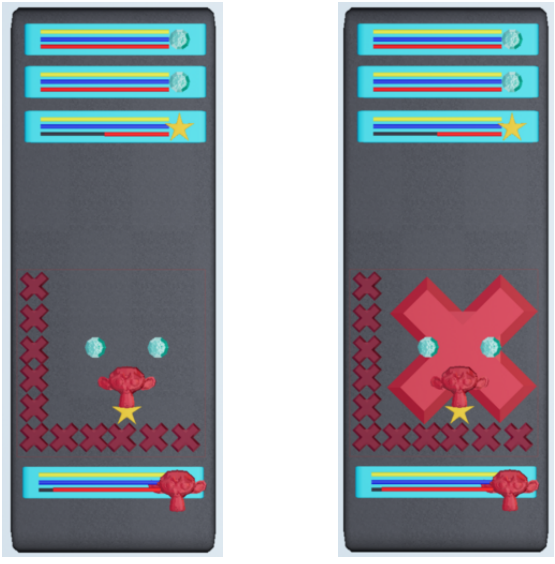
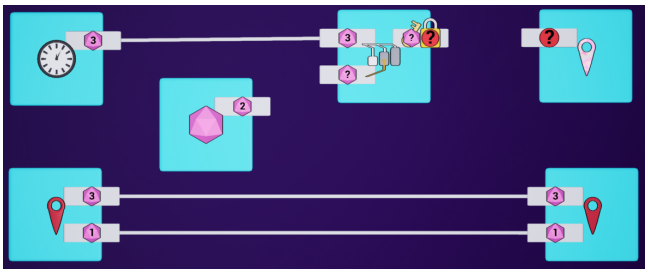
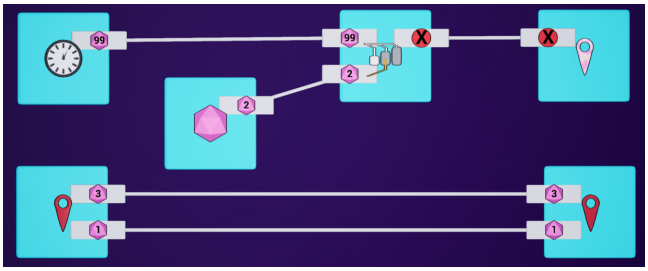


Figure 5: Level 2 - Board

This level has the same premise as Level-1, with the addition that the player will be hurt and killed if they ever set their position to $x = 0$ or $y = 0$.



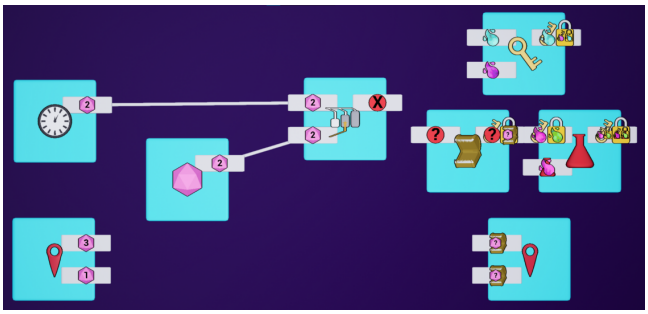
(a) Start of Phase



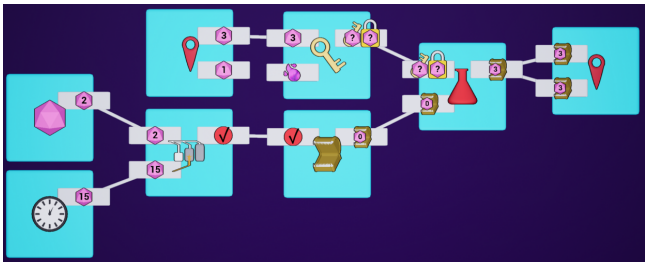
(b) Solution to Phase

Figure 6: Level 2 Phase 1

Level-2-1 is identical to Level-1-1 and all participants were able to solve the phase effortlessly.



(a) Start of Phase



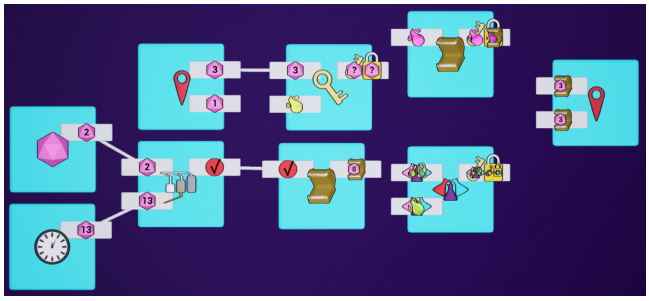
(b) Solution to Phase

Figure 7: Level 2 Phase 2

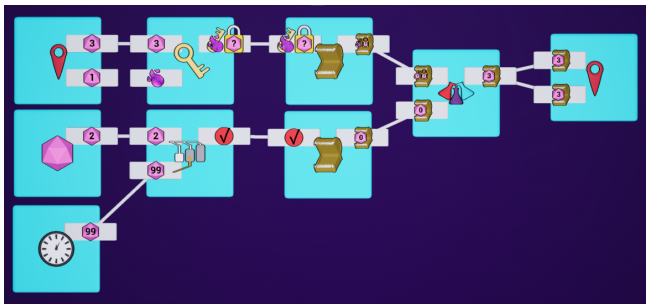
Level-2-2's solution is equivalent to the haskell code:

```haskell
gameTick :: UntilAttack -> Position -> DodgePosition
gameTick t (px, _) = (mPos, mPos) where
  mPos = const px <$> if' (2 < t) (Just 0) Nothing
```

Level-2-2 introduces the const and fmap function blocks, as well as the first required use of partial application, representing the first major hurdle to the participants without functional programming experience, Bob and Charlie. No participant was able to recognise the fmap or const functions from iconography, and partial application took some time for participants to be able to understand. Surprisingly, Charlie was the only participant to realize that the new blocks could be used to replace the value inside the Mimic before connecting any blocks. All participants except Dave arrived at a solution through trial and error rather than logically, however once seeing the updating values in the fmap function, all participants were able to logically reason about why and how the solution works.



