Visualization Kripke Models

Coen Mol, Haraldur Davidsson, Maurits Dijk, Thomas Sijpkens

Sunday 2nd June, 2024

Contents

1 Introduction

This report describes a full stack application consisting of a front-end written in Elm and a back-end written in Haskell. The application allows a user to insert and modify a Kripke model in a user friendly way, and save it to a file and validate it against a predefined propositions. A general explanation about Kripke Model is given and how we modeled this concept in our application.

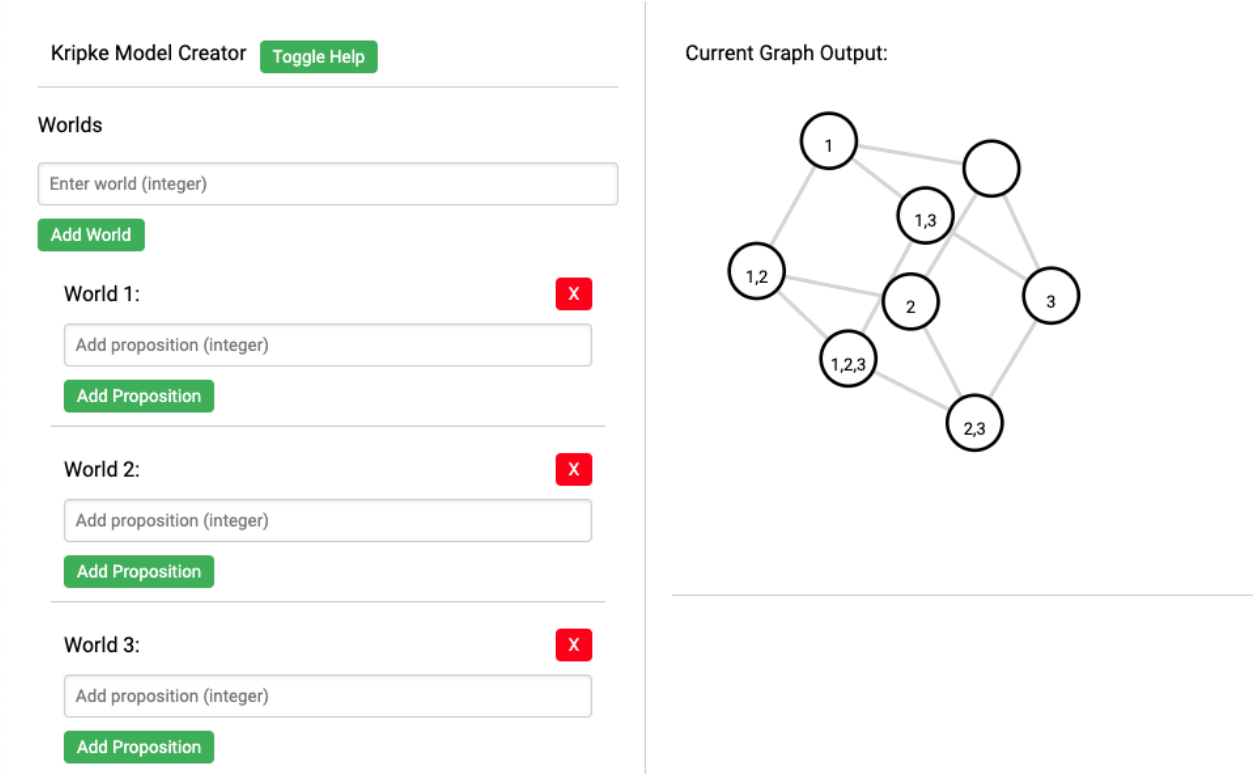

Figure 1: Example of the application state after inputting the Muddy Children model

