Christian-Albrechts-Universität zu Kiel
Christian-Albrechts-Platz 4
24118 Kiel

Seminararbeit für den Masterstudiengang Informatik

Typsichere Implementierung von Netzwerkprotokollen mittels Session Types und Scribble in der funktionalen Programmiersprache PureScript

Hendrik Oenings

23. April 2024

Technische Fakultät
Institut für Informatik

Inhaltsverzeichnis

1 Einleitung **3**

2 Grundlagen der Sprache PureScript **3**

2.1 Einbindung in praktische Anwendungen 4

2.1.1 Einbindung von JavaScript-Funktionen in PureScript-Module . . . 4

2.1.2 Einbindung von PureScript-Funktionen in JavaScript-Module . . . 6

2.2 Typsystem 6

2.2.1 Zeilenpolymorphismus (Row polymorphism) 7

2.2.2 Multiparameter-Typklassen 7

3 Session Types **8**

3.1 Globale Protokollspezifikation mit Scribble 8

3.2 Lokale Protokollspezifikation mit Scribble 11

3.2.1 Kodierung endlicher Automaten im PureScript-Typsystem 11

3.2.2 Implementierung eines Kommunikationsablaufs 13

4 Fazit **15**

1 Einleitung

PureScript ist eine funktionale Programmiersprache, welche an Haskell angelehnt ist und zu JavaScript kompiliert. JavaScript hat bereits viele funktionale Programmierelemente integriert, verfügt jedoch unter anderem über keine statische Typisierung. [1, § 1.1 f.]

Diese Seminararbeit beschäftigt sich schwerpunktmäßig mit der Realisierung von Session Types in PureScript. Session Types definieren hierbei, welche Art von Nachrichten wann in der Kommunikation erwartet beziehungsweise zulässig sind, und stellen somit eine formale Spezifikation des Kommunikationsprotokolls dar. In PureScript kann die Einhaltung dieser Spezifikation über das Typsystem erfolgen.

Hierfür wird in Abschnitt 2 zunächst eine Einführung in die Sprache PureScript gegeben, bevor sich Abschnitt 3 auf Seite 8 mit Session Types in PureScript über das Scribble-Framework befasst. Abschließend wird in Abschnitt 4 auf Seite 15 ein Fazit gezogen.

2 Grundlagen der Sprache PureScript

PureScript[1] ist eine stark typisierte rein funktionale Programmiersprache, welche an Haskell orientiert ist und in JavaScript übersetzt werden kann.

Der grundlegende Aufbau eines PureScript-Programms ist syntaktisch vergleichbar mit dem grundlegenden Aufbau eines Haskell-Programms, wobei in PureScript die Auswertung strikt erfolgt:

```
module Basic01 where
import Prelude

square :: Int -> Int
square x = x * x
```

Im Gegensatz zu Haskell werden jedoch die grundlegenden `Prelude`-Funktionalitäten nicht automatisch importiert, sondern müssen explizit importiert werden.

In diesem Abschnitt betrachten wir die Einbindung von PureScript in praktische Anwendungen (Abschnitt 2.1 auf der nächsten Seite) sowie das Typsystem von PureScript (Abschnitt 2.2 auf Seite 6).

[1] https://www.purescript.org/

2.1 Einbindung in praktische Anwendungen

PureScript-Code kann in Projekten gemeinsam mit JavaScript-Code[2] genutzt werden. Interaktion zwischen PureScript-Modulen und JavaScript-Modulen kann dabei in beide Richtungen erfolgen, das heißt, es können sowohl aus JavaScript-Modulen PureScript-Funktionen aufgerufen werden als auch als PureScript-Modulen JavaScript-Funktionen genutzt werden. Bei der Einbindung von JavaScript-Modulen ist darauf zu achten, diese unter Berücksichtigung der starken Typisierung von PureScript zu implementieren. [1, § 10.10]

2.1.1 Einbindung von JavaScript-Funktionen in PureScript-Module

Um mit PureScript auch mit der Welt interagieren zu können (bei Nutzung im Browser beispielsweise mit dem DOM), ist es notwendig, eine Schnittstelle zu schaffen. Hierfür können JavaScript-Funktionen unter Angabe einer Typsignatur in PureScript-Module eingebunden werden.

