Haskell SpriteKit

Transforming an Imperative Object-oriented API into a Purely Functional One

Manuel M T Chakravarty
UNSW Sydney
Australia
chak@cse.unsw.edu.au

Gabriele Keller
UNSW Sydney
Australia
gabriele.keller@unsw.edu.au

Abstract

In Haskell and many other functional languages, graphics libraries, animation frameworks, game engines, and the like usually have to choose between providing either state-of-the-art functionality or a purely functional API. In this paper, we will show that we can layer a purely functional interface on top of an object-oriented, imperative one in an efficient manner. We do so by computing the difference between the input and output values of purely functional transformation functions, and then, applying that difference to a mutable object graph. We will make good use of Haskell's by-default lazy evaluation in realising this scheme.

To demonstrate the feasibility of this approach, we implemented a Haskell binding to the SpriteKit animation system and game engine. We describe its interface, how to use it, and the methods underlying its implementation.

ACM Reference format:
Manuel M T Chakravarty and Gabriele Keller. DRAFT. Haskell SpriteKit. In *Proceedings of DRAFT, DRAFT, DRAFT,* 13 pages.
DOI: DRAFT

1 Introduction

Graphics libraries, animation frameworks, game engines, and the like face a dilemma in Haskell and many other functional languages. They can either build on an existing state-of-the-art framework (typically implemented in C++) and expose its imperative, object-oriented API by way of the language's foreign function interface [8]. Or, alternatively, they can be implemented from scratch with an elegant purely functional interface [13, 19], but generally rather limited functionality and visual appeal. That is not because we wouldn't be able to implement a fully-fledged framework in Haskell, but because doing so requires considerable effort, which we usually cannot expend, and specialised expertise, which is rare. This naturally raises the question of whether we can find a compromise. Can we build on an existing framework with an object-oriented, imperative API and endow it with a purely functional interface?

In this paper, we will answer this question in the affirmative. We will provide a constructive proof of our assertion by describing a purely functional interface for the Objective-C API of Apple's SpriteKit animation system, physics engine, and game engine [2]. More importantly, we will show how to efficiently translate between the imperative object-oriented and the functional API.

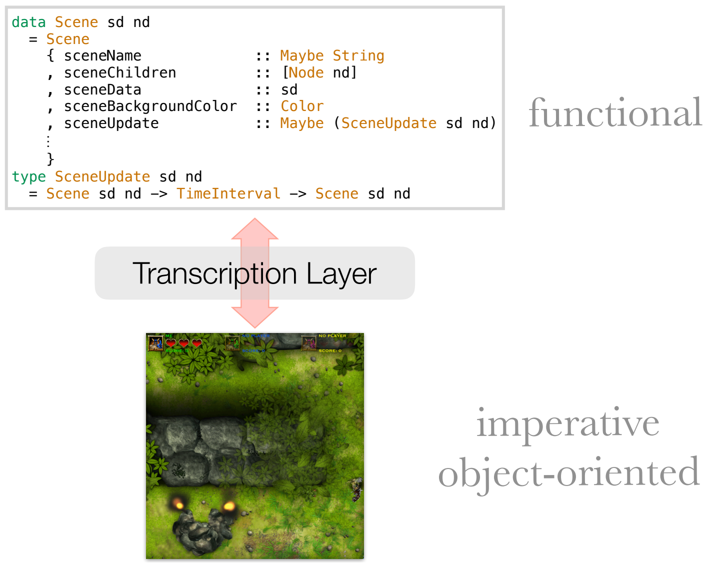
The core idea —illustrated in Figure 1— is the following: we replace mutating methods used to update the underlying object graph of the object-oriented framework with pure Haskell transformation functions that receive the original graph (or a portion thereof) as an argument and return a new, derived graph (or a portion thereof) as a result. Our *transcription layer* compares these two versions of the object graph and computes their difference. Then, it mutates the underlying object graph of the object-oriented framework according to the set of differences derived from the pure variant. A major challenge is to preserve hidden state in mutable objects and to achieve the computation of the difference in an efficient manner, especially when the transformation function only touches a small part of the object graph. We use Haskell's lazy evaluation semantics in combination with some low-level runtime system functionality to achieve this efficiency goal. More precisely, our approach ensures that the number of graph nodes touched by the transcription layer is asymptotically linear in the number of nodes touched by the user-provided pure transformation function.

In summary, this paper makes the following contributions:

- We provide a purely functional alternative to an object-oriented, imperative game engine API (Section 3).
- We demonstrate the utility of lazy evaluation for *lazy marshalling* of object graphs (Section 4).
- We show that lazy marshalling together with some low-level runtime functionality and object caching enables the asymptotically efficient calculation of the difference between a lazily marshalled version and its transformed variant (Section 4).

The Haskell SpriteKit source code is available as open source software from https://github.com/mchakravarty/HaskellSpriteKit.

DRAFT, DRAFT
DRAFT. DRAFT...$15.00
DOI: DRAFT



```
data Scene sd nd
  = Scene
    { sceneName             :: Maybe String
    , sceneChildren         :: [Node nd]
    , sceneData             :: sd
    , sceneBackgroundColor  :: Color
    , sceneUpdate           :: Maybe (SceneUpdate sd nd)
    ⋮
    }
type SceneUpdate sd nd
  = Scene sd nd -> TimeInterval -> Scene sd nd
```

Figure 1. Architecture

1.1 Related work

Our approach to layering a functional on top of an imperative object-oriented interface is related to treatment of the browser DOM by the Javascript framework React [11]. In particular, React also favours the use of pure transformation functions and uses *diffing* to compute a change set that gets applied to the rendered object tree (the browser DOM). However, in React we have a less clear separation between the imperative and functional world and the lack of statically enforced purity requires more discipline on the side of the application developer.

We are not aware of any other work that uses a similar approach in a Haskell or any other typed functional language. However, there exists a range of work on graphics and games programming. The most important work that aims to provide a functional interface is summarised in the following.

Gloss [5] provides a functional interface to a fragment of the 2D functionality of OpenGL. It was originally aimed at animations, but can also be used for simple games. It has a purely functional interface that is suitable for learners. However, its functionality is limited to the basics. All animations need to be hand-coded and there is no physics engine.

There exists a broad spectrum of work on the use of *functional reactive programming (FRP)* for animations and games in Haskell, from Elliott's seminal work [10] to the FRP libraries Yampa [9] and Reactive Banana [12] used in conjunction with several different graphics libraries including OpenGL and SDL. FRP provides a functional way to specify animations and reactive behaviour. However, much of the functionality that has been realised with FRP in previous work is already included ready-made in SpriteKit's animation system and physics engine (e.g., animate movement over time or a path, physical behaviour, collision detection). Hence, the same game functionality is significantly less code with SpriteKit and doesn't require an understanding of FRP.

Haskell graphics library wrappers, such wXHaskell [14] and Gtk2Hs [18], support basic game development, but require the use of similar programming idioms as the imperative object-oriented languages that the original libraries are based on.

2 The State of the Art

Animations and video games are often organised in *scenes*, comprising a collection of objects that represent both the visual as well as non-visual aspects of a portion of the animation or game. In the object-oriented, imperative languages commonly used to realise animation and games frameworks, a scene's objects are organised as a mutable object graph. Visual and non-visual properties as well as the graph structure itself change in-place as the game or animation progresses. Providing a functional representation of a scene graph and its change over time is the main challenge addressed in this paper.

