INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen

Universitätsstraße 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Portierung eines Haskell-Parsers nach Frege

Markus Brenneis

Bachelorarbeit

Beginn der Arbeit:	27. Juni 2016
Abgabe der Arbeit:	27. September 2016
Gutachter:	Prof. Dr. Michael Leuschel
	Prof. Dr. Jörg Rothe

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 27. September 2016

Markus Brenneis

Zusammenfassung

Mit der Sprache CSP$_M$ können in maschinenlesbarer Form nebenläufige, miteinander kommunizierende, sequentielle Prozesse beschrieben werden. Der am Lehrstuhl für Softwaretechnik und Programmiersprachen der Heinrich-Heine-Universität entwickelte ProB Animator and Model Checker kann eingesetzt werden, um z. B. sicherheitskritische Eigenschaften solcher Spezifikationen zu untersuchen.

ProB nutzt dafür Prolog, sodass die CSP$_M$-Eingabedatei zunächst nach Prolog übersetzt werden muss, was bisher ein externer, in Haskell geschriebener CSP$_M$-Parser tut. Zur besseren Integration der verschiedenen Parser für die von ProB unterstützten Sprachen ist geplant, alle Parser in einer Java Virtual Machine (JVM) laufen zu lassen.

Ziel dieser Arbeit war es, zu untersuchen, ob sich der existierende CSP$_M$-Parser nach Frege, einer Haskell-ähnlichen Sprache, die auf der JVM läuft, portieren lässt, und diese Portierung durchzuführen.

Eine besondere Herausforderung bei diesem Vorhaben war, dass Frege eine noch recht junge Sprache ist, die sich noch im Entwicklungsprozess befindet, und somit zu Beginn der Arbeit unklar war, ob diese bereits genug Spracheigenschaften unterstützt, um eine Portierung eines größeren Projekts zu ermöglichen. Damit war nicht nur die Portierung des CSP$_M$-Parsers, sondern auch das Vorantreiben der Entwicklung von Frege durch Portieren von Haskell-Bibliotheken und Finden von Fehlern im Compiler ein wichtiger Bestandteil dieser Arbeit.

Inhaltsverzeichnis

1 Einleitung

In diesem Abschnitt werden die wichtigsten Grundbegriffe im Zusammenhang mit Haskell und Frege eingeführt, die zum Verständnis dieser Arbeit notwendig sind, sowie die Portierung des CSP$_M$-Parsers motiviert.

1.1 Haskell

Haskell ist eine 1990 veröffentlichte, rein funktionale, nicht-strikte, stark typisierte Programmiersprache.[Lip11][Mar10]

Die nicht-strikte Auswertung wird in den meisten Haskell-Implementierungen mittels Lazy Evaluation erreicht: Ein Funktionswert wird nicht direkt beim Aufrufen einer Funktion berechnet, sondern erst, wenn das Ergebnis benötigt wird, z. B. wenn es in eine Datei geschrieben werden soll. Es ist somit möglich, dass ein Ausdruck nicht definierte Werte enthält. Zum Beispiel kann einer Konstanten der Wert `1/0` zugewiesen werden, aber solange der Wert der Konstanten nicht benötigt wird, tritt kein Fehler zur Laufzeit auf. Außerdem ist es so möglich, mit unendlich großen Datenstrukturen zu arbeiten, da nur die tatsächlich benötigten Teile berechnet werden.

Haskell hat ein statisches Typsystem und beim Kompilieren kann für jede Funktion (in den meisten Fällen automatisch) eindeutig geklärt werden, welchen Typen sie hat (Typinferenz). Beispielsweise hat eine Funktion, die einen Integer in einen String konvertiert, den Typen `Integer -> String`. Funktionen können auch andere Funktionen als Parameter haben, sogenannte Funktionen höherer Ordnung; ein Beispiel dafür ist die Funktion `map`, die eine Funktion und eine Liste als Parameter erhält und die Funktion auf jedes Element der Liste anwendet; sie hat den Typen `(a -> b) -> [a] -> [b]`. Bei diesem Beispiel handelt es sich um eine polymorphe Funktion mit den Typparametern `a` und `b`, die bei der Anwendung der Funktion konkretisiert werden.

```
lastElement :: [a] -> a
lastElement [e]   = e
lastElement (h:t) = lastElement t
lastElement []    = error "List␣is␣empty"
```