(a) Start of Phase



(b) Solution to Phase

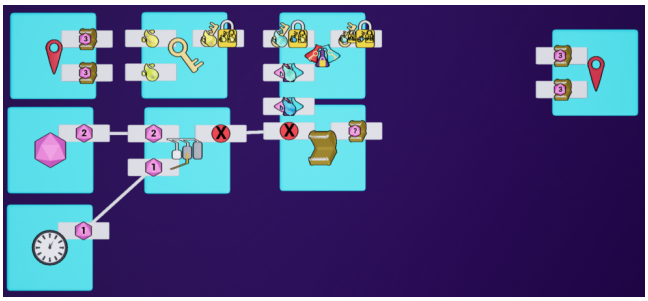
Figure 8: Level 2 Phase 3

Level-2-3's solution is equivalent to the haskell code:

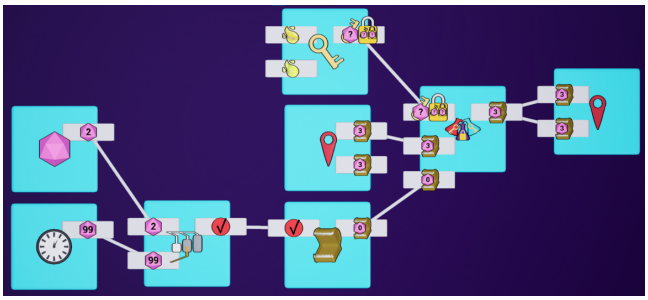
```haskell
gameTick :: UntilAttack -> Position -> DodgePosition
gameTick t (px, _) = (mPos, mPos) where
  mPos = Just (const px) <*> if' (2 < t) (Just 0) Nothing
```

Level-2-3 replaces fmap (<$>) with apply (<*>) and is the only instance of putting a function inside a Maybe. Alice was the only participant to do this unintentionally before assuming it was a mistake and undoing it, all other participants upon noticing the

difference in the type signature between fmap and apply were able to do this intentionally as a means of creating an acceptable input for apply despite boxing a function being something that is almost never done in imperative programming. Also of note is that all participants solved this level logically as compared to the trial and error solutions of Level-2-2.



(a) Start of Phase



(b) Solution to Phase

Figure 9: Level 2 Phase 4

Level-2-4's solution is equivalent to the haskell code:

```haskell
gameTick :: UntilAttack -> DodgePosition -> DodgePosition
gameTick t (mPx, _) = (mPos, mPos) where
  mPos = liftA2 const mPx (if' (2 < t) (Just 0) Nothing)
```

Level-2-4 replaces apply with liftA2, which means that const has to have its partial application removed. Various different solution paths arose to solve this problem; Erin used the iconography of liftA2 needing two keys to solve the phase. Alice, Dave, and Frank noticed that a two input function was needed as input to liftA2 from type signature. Boband Charlie phrased the problem in terms of Level-2-3's solution to find differences. All participants were able to explain the differences between fmap, apply, and liftA2 in terms of use at the end of the level, but no participants were able to identify the const function despite being able to use it well enough to solve the level.

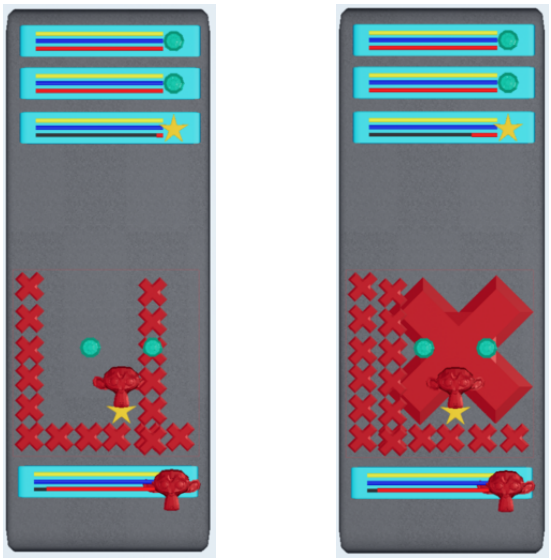
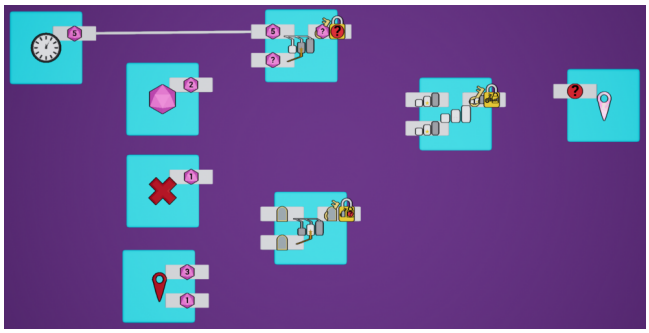
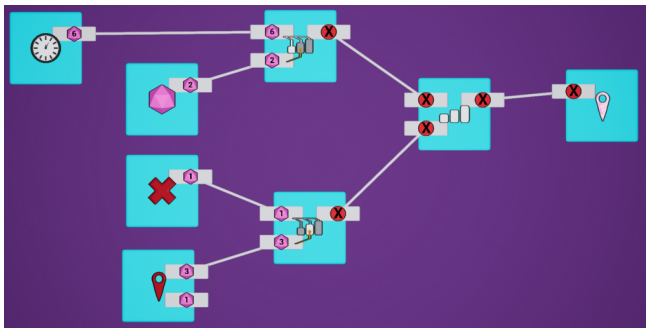


Figure 10: Level 3 - Board

This level adds a horizontally-scrolling wave attack that the player must dodge in addition to all the existing attacks from Level-2.