Betrachten wir dafür ein einfaches Beispiel. Gegeben sei eine Webseite, welche in einem `div`-Container einen Text enthält, den wir auf der Konsole ausgeben wollen:

```
<!doctype html>
<html>
        <body>
                <div id="page">hello, world!</div>

...

        </body>
</html>
```

Um auf das DOM zugreifen zu können, benötigen wir nun zunächst eine (native) JavaScript-Funktion, welche wir aus unserem PureScript-Modul nutzen können:[3]

```
export const getPageContent = () =>
↪  document.getElementById('page')?.textContent || ''
```

Diese Funktion findet den Container mit der ID `page` und liest das Attribut `textContent` aus, sofern der Container existiert. Sofern der Container nicht existiert oder das Attribut nicht gesetzt ist, wird stattdessen ein Leerstring zurückgegeben. In jedem Fall erhalten wir also einen `String` aus dieser Funktion.

Dies ermöglicht es uns, eine Typsignatur für diese Funktion anzugeben.

[2] Natürlich ist hier auch eine gemeinsame Nutzung mit anderen Sprachen denkbar, welche in JavaScript übersetzt werden, möglich. Insbesondere kommen hier beispielsweise TypeScript oder JSX in Betracht.

[3] Üblicherweise wird man hierfür bereits existierende Bibliotheken nutzen, welche den Zugriff auf das DOM allgemein bereitstellen, beispielsweise `purescript-web-dom` und `purescript-web-html`.

```
foreign import getPageContent :: Effect String
```

Mit dem Schlüsselwort `foreign import` geben wir an, dass wir eine native JavaScript-Funktion nutzen wollen. Diese Funktion wird in der Datei mit demselben Dateinamen gesucht, das heißt, für unsere PureScript-Datei `Embed01.purs` wird die Funktion `getPageContent` aus der JavaScript-Datei `Embed01.js` eingebunden.

Mit dieser Funktion sowie der Funktion `log` aus dem Modul `Effect.Console` können wir nun die gewünschte Funktionalität umsetzen:

```
module Embed01 (main) where
import Prelude
import Effect (Effect)
import Effect.Console (log)

foreign import getPageContent :: Effect String

main :: Effect Unit
main = do
  txt <- getPageContent
  log txt
```

`Effect` erfüllt hierbei denselben Zweck wie die `IO`-Monade aus Haskell, nämlich die Abbildung von Seiteneffekten. Im Kontext der Webfrontendentwicklung sind Seiteneffekte insbesondere Zugriffe auf das DOM (siehe Beispiel), Netzwerkzugriffe oder Zugriffe auf einen globalen Zustand (beispielsweise `window`, `localStorage`, IndexedDB oder Cookies).

Abschließend müssen wir noch unseren Code in unsere HTML-Datei einbinden. Es gibt verschiedene Möglichkeiten, PureScript-Code zu übersetzen und anschließend einzubinden.

Mit dem Kommando `spago build` wird das Projekt in JavaScript-Dateien übersetzt, wobei die Modulstruktur beibehalten wird (ein Ordner pro Modul). Die Einbindung der anderen PureScript-Module erfolgt dabei über JavaScript-Module (`import`/`export`) durch den Webbrowser. Diese Variante ist zwar langsam (da unter Umständen viele Dateien geladen werden müssen), veranschaulicht aber die Übersetzung von PureScript zu JavaScript gut.

```
                <script type="module">
import { main } from '../output/Embed01/index.js'
main()
                </script>
```

Das Bundling der einzelnen Module zu einer einzelnen JavaScript-Datei ist mittels `spago bundle-app` möglich. Alternativ kann hierfür auch ein klassischer JavaScript-Bundler (beispielsweise Webpack) genutzt werden. Bundling verhindert, dass die Ladezeiten des Frontends durch zu viele Einzelanfragen zu lang werden.

Als Ergebnis erhalten wir in beiden Fällen in der Konsole das erwartete `hello, world!`.

2.1.2 Einbindung von PureScript-Funktionen in JavaScript-Module

Auch eine Integration von PureScript-Funktionen in JavaScript-Module ist möglich. Hierfür können Funktionen aus einem PureScript-Modul in einer JavaScript-Datei eingebunden werden.

Betrachten wir dafür erneut ein Beispiel.