Listing 1: Beispiel zur Verwendung von Pattern Matching

Bei Funktionsparametern (und auch an anderen Stellen) kann sogenanntes Pattern Matching verwendet werden, um Funktionen für bestimmte Fälle zu definieren, wie es z. B. auch in Prolog und Erlang möglich ist. Listing 1 zeigt dies exemplarisch für Listen: Der erste Fall der Definition wird verwendet, wenn die Liste genau ein Element `e` hat, der zweite, wenn sie ein erstes Element `h` und eine (möglicherweise leere) Restliste `t` hat, und der letzte, wenn sie leer ist.

```
data Tree a
   = Empty
   | Node (Tree a) a (Tree a)
```

```
data Tree a
   = Empty
   | Node {
       left  :: (Tree a),
       value :: a,
       right :: (Tree a)
     }
```

Listing 2: Ein algebraischer Datentyp zur Repräsentation von Binärbäumen, ohne und mit Feldnamen

Es ist möglich, eingebaute Datentypen in benutzerdefinierten Datentypen, sogenannten algebraischen Datentypen, zusammenzufassen. Listing 2 zeigt, wie ein (rekursiver) Datentyp zur Repräsentation eines Binärbaumes definiert werden kann, der die *Konstruktoren* `Empty` für einen leeren Baum und `Node` für einen Knoten mit Wert und linkem und rechtem Teilbaum enthält. `a` ist eine Typvariable, d. h. in einem `Tree a` können beliebige, aber innerhalb eines Baumes gleiche Elemente vom Typ `a` gespeichert werden. Weiterhin ist es möglich, den Bestandteilen des Types Feldnamen zu geben (Record-Syntax). Wenn `t` ein nicht-leerer `Tree` ist, kann man mittels `value t` auf den gespeicherten Wert zugreifen, ohne Pattern Matching verwenden zu müssen.

Mit dem Schlüsselwort `type` ist es möglich, Typsynonyme zu definieren. Beispielsweise ist der Typ `String` ein Synonym für `[Char]`, d. h. jede Funktion, die eine Liste von Charactern als Argument nimmt, akzeptiert auch Strings und andersherum. Insbesondere kann auch (anders als in Frege, wie später erklärt wird) List-Pattern-Matching auf Strings angewendet werden. Mittels `newtype` kann ein neuer Datentyp mit nur einem Konstruktor definiert werden. Dies ist prinzipiell äquivalent zur Verwendung von `data`, erlaubt dem Compiler aber, den Konstruktor wegzuoptimieren, wodurch es keine Performanceeinbußen gegenüber `type` gibt; `newtype` bietet dafür aber eine höhere Typsicherheit.

Mehrere Datentypen können in einer Typklasse zusammengefasst werden. Typklassen sind vergleichbar mit Interfaces in Java. Beispielsweise gibt es die Typklasse `Eq`, in der Datentypen sind, für die die Vergleichsoperatoren == und `/=` definiert sind, wobei es jeweils eine Standardimplementierung mithilfe des jeweils anderen Operators gibt. Für einige Standard-Typklassen wie z. B. `Eq` kann der Haskell-Compiler automatisch für einen benutzerdefinierten Typen die Funktionen einer Typklasse implementieren (Deriving-Mechanismus).

Funktionen und Datentypen werden in Haskell in Modulen zusammengefasst, die in anderen Modulen, ähnlich wie Klassen in Java, importiert werden können. Beispielsweise enthält das Modul `Data.List` Funktionen zum Arbeiten mit Listen. `Prelude` ist ein Modul mit vielen Standardfunktionen und -typen, das, so wie `java.lang` in Java, automatisch importiert wird. Viele zusätzliche Haskell-Module sind in Form von Paketen im Internet auf der Plattform `hackage.haskell.org` verfügbar.