1.1 Kripke Models

Kripke models provide a frame to interpret modal statements and reason about necessity and possibility across different "worlds". A Kripke model is a tuple, M=(W,R,V), where: W is a non-empty set of possible worlds, R is a binary accessibility relation on W(i.e. if If wRv, then world v is accessible from world w), and V is a valuation function that assigns truth values to each propositional variable at each world. In epistemic logic the relations are binary accessibility relations for the agents in the model containing the worlds that a particular agent considers to be possible. These relations are also called indistinguishably relations because they show which worlds an agent thinks is accessible, if they are related then these two worlds are indistinguishable to the agent creating uncertainty between these worlds. In our program worlds are implemented as Int so a set of worlds is just a list of Ints. Relations are lists of tuples of an agent and a list of world pairs. Mapping agents to accessibility relations.

The truth of a modal formula in a Kripke model is defined recursively as:

$$
\begin{aligned}
M, w \models p : & \quad \Leftrightarrow p \in V(w) \\
M, w \models \neg\varphi : & \quad \Leftrightarrow \textit{ not } M, w \models \varphi \\
M, w \models \varphi \wedge \psi : & \quad \Leftrightarrow M, w \models \varphi \textit{ and } M, w \models \psi \\
M, w \models K_i\varphi : & \quad \Leftrightarrow M, w\prime \models \varphi \textit{ for all } w\prime \textit{ such that } R_i w w\prime \\
M, w \models [!\varphi]\psi : & \quad \Leftrightarrow M, w \models \varphi \textit{ implies } M^{\varphi}, w \models \psi
\end{aligned}
$$

1.2 Data Model

To make the communication between the client and the server, we have defined the data model in JSON to communicate such Kripe Models. JavaScript Object Notation (JSON) is an open standard data exchange format for the web. It is popular due to it's readability and efficiency. JSON uses key-value pairs and has the following shape:

```
{
    "worlds": [Integer],
    "relations": [
        {
            "agentName": string,
            "worldRelations": [[Integer]]
        }
    ],
    "valuations": {
        "world": Integer,
        "propositions": [Integer]
    }
}
```

2 System requirements

The requirements of the application define the applications scope and capabilities along with its quality guarantees, where basic functionality is listed in the Functional Requirements, and the quailty functionality listed in the Nonfunctional Requirements. The application consists of the following Functional and Nonfunctional requirements:

2.1 Functional Requirements

- **FR1**: The system shall provide functionality for users to save their Kripke models to a database.
- **FR2**: The system shall allow users to input Kripke models via designated input fields for worlds, agents, and relations.
- **FR3**: Upon user request, the system shall visually represent the Kripke model showing worlds, agents, and their relations, along with a JSON representation of it.
- **FR4**: The system shall allow users to validate Kripke models against a single specified proposition

2.2 Nonfunctional Requirements

- **NFR 1**: The system shall provide an intuitive user interface that allows new users to input a Kripke model and validate propositions with no more than three clicks
- **NFR 2**: The system shall provide tooltips and help documents accessible directly from the interface for guidance on creating and validating Kripke models.
- **NFR 3**: The system shall display the visual representation of the Kripke as soon as a user changes the model (within 200 ms).

3 Elm UI

3.1 Paradigm

Elm is a functional programming language designed for creating reliable web application with an emphasis on simplicity and robustness [CC13]. Compiling to JavaScript, Elm aims to eliminate common pitfalls associated with web development, such as runtime errors and type inconsistencies by taking inspiration from Haskell. Elm features a strong static type system, immutability by default, and a straightforward architecture encouraging clear and maintainable code. The goal of Elm is to create web applications that are functionally sound and semantically unbreakable.