In PureScript definieren wir eine Funktion `square`, welche eine Ganzzahl quadriert:

```
square :: Int -> Int
square x = x * x
```

Dann können wir diese Funktion im JavaScript-Code nutzen, indem sie als normales JavaScript-Modul importiert wird:

```
import { square } from './index.js'

export function main() {
        console.log(square(5))
}
```

Die aus dem JavaScript-Code exportierten Funktionen sind jedoch nach dem Bau des Projekts standardmäßig nicht sichtbar. Um die Funktionen aus dem JavaScript-Code nutzen zu können, müssen wir diese in PureScript importieren und re-exportieren:

```
module Embed02 (main, square) where

...

foreign import main :: Effect Unit
```

2.2 Typsystem

Im Folgenden wollen wir zunächst das Konzept des Zeilenpolymorphismus betrachten und anschließend Multiparameter-Typklassen näher beleuchten. In diesem Zuge führen wir außerdem funktionale Abhängigkeiten in Multiparameter-Typklassen ein, welche eine wesentliche Grundlage für Abschnitt 3 auf Seite 8 bildet.

2.2.1 Zeilenpolymorphismus (Row polymorphism)

Mit Zeilenpolymorphismus werden polymorphe Programme über Verbunddatentypen ermöglicht. [1, § 5.7] Zeilenpolymorphismus ermöglicht es, Funktionen auf unterschiedlichen, aber strukturell ähnlichen Datentypen zu definieren. Bei JavaScript respektive PureScript wird JSON als Verbunddatentyp genutzt.

Ein Verbunddatentyp kann in PureScript wie folgt repräsentiert werden:

```
type Person = { firstName :: String
              , lastName  :: String
              }
type AcademicPerson = { firstName     :: String
                      , lastName      :: String
                      , academicTitle :: String
                      }
```

Zeilenpolymorphismus ermöglicht es nun, Funktionen auf beliebigen Verbunddatentypen zu definieren, welche bestimmte Eigenschaften erfüllen. Beispielsweise kann eine Funktion, welche nur das Feld `firstName` eines Verbunddatentyps nutzt, auf jedem Verbunddatentyp arbeiten, welcher dieses Feld enthält:

```
greet :: forall (r :: Row Type). { firstName :: String | r } -> String
greet { firstName: name } = "Hello " <> name <> "!"
```

Diese Funktion kann unter anderem mit den oben definierten Datentypen `Person` und `AcademicPerson` aufgerufen werden.

2.2.2 Multiparameter-Typklassen

Typklassen in PureScript können nicht nur einen Typen als Argument erhalten, sondern beliebig viele. [1, § 6.8] Die Syntax hierfür ist vergleichbar mit den klassischen Ein-Typ-Typklassen:

```
class Serializable source target where
    serialize :: source -> target

instance Serializable Int String where
    serialize x = show x
```

Nun ist allerdings der Aufruf von `serialize` nicht mehr eindeutig. Damit `serialize` genutzt werden kann, muss der zweite Typ ebenfalls fixiert werden:

```
> serialize 42
Error found: [...] The instance head contains unknown type variables. [...]
> serialize 42 :: String
"42"
```

Funktionale Abhängigkeiten

Falls allerdings ein Typ einer Multiparameter-Typklasse immer einen anderen Typen impliziert, können wir diese Implikation als funktionale Abhängigkeit implementieren. In unserem Beispiel bedeutet das, dass ein bestimmter Datentyp `source` immer in denselben bestimmten Datentypen `target` übersetzt wird:

```
class UniqueSerializable source target | source -> target where
    uniqueSerialize :: source -> target

instance UniqueSerializable Int String where
    uniqueSerialize x = show x
```

Danach können wir zwar keine anderen Instanzen `instance UniqueSerializable Int ...` mehr definieren, allerdings ist dafür der Aufruf von `uniqueSerialize` direkt eindeutig:

```
> uniqueSerialize 42
"42"
```

3 Session Types

Das Konzept der Session Types dient dazu, bei Programmen, welche Netzwerkkommunikation betreiben, die Einhaltung von Protokollen, welche ebendiese Netzwerkkommunikation beschreiben, über das Typsystem zu überprüfen. [2, § 1]

Hierfür bedarf es zunächst einer standardisierten Darstellung von Netzwerkprotokollen. Die standardisierte Darstellung mit Scribble betrachten wir in Abschnitt 3.1. Abschnitt 3.2 auf Seite 11 befasst sich dann mit der konkreten Umsetzung der Scribble-Spezifikation in PureScript.