Eine der wichtigsten Haskell-Implementierungen ist der Glasgow Haskell Compiler

```
applyToBoth :: (a,b) -> (forall c. c -> [c]) -> ([a], [b])
applyToBoth (x,y) f = (f x, f y)
```

Listing 3: Beispiel zur Verwendung von Rank-N-Types

(GHC). GHC unterstützt neben dem aktuellen Standard Haskell 2010 eine Reihe von Spracherweiterungen, die beim Kompilieren einzeln oder in einer Datei mit einem Pragma aktiviert werden können. Ein Beispiel dafür ist die Erweiterung zur Unterstützung von Rank-N-Types, d. h. Typen, die beliebig verschachtelte, explizite Quantifizierungen von Typvariablen mittels `forall` zulassen. Ein kleines Beispiel für diese häufig verwendete Erweiterung zeigt Listing 3: Die Funktion `applyToBoth` erhält ein Tupel mit zwei Werten beliebiger Typen `a` und `b` und eine Funktion, die aus einem Wert beliebigen Typs `c` eine Liste mit Elementen desselben Typs macht, und gibt ein Paar zurück, das eine Liste mit Elementen von Typ `a` und eine Liste mit Elementen von Typ `b` enthält. Ohne Verwendung von `forall` wäre diese Definition nicht gültig, da durch die Anwendung der Funktion auf eine Variable vom Typ `a` inferiert würde, dass diese Funktion nur Argumente vom Typ `a` akzeptiert, d. h. die Anwendung auf die Variable vom Typ `b` wäre nicht mehr möglich.

1.2 Frege

Die von Ingo Wechsung entworfene Programmiersprache Frege ist nach eigener Beschreibung „a Haskell for the JVM“ und hat damit dieselben, im letzten Abschnitt erklärten Grundeigenschaften wie Haskell.[Wec14] Der Frege-Compiler, der in dieser Form seit 2011 entwickelt wird, transpiliert den Frege-Quellcode zu Java-Code und strebt eine große Kompatibilität mit Haskell und GHC an. Dadurch ist es möglich, rein funktionalen Code in Haskell-Syntax zu schreiben und auf der Java Virtual Machine (JVM) auszuführen. Damit hat Frege gegenüber Haskell den Vorteil, dass das Kompilat auf allen Plattformen, für die eine JVM verfügbar ist, lauffähig ist, und nicht für jede Zielplattform separat kompiliert werden muss.

Auch wenn Frege große Haskell-Kompatibilität anstrebt, so soll die Sprache vor allem praktisch auf der JVM verwendbar sein: Alle primitiven Typen entsprechen ihren Java-Typen, wodurch es (relativ) einfach möglich ist, Frege-Funktionen in Java-Klassen und auch Java-Klassen als Datentypen in Frege zu verwenden. Eine Konsequenz daraus ist insbesondere, dass Strings durch Java-Strings dargestellt werden, also keine Characterlisten sind, weshalb sie nicht wie Listen behandelt werden können. Dafür steht aber eine große Anzahl von aus Java bekannten Funktionen für Strings zur Verfügung, wie zum Beispiel `startsWith`, `substr` und `toUpperCase`, die somit zum Teil Haskell-Listenfunktionen wie `map toUpper` ersetzen können.

```
containsGreeting :: String -> Bool
containsGreeting ´Hi|Hello´ = True
containsGreeting _          = False
```

Listing 4: Verwendung eines regulären Ausdrucks als Pattern

Zusätzlich gibt es in Frege gegenüber Haskell 2010 reguläre Ausdrücke als primitiven Datentyp. Es ist u. a. möglich, diese zum Pattern Matching auf Strings zu benutzen, wie in Listing 4 gezeigt wird: Die Funktion liefert `True`, wenn der übergebene String `Hi` oder `Hello` enthält.

In Frege benutzte Java-Funktionen, die Parameter oder Rückgabewerte haben, die mutable sind, benutzen die `ST`-Monade. Monaden werden in Haskell u. a. zum Trennen von Code mit und ohne Nebeneffekten benutzt.

Weitere praxisrelevante Unterschiede zu Haskell und wie mit diesen umgegangen werden kann, werden im zweiten Kapitel beschrieben.