2.1 SpriteKit

To keep the discussion concrete, we will focus on one particular animation framework and game engine as well as on a simple sample game for the majority of this paper. Specifically, we discuss Apple's SpriteKit framework [2], a popular, fully featured, easy-to-set-up 2D game engine including a state-of-the-art animation subsystem and physics engine. Nevertheless, the concepts explained in this paper transcend this specific technology and are generally applicable.

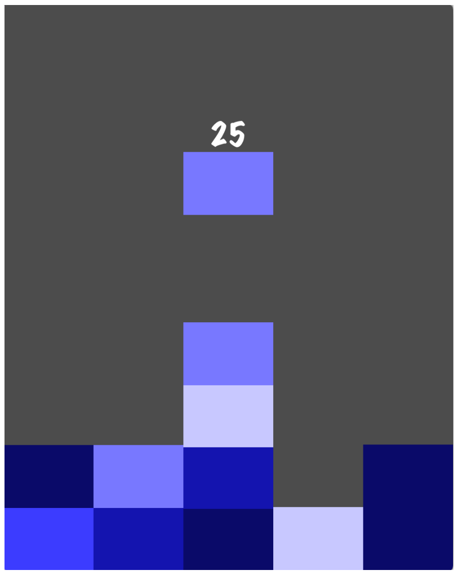


Figure 2. Shades — our running example

2.2 Shades

As a running example, we will use a Haskell clone of the mobile game *Shades* (see Figure 2). It is sufficiently simple to explain in a paper, while covering all the mechanisms of SpriteKit that are crucial to the methods explained here.

In Shades, the player moves a falling block, coloured in one of five random shades of a given colour, to the left or right. Blocks of different colours stack up on top of each other as they reach the bottom of the play area. If a block comes to rest on a block of the same shade, it merges with that blocks and turns into the next darker shade, possibly triggering a chain reaction with the next block below it. If the colour of the two blocks is already the darkest shade, they do not merge and instead stack on top of each other. Whenever the player manages to fill a row with blocks of the same shade, all blocks in that row disappear. Once all blocks come to rest, a new block to be placed by the player is spawned at the top of the play area. The game is over once a block lands in the top row of the playing field.

In the reminder of this section, we shall discuss the three main problems that we need to overcome to provide a purely functional interface to SpriteKit and similar frameworks. They are: (1) subclassing; (2) mutable node properties in the scene graph; and (3) in-place mutation of the structure of the scene graph.

The source code for Shades is available at https://github.com/gckeller/shades

2.3 Problem #1: subclassing

Figure 3 contains the *scene graph* corresponding to the scene rendered in the screenshot of Figure 2. The scene graph is always rooted in a *scene object*, with *node objects* representing the various scene elements, as children. These nodes have varying visual representations, depending on their type, and can have further nodes as children.

Moreover, nodes can optionally have *actions* attached to them, which can change a node's properties and behaviours over time and are often used to implement animations. Nodes can also have

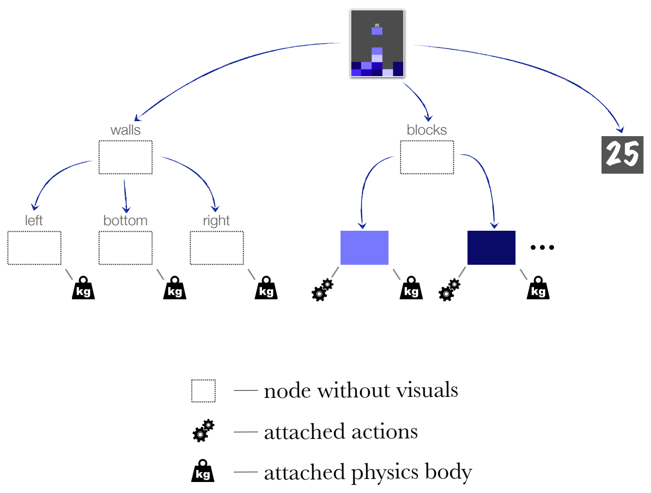
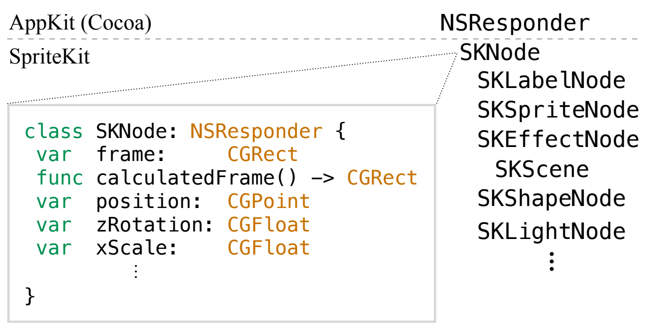


Figure 3. Scene graph of Shades



```
class SKNode: NSResponder {
 var  frame:     CGRect
 func calculatedFrame() -> CGRect
 var  position:  CGPoint
 var  zRotation: CGFloat
 var  xScale:    CGFloat
          ⋮
}
```

Figure 4. SpriteKit class hierarchy

a *physics body* attached, in which case they partake in the physics simulation implemented by the physics engine — that is, they may be affected by gravity, collisions, force fields, and so forth.

In our case, the scene has three children (from left to right): (1) a node grouping the three walls (which are represented by edge-based physics bodies to keep the blocks inside the scene); (2) a node grouping the blocks; and (3) a label node for the current score. There are a variable number of blocks in the scene, indicated by the three dots next to the rightmost block in the scene graph.

Although, we have drawn the scene graph as a tree (which is how a SpriteKit API user usually thinks about it), the underlying Objective-C heap structure is actually a graph as all nodes contain back edges to reach their parent node and the scene node at the root.

2.3.1 The class hierarchy

The SpriteKit class hierarchy is displayed in Figure 4. All objects occurring in a SpriteKit scene are of type `SKNode` or one of its subclasses, where node objects of type `SKNode` itself contain no visuals, but are useful to group other nodes — in Figure 3, these are the white boxes with a dotted outline. The root in Figure 3 (and any SpriteKit scene) is of type `SKScene`, the blocks in the middle (blue in colour version) are of type `SKSpriteNode`, and the score text is of type `SKLabelNode`.

All nodes inherit common properties (such as `position`) and methods (such as `calculatedFrame()`) from `SKNode` — the excerpt of its class definition, given in Figure 4, is rendered in Swift [4].[1]

There is one oddity in the SpriteKit class hierarchy. `SKScene` is used rather differently from all other types of nodes and arguably ought to have been separate.

2.3.2 Event handling

Figure 4 also includes `SKNode`'s superclass, `NSResponder`. This class is not part of SpriteKit, but of *AppKit*, which is the application and GUI toolkit of Apple's collection of application frameworks for macOS, called *Cocoa*.[2] The purpose of `NSResponder` is to enable subclasses to intercept input events, such as mouse clicks, keyboard key presses, and so forth. Hence, Objective-C (and Swift) programmers typically subclass `SKScene` and other `SKNode` subclasses to receive input events.

Haskell does not directly support class-based inheritance in a manner that aligns with Objective-C. Hence, we will need to find alternative means to provide that functionality in the Haskell SpriteKit interface.