The Elm architecture(TEA) is a pattern for structuring web applications, manage packaging, bundling, architectural code (model-view-container), state management, type checking, and immutability. TEA is interoperable with JavaScript and eliminates the need for a complex stack involving NPM (a package manager), Webpack (a module bundler), React (a user interface library), Redux(a state management library), Typescript/Flow(for type checking), and Immutable.js(for data immutability) to handle these core functions. It consists of three parts: `Model` (state), `Update` (function to update the state based on messages), and `View` (function that turns the state into HTML). This unidirectional data flow ensures predictable behavior. This components as have the following type definition:

```
-- Initial state
init : () -> ( Model, Cmd Msg )
-- actions that app supports (events or messages)
type Msg
 -- business logic (response to actions)
update : Msg -> Model -> ( Model, Cmd Msg )
-- display logic
view : Model -> Html Msg
```

3.2 Model

We defined our Elm model, using a *type alias*, each parameter is responsible for a certain section of the "state" of the application:

```
-- Define a type alias for the Model
type alias Model =
    { worlds : List (Int, String)
    , agents : List String
    , relations : List (String, List (List Int))
    , jsonOutput : String
    , worldInput : String
    , agentInput : String
    , propositionInputs : List String
    , relationInputs : List String
    , readMeContent : String
    , showPopup : Bool
    , showReadMe : Bool
    }
```

which is initialized with the default values

```
init _ =
    ( { worlds = []
      , agents = []
      , relations = []
      , jsonOutput = ""
      , worldInput = ""
      , agentInput = ""
      , propositionInputs = []
      , relationInputs = []
      , readMeContent = ""
      , showPopup = False
      , showReadMe = False
      }
    , Cmd.none
    )
```

3.3 Messages

Effects in Elm refer to tasks that need be performed outside of the normal flow of the web application itself, such as making a HTTP request or interacting with the browser externally. When the `update` function in Elm is called to update the application state, it returns a new state (Model) and can also return a `Cmd` (command), which represents a task for Elm to execute later. A `Cmd` is similar to a promise, but Elm manages it as a side effect safely and predictably.

A message `Msg` is a *custom type* representing all possible actions that can be performed in the application which are, these actions then contain some logic and update the model correspondingly.

```
type Msg
    = UpdateWorldInput String
    | AddWorld
    | UpdateAgentInput String
    | AddAgent
    | UpdatePropositionInput Int String
    | AddProposition Int
    | UpdateRelationInput Int String
    | AddRelation Int
    | RecieveReadMe (Result Http.Error String)
    | ToggleReadMe
    | FetchReadMe
    | PostKripkeModel
    | PostedKripkeModel (Result Http.Error String)
    | GotKripkeModel (Result Http.Error String)
```

Two example messages can be seen in the following code snippet from `update`:

```
update msg model =
    case msg of
        ...
        UpdateRelationInput index input ->
            let
                updatedCurrentRelationInputs =
                    List.Extra.updateAt index (\a -> input) model.currentRelationInputs
            in
            ( { model | currentRelationInputs = updatedCurrentRelationInputs }, Cmd.none )

        AddRelation agentIndex ->
            let
                maybeCurrentInput =
                    List.Extra.getAt agentIndex model.currentRelationInputs

                maybeAgentRelations =
                    List.Extra.getAt agentIndex model.relations
            in
            case (maybeCurrentInput, maybeAgentRelations) of
                (Just currentInput, Just (agentName, existingRelations)) ->
                    case parseInputToRelation currentInput of
                        Just parsedInput ->
                            let
                                relationExists =
                                    List.member parsedInput existingRelations

                                worldsExist =
                                    List.all (\w -> List.any (\(world, _) -> world == w)
                                        model.worlds) parsedInput

                                updatedRelations =
                                    if relationExists then
                                        model.relations
                                    else
                                        List.Extra.updateAt agentIndex (\_ -> (agentName,
                                            existingRelations ++ [parsedInput])) model.
                                            relations

                                updatedCurrentRelationInputs =
                                    List.Extra.updateAt agentIndex (\a -> "") model.
                                        currentRelationInputs

                                updatedModel =
                                    if relationExists then
                                        { model | error = Just Error.RelationExists,
                                            currentRelationInputs =
                                            updatedCurrentRelationInputs }
                                    else if not worldsExist then
                                        { model | error = Just Error.WorldNotExists }
                                    else
                                        { model
                                            | relations = updatedRelations
                                            , currentRelationInputs =
                                                updatedCurrentRelationInputs
                                            , error = Nothing
                                            , jsonOutput = toJson { model | relations =
                                                updatedRelations }
                                        }
                            in
                            ( updatedModel, Cmd.none )

                        Nothing ->
                            ( { model | error = Just Error.InvalidRelationInput }, Cmd.none
                                )

                (Nothing, _) ->
                    ( { model | error = Just Error.InvalidRelationInput }, Cmd.none )

                (_, Nothing) ->
                    ( { model | error = Just Error.AgentDoesNotExist }, Cmd.none )
```