1.3 CSP$_M$ und ProB

CSP (Communicating Sequential Processes) ist eine Sprache zur Beschreibung komplexerer, nebenläufiger Systeme, in denen sequentielle Prozesse miteinander kommunizieren; CSP$_M$ ist der maschinenlesbare Dialekt von CSP.[RHB97] Der ProB Animator and Model Checker ist ein auf Prolog basierendes Validierungsprogramm für die B-Methode, einer Theorie zur formalen Entwicklung von Computersystemen.[LB08] Das Programm unterstützt verschiedene Sprachen, darunter B und CSP$_M$, und auch die kombinierte Verwendung von B und CSP$_M$. Es besitzt einen Interaktivmodus, bei dem z. B. CSP$_M$-Ausdrücke basierend auf der aktuell geladenen CSP-Spezifikation ausgewertet werden können. Mit ProB ist es beispielsweise möglich herauszufinden, ob und wie Invarianten eines Systems, z. B. sicherheitsrelevante Eigenschaften eines Eisenbahnkontrollsystems, verletzt werden können.

1.4 Ziel der Arbeit

Zur Zeit gibt es für ProB einen in Java geschriebenen B-Parser und einen in Haskell geschriebenen CSP$_M$-Parser. Bisher haben ProB und der CSP$_M$-Parser miteinander „kommuniziert“, indem ProB den Parser aufgerufen und die erstellte Prolog-Ausgabedatei bzw. die Standardausgabe gelesen hat. Nun ist eine Vereinheitlichung der Schnittstellen angedacht, indem alle Parser für die von ProB unterstützten Sprachen in derselben JVM laufen und über Sockets mit ProB kommunizieren.

Daher ist es naheliegend, statt den bereits vorhandenen CSP$_M$-Parser in Haskell nach Java zu portieren, was insbesondere wegen der unterschiedlichen Programmierparadigmen mit dem Neuschreiben des gesamten Codes verbunden wäre, den Haskell-Code nach Frege zu portieren. Dazu sind theoretisch keine umfangreichen Änderungen am Code notwendig, sodass die bereits ausgiebig getestete Codebasis, mit der die bisherigen Entwickler vertraut sind, erhalten bleiben kann.

Dass die relativ junge Sprache Frege in der Praxis genutzt werden kann, zeigt u. a. schon der in Frege geschriebene Frege-Compiler. Unklar war allerdings vor Beginn dieser Arbeit, ob Frege auch genug Möglichkeiten bietet, um eine reibungslose Portierung von bereits existierendem, umfangreicherem Haskell-Code zu ermöglichen, und ob das Resultat in einer produktiven Umgebung effizient eingesetzt werden kann. Genau dies galt es in dieser Arbeit herauszufinden.

2 Portierung des Haskell-Codes

Da Frege eine hohe Kompatibilität mit dem Haskell 2010-Standard anstrebt, ist es grundsätzlich möglich, dass Haskell-Dateien ohne Modifikation mit Frege übersetzt werden können. In der Praxis sind die kleinen Syntax-Unterschiede zwischen Haskell und Frege aber schon in einfachen Programmen relevant. Darüber hinaus sind sehr viele Module, die für Haskell verfügbar sind, nicht in Frege verfügbar. In diesem Abschnitt wird zunächst dargestellt, wie im Allgemeinen Haskell-Code an Frege angepasst werden muss, und anschließend werden komplexere Herangehensweisen erläutert, die bei der Portierung des CSP$_M$-Parsers notwendig waren.

Beim Portieren wurde die Frege-Version 3.23.423 verwendet. Sowohl in der letzten veröffentlichten Version 3.23.888 des stabilen 3.23-Zweigs als auch in den Versionen des zur Zeit in Entwicklung befindlichen 3.24-Zweigs wurden zu Beginn und im Verlauf der Arbeit verschiedene „nervige“ bis entscheidende Fehler gefunden und den Frege-Entwicklern gemeldet[1][2][3], sodass mit diesen Versionen nicht produktiv gearbeitet werden konnte und auf eine ältere, aber dafür zuverlässigere Version zurückgegriffen wurde.