(a) Start of Phase



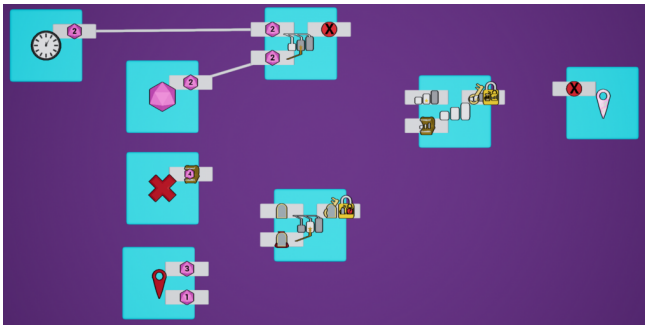
(b) Solution to Phase

Figure 11: Level 3 Phase 1

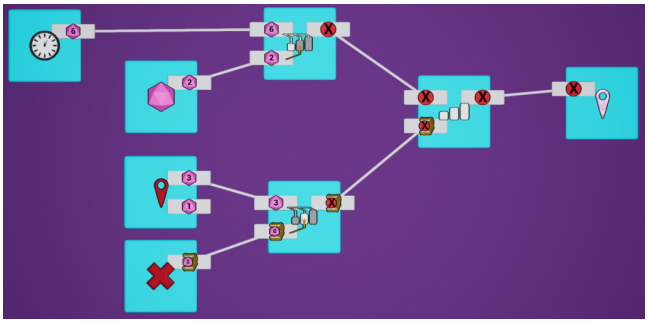
Level-3-1's solution is equivalent to the haskell code:

```haskell
type AttackX = Int
gameTick :: UntilAttack -> AttackX -> Position -> Dodge
gameTick t x (px, _) = dodge where
  dodge = (t < 2) `max` (x == px)
```

Level-3-1 re-uses the Less Than block while also introducing the Equals and Maximum blocks which all use the same iconography. Alice, Charlie, Dave, and Frank were able to make strong guesses about what each function was purely from iconography, and all participants were able to confidently identify all comparison blocks after some minor value testing. From this, all participants logically constructed a solution to the level based on the comparison functions. Also of note, Dave was the only participant who showed clear interest in the new type icon on the Equals inputs.



(a) Start of Phase



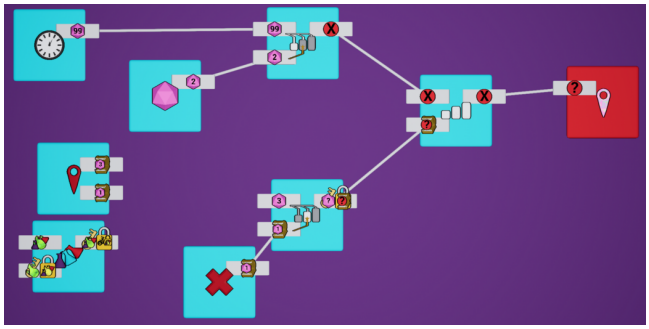
(b) Solution to Phase

Figure 12: Level 3 Phase 2

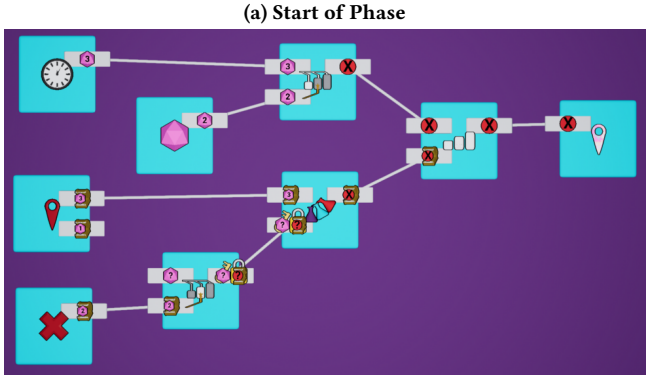
Level-3-2's solution is equivalent to the haskell code:

```haskell
gameTick :: UntilAttack -> Maybe AttackX -> Position -> Dodge
gameTick t mX (px, _) = dodge where
  dodge = (t < 2) `mMax` (px `mEq` mX)
```

Level-3-2 is a small preparatory stage that all participants solved quickly. Alice, Dave, and Frank were able to pick up on this preparatory nature, and all commented on how "We are running the comparison as a function this time" as compared to a simple operator.



(a) Start of Phase



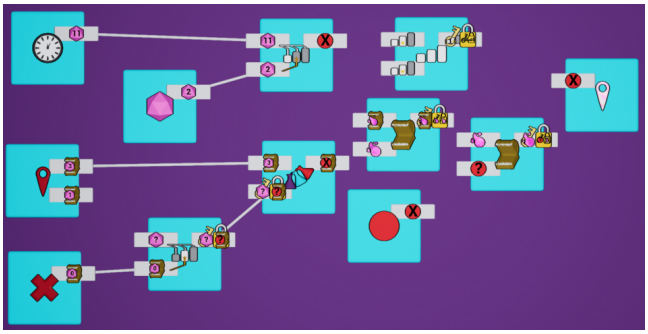
(b) Solution to Phase

Figure 13: Level 3 Phase 3

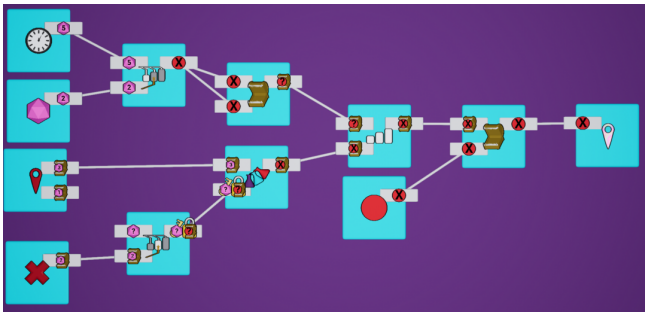
Level-3-3's solution is equivalent to the haskell code:

```haskell
gameTick :: UntilAttack -> Maybe AttackX -> DodgePosition -> Dodge
gameTick t mX (mPx, _) = dodge where
  dodge = (t < 2) `mMax` (mPx >>= (`mEq` mX))
```