3.1 Globale Protokollspezifikation mit Scribble

Das Scribble-Projekt ist ein Framework zur Spezifikation von Protokollen. [3, § 1] Es wird hierbei die Kommunikation zwischen allen beteiligten Rollen beschrieben (globale Protokollspezifikation). [3, § 3.1]

Betrachten wir das Protokoll „Schiffe versenken“ als Fallstudie. Wir definieren das Protokoll über die Rollen und Zustände der Teilnehmer sowie über die Nachrichtenflüsse (beispielsweise „Angriffe“). Diese detaillierte Protokollspezifikation ermöglicht es, die Interaktionen innerhalb des Spiels genau zu kontrollieren und zu verifizieren, dass alle Nachrichten entsprechend den Spielregeln ausgetauscht werden.

Ein Protokoll ist dabei zunächst über die beteiligten Rollen definiert:

```
explicit global protocol BattleShips(role P1, role P2, role GameServer) {
...
}
```

Der Ablauf des Protokolls kann dabei verschiedene Arten von Anweisungen enthalten:

- **Verbindungsaufbau mit assoziierten Daten:**

  `Init(Config) connect P1 to GameServer;`

  Dabei sind die einzelnen Bestandteile der Anweisung wie folgt zu interpretieren:

  - `Init`: Name der Aktion
  - `Config`: Typ der gesendeten Daten; der Typ wird in PureScript spezifiziert (siehe unten) und kann daher so in Scribble genutzt werden.
  - `connect A to B`: Schlüsselwort für Verbindungsaufbau; A baut die Verbindung auf, B wartet auf den Verbindungsaufbau.
  - `P1, GameServer`: Rollen der Kommunikationsbeteiligten (siehe oben)

- **Verbindungsabbau:**

  `disconnect P1 and GameServer;`

  Hierbei ist `disconnect ... and ...` das Schlüsselwort; `P1` und `GameServer` bezeichnen wie beim Verbindungsaufbau die Rollen, deren Verbindung betrachtet wird.

- **Aufruf eines anderen Protokolls unter Angabe der Rollen:**

  `do Game(P1, GameServer, P2);`

  Protokollspezifikationen können modular gestaltet werden. Hierfür kann ein Bestandteil ausgelagert werden, analog zu Funktionen in Programmiersprachen. Dies ermöglicht es auch, Abläufe rekursiv zu beschreiben und so Wiederholungen darzustellen. `do` fungiert hierbei als Schlüsselwort; in Klammern werden die Rollen als Parameter übergeben.

- **Nachricht von Ausgangsrolle zu Zielrolle:**

  `Attack(Location) from Atk to Svr;`

  Der Versand von Nachrichten funktioniert ähnlich zum Verbindungsaufbau. Auch hier wird zunächst die Art der Nachricht spezifiziert (`Attack`) und anschließend der Typ der übergebenen Daten (`Location`). Anschließend wird mit dem Schlüsselwort `from SENDER to EMPFÄNGER` beschrieben, welche beiden Rollen in welche Richtung kommunizieren. Eine bereits aufgebaute Verbindung ist erforderlich.

- **Verschiedene Optionen einer Rolle über den weiteren Verlauf:**

```
choice at Svr { // Svr knows if it's a hit
...
} or {
...
}
```

Mit diesem Block wird es ermöglicht, dass eine Rolle eine Entscheidung über den weiteren Protokollverlauf treffen kann.[4] Im vorliegenden Beispiel kann der Server entscheiden beziehungsweise feststellen, ob ein Treffer vorliegt oder nicht.

Die in der Protokollspezifikation genutzten Datenstrukturen werden nicht über Scribble, sondern in der Implementierungssprache spezifiziert und in Scribble importiert.

Spezifikation eines Datentyps in PureScript:

```
-- An initial game configuration
newtype Config = Config Int
```

Import der Datentypdefinition in Scribble:

```
type <purescript> "Config" from "Game.BattleShips.Types" as Config;
```

In Scribble ist also nur bekannt, dass Daten eines Typs übertragen werden. Die konkrete Beschaffenheit wird nicht näher betrachtet.