The code snippet above showcases two scenario's of the update model, when the a relation input is changed, and when it is submitted. The former case handles the updates from the input and updates the model with the corresponding string, while the latter, parses the the input and checks for all errors in an idiomatic way, handling cases such as and invalid input, trying to add a relation to a world which doesn't exist etc. Errors are handled gracefully within the cases in the and displayed at the top of the UI (2), notifying the user of what was wrong with the input.

The complete list of errors can be seen in the following code snippet:

```
    type KMError
    = InvalidInput
    | WorldExists
    | AgentExists
    | AgentDoesNotExist
    | PropositionExists
    | RelationExists
    | InvalidRelationInput
    | WorldNotExists
    | NetworkError Http.Error
    | OtherError String
    | PostError
```

Figure 2: Error message in the UI

3.4 API communication

Elm has a built-in effect managers and when an effect is executed, the effect manager will send a message back to the application when the task is completed. We use the HTTP effect manager to make requests as follows:

```
postModel : Model -> (Result Http.Error String -> msg) -> Cmd msg
postModel model onResponse =
  Http.request
    { method = "POST"
    , headers = []
    , url = "http://127.0.0.1:3000/model"
    , body = Http.jsonBody (newModelEncoder model)
    , expect = Http.expectString onResponse
    , timeout = Nothing
    , tracker = Nothing
    }

evaluateModel : Model -> (Result Http.Error String -> msg) -> Cmd msg
evaluateModel model onResponse =
    Http.post
        { body = Http.jsonBody jsonValue
        , url = "http://127.0.0.1:3000/evaluate"
        , expect = Http.expectString onResponse
        }
```

These functions handle the cases where a model is posted, and where a model is evaluated against a given proposition, respectively.

The JSON analog for Elm are records. We define a JSON serialization function for mapping the Elm model to a Kripke models as follows:

```
toJson : Model -> String
toJson model =
    object
        [ ( "worlds", list (\\( w, _ ) -> int w) model.worlds )
        , ( "valuations", list (\\( w, ps ) -> object [ ( "world", int w ), ( "propositions
            ", list int ps ) ]) model.worlds )
        , ( "relations", list (\\( a, rs ) -> object [ ( "agentName", string a ), ( "
            worldRelations", list (list int) rs ) ]) model.relations )
        ]
        |> encode 4
```

When the action `PostKripkeModel` is called:

```
        PostKripkeModel ->
            (model, postModel model PostedKripkeModel)

        PostedKripkeModel (Ok response) ->
            ( { model | successMsg = "Successfully posted model"}, Cmd.none )

        PostedKripkeModel (Err httpError) ->
            ( { model | error = Just Error.PostError }, Cmd.none )
```

The aforementioned `postModel` is function is called with the model as a parameter, which then parses the model into the correct json format and sends an API request to the server. The response is either and `Ok response` or a `Err httpError`, which the two latter a cases (`PostedKripkeModel`) handle.