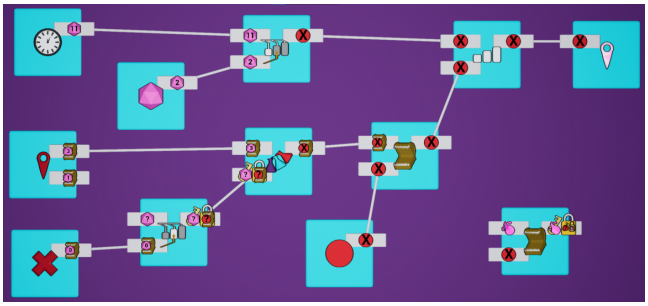
Level-3-3 introduces the bind (>>=) function block, and as previously noticed by some participants requires using the Equals block as a function, specifically by partially applying the Equals block to satisfy bind's lower input. Participants had extremely varied reactions to this phase, with Alice and Dave solving the phase due to recognising bind, Frank and Erin solving the phase through type signature matching, Charlie came to an epiphany about functional programming's differences to imperative programming in that "Rather than manipulating things with variables, you pass them around to functions, and chain functions until you eventually get what you want to get" and solving the phase through that method. Finally, Bob was unable to solve the phase within a time they deemed acceptable and gave up on the rest of this level.



(a) Start of Phase



(b) The Canonical Solution to Phase



(c) The Common Solution to Phase

Figure 14: Level 3 Phase 4

Level-3-4's canonical solution is equivalent to the haskell code:

```haskell
gameTick :: UntilAttack -> Maybe AttackX -> DodgePosition -> Dodge
gameTick t mX (mPx, _) = dodge where
  tDodge = (\x -> if' x Just x Nothing) (t < 2)
  mDodge = tDodge `max` (mPx >>= (`mEq` mX))
  dodge = fromMaybe mDodge False
```

While Level-3-4's common solution, the solution all participants found, is equivalent to the haskell code:

```haskell
gameTick :: UntilAttack -> Maybe AttackX -> DodgePosition -> Dodge
gameTick t mX (mPx, _) = dodge where
  dodge = (t < 2) `max` fromMaybe False (mPx >>= (`mEq` mX))
```

Level-3-4's canonical solution introduces bare comparison functions on Mimics, and extracting values out of Mimics. The participants who solved this level, Dave, Erin, and Frank, all found Level-3-4's common solution which only introduces extracting values out of Mimics. Ironically this common solution arises because the Maximum function works on any Ordinal Dataclass instead of only Mimics, which is the reverse of what the phase was trying to teach. Dave, and Frank found this solution without much time or effort, but Erin took the time to work backwards from the Maximum function to solve the level logically. In doing so, Erin belatedly recognised the bind function from looking at its icon, and from there was able to make connections to the flask icons used for the fmap and apply functions. This finally resulted in Erin giving a perfect explanation on how each flask visual metaphor was used to represent the Functor, Applicative, and Monad Typeclasses.

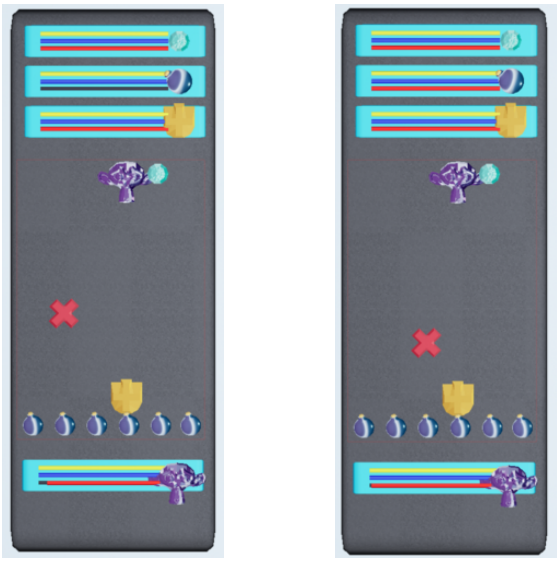
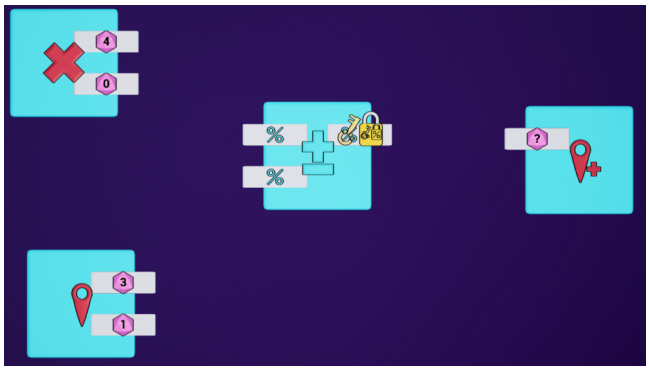
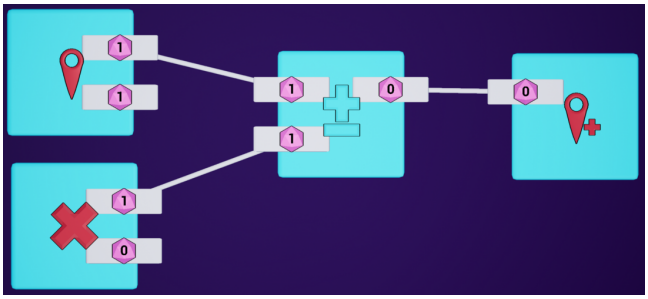


Figure 15: Level 4 - Board

This level introduces a new class, the Shielder. Represented by a shield icon, the Shielder has to move and block the projectile attacks sent by the boss from killing the mages in the back line by taking the hits for them. The Shielder is only able to move left or right one space at a time.