Die gesamte Spezifikation des Protokolls ergibt sich damit wie folgt:

```
type <purescript> "Int" from "Prim" as int;
type <purescript> "Config" from "Game.BattleShips.Types" as Config;
type <purescript> "Location" from "Game.BattleShips.Types" as Location;

explicit global protocol BattleShips(role P1, role P2, role GameServer) {
    Init(Config) connect P1 to GameServer;
    Init(Config) connect P2 to GameServer;
    do Game(P1, GameServer, P2);
    disconnect P1 and GameServer;
    disconnect P2 and GameServer;
}

global protocol Game(role Atk, role Svr, role Def) {
    Attack(Location) from Atk to Svr;
    choice at Svr { // Svr knows if it's a hit
        Hit(Location) from Svr to Atk;
        Hit(Location) from Svr to Def;
```

[4] Diese Entscheidung ist also durch eine entsprechende Implementierung zu treffen.

```
        do Game(Def, Svr, Atk);
    } or {
        Miss(Location) from Svr to Atk;
        Miss(Location) from Svr to Def;
        do Game(Def, Svr, Atk);
    } or {
        Winner() from Svr to Atk;
        Loser() from Svr to Def;
    }
}
```

3.2 Lokale Protokollspezifikation mit Scribble

Aus der globalen Protokollspezifikation werden für die beteiligten Rollen lokale Protokollspezifikationen automatisiert abgeleitet. Der große Vorteil der globalen Protokollspezifikation besteht also darin, dass der Ablauf des Protokolls nur einmal (konsistent) festgelegt werden muss und die lokalen Protokollspezifikationen so erstellt werden, dass sie mit der globalen Spezifikation (und damit auch zueinander) konsistent sind.

Die jeweiligen Endpunkte können dabei als endliche Automaten dargestellt werden.

3.2.1 Kodierung endlicher Automaten im PureScript-Typsystem

Die zentrale Idee in [2] besteht darin, diese endlichen Automaten über das PureScript-Typsystem abzubilden. Hierfür werden Multiparameter-Typklassen mit funktionalen Abhängigkeiten genutzt.

Typklassen

Die Typklassen entsprechen dabei den innerhalb der Kommunikation möglichen Ereignissen:

- `Connect`: Hiermit wird der (ausgehende) Verbindungsaufbau von der Rolle `r` zur Rolle `r'` abgebildet. Zusätzlich zu den Rollen wird der Anfangszustand `s` und der Zielzustand `t` der Operation erwartet. Der Anfangszustand bestimmt hierbei eindeutig die beiden Rollen und den Zielzustand (*funktionale Abhängigkeit*).

  ```
  class Connect (r :: Role) (r' :: Role) s t | s -> r r' t
  ```

- `Disconnect`: Hiermit wird entsprechend der Verbindungsabbau zwischen zwei Rollen abgebildet.

  ```
  class Disconnect (r :: Role) (r' :: Role) s t | s -> r r' t
  ```

- `Accept`: Hiermit wird ein eingehender Verbindungsaufbau abgebildet.

```
class Accept (r :: Role) (r' :: Role) s t | s -> r r' t
```

- `Send`: Hiermit wird der Versand einer Nachricht vom Typ `a` durch die Rolle `r` abgebildet. Der Anfangszustand (`s`) bestimmt hierbei erneut die Rolle (`r`) und den Nachrichtentyp (`a`) sowie den Zielzustand (`t`).

  ```
  class Send (r :: Role) s t a | a -> t, s -> r a
  ```

- `Receive`: Hiermit wird der Empfang einer Nachricht vom Typ `a` durch die Rolle `r` abgebildet.

  ```
  class Receive (r :: Role) s t a | a -> t, s -> r a
  ```

- `Select`: Hiermit kann der Operator `choice` aus der Scribble-Spezifikation umgesetzt werden; das heißt, die Rolle `r` kann aus einem Anfangszustand `s` in einen von mehreren Zuständen `ts` wechseln.

  ```
  class Select (r :: Role) s (ts :: RowList) | s -> ts r
  ```

  Der Typ `RowList` wird genutzt, um eine Liste von Typen darzustellen. Da die Zustände später als Typen modelliert werden, können wir mit `RowList` eine Liste von Zuständen abbilden.