2.4 Problem #2: mutable scene graph properties

During most of the gameplay of the sample game Shades, only one block is active — that is, the one falling in from the top. The player can move the active block left and right with key presses; in other words, in response to those key presses, the active block needs to change its position abruptly. As indicated in Figure 3, blocks in Shades are sprites represented by objects of type `SKSpriteNode`. As a subclass of `SKNode`, they inherit the `position` property displayed in the class declaration excerpt in Figure 4. Put differently, whenever the user presses a movement key while a block is falling, the event handler mutates the `position` property of the `SKSpriteNode` representing the falling block.

In-place mutating the scene description on each key press is not a particular functional approach. The SpriteKit API has several occurrences of methods of node classes and similar that are invoked at specific times by SpriteKit (they are essentially callbacks) and which can mutate properties in the scene graph in an entirely unstructured manner. The most prominent of these methods is the `update(_:)` method of `SKScene`:

```
class SKScene: SKEffectNode {
  var  backgroundColor: NSColor
  func update(_ currentTime: TimeInterval)
   ⋮
}
```

The `update(_:)` method,[3] by way of `self`, has access to the entire scene graph and may mutate any and all mutable properties (which are most of them) of all nodes. In the Haskell SpriteKit interface,

[1]Almost all Objective-C declarations can be directly translated into Swift, which typically makes them more readable, unless you are fluent in Objective-C. Hence, we chose to present SpriteKit interface code in Swift, although it is an Objective-C framework.

[2]The iOS variant of Cocoa, called *Cocoa Touch*, replaces AppKit by *UIKit*, which is a somewhat simplified and modernised variant. In UIKit, the superclass of `SKNode` is `UIResponder`, but it serves the same general purpose as `NSResponder`.

[3]Function arguments in Swift (and Objective-C) are labelled; i.e., `update(_:)` is a unary function whose single argument is named `currentTime` and is of type `TimeInterval`. The absence of an explicit return type implies that the return type is `()`; in Haskell terms essentially `IO ()`.

we need to find a way to avoid this unstructured in-place mutation entirely.

2.5 Problem #3: in-place scene graph edits

In addition to changing visual and physical properties of individual nodes, the scene graph is even more deeply affected by changes to the structure of the graph. For example, whenever a block falling in from the top hits another block, Shades determines whether any blocks need to be deleted due to matching colours. Once the system comes to a rest —i.e., when all reshuffling triggered by deleted blocks has ceased— the game spawns a new active block falling in from the top.

Both deleting and spawning blocks affects the graph structure and, in Swift or Objective-C, is accomplished by a number of methods which all node classes inherit from SKNode. Here are a few of them:

```
class SKNode: NSResponder {
  var  children: [SKNode] { get }
  var  parent:   SKNode?  { get }

  func addChild(_ node: SKNode)
  func removeChildren(in nodes: [SKNode])
  ⋮
}
```

The properties `children` and `parent`[4] are mutable (as indicated by the keyword `var`), but they can only be mutated from code inside the class as they are marked as `get` only. The methods `addChild(_:)` and `removeChildren(in:)` will alter these properties to perform graph edits. Just like with directly mutable properties, we need to find a functional way to represent these actions in Haskell SpriteKit.

3 A Purely Functional Interface

To provide a purely functional interface for SpriteKit and similar frameworks, we need to address the three issues introduced in the previous section: (1) we need to resolve the need for subclassing; (2) we need to deal with changes of scene graph properties; and (3) we need to handle edits to the structure of the scene graph itself.

3.1 Algebraic datatypes and pure functions

For the functional API, we intentionally refrain from using advanced language features and extensions. Instead, we stick to functional programming fundamentals, namely algebraic datatypes and pure functions. Among other things, visual applications tend to be of appeal to beginners (both of programming, in general, and of functional programming, in particular). A simple interface is a prerequisite for effective use in teaching.

3.2 Scenes and nodes

We begin by modelling SpriteKit scenes and nodes as Haskell datatypes, addressing Problem #1 from Section 2.3. We can simply ignore the fact that the Objective-C `SKScene` class is a subclass of `SKEffectNode`, as this subclass relationship isn't particularly meaningful anyway (as already noted in Section 2.3). This leaves us with SKNode and its direct subclasses, which is the object-oriented equivalent of trying to model a sum type, while simultaneously sharing the fields of SKNode with all variants of this sum.

[4]The syntax SKNode? is equivalent to `Maybe SKNode` in Haskell.

```
1  data Node u
2    = Node
3      { nodeName               :: Maybe String
4      , nodePosition           :: Point
5      , nodeChildren           :: [Node u]
6      , nodeActionDirectives   :: [Directive (Node u)]
7      , nodePhysicsBody        :: Maybe PhysicsBody
8      , nodeUserData           :: u
9      , ...
10     }
11   | Label
12     { ...  — repeats all fields of the 'Node' constructor
13     , labelText              :: String
14     , labelFontColor         :: Color
15     , labelFontName          :: Maybe String
16     , ...
17     }
18   | Shape
19     { ...  — repeats all fields of the 'Node' constructor
20     , shapePath              :: Path
21     , shapeFillColor         :: Color
22     , ...
23     }
24   | Sprite
25     { ...  — repeats all fields of the 'Node' constructor
26     , spriteSize             :: Size
27     , spriteAnchorPoint      :: Point
28     , spriteTexture          :: Maybe Texture
29     , ...
30     }
```

Figure 5. Haskell definition of scene nodes

In Haskell, we render this directtly as a sum with named record fields with a plain Node alternative (representing SKNode) and one further alternative for each direct subclass (here the `Label`, `Shape` and `Sprite` alternatives) as shown in Figure 5. SpriteKit does include a few more node variants, however our Haskell bindings currently only supports a subset of them. The missing variants can be added in much the same way; it is just a matter of implementing the binding.

Most importantly, all fields of the node constructor (`nodeName`, `nodePosition`, and so forth) are repeated in every single alternative. This enables us to use the corresponding projection functions uniformly on all values of type Node, partially replicating the functionality provided by subclassing in the Objective-C API.

Moreover, the Node datatype in Figure 5 includes a type parameter `u` to type the `nodeUserData` field (Line 8) included in each variant of Node. This type parameter allows users to equip nodes with application specific information. In Objective-C or Swift, this would typically be achieved by providing application-specific subclasses of the assorted node variants.

```
 1 data Scene sceneData nodeData
 2   = Scene
 3     { sceneName              :: Maybe String
 4     , sceneChildren          :: [Node nodeData]
 5     , sceneActionDirectives :: [SDirective (Scene sceneData nodeData) (Node nodeData)]
 6     , sceneData              :: sceneData
 7     , sceneUpdate            :: Maybe (SceneUpdate sceneData nodeData)
 8     , scenePhysicsWorld      :: PhysicsWorld sceneData nodeData
 9     , sceneHandleEvent       :: Maybe (EventHandler sceneData)
10     , ... — more fields
11     }
12
13 type SceneUpdate sceneData nodeData = Scene sceneData nodeData → TimeInterval → Scene sceneData nodeData
14
15 type EventHandler userData = Event → userData → Maybe userData
16
17 data PhysicsWorld sceneData nodeData
18   = PhysicsWorld
19     { worldGravity          :: Vector
20     , worldSpeed            :: GFloat
21     , worldContactDidBegin :: Maybe (PhysicsContactHandler sceneData nodeData)
22     , worldContactDidEnd   :: Maybe (PhysicsContactHandler sceneData nodeData)
23     }
24
25 type PhysicsContactHandler sceneData nodeData
26   = sceneData → PhysicsContact nodeData → (Maybe sceneData, Maybe (Node nodeData), Maybe (Node nodeData))
27
28 data PhysicsContact nodeData = PhysicsContact{ contactBodyA :: Node nodeData, contactBodyB :: Node nodeData, ...}
```

Figure 6. Haskell definition of scenes

As we are ignoring SKScene's subclass relationship with SKNode by way of SKEffectNode, we simply represent it in a datatype Scene of its own, as outlined in Figure 6. This is perfectly fine as a scene node always and only occurs in the form of the root node of a scene. The Scene datatype includes a list of children, which are Nodes, in addition to a range of scene-wide properties. It is parameterised with two types, represented by the type variables sceneData and nodeData. The latter is used to parameterise the scene's child nodes and the former to add application-specific state to the scene itself (Figure 6, Line 6), again obviating the need for subclassing.