(a) Start of Phase



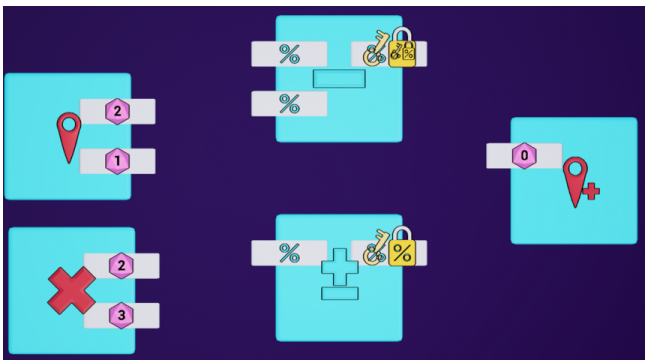
(b) Solution to Phase

Figure 16: Level 4 Phase 1

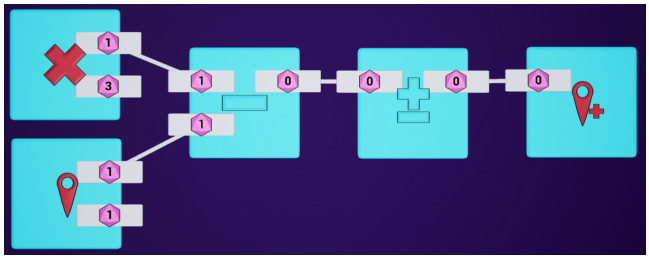
Level-4-1's solution is equivalent to the haskell code:

```haskell
type AttackPos = (Int, Int)
type Movement = Int
-- Functionality of output movement block
move :: Int -> Movement
move -1 = -1
move  1 =  1
move  _ =  0
-- Main Game Tick
gameTick :: Position -> AttackPos -> Movement
gameTick (px, _) (ax, _) = move mv where
  mv = signDiff px ax
```

Level-4-1 was solved by all participants logically except Charlie who used trial and error. As the study was not targeting specific iconography choices, and the symbology was somewhat incorrect, all participants were offered to be told what the symbol on the middle block was being used as (sign function), even if it wasn't necessarily what the function block was. Despite this, all participants took Dave's stance of "I'm gonna check what this does with my own eyes", refusing the explanation except as confirmation.



(a) Start of Phase



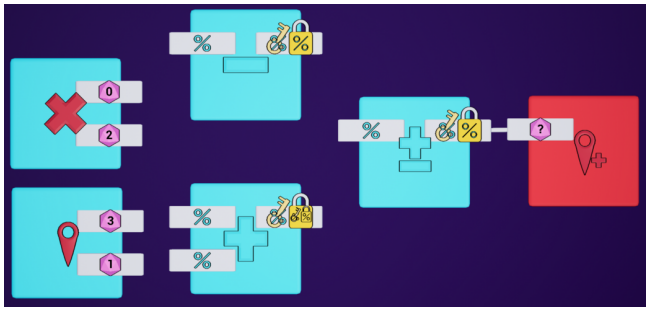
(b) Solution to Phase

Figure 17: Level 4 Phase 2

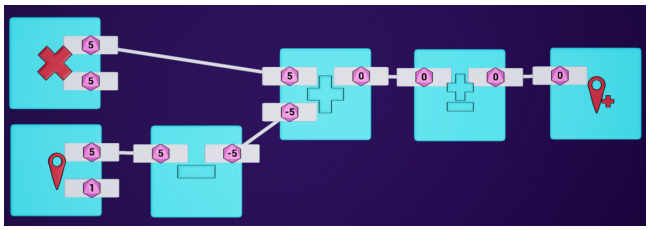
Level-4-2's solution is equivalent to the haskell code:

```haskell
gameTick :: Position -> AttackPos -> Movement
gameTick (px, _) (ax, _) = move mv where
  mv = sign $ ax - px
```

Level-4-2 was by far the quickest solved phase in the entire game by all participants, who all could "see what to do"(Frank) as soon as they saw the two new blocks.



(a) Start of Phase



(b) Solution to Phase

Figure 18: Level 4 Phase 3

Level-4-3's solution is equivalent to the haskell code:

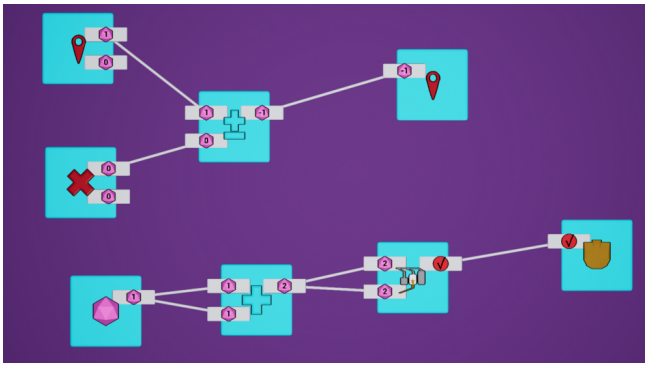
```haskell
gameTick :: Position -> AttackPos -> Movement
gameTick (px, _) (ax, _) = move mv where
  mv = sign $ ax + (-px)
```

Level-4-3 did not fall far behind Level-4-2 with participants similarly solving it quickly thanks to the level "splitting the maths operations into ways to construct it from smaller bits"(Bob) and

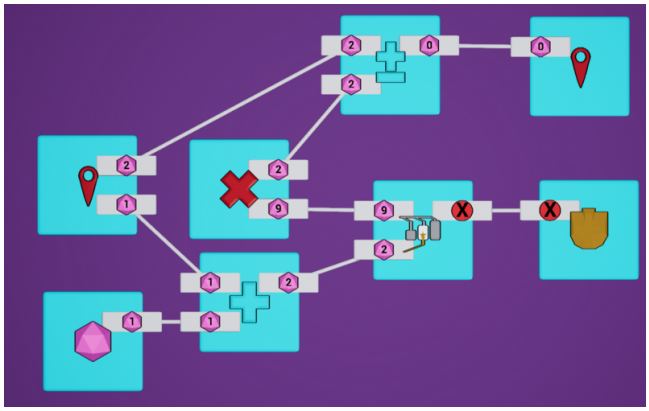
not “[feeling] functional programming inspired”(Erin). Participants were “invested in the level” and found it “fun because [they] knew how to solve it” and “completing levels ... accomplishment is fun” as was so eloquently stated by “the maths ones are so easy, I love maths” Alice. Or as Charlie summarized: “making sense is what makes it fun”.

