DOKUMENTATION memacs

March 20, 2022

Contents

1 Memacs - Gui Anwendungsprogrammierung **1**
1.1 Aufbau Memacs 1
1.2 Prozess und Problemlösung 2

2 Funktionale Referenzen (Linsen) **2**
2.1 Prozess und Problemlösung 2
2.2 Relevanten Bereiche im src code 2

3 Template Haskell **2**
3.1 Prozess und Problemlösung 2
3.2 Relevanten Bereiche im src code 3

4 Build instructions **3**

1 Memacs - Gui Anwendungsprogrammierung

Memacs ist eine monomer haskell app, für mehr infos über monomer siehe die github seite von monomer, aber hier ist die grob zusammenfassung.
Eine monomer app hat eine start funktion, die

- ein model (type),
- events(type),
- ui (function),
- handleEvent (function) und
- eine config erwartet.

Das model repräsentiert die Daten, die Events können durch user input bzw. Task oder Producer asynchron ausgelöst werden. Die ui function definiert wie die GUI aussieht. Der Eventhandler, spezifiziert was als folge auf welches Event passieren soll. Das kann entweder sein das die Werte im Model angepasst werden, dann würde die UI neu gerendert. Oder es werden asynchrone temporäre oder permanent laufende zusatz threads gestartet, die IO operationen ausführen können. Die Config ist nötig um dem Programm z.B. zugriff auf Schriftarten zu geben - das merkt man ganz schnell wenn man stack run in der src direcotry ausführt und die monomer app leer ist - weil keine Schriftarten existieren.

1.1 Aufbau Memacs

Bei Memacs habe ich den Ansatz verfolgt, den code in mehrere logische einheiten zu unterteilen.
Es gibt die UI.* module, das sind alle ui functions die, gebündelt in MainUI.hs zusammengefasst werden. Analog gilt das gleiche für Events und MainEvents.hs. Zusätzlich gibt es Types.* darin sind allgemein genutzte typen, Calendar und Task definiert. Und dann gibt es noch Utils für alle möglichen helper functions. Finale haben wir noch das submodule Parser - darin ist der ParserCombinator Parser definiert und meine QuasiQuotes und das submodule Widget - worin sich das tetrismodule befindet - meine eigene implementation eines Monomer widgets.
Memacs hat eine Sidebar, die einen von 4 Bereichen per Klick auf den Icon darstellen kann. File/Write, Calendar/-Tasks, Tetris und Settings. Für die Bereiche sind verschiedene Tätigkeiten in das Projekt eingeflossen.

- Settingsbereich Parsen von Json
- Calendar Definieren von Calendar datatype, und Task datatyp, einlesen von xml und schreiben von xml

- Tetris spiel
- commandoZeile (Alt-x, Esc, Enter)
- Filebrowser und Texteditor Es gibt einen Filetree aus dem heraus dateien geoeffnet werden koennen, direcotries gewechselt werden koennen

1.2 Prozess und Problemlösung

Basierend auf der hackage dokumentation von Monomer bin ich schnell voran gekommen viel code zu schreiben, der innerhalb von kürzester Zeit kaum noch zu maintainen war - und refactoring hat das ganz unkompilierbar gemacht - also war ich gezwungen das projekt nochmal anzufangen. Die Erfahrung hat mich leider nicht gelehrt - und das gleich ist noch mal passiert. Der Hauptgrund wieso in der aktuellen form so viele so kleine Module existieren, die übersichtlichkeit gegenüber weniger sehr großen. Zu mindest für mich.
Neben dem bloßen bauen von UI elementen habe ich basierend auf den tutorial in der Monomer mein eigenes widget - TetrisWidget angelegt
Ursprünglich war ich dabei das TextArea widget anzupassen, damit man syntax highlighting und line numbers hat usw. aber zeitlich und vom Verständnis her bin ich da an meine grenzen gestoßen. Ich habe mich darauf beschränkt, syntaxhighlighting für Haskell files hinzuzufügen. Meine Änderungen sind in Wiget/My* enthalten.
Eine Funktion die ich in Memacs eingebaut habe, und die wohl etwas zu wenig effektiv genutzt wird, ist ein minimales log system, das erlaubt bei Events eine Nachricht zu loggen. Die Idee war grob die Ursache von Fehlern zu finden.
Am Effektivsten um Problemen vorzubeugen und diese zu lösen, ist für mich nun die positive erfahrung mit der Aufteilung von meinem Code in kleinere Module, die wesentlich übersichtlicher sind und weniger unklarheit zulassen welche Änderung und wo den fehlern introducted hat.

2 Funktionale Referenzen (Linsen)

2.1 Prozess und Problemlösung

Begründet durch die Gui library monomer werden linsen praktisch überall verwendet. Zum einen weil permanent Werte benötigt werden bzw. verändert werden als reaktion auf ein Event in der Gui. Und zum anderen weil die Widgets in Monomer i.d.R. eine Linse als input annehmen. Ein Textfield bekommt eine Linse zu Text im Model und wenn der Text im TextField angepasst wird, wird der Wert im Model durch die Linse angepasst.

2.2 Relevanten Bereiche im src code

- Erzeugung von Linsen ist vorallem in Model.hs und die submodules in Model/*
- Die Verwendung von Linsen findet statt in
  - UI/* und Widget/* um Daten aus dem Model zu beziehen
  - Events/*

3 Template Haskell

3.1 Prozess und Problemlösung

Allgemein könnte der Prozess mit sehr viel Frustation und die Lösung mit Dankbarkeit umrissen werden. Nach dem die Vorlesung schon ein bisschen her war, habe ich erstmal das skript nochmal gelesen. Das hat mir leider nicht gereicht um wirklich weiter zu kommen. Dann habe ich mich ins Internet gestürzt und ein tutorial, blockpost youtube video nach dem nächsten gesucht um zu raffen wie das mit quasiquotern funktionieren kann. Ich denke ich hab bestimmt 2-3 Tage gebraucht um irgendwas sinnvolles im zusammenhang von quasiquoter hinzubekommen.
Ich wollte dann quasiquoter benutzen um Commandos, die ein user in memacs in die eingabezeile eintippt per quasiquoter, auszuführen, über mehrere videos bin ich dann auf das Thema Parser combinators gekommen und dann gabe es auf einmal auch eine Verbindung zu quasiquotern. Und basierend von einem Blockpost (Im source code ist die quelle zitiert) habe ich dann einen Parser und darauf basierend meinen Typ Open implementiert. Und zu Open habe ich dann den quasiquoter openQQ geschrieben. Damit kann man im source code einen Quasiquoter benutzen um einen string zum Open typ parsen ... der Variablen hat, die im quasiquoter von der umgebung eingesammelt werden ...
Dann ist mir aber aufgefallen, dass man einen quasiquoter schreiben kann, der direkt einen eingabe string parsed - und daraus ist dann der banale quasiquoter settingsQQ geworden, der es moeglich macht basierend auf dem Parser json daten direkt in einen JValuewert zu parsen. Und dass ist in memacs jetzt die Implementation der defaultsettings.

3.2 Relevanten Bereiche im src code

- Utils/Style.hs
- Parser/Parser.hs
- Parser/Json.hs
- Parser/QQ.hs
- Parser/Open.hs

4 Build instructions

Informationen zu den allgemeinen dependencies von monomer kann die github seite `https://github.com/fjvallarino/monomer/blob/main/docs/tutorials/00-setup.md` gelesen werden.
Memacs benutzt `https://github.com/fjvallarino/monomer-starter` als Basis (benutzt aber keinen Haskell code aus der Basis)