3.3 Game state in Shades

In Shades, we use the types NodeState and SceneState (Figure 7, Lines 1 & 7) to define ShadesNode and ShadesScene by instantiating Node and Scene, respectively (Lines 16 & 17). This enables us to keep track of the colour of each block (by way of the NodeState).

Shades can be in one of three basic GameStates, as illustrated in Figure 8:

1. Running: an active block is currently falling down and can be moved to the left or right by the player;
2. Landed: the active block made contact with the ground or another block. If this happens in the topmost row of the field, the game is over. If it fills a row of blocks of the same colour, or merges with other blocks, blocks are removed or change colour, and the score is updated. The player can't interact with the game in this state. Once none of the blocks move anymore, a new block is spawned and the game is back in Running mode.
3. GameOver: one column of blocks reaches the top of the play area.

The current state is maintained as one field of SceneState (Figure 7, Line 12), together with the current score, flags signalling whether the key to move left or right has been pressed, an optional value indicating the score increase during the last frame, and randomInts, which contains a stream of pseudo random numbers used to pick the shade of newly spawned blocks.

The remaining code in Figure 7 (from Line 19 onwards) initialises a Shades scene at the start of the game, where the scene has three children: a subtree containing the walls, a subtree for blocks, and a node for the score. In the initial scene, we have no blocks yet. The function `node :: [Node u] → Node u` constructs a new group node from a list of child nodes and initialises it with the SpriteKit node defaults.

3.4 Physics

As a game progresses, its scene graph changes continuously. Part of these changes are due to the simulation of the physical behaviour

```
data NodeState = NoState               — for non-block nodes
               | NodeCol NodeColour    — for block nodes
               deriving (Eq, Show)

data GameState = Running | Landed | GameOver

data SceneState = SceneState
                  { sceneScore   :: Int
                  , leftPressed  :: Bool
                  , rightPressed :: Bool
                  , bumpScore    :: Maybe Int
                  , gameState    :: GameState
                  , randomInts   :: [Int]
                  }

type ShadesNode  = Node NodeState
type ShadesScene = Scene SceneState NodeState

initialShadesScene :: ShadesScene
initialShadesScene
  = (sceneWithSize (Size width height))
    {sceneChildren = [blocks, walls, score], ...}

blocks :: ShadesNode
blocks = (node []){ nodeUserData = NoState
                  , nodeName     = Just "Blocks"
                  }

walls :: ShadesNode
walls = (node [left, bottom, right])
        {nodeUserData = NoState}

left :: ShadesNode
left = ...
```

Figure 7. Scene and node type for the Shades game

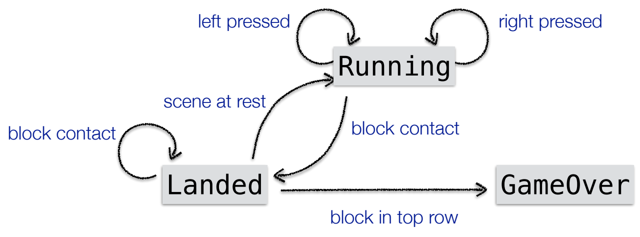


Figure 8. Game states and transitions

of the nodes. These changes don't require the programmer to actively update the scene graph, as they are declaratively specified by values of type PhysicsBody, which specifies properties such as mass, restitution, friction, which categories of objects it interacts with, and so on. Each node may optionally be associated with a physics body by way of the nodePhysicsBody field (Figure 5, Line 7). Similarly, scenes may be associated with a PhysicsWorld (Figure 6, Line 27), specifying the gravity of the scene, the simulation speed, and so on by way of scenePhysicsWorld (Figure 6, Line 8).

SpriteKit takes care that property changes by the physics engine are properly sequenced with the execution of user code that can observe these properties — anything else would result in a concurrency control nightmare. Hence, Haskell code will never get into a situation where properties change behind the scenes, while that Haskell code is running. This is not just a lucky break provided by SpriteKit, but generally how similar frameworks are designed to avoid costly locking on a property by property basis.

Shades uses the physics engine only lightly; to simulate gravity affecting blocks, boxing the blocks into the scene, and handling collisions between blocks. This automatically ensures that gaps are filled when blocks disappear. Hence, we associate physics bodies with blocks and the (invisible) walls. In contrast, the score node doesn't interact with other nodes in a physical manner, and hence, is not associated with a physics body.

3.5 Transformation instead of mutation

Changes to the scene graph are, of course, not restricted to the mere physical simulation of objects. We also have: (1) reaction to external events, such as mouse or keyboard events; (2) the effect of collisions which go beyond a purely physical collision behaviour; and (3) the continuously evolving game state as the game progresses.

These changes are the root cause for Problem #2 (Section 2.4), where the properties of individual nodes get updated, and Problem #3 (Section 2.5), where the structure of the scene graph changes. To preserve a functional interface, all these changes are expressed in terms of purely functional transformations in Haskell, regardless of whether individual nodes or the entire scene gets transformed.

3.6 Transformation by animation

Let us first focus on Problem #2, where we change the properties of individual nodes. One way of doing so is by explicitly using record updates to change fields of a Node value. For example, we can move a Sprite node by changing its position field:

```
moveLeftUpd :: ShadesNode → ShadesNode
moveLeftUpd sprite@Sprite{nodePosition = Point x y} =
  let x' = (blockWidth / 2) `max` (x - blockWidth)
  in  sprite{ nodePosition = Point x' y }
```

Alternatively, we use the nodeActionDirectives field of the node (Figure 5, Line 6) to add an animation action. SpriteKit provides a variety of smart constructors for SAction values, such as moving, resizing, fading, colour changes, playing sound files, animating a node with a sequence of textures, and more:

```
moveTo  :: Point → SAction node children
scaleTo :: GFloat → SAction node children
fadeOut :: SAction node children
...  — there are several more such functions in SpriteKit
```

In contrast to simple field updates, these actions can be animated over a user-specified duration — for example, this is how we implement the melting away of blocks with matching colours in Shades.

The function:

```
runAction :: SAction node children
          → SDirective node children
```

converts a SAction into a SDirective, so that we can use it on a node:

```
moveLeft :: ShadesNode → ShadesNode
moveLeft sprite@Sprite{nodePosition =  Point x y}
  = sprite{ nodeActionDirectives =
              [runAction $ moveTo $
                  Point ((blockWidth/2) `max`
                         (x - blockWidth)) y]
          }
```

Actions are more versatile than just simple updates of node properties. Not only because most properties can be animated over a duration, some properties, such as playing a sound file, have no corresponding node record field. Multiple actions can also be combined sequentially or in parallel:

```
sequenceActions    — sequential composition of actions
  :: [SAction node children] → SAction node children
groupActions       — parallel composition of actions
  :: [SAction node children] → SAction node children
```

or applied repeatedly:

```
repeatActionCount
  :: SAction node children → Int
  → SAction node children
repeatActionForever
  :: SAction node children → SAction node children
```

We can specify the duration over which a change gets animated by setting an action's actionDuration field.