Figure 19: Level 5 - Board

This level builds upon Level-4’s challenges by introducing a shield that the Shielder must use to block attacks rather than simply using their body as the attacks are now powerful enough to hurt the Shielder. Holding the shield out both costs 10% of their stamina, as well as stops any attempted movement, as such this level can only be solved by using the shield exactly when needed.



(a) Start of Phase



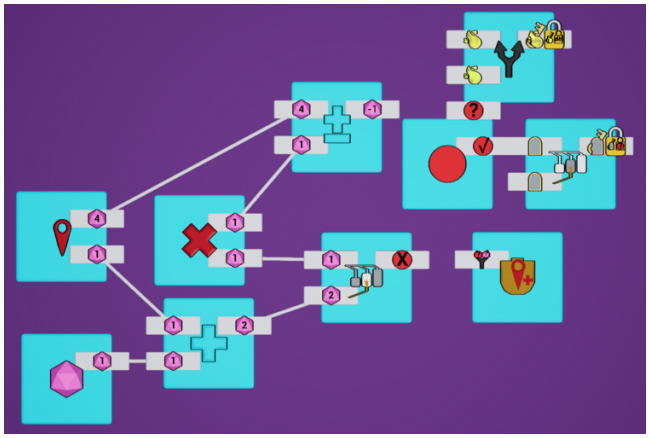
(b) Solution to Phase

Figure 20: Level 5 Phase 1

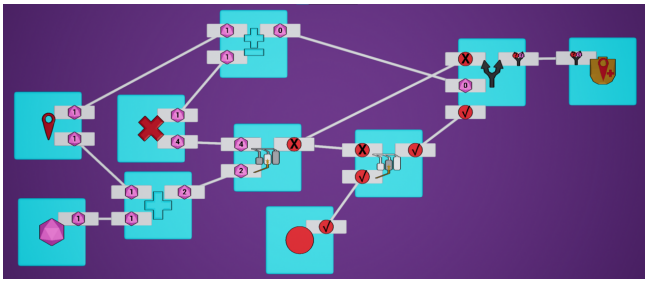
Level-5-1’s solution is equivalent to the haskell code:

```haskell
type Block = Bool
-- Updated Functionality of output movement block
move :: Block -> Movement -> (Block, Movement)
move False -1 = (b, -1)
move False  1 = (b,  1)
move b      _ = (b,  0)
-- Main Game Tick
gameTick :: Position -> AttackPos -> (Movement, Block)
gameTick (px, py) (ax, ay) = move block mv where
  mv = signDiff px ax
  block = ay == (py + 1)
```

Level-5-1 has the movement part already solved, while the shielding part can be solved through either the absolute solution Alice and Charlie found of comparing the attacks Y position against 1 + 1 or the relative solution the other participants found of comparing it against the players Y position +1. If the players Y position was 2 or greater the absolute solution would not be possible, but as it is not this phase is able to suggest which participants are still using value-based solutions rather than logic-based solutions.



(a) Start of Phase



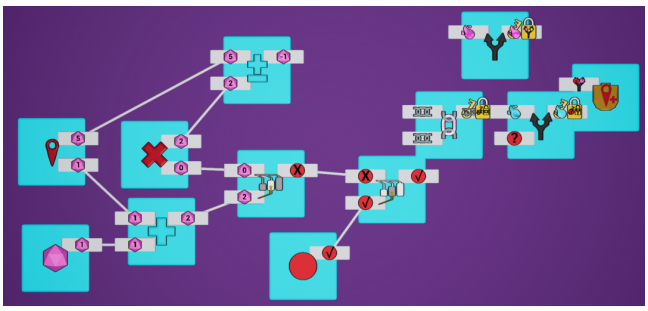
(b) Solution to Phase

Figure 21: Level 5 Phase 2

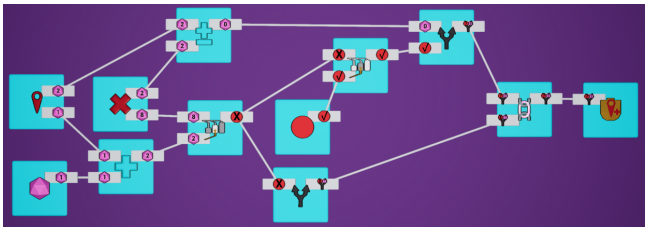
Level-5-2's solution is equivalent to the haskell code:

```haskell
-- Either Constructor
eitherSide :: a -> b -> Bool -> Either a b
eitherSide a _ False = a
eitherSide _ b True  = b
-- Main Game Tick
gameTick :: Position -> AttackPos -> Either Block Movement
gameTick (px, py) (ax, ay) = either block mv notBlock where
  mv = signDiff px ax
  block = ay == (py + 1)
  notBlock = block /= True
```

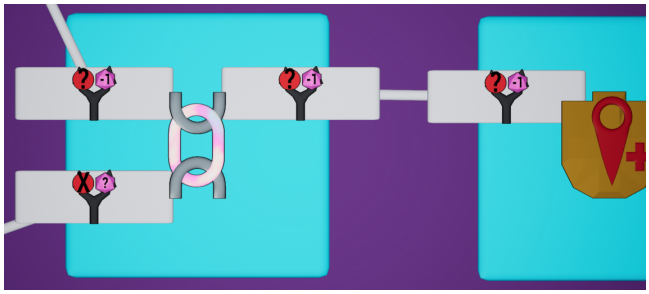
Level-5-2 introduces the Either type with the split path icon. This iconography was not helpful to any participants, and better iconography such as a skill check or saving throw was suggested which has a clear success and failure implication as Either is commonly used for in Haskell, instead of its purely definitional left and right are equally valid interpretation. Despite this by matching the type icons, all participants were able to solve the phase after some hinting that the Not Equals block that they had recognized from iconography could be used to make a boolean Not operator.