2.1 Allgemeine Herangehensweise

Im Allgemeinen kann man den Frege-Compiler die Haskell-Ausgangsdatei kompilieren lassen und die vom Compiler bemängelten „Fehler“ nach und nach beheben. Dabei wird man durch auf den ersten Blick mehr oder weniger hilfreiche Fehlermeldungen auf folgende Unterschiede zu Haskell aufmerksam gemacht.

2.1.1 Deriving, Datentypen und Records

```
data Rename = Rename LExp LExp
    deriving (Eq, Ord, Show)
```

```
data Rename = Rename LExp LExp
derive Eq Rename
derive Ord Rename
derive Show Rename
```

Listing 5: Deriving von Instanzen in Haskell (links) und Frege (rechts) im Vergleich

Frege kennt das Schlüsselwort `deriving` nicht, stattdessen gibt es eine der Standalone Deriving-Deklarationen von GHC ähnliche Syntax mit dem Schlüsselwort `derive`. `derive`-Deklarationen müssen nicht direkt bei der `data`-Deklaration stehen, sondern können sich auch in einem anderen Modul befinden.

[1] Issue #271 (closed): guard may evaluate to false, but _ case present, `https://github.com/Frege/frege/issues/271`, abgerufen am 17.09.2016

[2] Issue #284: `'` (prime character) in field label translated to invalid java code, `https://github.com/Frege/frege/issues/284`, abgerufen am 17.09.2016

[3] Issue #285: cannot find symbol `class` β in generated java code, `https://github.com/Frege/frege/issues/285`, abgerufen am 17.09.2016

In Frege können, wie in Haskell auch, Instanzen von `Eq`, `Ord`, `Show`, `Enum` und `Bounded` automatisch abgeleitet werden; insbesondere kennt Frege aber die aus Haskell bekannte Typklasse `Read` nicht, die eine Funktion `read` anbietet, um die String-Repräsentation eines Wertes eines Datentypes wieder in einen Wert des Datentypes zu konvertieren. Als Ersatz für `read` kann oft auf spezielle Konvertierungsfunktionen von Strings zu anderen Typen zurückgegriffen werden, wie z. B. `atoi :: String -> Int`; für algebraische Datentypen muss `read` selbst implementiert werden und kann nicht wie in Haskell automatisch erzeugt werden.

```
data Labeled t = Labeled {
    unLabel :: t
}

f :: Labeled a -> a
f = unLabel


g :: Labeled a -> a
g o = unLabel o

h :: Labeled a -> a -> Labeled a
h o v = o {unLabel = v}
```

```
data Labeled t = Labeled {
    unLabel :: t
}

f :: Labeled a -> a
f = Labeled.unLabel

f' :: Labeled a -> a
f' = _.unLabel

g :: Labeled a -> a
g o = o.unLabel

h :: Labeled a -> a -> Labeled a
h o v = o.{unLabel = v}
```

Listing 6: Möglichkeiten des Zugriffs auf Record-Felder in Haskell (links) und Frege (rechts) im Vergleich

Wenn algebraische Datentypen in Record-Syntax deklariert werden, sind in Frege die Record-Felder in einem eigenen Namespace des Datentypes. Dies erlaubt es einerseits, im Gegensatz zu Haskell, in verschiedenen Datentypen gleiche Feldnamen zu verwenden, aber andererseits hat es zur Folge, dass die Felder, wie in Funktion `f` in Listing 6 gezeigt, mit qualifiziertem Namen als Funktion aufgerufen werden müssen. Um die Verwendung des langen, qualifizierten Namens zu vermeiden, kann alternativ auch eine „objektorientierte“ Notation mithilfe eines Punktes benutzt werden (vgl. Funktion `g`). Auch beim Ändern eines Record-Feldes ist im Gegensatz zu Haskell ein zusätzlicher Punkt erforderlich (Funktion `h`).