In fact, by way of custom actions, we can turn any Haskell function on a node into an action:

```
type TimedUpdate node = node → GFloat → node
customAction
  :: TimedUpdate node → SAction node children

moveLeftAction =
  customAction (const (flip moveLeftUpd))
```

In Shades, whenever a block contacts with another block of the same colour, we first change the colour of both blocks to the next darker shade; then, we let the lower block slowly disappear by scaling its height to zero over one second (using an action with an actionDuration of one second), which provides the visual effect of the top block melting into the lower one as it continues to fall down the screen. Finally, when the node is invisible, we use the removeFromParent action to detach the node from its parent node.

```
meltBlock node
  = node{nodeActionDirectives
         = [runAction $ sequenceActions
             [ customAction darkenBlock
             , (scaleYTo 0){actionDuration = 1}
             , removeFromParent
             ] ]}
```

3.7 Transformative callbacks

The application of node transformations, such as those described previously, are in response to events which trigger callback handlers — such events arise, for example, from keyboard or mouse input, node collisions, or the start of a new animation frame. We realise all these callback handlers as pure transformation functions.

Handling physics contacts. Two examples of such handlers are worldContactDidBegin/End fields in PhysicsWorld (Figure 6, Line 21 and 22). They are invoked whenever the physics engine detects the start or end of contact (i.e., overlap) between two physics bodies, and are used to implement contact behaviours beyond the bodies physically bouncing off each other.

In Shades, we use worldContactDidBegin to check whether a block landed on a block of the same colour. Whenever that happens, the contact handler transforms the scene state to increment the score and the contacting nodes by darkening both blocks and "melting" the lowermost block.

```
contact :: SceneState
        → PhysicsContact NodeState
        → (Maybe SceneState,
            Maybe ShadesNode,
            Maybe ShadesNode)
contact state@SceneState{..} PhysicsContact{..}
  | (sameColour contactBodyA contactBodyB) &&
    (sameColumn contactBodyA contactBodyB)
  = if (above contactBodyA contactBodyB)
      then  (Just $ incScore state,
             Just $ darkenBlock contactBodyA,
             Just $ meltBlock contactBodyB)
      else  (Just $ incScore state,
             Just $ meltBlock contactBodyA ,
             Just $ darkenBlock contactBodyB)
  ...
  | otherwise
  = (Nothing, Nothing, Nothing)
```

Handling input events. Another important callback handler is sceneHandleEvent (Figure 6, Line 9), which SpriteKit invokes for every input event (from a keyboard, mouse, touch screen, etc.). The event handler may transform the scene state or, if it is not prepared to handle the event, may just return Nothing — in which case the event gets propagated up Cocoa's responder chain [1].

In Shades, the event handler simply checks whether the event indicates a left or right arrow keypress. If so, it sets the corresponding flag in the SceneState.

```
handleEvent :: Event
            → SceneState
            → Maybe SceneState
handleEvent KeyEvent{ keyEventType = KeyDown
                    , keyEventKeyCode = code } state
  | code == leftArrowKey
  = Just state { leftPressed = True }
  | code == rightArrowKey
  = Just state { rightPressed = True }
handleEvent _ _ = Nothing
```

Handling the start of an animation frame. In the handleEvent code, we see that it does not directly move the active block left or right, but only sets a flag. The actual transformation of the block position is left to the scene's main update handler determined by the sceneUpdate field of a Scene (Figure 6, Line 7). It is invoked each frame before the animation system and physics engine are allowed to change the scene state.

In Shades, we use the function update from Figure 9. In the first alternative of the case expression dispatching on the current game state, the game reacts to a left arrow press flagged by the event handler, by running a custom action on the active block that moves it to the left. Together with the contact and event handlers, the update function implements the state transition diagram from Figure 8.

3.8 Editing trees

As we saw in Figure 7, from Line 29 onwards, the initial scene for Shades only consist of nodes for the walls, the score, and a blocks node with no children yet. During game play, we need to add new nodes to the scene graph (whenever we spawn a new block) and delete nodes (whenever a node melts into another or when a row of blocks of the same colour disappears).

As we saw in Section 2.5, the Objective-C API of SpriteKit treats adding and removing children of nodes differently to changing other properties. In contrast, in Haskell, the nodeChildren field of Node (Figure 5, Line 5) is just a normal list. This keeps scene tree edits, just like other property changes, straight forward in the Haskell interface. As an example, consider the following code that spawns a new active block that falls in from the top:

```
spawnNewBlock :: ShadesScene → ShadesScene
spawnNewBlock scene0 =
  scene
  { sceneActionDirectives
              = [runCustomActionOn "Blocks" addBlock]
  , sceneData = sceneState { gameState = Running } }
  where
    addBlock node{ nodeChildren = children } _ =
      node{ nodeChildren = block col : children }
    (scene@Scene{sceneData = sceneState}, col) =
      randomColour scene0

block :: NodeColour → ShadesNode
block colour = (spriteWithTexture texture)
               { nodeUserData = NodeCol colour, ... }
```

Finally, Shades uses a function resting that is invoked when all physical activity has come to a rest after a block landed on the ground or on other blocks. It checks whether there is a row of blocks of the same colour. If that is the case, that row gets deleted; if not, it invokes the spawnNewBlock function, which we just discussed, to start new activity.

In the second alternative of the case expression, we see how resting deletes a row of blocks. Specifically, it sets the new list of blocks, determined by removeColouredRow, via a helper function setNodeChildren and a custom action:

```
resting :: Int → ShadesScene → ShadesScene
resting row scene{..}
  | maxRow scene > blocksInCol
  = scene{ sceneData = sceneData{ gameState = GameOver}}
  | otherwise
  =  case removeColouredRow scene of
       Nothing     → spawnNewblock scene
       Just blocks →
         scene { sceneData = incRowScore sceneData
               , sceneActionDirectives =
                   [ runCustomActionOn "Blocks"
                       (setNodeChildren blocks) ] }
  where
   -- Return top most row number with block in it
   maxRow :: ShadesScene → Int

   -- Return Nothing if no row of same colour can be found.
   -- Otherwise, return list of blocks without that row.
   removeColouredRow :: ShadesScene
                     → Maybe [ShadesNode]

   setNodeChildren :: [ShadesNode] → ShadesNode
                   → TimeInterval → ShadesNode
   setNodeChildren newKids node _ =
     node{ nodeChildren = newKids }
```

Once again, this is a pure function, mapping the current scene to the follow up scene.

4 Behind the Scenes

Now that we have settled on a purely functional interface for SpriteKit, we need to look at implementing it correctly and efficiently. To understand the challenges, let us look in some more detail at the process of invoking the update handler passed in a Scene's sceneUpdate function. Figure 10 illustrates what happens in this process if we implement it naïvely, for the situation where we spawn a single new block in our Shades example program. The naïve method performs the following steps:

1. Marshal the Objective-C scene graph to a value of the Haskell datatype Scene.
2. Apply the sceneUpdate transformation function to this Haskell value, which will result in a new Scene value.
3. Marshal the updated Scene value from Haskell back to Objective-C.