(a) Start of Phase



(b) Solution to Phase



(c) A Close-Up of the 'mappend' function

Figure 22: Level 5 Phase 3

Level-5-3's solution is equivalent to the haskell code:

```haskell
gameTick :: Position -> AttackPos -> Either Block Movement
gameTick (px, py) (ax, ay) = action where
  mv = signDiff px ax
  block = ay == (py + 1)
  notBlock = block /= True
  action = either False mv notBlock <> Left block
```

Level-5-3 introduces mappend (<>) which all participants understood from iconography; "when taking two values, [the iconography of] chaining them together gave [them] a really good idea of what it was doing"(Charlie). However "chaining together two Eithers is hard to understand"(Frank) and therefore participants struggled with this phase. This is where the iconography of a saving throw would be clearer, as you only do more saving throws when the last one failed. Despite this, all participants were able to reason the phase was about replacing the functionality of the three-input Either constructor with combining two less powerful Either constructors, which led to all participants bar Bob being able to solve the level with minor help.

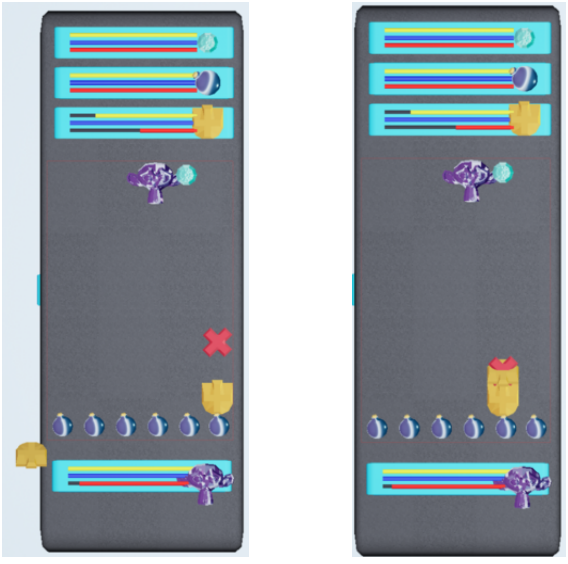
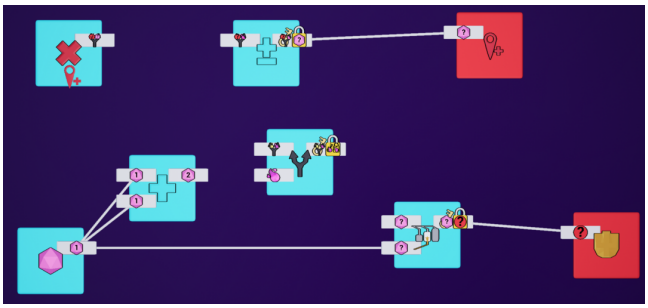
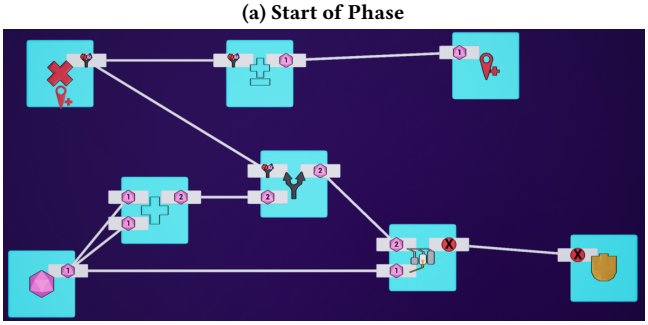


Figure 23: Level 6 - Board

This level does not add anything to Level-5's challenge, and instead changes the functions through the phases in a different way.



(a) Start of Phase



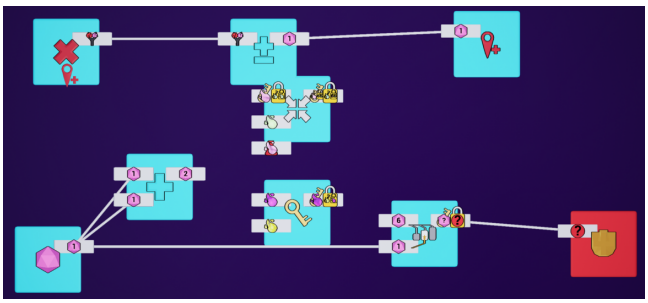
(b) Solution to Phase

Figure 24: Level 6 Phase 1

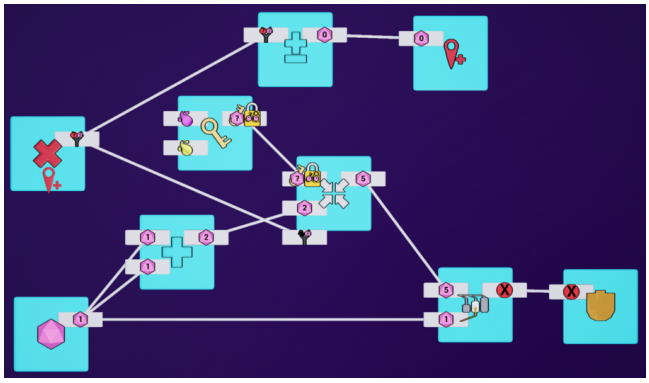
Level-6-1's solution is equivalent to the haskell code:

```haskell
type PositionDiff = Either Bool Int
gameTick :: PositionDiff -> (Movement, Block)
gameTick dPos = move block mv where
  mv = signBoolLeft dPos
  block = fromRight dPos (1+1) == 1
```

Level-6-1 rather than giving the attack and Shielder positions separately, gives the vertical difference when aligned, and the sign of horizontal difference when not. All participants were able to adapt to this change easily and solve the phase without issue, potentially due to the small problem space of the phase. All participants were able to reason about how fromRight was being used to get the Right value out of the Either, using a fallback when given a Left value.