Da der Punkt hier eine besondere Funktion hat, ist ein Haskell-Ausdruck wie `succ.fst`, in dem der Punkt für die Verkettung zweier Funktionen steht, in Frege nicht gültig. In Frege wird stattdessen in Anlehnung an das Verkettungszeichen in der Mathematik das Zeichen • (U+2022) verwendet, also `succ•fst` geschrieben. Zur besseren Kompatibilität mit Haskell und weil sich das Unicode-Zeichen nur schwierig über die Tastatur eingeben lässt, wird aber alternativ auch ein Punkt unterstützt, wenn dieser von Leerzeichen umgeben ist, das heißt `succ . fst` funktioniert sowohl in Haskell als auch in Frege.

```
data ParseError = PE {
    parseErrorMsg :: String,
    isCritical    :: Bool
} deriving Show

-- "PE {parseErrorMsg = \"foo\",
   ↪  isCritical = True}"
f = show $ PE "foo" True
```

```
data ParseError = PE {
    parseErrorMsg :: String,
    isCritical    :: Bool
}
derive Show ParseError

-- "PE \"foo\" true"
f = show $ PE "foo" True
```

Listing 7: Beispiel für die unterschiedlich abgeleiteten `Show`-Instanzen in Haskell (links) und Frege (rechts). In Frege werden die Feldnamen nicht mit ausgegeben und boolesche Konstanten sind kleingeschrieben.

Darüber hinaus unterscheiden sich die automatisch abgeleiteten Implementierungen für die Typklasse `Show`: Bei der Konvertierung eines Wertes eines in Record-Syntax definierten Datentyps in einen String sind bei Haskell die Feldnamen mit enthalten, bei Frege fehlen diese. Im CSP$_M$-Parser ist dies nur ein sichtbarer Unterschied, wenn bei Nutzung der Optionen für den Interaktivmodus ein Fehler ausgegeben wird. Da die genaue Formatierung dieser Ausgabe aber für ProB nicht relevant ist, mussten die `Show`-Instanzen nicht angepasst werden. Ferner werden in Frege auch die Konstanten `true` und `false` wie in Java kleingeschrieben; der Compiler akzeptiert aus Kompatibilitätsgründen inzwischen auch Großschreibung, die abgeleiteten `Show`-Instanzen verwenden aber weiterhin Kleinschreibung, weshalb diese manuell überschrieben werden mussten.

Frege kennt vor Version 3.24 das Schlüsselwort `newtype` nicht, stattdessen muss `data` verwendet werden, was äquivalent zu Haskells `newtype` ist, wenn es nur einen einstelligen Konstruktor gibt.

```
data S = S !Int Int
```

```
data S = S {
    a :: !Int,
    b ::  Int
}
```

```
data S = S {
    !a :: Int,
     b :: Int
}
```

Listing 8: Möglichkeiten zur Deklaration strikter Konstruktor-Parameter in Haskell (links) und Frege (rechts)

In Haskell ist es möglich, Parameter eines Konstruktors mit einem Strictness-Flag zu versehen, sodass dieser beim Anwenden des Konstruktors evaluiert wird. Frege unterstützt dies momentan nur in leicht abweichender Syntax, wenn der Parameter ein Feldlabel hat.[4]

[4] Issue #290: Fully support strictness flags in data declarations, `https://github.com/Frege/frege/issues/290`, abgerufen am 17.09.2016

```
f = (g i, g q)
    where
        g :: a -> a
        g x = x
        i = 1
        q = true
```

Listing 9: Beispiel zur Nicht-Generalisierung `let`-gebundener Funktionen in Frege. (Die `where`-Konstruktion wird vom Compiler in eine `let`-Konstruktion übersetzt.)

Frege generalisiert keine `let`-gebundenen Funktionen, was z. B. im Code in Listing 9 relevant ist: Ohne die Typsignatur für die Funktion `g` würde der Code nicht kompilieren, da Frege für `g` den zu konkreten Typen `Int -> Int` inferiert, weil `g` zunächst auf einen Wert vom Typ `Int` angewendet wird. GHC dagegen würde generalisieren und den allgemeineren Typen `forall a. a -> a` (äquivalent zu `a -> a`) inferieren, der in Frege explizit durch eine Typsignatur gegeben sein muss. Fehlt diese, kann es oft beim Portieren von Haskell-Code zu zunächst unverständlichen Fehlermeldungen des Typecheckers kommen.[5] Durch dieses Verhalten, was nur in seltenen Fällen relevant ist, werden die Codeoptimierung und Typinferenz im Frege-Compiler vereinfacht.[Wec14]