The new scene includes one new Node, namely the newly spawned block, but all the other nodes will be the same.

This is obviously wasteful. Most nodes are needlessly marshalled back and forth between Objective-C and Haskell without any change. Moreover, replacing nodes in the scene graph by marshalled copies is often incorrect. Many nodes can have hidden state; i.e., information that is not part of their public interface and hence not represented in the Haskell rendering of a node. As a consequence, that state is not reconstructed when such a node is marshalled back from Haskell to its Objective-C representation. A common example of hidden state are long running actions attached to a node. This can, for example, be a movement action which is not instantaneous, but rather proceeds more slowly over several frames of animation, or a sound that plays for a short while.

```
update :: ShadesScene → TimeInterval → ShadesScene
update scene@Scene{ sceneData = sceneState@SceneState{..} } _dt
  = case gameState of
      Running | leftPressed          → scene
                                       { sceneActionDirectives = [runCustomActionOn "Block" (const moveLeft)]}
                                       , sceneData             = sceneState{ leftPressed = False}
                                       }
              | rightPressed         → scene{...much like above...}
              | Just n ← bumpScore  → incScore n scene
              | otherwise            → scene
      Landed                         → if sceneResting scene then resting scene else scene
      GameOver                       → gameOver scene
```

Figure 9. The main update handler of Shades

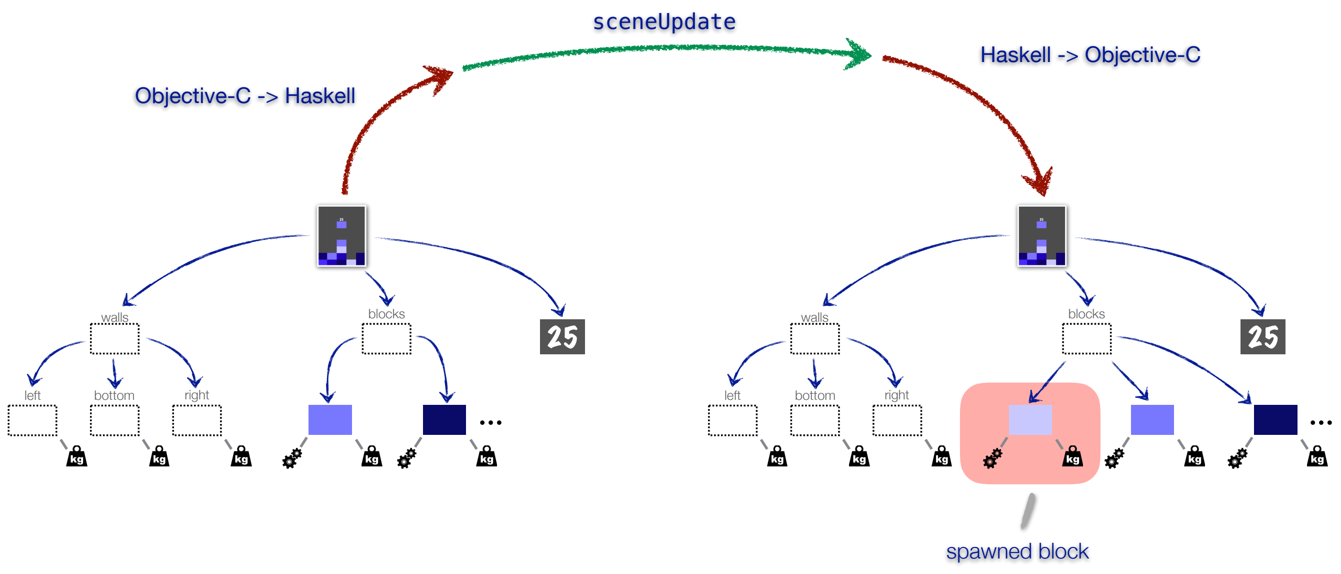


Figure 10. Naïve marshalling for sceneUpdate, for the process of spawning a single new block.

If we marshal a node with a currently attached long-running action to Haskell and back to Objective-C, the action will be lost—i.e., the animation will stop before it has completed, or the sound stops playing before it is finished. Hence, it is crucial for efficiency as well as for correctness that we re-use existing Objective-C node representations when marshalling back and forth between the two languages. Moreover, we want to avoid marshalling nodes that are not used during a particular invocation of sceneUpdate — more precisely, our aim is to ensure that marshalling will only impose a linear overhead in sceneUpdate and similar callbacks, independent of the size of the scene graph. These requirements lead us to the following approach, which is illustrated in Figure 11:

1. *Lazily marshal* the Objective-C scene graph to a value of the Haskell datatype Scene.
2. Apply the sceneUpdate transformation function to this Haskell value, which will result in a new Scene value.
3. *Compute the difference* between the old and new value, corresponding to the changes effected by the transformation function sceneUpdate.
4. *Apply those changes* to the original Objective-C scene graph.

In other words, we do not create a new scene graph from the Haskell scene description returned by sceneUpdate. Instead, we directly apply the changes derived from the difference between the old and new scene value and in-place mutate the existing scene — that is, based on the information derived from the output of the pure transformation function, the adaptation layer replicates the in-place mutation that an Objective-C program would have directly performed on the object graph representing the scene. In contrast to the purely functional API offered to a user of Haskell SpriteKit, the implementation of the adaptation layer will have to resort to unsafe low-level programming techniques — after all, all magic has its price. In the following, we describe the three main techniques to realise this scheme in more detail.

4.1 Lazy marshalling

A scene description is a tree in its Haskell representation. On invoking sceneUpdate, we only marshal the root of that tree from its Objective-C representation to its Haskell representation; the