3.5 View

We used Elm's Html library to create the styles and output the view. Non-standard styles need to be defined in the style.css, which are "left-column" and "right-column" classes given below. The `onClick` actions of the buttons and the inputs correspond to the actions defined in the `update` part of the application.

```
view model =
    div [ class "container-flex" ]
        [ div [ class "left-column" ]
            [ div [ class "container" ] [ text "Kripke Model Creator" ]
            , input [ class "input", placeholder "Enter world (integer)", onInput
                UpdateWorldInput, value model.worldInput ] []
            , button [ class "button", onClick AddWorld ] [ text "Add World" ]
            , br [] []
            , div [ class "container" ] (List.indexedMap (worldInputView model) model.
                worlds)
            , input [ class "input", placeholder "Enter agent name", onInput
                UpdateAgentInput, value model.agentInput ] []
            , button [ class "button", onClick AddAgent ] [ text "Add Agent" ]
            , br [] []
            , div [ class "container" ] (List.indexedMap agentInputView model.agents)
            , button [ class "button", onClick ToggleReadMe ] [ text "Toggle README/JSON" ]
            , button [ class "button", onClick FetchReadMe ] [ text "Fetch README" ]
            , button [ class "button", onClick PostKripkeModel ] [ text "Post Model" ]
            ]

        , div [ class "right-column" ]
            [ if model.showReadMe then
                div []
                    [ h1 [] [ text "REPORT" ]
                    , div [] [ lazy (Markdown.toHtml []) model.readMeContent ]
                    ]

              else
                div [ class "container" ]
                    [ text "Current JSON Output:", br [] [], text model.jsonOutput ]
            ]
        ]


worldInputView : Model -> Int -> ( Int, List Int ) -> Html Msg
worldInputView model index ( world, propositions ) =
    div [ class "container" ]
        [ text <| "World " ++ String.fromInt world ++ ": "
        , input [ class "input", placeholder "Add proposition (integer)", onInput (
            UpdatePropositionInput index), value (List.Extra.getAt index model.
            propositionInputs |> Maybe.withDefault "") ] []
        , button [ class "button", onClick (AddProposition index) ] [ text "Add Proposition
            " ]
        , text <| " Propositions: " ++ String.join ", " (List.map String.fromInt
            propositions)
        , br [] []
        ]


agentInputView : Int -> String -> Html Msg
agentInputView index agent =
    div [ class "container" ]
        [ text <| "Agent " ++ agent ++ ": "
        , input
            [ class "input"
            , placeholder "Add relation (list of worlds)"
            , onInput (UpdateRelationInput index)

            -- , value (List.Extra.getAt index model.re)
            ]
            []
        , button [ class "button", onClick (AddRelation index) ] [ text "Add Relation" ]
        , br [] []
        ]
```

In Elm, `Browser.element` is a function from the `Browser` module that setups the application. It takes a record with four fields:

```
main : Program () Model Msg
main =
    Browser.element
        { init = init
        , update = update
        , subscriptions = subscriptions
        , view = view
        }
```

The `subscriptions` field is used to handle asynchronous events, such as WebSocket messages or time-based events. These subscriptions are managed by Elm's effect manager. However, for this project, subcriptions are not relevant and is set to `Sub.none`.

3.6 Graph Visualization

For visualizing the Kripke Model, we used various libraries to convert our defined data model to a Graph. First, the elm-community/Graph library [Gra18] is used to convert the Kripke Model to a Graph representation.

```
fromKripkeModelToGraph : KripkeModel -> Graph String String
fromKripkeModelToGraph { relations, evaluations } =
    let
        combineWorlds : String -> List World -> List (Edge String)
        combineWorlds label xs =
            List.concatMap
                (\e1 ->
                    List.map (\e2 -> Edge e1 e2 label) xs
                )
                xs
    in
    fromNodesAndEdges
        (List.map
            ({ propositions, world } -> Node world (String.join "," <| List.map String.
                fromInt propositions))
            evaluations
        )
        (List.concatMap ({ agentName, worldRelations } -> List.concatMap (combineWorlds
            agentName) worldRelations) relations)
```

List comprehension is not possible in Elm, therefore the function `combineWorlds` is done to convert from a Kripke Model to a list of tuples. The edges are derived from the relations and the nodes is related to the evaluations.