2.1.2 Module, Im- und Exporte

Andere Module werden in Haskell mittels `import`-Statements importiert und die vom Modul exportierten Funktionen sind im Namensraum des aktuellen Moduls und mittels des qualifizierten Namens, der den Modulnamen enthält, sichtbar. Wird beim Import das Schlüsselwort `qualified` verwendet, sind die Funktionen nur mit qualifiziertem Namen verfügbar. `qualified` gibt es in Frege nicht, stattdessen kann derselbe Effekt mittels einer leeren Import-Liste erreicht werden, also z. B. `import Lexer()` statt `import qualified Lexer`. Möchte man `Prelude` explizit importieren, um z. B. bestimmte Standardfunktionen aus dem Modulnamensraum auszublenden, macht man dies mit `import frege.Prelude hiding(...)`.

In Haskell werden öffentliche Funktionen mittels einer Exportliste nach dem Modulnamen am Anfang einer Datei exportiert. In Frege werden dagegen Funktionen ähnlich wie in Java als `private`, `protected` oder `public` deklariert, wobei das Weglassen der Angabe – anders als in Java – einer Deklaration als `public` entspricht. Private Funktionen entsprechen Haskell-Funktionen, die nicht in der Exportliste stehen; diese können also nicht in anderen Modulen verwendet werden. Als `protected` deklarierte Funktionen werden beim Import eines ganzen Moduls nicht automatisch mitimportiert, können aber explizit in einer Importliste aufgeführt werden. Weiterhin gibt es in Frege für `Data`-Deklarationen das Schlüsselwort `abstract`, welches alle Konstruktoren des Datentyps privat macht. Tabelle 1 zeigt, welche Frege-Schlüsselwörter den Angaben in den Export-Listen von Haskell entsprechen.

[5] Ein Beispiel für eine solche Fehlermeldung ist `type error in expression lexeme int: type is: CharParser st Integer; expected: CharParser st Integer`, die, zusammen mit anderen Typfehlern, auf eine nicht-generalisierte `let`-gebundene Funktion hingedeutet hat.

Export-Liste (Haskell)	Schlüsselwort (Frege)	bei Import des Moduls sichtbar
`(f, T(..))`	-	ja
`(f, T(..))`	`public`	ja
n/a	`protected`	nur wenn explizit importiert
`()`	`private`	nein
`(T)`	`abstract`	nur Datentyp sichtbar, keine Konstruktoren sichtbar

Tabelle 1: Vergleich der Möglichkeiten zur Festlegung der Sichtbarkeit einer Funktion `f` und eines Datentyps `T` in einem Modul

Sollen Funktionen eines importierten Moduls exportiert werden, fügt man nach dem Eintrag im `import`-Statement das Schlüsselwort `public` ein.

Das von GHC unterstützte `{-# INLINE #-}`-Pragma gibt es in Frege nicht, stattdessen werden Funktionen, die geinlinet werden sollen, nach dem Modulnamen in einer `inline`-Liste aufgeführt.

Bei Modulen ist darüber hinaus zu beachten, dass in Frege der Modulname Package- und Klassennamen der resultierenden Java-Klasse entspricht. Beispielsweise wird das Modul `Language.CSPM.Rename` zu einer Klasse `Rename` im Package `frege.language.CSPM` (importierbar als `Language.CSPM.Rename` oder `frege.language.CSPM.Rename`). Zu beachten ist, dass ein Modul `M` zu einer Klasse `M` im Default-Package wird, wodurch es nicht in Java-Klassen importiert werden kann; daher sollte man Modulnamen ohne Punkt vermeiden. Außerdem müssen Modulnamen wie Klassennamen in Java mit einem Großbuchstaben beginnen.