(a) Start of Phase



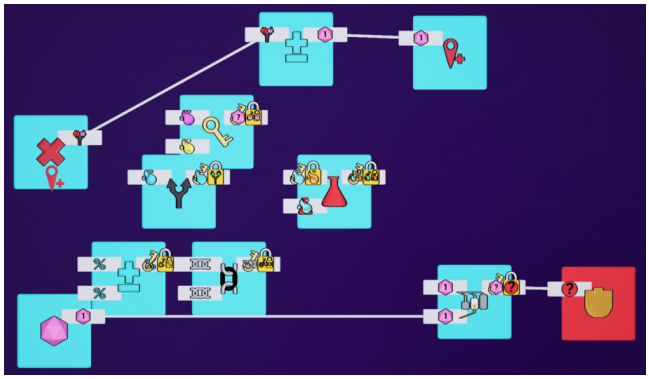
(b) Solution to Phase

Figure 25: Level 6 Phase 2

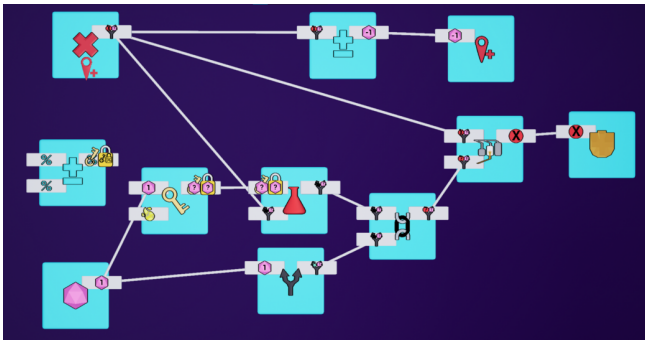
Level-6-2's solution is equivalent to the haskell code:

```haskell
gameTick :: PositionDiff -> (Movement, Block)
gameTick dPos = move block mv where
  mv = signBoolLeft dPos
  block = foldr const (1+1) dPos == 1
```

Level-6-2 attempts to teach participants that fromRight can be replaced with foldr const, however Alice and Bob were unable to solve this level, and only Frank was able to solve the phase without hints. Frank was also the only participant to reason that since so little had changed, the two new blocks must replace the function of the previous one, whereas all other participants had to be given this hint, in addition to being reminded what const does.



(a) Start of Phase



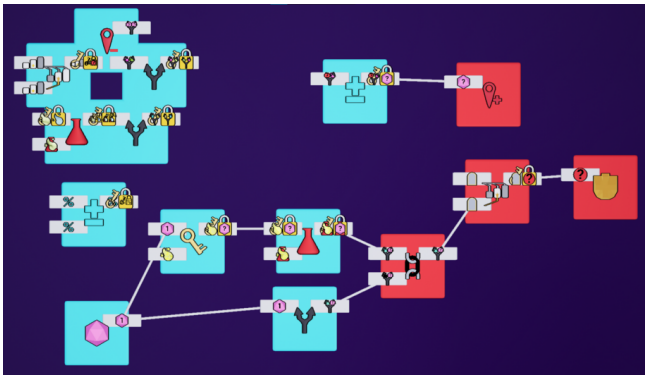
(b) Solution to Phase

Figure 26: Level 6 Phase 3

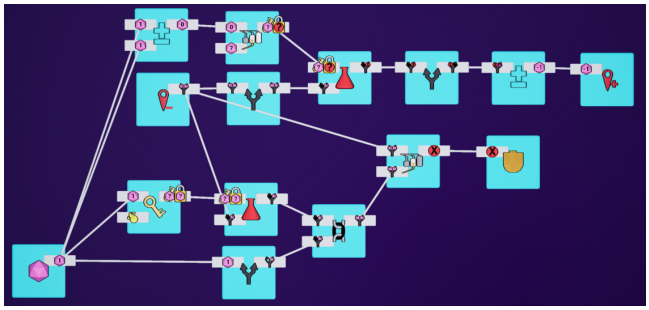
Level-6-3's solution is equivalent to the haskell code:

```haskell
gameTick :: PositionDiff -> (Movement, Block)
gameTick dPos = move block mv where
  mv = signBoolLeft dPos
  block = dPos == (fmap (const 1) dPos <> Right 1)
```

Level-6-3 reintroduces mappend to join two Either values, but this time the remaining participants were able to reason about how to use it correctly due to learning what it does in Level-5-3. In addition, they were also able to realise that the new blocks could be used to exactly replace the lost blocks function this time. As Dave explains, "The flask is used to get a 1 or question mark" inside an Either's Right, and then "The comparison is used to get a bool out of the Either based on when [the vertical position difference is] 1".



(a) Start of Phase



(b) Solution to Phase

Figure 27: Level 6 Phase 4

Level-6-4's solution is equivalent to the haskell code:

```haskell
type PositionDelta = Either Int Int
gameTick :: PositionDelta -> (Movement, Block)
gameTick dPos = move block mv where
  mv = signBoolLeft $ swap $ (signDiff 1 1 >=) <$> swap dPos
  block = dPos == (fmap (const 1) dPos <> Right 1)
```

The sight of Level-6-4 caused Charlie and Dave to instantly give up on the level, leaving only Erin and Frank as the participants who were able to solve the level. In theory this phase is simple as all it requires is creating the Left boolean in the Either from the raw Left Int that is now given, however the phase is made much more complex as the Either needs to be flipped around to operate on the Left value while it is on the Right, and the flipped back afterwards. Erin struggled with this concept significantly, as they could not logically reason about why "flipping around [the Either] to operate on the right [value]" was necessary as Frank was able to. Erin suggested this may be due to having Int values on both sides of the Either instead of two distinct types.

Following the levels, all participants filled out a survey adapted from the NASA-TLX while having guided and unguided conversations about the game. The Survey is also available as an appendix.