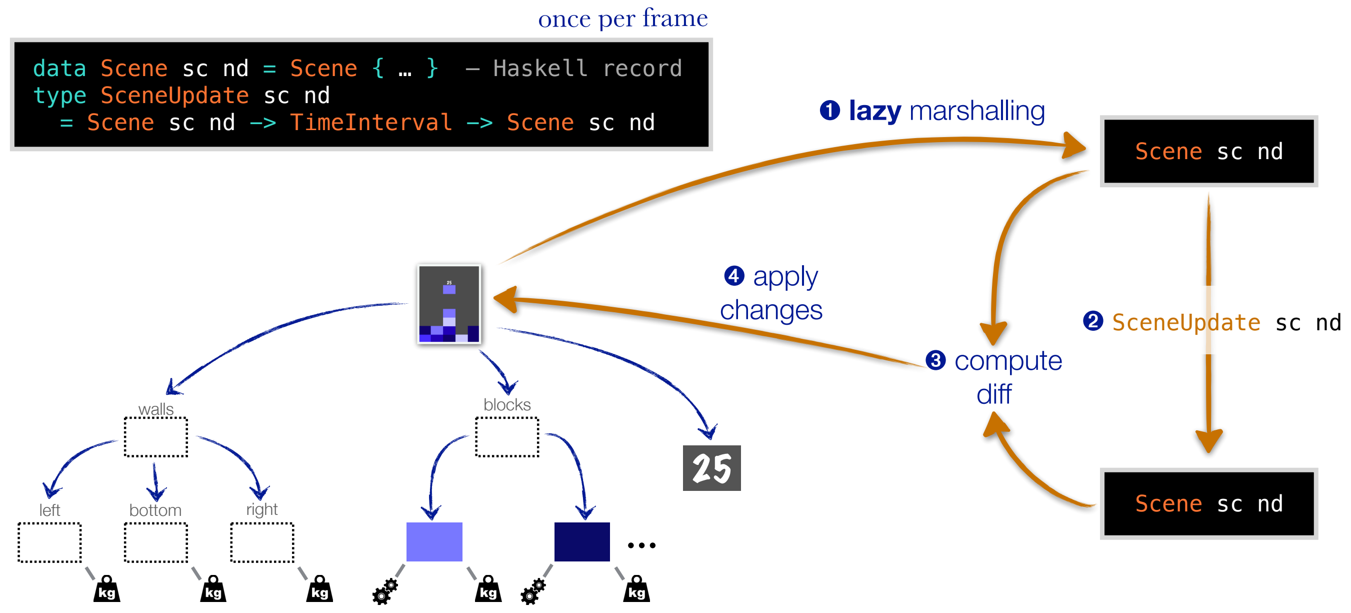


Figure 11. Efficient implementation of `sceneUpdate`, as a transcription of pure transformations into direct graph edits.

marshalling of all subtrees and, indeed, the marshalling of all the individual properties of the Scene record are deferred. Conveniently, Haskell, by being a lazy language, already comes with all the facilities that we need to realise this scheme. All that is left for us to do is to ensure that all fields of the Scene constructor are initialised with thunks, and that these thunks perform the marshalling of the scene properties and subtrees only when demanded.

However, FFI marshalling typically involves functions in the `IO` monad, such as the methods of the `Storable` class. Hence, to perform marshalling for a pure result, we need to resort to `unsafePerformIO`. This is not unusual when implementing a Haskell binding for a foreign library. However, typically, we only use one occurrence of `unsafePerformIO` to perform the entire marshalling of a foreign structure or object, following the below pattern:

```
data Struct = Struct {field1 :: T1, ..., fieldn :: Tn}

marshalStruct :: Ptr Struct → Struct
marshalStruct ptr = unsafePerformIO $ do
  field1 ← <get first field>
  ⋮
  fieldn ← <get nth field>
  return Struct{..}
```

This code marshals the entire structure eagerly. To marshal lazily, we push the use of `unsafePerformIO` into the leaves — i.e., into the marshalling of the individual fields. Hence, we arrive get:

```
marshalStruct :: Ptr Struct → Struct
marshalStruct ptr = Struct{..}
  where
    field1 = unsafePerformIO <get first field>
    ⋮
    fieldn = unsafePerformIO <get nth field>
```

Here we return the outermost structure right away and defer marshalling of the components until they are demanded. This is exactly the approach that we take in Haskell SpriteKit. In fact, the marshalling of scenes is achieved as follows:

```
marshalSKScene :: SKScene → Scene sd nd
marshalSKScene skScene
  = Scene
    { sceneName      = unsafePerformIO $(objc ... )
    , sceneChildren =
        unsafePerformIO $ do
        { nodes ← $(objc ... )
        ; unsafeInterleaveNSArrayToListOfNode nodes
        }
    ... }
```

Here we use the package `language-c-inline` for the actual marshalling of values, which includes facilities for using Objective-C inline in Haskell by way of quasi-quotation in Template Haskell [7, 15, 17]. The splices of inline Objective-C are denoted by `$(objc ⋯)` above. Here, `SKScene` is the type of a reference to an Objective-C SpriteKit scene node; a newtype wrapping of a foreign pointer [6].

The function `unsafeInterleaveNSArrayToListOfNode` builds a list of thunks that marshal subtrees, so that descending into one subtree does not cause sibling subtrees to be marshalled. We use the same approach with the various flavours of SpriteKit nodes, regardless of whether they appear as subtrees to a scene (as above) or whether they are directly passed to callbacks, such as contact handlers or custom actions.

4.2 Change detection

Once the `sceneUpdate` transformation function yields a transformed scene (of type `Scene`), we need to compare that transformed scene to the original scene value passed as an argument to `sceneUpdate` to determine the difference. Again, we need to be careful. If we compare both trees by a conventional tree walk, we will perform work in the order of the size of the scene graph, even if only one property changed. Moreover, we will undo the benefits of lazy marshalling, as the comparison will demand all marshalling to be performed.

In order to avoid the later, we need to be able to determine whether a value has changed without demanding the thunk that may encapsulate the marshalling of that value. There is no precise method to check that property, but we can approximate. The GHC primitive `reallyUnsafePtrEquality#`, determines whether the pointers to the heap node representing the two arguments are the same pointer (i.e., the heap nodes are located at the same address in the Haskell heap). If that is the case, we definitely know that the two arguments are the same thing, whether evaluated or not. Hence, the arguments represent equal values.

Conversely, if `reallyUnsafePtrEquality#` flags the two pointers as not equal, we haven't learnt anything useful. After all, two different pointers can still point to two Haskell structures that are equivalent. However, for our use of `reallyUnsafePtrEquality#` that is not an issue. In the worst case, if we update a property in the scene graph with a value that it already contained, we have performed superfluous work, but we haven't changed the semantics of the computation.

In other words, whenever `reallyUnsafePtrEquality#` deems the values of a field in the `Scene` or `Node` type to be represented by the same pointer in both the original and the transformed version, we know that this field was not changed and we don't need to update it in the Objective-C version of the scene graph. However, if `reallyUnsafePtrEquality#` finds the pointers to be different, we update the corresponding property in the scene graph with the value projected from the transformed `Scene` or `Node` value. That update may be superfluous, but it is never wrong.

The following code excerpt illustrates the idea:

```
marshalScene :: Scene sd nd     -- original scene
             → Scene sd nd      -- updated scene
             → IO SKScene
marshalScene originalScene Scene{..} =
  do
  { ...
  ; case reallyUnsafePtrEquality#
           originalName sceneName of
      1# → return ()
      _  → $(objc ...)         -- update scene graph
  ; updateChildren skNode
      originalChildren sceneChildren
  ⋮
  }
```

While, in principle, several operations of the runtime system, such as garbage collection, and of heap manipulations of the generated code, such as a record updates, may introduce new pointers to identical structures, this rarely happens in practice. After all, both the compiler generated code and the runtime system are quite careful not to unnecessarily demand thunks or copy structures as this carries the risk of non-termination or at least increased resource consumption. As we only use `reallyUnsafePtrEquality#` to avoid superfluous work, and not to make decisions affecting semantics, we are on the safe side and generally achieve our performance goal.

Moreover, SpriteKit callbacks, such as `sceneUpdate`, are necessarily short running. They are executed often —in the case of `sceneUpdate`, once per frame— and if they are long running, animation performance will be compromised anyway.

It would be interesting to investigate whether the use of GHC's `StableNames` might be helpful in this context. The runtime system is more careful about duplicating pointers that have been used to create a `StableName` and `StableNames` can be compared for equality. However, `StableNames` in turn also carry a runtime cost and SpriteKit would need to allocate quite a few `StableNames` to marshal and diff one node. As unnecessarily updating one or multiple scalar values in a node, once in a while, is not going carry a significant performance penalty, it doesn't seem worthwhile to generally use `StableNames`. However, unnecessarily updating an entire subtree in a large scene might potentially be sufficiently costly to lead to the occasional dropped frame. If that is the case, using `StableNames` exclusively to improve the accuracy of the change detection for subtree pointers might be worthwhile.