After converting the model to a graph, the resulting graph is used to show the model. Hereby, the library elm-visualization was used to show the graph [Ham24]. To show how the graph is built up, the following code results in the visualization. While there are many helper functions to calculate the coordinates on the canvas, the essence is down below.

```
    svg
        [ viewBox -10 -10 150 150
        , Attrs.width <| Percent 100
        , Attrs.height <| Percent 50
        ]
        [ edges
            |> List.map
                (\ ( source, target ) ->
                    line
                        [ strokeWidth 1
                        , stroke <| Paint <| Color.gray
                        , x1 source.x
                        , y1 source.y
                        , x2 target.x
                        , y2 target.y
                        ]
                        []
                )
            |> g [ class [ "links" ] ]
        , nodes
            |> List.map
                (\node ->
                    g [ class [ "node" ] ]
                        [ circle
                            [ r 8
                            , strokeWidth 1
                            , fill (Paint Color.white)
                            , stroke (Paint Color.black)
                            , cx node.x
                            , cy node.y

                            -- The coordinates are initialized and updated by `Force.
                                simulation`
                            -- and `Force.tick`, respectively.
                            -- Add event handler for starting a drag on the node.
                            ]
                            [ title [] [ text node.value ]
                            ]
                        , text_
                            [ -- Align text label at the center of the circle.\
                              dx <| node.x
                            , dy <| node.y + 2
                            , Attrs.alignmentBaseline
                                AlignmentMiddle
                            , Attrs.textAnchor AnchorMiddle

                            -- styling
                            , fontSize <| Px 5
                            , fill (Paint Color.black)
                            ]
                            [ text node.value ]
                        ]
                )
            |> g [ class [ "nodes" ] ]
        ]
```

This element is reactively created each time after the user changes the Graph. Therefore, it is always up to date.

4 Haskell (backend)

The section gives a general overview how we translate the defined JSON Kripke Model data structure that is defined in the introduction into an usable type that can be used by Haskell. Furthermore, we give a general overview how this backend is communicating with the Elm UI and how it sends the information.

4.1 Models & Forms

We have defined the Kripke Model in the following in our Haskell application:

```
type Prop = Int
type Agent = String
type World = Int
type Relations = [(Agent, [[World]])]
type Valuation = [(World, [Prop])]

data Model = Mo
  { worlds :: [World],
    rel :: Relations,
    val :: Valuation
  }
  deriving (Eq, Ord, Show)
```

However, this is different compared to the defined data model which we saw in the data model. The models and the formulas between the front-end and the back-end are defined and communicated as JSON. The front-end sends a request with a model or formula formatted as a JSON to the back-end to be used in evaluating the formulas on the models. The JSON is parsed to convert to the defined model that is usable by Haskell. This can be encoded as a JSON again and sent to back to the front-end to display to the user.

To encode and parse the JSON we used a library called Aeson[O'S12]. There are other libraries available to work with JSON in Haskell but Aeson is easy to use and there are relatively many resources online with information about the libary. This is why we choose to work with Aeson over another library. When encoding a type into JSON, a `ToJSON` instance needs to be defined for the type you are trying to encode. This instance creates an object list with the correct key-value pairs for a given world by mapping over the the relations and valuations and creating JSON objects for each one of them. It uses this to create a list of objects which in our case are the different parts of the model, so the worlds, relations and valuations.

```
instance ToJSON Model where
  toJSON (Mo worlds' rel' val') =
    object
      [ "worlds" .= worlds',
        "relations" .= map (\ (a, ws) -> object ["agentName" .= a, "worldRelations" .= ws])
            rel',
        "valuations" .= map (\ (w, ps) -> object ["world" .= w, "propositions" .= ps]) val'
      ]
```