- `Branch`: Hiermit wird der Operator `choice` ebenfalls umgesetzt, jedoch von der passiven Seite; das heißt, die (entfernte) Rolle `r'` wählt aus, in welchem der Zustände `ts` das Protokoll fortgesetzt wird.

  ```
  class Branch (r :: Role) (r' :: Role) s (ts :: RowList) | s -> ts r r'
  ```

Zusätzlich werden für jede Rolle der Start- und Endzustand über eine Typklasse abgebildet:

```
class Initial (r :: Role) a | r -> a
class Terminal (r :: Role) a | r -> a
```

[2, § 3.1]

Instanzen

Auf Basis der globalen Scribble-Spezifikation werden nun Instanzen ebendieser Typklassen definiert.[5] Diese Instanzen bilden ab, welche Zustandsübergänge für die Rolle möglich sind.

Wir betrachten im Folgenden beispielhaft einen Auszug des abgeleiteten Schemas für den Spieler P2. Der Startzustand ist als `S34` und der Zielzustand als `S35` festgelegt:[6]

[5] Da die Typklassen keine Funktionen definieren, beschränkt sich die Definition der Instanz auf die Angabe der konkreten Typen.

[6] Die Namen der Instanzen sind hierbei nur informativ und haben keine nähere Semantik; in aktuellen PureScript-Versionen sind sie optional.

```
instance initialP2 :: Initial P2 S34
instance terminalP2 :: Terminal P2 S35
```

Ausgehend vom Startzustand (`S34`) wird nun der Verbindungsaufbau zum Server als Schritt erlaubt:

```
instance connectS34 :: Connect P2 GameServer S34 S34Connected
```

Analog werden diese Instanzen für alle Zustandsübergänge des endlichen Automaten generiert.

[2, § 3.2.1]

3.2.2 Implementierung eines Kommunikationsablaufs

Im Folgenden wird betrachtet, wie diese Instanzen genutzt werden können, um die Einhaltung des Protokolls tatsächlich sicherzustellen. Die konkrete Implementierung des Protokolls kann hierbei keine neuen Instanzen definieren, sondern nur die Session Runtime nutzen.

Wesentlich zum weiteren Verständnis ist zunächst der neue Datentyp `Session`, welcher einen Kommunikationskanal aus einem Eingangszustand in einen Ausgangszustand überführt und dabei (gegebenenfalls[7]) ein Ergebnis zurückliefert.

```
newtype Session m c i t a = Session ((Channels c i) -> m (Tuple
↪  (Channels c t) a))
```

Dieser Datentyp kann nur durch die vordefinierten Funktionen der Session Runtime erzeugt werden.

Beispielsweise steht zum Senden der Daten die Funktion `send` zur Verfügung:

```
send :: forall r rn c a s t m p.
     Send r s t a
  => RoleName r rn
  => IsSymbol rn
  => Transport c p
  => EncodeJson a
  => MonadAff m
  => a -> Session m c s t Unit
```

Wir betrachten im Folgenden nur die beiden Constraints `Send` und `EncodeJson` näher:

- `EncodeJson a` stellt sicher, dass die zu sendenden Daten JSON-kodierbar sind, also eine Möglichkeit besteht, diese sinnvoll zu übertragen.

[7] Der Typ `a` kann `Unit` sein, es muss also kein Ergebnis existieren.

- `Send r s t a` fordert, dass eine `Send`-Instanz für die Rolle `r` vom Zustand `s` in den Zustand `t` mit dem Nachrichtentyp `a` existiert.

Wenn alle Constraints erfüllt sind, können wir der Funktion `send` Daten übergeben und erhalten eine `Session`.

Analog existieren Funktionen für die weiteren möglichen Operationen (beispielsweise `connect`, `receive` und `choice`).

Die `Session`s lassen sich über eine monadische `bind`-Operation miteinander verbinden:

```
bind :: Monad m => Session m c i s a -> (a -> Session m c s t b) ->
↪  Session m c i t b
```

Um jedoch die Funktionen, welche in den jeweiligen `Session`s gekapselt sind, tatsächlich nutzen zu können, muss die Funktion `session` genutzt werden:

```
session :: forall r c p s t m a.
     Transport c p
  => Initial r s
  => Terminal r t
  => MonadAff m
  => Proxy c
  -> Role r
  -> Session m c s t a
  -> m a
```

Hier sind insbesondere die Constraints `Initial` und `Terminal` interessant:

- `Initial r s` stellt sicher, dass der Zustand `s` der Startzustand für die Rolle `r` ist.
- `Terminal r t` stellt analog sicher, dass der Zustand `t` der Endzustand für die Rolle `r` ist.