4.3 Object caching

To avoid re-creating SpriteKit nodes in an Objective-C to Haskell to Objective-C roundtrip —and, as we have discussed earlier, lose vital hidden state in the process— we need to keep track of the association between Objective-C nodes and their representations in Haskell. This allows us to identify the Objective-C node which needs to be updated when we marshal a Haskell node back to Objective-C.

Unfortunately, it is difficult to maintain this association between Haskell nodes and the Objective-C nodes they originate from in a manner that is completely transparent to a user of the Haskell SpriteKit API. After all, when the user code updates a node in a transformation function, it will ultimately allocate a new copy of that node and we cannot keep track of that relationship from the outside. We address this issue by including a field `nodeForeign` in every variant of the `Node` datatype:

```
data Node u
  = Node
    { nodeName        :: Maybe String
    , nodePosition    :: Point
    ⋮
    , nodeForeign     :: Maybe SKNode
    }
  | Label
    { nodeName        :: Maybe String
    , nodePosition    :: Point
    ⋮
    , nodeForeign     :: Maybe SKNode
    , labelText       :: String
    ⋮
    }
    ⋮
```

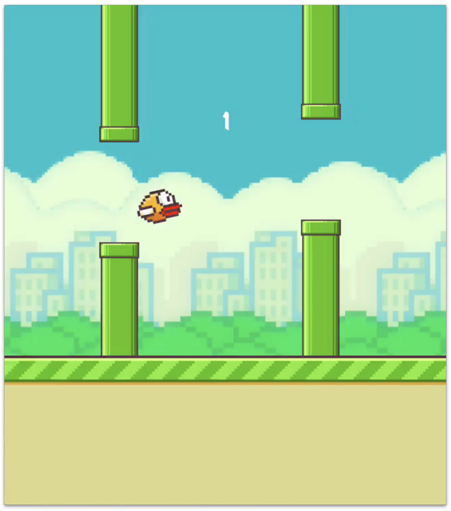


Figure 12. Lazy Lambda — a Haskell clone of Flappy Bird

On marshalling an Objective-C node to Haskell, a foreign pointer to the original Objective-C node is placed into the `nodeForeign` field. In contrast, nodes which have been newly created from the Haskell side, will have no such foreign pointer.

When marshalling the Haskell representation back to Objective-C, the presence of a foreign pointer in `nodeForeign` indicates that we ought to update the existing Objective-C node. Conversely, if `nodeForeign` is `Nothing`, we must create a new Objective-C node from scratch.

The pitfall in this scheme is that duplicating a node with an associated foreign pointer in Haskell land and marshalling both of these nodes back into Objective-C creates an ambiguity. Only one of the two Haskell nodes can be marshalled back to Objective-C by reusing the existing Objective-C node, and Haskell SpriteKit has no means to determine which one it ought to be. Note that this choice determines which version of the Haskell node inherits the hidden state in the original Objective-C node, and thus the choice may lead to a behavioural difference. Hence, when duplicating nodes in Haskell code, deterministic behaviour requires setting the `nodeForeign` field of all but one node to `Nothing`. This is a rare situation, but there is no way to check for it statically.

5 Conclusions

We demonstrated how we can put a purely functional interface on imperative, object-oriented frameworks based on a mutable scene graph at the example of SpriteKit. Although our Haskell interface does not currently expose all functionality supported by the native SpriteKit library, it covers all core functionality and is sufficient to implement interesting games. We used Shades as a simple example and have also implemented a Haskell clone of Flappy Bird [16] (see Figure 12).

Most of the functionality of the native SpriteKit library that is currently not supported —such as positional audio, light sources, particle effects, and inverse kinematics— can be easily added by following the exact same approach presented in this paper; it is simply the size of the API, not the difficulty of the task, that makes this a significant amount of work.

Future work. In addition to extending Haskell SpriteKit to cover the complete set of functionality of the native library, the most interesting topic for future work would be to investigate avenues to automate the implementation of lazy marshalling and change detection, possibly by way of Template Haskell or another generic programming mechanism. Most of the code to lazily marshal individual record fields and to detect changes is both (a) low-level, and (b) repetitive. Hence, it seems like an ideal target for automation.

It remains to be explored how some aspects of SpriteKit can be provided nicely from Haskell. For example, `SKEffectNode` enables the application of image filters onto parts (or all) of a scene. In order to use these, another Cocoa subsystem, CoreImage [3], would need to be supported as well. Moreover, we have ignored the fact that SpriteKit scene graphs contain back edges. In principle we could introduce cycles into the Haskell representation as well, but this seems unwieldy. Alternatively, we could provide query functions to determine the parent node and enclosing scene of a given `Node` value. This is certainly sufficient for inspection, but becomes trickier if we want to transform these nodes as well. However, our experience with SpriteKit suggests that it is usually sufficient to add actions to transform the user data of these nodes, which again seems perfectly feasible within the approach outlined in this paper. Finally, SpriteKit contains node query functions and related functionality that poses similar issues to back edges.

It would be interesting to benchmark using `StableNames` to improve change detection as discussed at the end of Section 4.2.

Acknowledgments

We warmly thank Robert Clifton-Everest and Trevor L. McDonell for their insightful feedback and help with proofreading.

References

[1] Apple Inc. 2016. Cocoa Event Handling Guide. (2016). https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/EventOverview/
[2] Apple Inc. 2016. SpriteKit. (2016). https://developer.apple.com/reference/spritekit
[3] Apple Inc. 2017. Core Image. (2017). https://developer.apple.com/reference/coreimage/ciimage
[4] Apple Inc. 2017. *The Swift Programming Language (Swift 3.1)*. iBooks Store. https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/
[5] Ben Lippmeier. 2014. Gloss. (2014). http://trac.ouroborus.net/gloss
[6] Manuel M T Chakravarty. 2003. The Haskell Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report. (2003).
[7] Manuel M T Chakravarty. 2014. Foreign Inline Code in Haskell. In *Haskell Symposium*. ACM.
[8] Mun Hon Cheong. 2005. *Functional programming and 3D games*. Master's thesis. School of Computer Scienes and Engineering, UNSW Sydney.
[9] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*.
[10] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming*. http://conal.net/papers/icfp97/
[11] Facebook Inc. 2017. Tutorial: Intro To React. (2017). https://facebook.github.io/react/tutorial/tutorial.html
[12] Heinrich Apfelmus. 2016. Reactive Banana. (2016). https://wiki.haskell.org/Reactive-banana
[13] Heinrich Apfelmus. 2017. The reactive-banana package. (2017). http://hackage.haskell.org/package/reactive-banana
[14] Daan Leijen. 2004. wxHaskell: A Portable and Concise GUI Library for Haskell. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*.
[15] Geoffrey Mainland. 2007. Why it's nice to be quoted. In *Haskell Symposium*.
[16] Manuel Chakravarty. 2017. Lazy Lambda: a Flappy Bird clone in Haskell with SpriteKit. (2017). https://github.com/mchakravarty/lazy-lambda
[17] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*.
[18] The Gtk2Hs Team. 2016. The gtk package. (2016). https://hackage.haskell.org/package/gtk
[19] Mike Wiering. 1999. *The Clean Game Library*. Master's thesis. Katholieke Universiteit Nijmegen.