For decoding you need to define a `parseJSON` function in the `FromJSON` instance for the type you are trying to decode. This `parseJSON` function looks at the type of the objects in the JSON and extracts the objects according to their key and returns a model with the values from the JSON. It parses the object list and each element extracted according to their key.

```
instance FromJSON Model where
  parseJSON =
    withObject
      "Model"
      ( \o ->
          do
            _worlds <- o .: "worlds"
            _relations <- o .: "relations" >>= mapM parseRel
            _valutations <- o .: "valuations" >>= mapM parseVal
            return (Mo _worlds _relations _valutations)
      )
    where
      parseRel = withObject "Relations" (\o' -> (,) <$> (o' .: "agentName") <*> (o' .: "
          worldRelations"))
      parseVal = withObject "Valuations" (\o' -> (,) <$> (o' .: "world") <*> (o' .: "
          propositions"))
```

We do the same for formulas. We use the implementation for formulas and models as in homework 4 so they are defined as follows.

```
data Form = Top | P Prop | Neg Form | Con Form Form | K Agent Form | Ann Form Form
  deriving (Eq, Ord, Show)
```

The `ToJSON` instance matches on the Form type to encode all constructors to JSON and recursively builds a list of objects that match the given formula.

```
instance ToJSON Form where
  toJSON Top = object ["top" .= True]
  toJSON (P p) = object ["p" .= p]
  toJSON (Neg f) = object ["neg" .= toJSON f]
  toJSON (Con f g) = object ["con" .= toJSON [f, g]]
  toJSON (K i f) = object ["knows" .= object ["agent" .= i, "formula" .= f]]
  toJSON (Ann f g) = object ["announce" .= object ["formula" .= f, "result" .= g]]
```

`FromJSON` parses this list of objects recursively to unwrap the objects back to a Form.

```
instance FromJSON Form where
  parseJSON = withObject "form" $ \o -> do
    let parseTop = return Top
        parseP = P <$> o .: "p"
        parseNeg = Neg <$> (o .: "neg" >>= parseJSON)
        parseCon = do
          subformulas <- o .: "con"
          case subformulas of
            [f, g] -> Con <$> parseJSON f <*> parseJSON g
            _ -> fail "Con must contain exactly two subformulas"
        parseK = K <$> o .: "agent" <*> (o .: "formula" >>= parseJSON)
        parseAnn = Ann <$> o .: "announce" <*> (o .: "formula" >>= parseJSON)
```

4.2 Requests

On their own these JSON objects are not really usefull within the Haskell application itself. The purpose of this encoding an decoding is to accommodated the communication. Therefore, we use the web framework scotty [Far24]. This library allowed us to write a declarative web application.

Through this framework, we defined a set of endpoints. An endpoint is the resource locator point where the server provides information about a resource or accepts messages that indicate that data has changed or the user want to change some data. To define, if a resource is going to

change or mutated, a HTTP method is defined. `GET` shows that information is requested from the server and `POST` for muting data [FNR22]. For this application, we have defined 3 endpoints:

- `GET /model`: The current saved model is sent back.

```
        get "/model" $ do
            model <- liftIO getModel
            json model -- Call Model constructor and encode the result as JSON
```

- `POST /model`: The new model is saved on the server.

```
        post "/model" $ do
            model <- jsonData :: ActionM Model -- Decode body of the POST request as
                an Model object
            liftIO $ saveModel model
            json model
```

- `POST /evaluate`: Here a model is evaluated against a current propositions.

```
         post "/evaluate" $ do
            r <- jsonData :: ActionM EvaluationRequest
            let trueWorlds = getFormFromRequest r `trueIn` getModelFromRequest r
            json trueWorlds
```

The JSON that were provided in the previous section, were used to get and mutate information, such as saving the model, and used to return a response front end, such returning the saved model.

5 Haskell vs. Elm

While Haskell and Elm are both functional languages, there are still a few differences between the languages. In this section, we give a general overview, which kind of different were observed and how they affected both programs.