Wir können diese Funktion also nur mit einer `Session` aufrufen, welche im Startzustand der Rolle startet und im Zielzustand der Rolle endet und erhalten dann das „Gesamtergebnis“ der Kommunikation als Rückgabewert.

Damit wird – in Verbindung mit den Constraints bei der Erzeugung der Sessions – sichergestellt, dass nur Verbindungen aufgebaut werden können, welche vollständig und korrekt vom Startzustand bis zum Endzustand implementiert sind:

- `Session`s können nur durch die vordefinierten Operationen (beispielsweise `connect`, `send`) erzeugt beziehungsweise verbunden werden. Da diese Funktionen ihrerseits prüfen, dass die Session ein gültiger Teildurchlauf des endlichen Automaten ist, können nur protokollkonforme `Session`s erzeugt werden.

- Beim Aufruf von `session` wird über die Constraints gefordert, dass wir einen gültigen Start- und Zielzustand haben. Das heißt, die Session startet in einem Startzustand, wird über die zugelassenen Operationen fortgesetzt und endet in einem Endzustand.
- Die Funktion `session` ist die einzige Funktion, welche eine (`Effect`-)Monade zurückliefert, welche wir auswerten können. Die übrigen Funktionen (`send`, `bind` et cetera) liefern „nur“ eine `Session`, welche wir zwar weiter kombinieren, jedoch nicht auswerten können.

[2, § 3.2.2]

Beispiel „Schiffe versenken“

Betrachten wir nun erneut das Beispiel „Schiffe versenken“. Die Implementierung für die Rolle `P1` in unserem Beispiel kann wie folgt aussehen:

```
battleShipsWidgetP1 port = session
  (Proxy :: Proxy WebSocket)
  (Role :: Role BS.P1) $ do
    connect (Role :: Role BS.GameServer) (URL $ "ws://localhost:" <>
    ↪  show port)
    config <- lift setupGameWidget
    send $ BS.Init config
    let pb = BS.mkBoard config
    let ob = mempty :: BS.Board BS.OpponentTile
    attack pb ob
```

An `session` wird der Transportkanal (hier Websocket), die Rolle (hier P1) und die Session übergeben. Die Session wird im Beispiel über die `do`-Notation erzeugt:

1. Zunächst wird eine Verbindung mit der Rolle `GameServer` aufgebaut. Anschließend wird die initiale Konfiguration gesendet. (In Scribble-Spezifikation: `Init(Config) connect P1 to GameServer`)
2. Anschließend wird die `Session` aufgerufen, welche von der rekursiven Funktion `attack` zurückgegeben wird.

4 Fazit

Im Rahmen dieser Arbeit haben wir die stark typisierte rein funktionale Programmiersprache PureScript kennengelernt, welche sich im JavaScript-Umfeld bewegt. Das Typsystem der Sprache ähnelt Haskell, bietet jedoch mit Zeilenpolymorphismus auch weitergehende Konzepte. Außerdem werden Multiparameter-Typklassen und funktionale Abhängigkeiten unterstützt, welche die Grundlage der Session Types bilden.

Session Types ermöglichen in PureScript die Prüfung der Konformität einer Anwendung zu einem definierten Kommunikationsprotokoll. Zur Spezifikation von Kommunikationsprotokollen wurde dabei kurz in das Scribble-Framework eingeführt. Abschließend wurden die Übersetzung der globalen Scribble-Spezifikation in Multiparameter-Typklassen-Instanzen sowie die Sicherstellung der Konformität über das Typsystem erklärt.

Literatur

[1] Phil Freeman. „PureScript by Example“. In: *Leanpub. Retrieved* 23 (2017). URL: http://samples.leanpub.com.s3.amazonaws.com/purescript-sample.pdf (besucht am 02. 12. 2023).

[2] Jonathan King, Nicholas Ng und Nobuko Yoshida. „Multiparty session type-safe web development with static linearity“. In: *arXiv preprint arXiv:1904.01287* (2019). URL: https://arxiv.org/pdf/1904.01287 (besucht am 20. 01. 2024).

[3] Nobuko Yoshida u. a. „The Scribble protocol language“. In: *Trustworthy Global Computing: 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers 8*. Springer. 2014, S. 22–41. URL: http://mrg.doc.ic.ac.uk/publications/the-scribble-protocol-language/invited.pdf (besucht am 02. 02. 2024).