Elm does not have type classes in the way that Haskell does [Cza]. In Elm, ad hoc polymorphism, which is typically handled by type classes in Haskell, is instead managed through record types and extensible records. While Elm's type system is quite powerful and provides features like type inference, custom types, and type aliases, it consciously avoids type classes to maintain simplicity and beginner-friendliness [Cza]. A *type alias* in Elm allows you to create a meaningful name for an existing type.

Custom types in Elm define union types that describe possible variations. The Elm they are annotated with `type` and not to be confused with Haskell's `type`. Custom types are used to define a message `Msg`, modeling situations and error handling. For instance, in Elm, a `Maybe` type is defined as:

```
type Maybe a
  = Just a
  | Nothing
```

At least, list comprehension were not available in Elm. While in essence the functionality is syntax sugar and can also be achieved through the `map` functionality, we noticed that the readability increased when using the list comprehensions in Haskell in comparison to the `map` way in Elm.

6 Conclusion

An overview is given what our experiences were related to the development with the Elm frontend and the Haskell backend. We also discuss the possible features that could be introduced in the project. At least, we sum up what we have done and give a general overview about what is done.

6.1 Experience

Working with Elm was easier than expected. The model-view-update framework is very intuitive to work with for programmers who have worked with front-end frameworks before and the language syntax, being so similar to Haskell, was also a joy to write. One thing that did come back to bite us was the lack of experience writing larger Elm projects. At some point the main file started to exceed 600 lines of code which made it hard to debug and read. A more idiomatic way would be to decouple the application better, however it seems that the internet had a lot more proof-of-concept simple applications rather than examples of larger, production scale applications, which would have given us a clearer picture of how the system architecture should ideally look like.

The functional nature of the language also allowed for a clearer structure in regards to error handling and form validation. The form consisted of a lot of different cases and a lot of different errors were possible in every case. Thus, the pattern matching approach was very intuitive and lead to correct executions earlier than we are used to with imperative languages.

Working with http requests was also easier to work with than expected and we could go as far as saying it is better than most libraries in other languages such as python or JavaScript.

What we think is missing is a better community support and better tooling, documentation is decent, but could also be improved as well. While the core team provides support, it is somewhat sparse, and no significant updates or new features are expected in the foreseeable future.

6.2 Future work

While a interactive model builder was developed, whereby a model could be saved and received, there are still several features that could be introduced per user. First, instead of that a model is persisted in a simple text file, saving the multiple models in a database would be the next feature. The user could save multiple models and modify them through the interface. Also, this gives the possibility that multiple users could use the system and introduces the possibility to test various configurations of Kripke Models. When we would save the models to a database, displaying a list of all saved models would be a fun feature, where you could perhaps add a title to your model and compare models.

For now, the system only allows for checking whether a given a proposition is true in a world, a logical next step in the application would be extending the model with Public Announcements.

6.3 Final

We have developed an application whereby Kripke Models can be made and shown. The Elm frontend made it possible that the input of the user could be parsed and be reactively shown. The Haskell backend saved the model and could process the forms send by the user. Through the libraries and frameworks, provided by the Haskell and Elm community, we could produce this application. Hopefully, this project can be used to show these kind of functional programs can be developed.

References

[CC13] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. *ACM SIGPLAN Notices*, 48(6):411–422, 2013.

[Cza] Evan Czaplicki. The elm architecture.

[Far24] Andrew Farmer. scotty: Haskell web framework, 3 2024.

[FNR22] Roy T. Fielding, Mark Nottingham, and Julian Reschke. HTTP Semantics. RFC 9110, June 2022.

[Gra18] Sebastian Graf. Elm Community Graph, 12 2018.

[Ham24] Jakub Hampl. Elm Visualization, 3 2024.

[O'S12] Bryan O'Sullivan. Aeson: Fast JSON parsing and encoding, 12 2012.