2.1.3 Umgang mit fehlenden Modulen

Genau wie bei Haskell sind bei der Frege-Distribution nicht alle verfügbaren Frege-Module mitgeliefert, sondern zusätzliche Module können aus dem Internet heruntergeladen werden. Im Gegensatz zu Haskell gibt es dafür aber keine zentrale Seite wie Hackage, weshalb man z. B. per Suchmaschine nach den Modulen suchen muss.

Insbesondere wenn es sich um Module handelt, die zum Standardumfang von GHC gehören, lohnt es sich, zunächst im GitHub-Repository von Frege nach dem Modul zu suchen. Im Verzeichnis `contrib` gibt es eine Reihe von Modulen, die noch nicht vollständig portiert, nicht ausreichend getestet, optimiert oder noch nicht syntaktisch an die aktuelle Frege-Version angepasst sind. Vor allem dann, wenn nicht die ganze Funktionalität eines Moduls gebraucht wird, kann dieser Code schon vollkommen ausreichend sein.

Hilfreich kann auch die Frege-Suchmaschine *Hoogle for Frege*[6] sein, mit der u. a. Funktionen auch über ihre Typsignatur gefunden werden können; z. B. findet man mit `String -> Int` die Funktion `atoi`, die es in Haskell nicht gibt, wo stattdessen `read` aus der Typklasse `Read` benutzt wird. Ferner können einzelne Funktionen auch durch native Java-

[6] `http://hoogle.haskell.org:8081`, abgerufen am 17.09.2016

Methoden ersetzt werden. Beispielsweise kann `System.CPUTime.getCPUTime` durch die Java-Methode `currentTimeMillis` ersetzt werden, wie es in Listing 10 dargestellt ist.

```
native getMs java.lang.System.currentTimeMillis :: () -> IO Long

getCPUTime :: IO Integer
getCPUTime = fmap ((1_000_000 *) . Integer.valueOf) $ getMs ()
```

Listing 10: Verwendung der nativen Java-Methode `currentTimeMillis` in Frege

Wenn „nur“ ein spezieller Datentyp wie `Word` oder `IntMap` fehlt, kann dieser zum Teil auch durch ein einfaches `type`-Statement ersetzt werden. Der Typ `Word` enthält beliebig große, vorzeichenlose Ganzzahlen und kann ohne funktionalen Unterschied als `Integer` repräsentiert werden (dabei ginge „nur“ die Typsicherheit verloren, da nun auch negative Zahlen erlaubt wären). `IntMap` ist eine effiziente Implementierung von `Map`s, die Integer als Schlüssel haben, ist also funktional äquivalent zu `Map Int`.

Außerdem lohnt sich auch ein Blick in den Bugtracker oder auf die Mailinglist, um zu erfahren, ob ein Modul bereits portiert wird oder warum ein Modul nicht portiert wurde. Beispielsweise unterscheidet sich, wie später in Abschnitt 2.4 beschrieben, das Exception-Handling in Frege von dem in Haskell, weshalb das Modul `Control.Exception` nicht zur Verfügung steht.

Die Existenz einer Funktion in Frege bedeutet nicht immer, dass sie sich genauso verhält wie in Haskell; in wenigen Fällen wurde unterschiedliches Verhalten festgestellt. Das betraf z. B. die Operatorpräzedenzen von `.&.` (binäres Und) und + (Addition)[7], die gegenüber Haskell vertauschte Prioritäten haben. Außerdem kann die Funktion `listArray`, die ein immutable Array aus einer Liste erstellt, nicht mit unendlich langen Listen umgehen, was sich in einer Endlosschleife äußert und durch manuelles Kürzen der List umgangen werden kann.[8] Weiterhin gibt in der verwendeten Frege-Version die Funktion `showChar`, die einen Character in einen String umwandelt, anders als Haskell zusätzliche Anführungszeichen aus.[9]

Sollte ein benötigtes Modul in Frege nicht existieren, so kann in der Regel einfach der unter einer BSD-Lizenz stehende „Originalcode“ von GHC nach Frege portiert werden. So ist auch ein Großteil des Codes der Frege-Standardbibliothek entstanden.

Im Rahmen der Arbeit wurden die Module `Data.DList` Version 0.7.1.2[10] (Differenzlisten), `Data.Version` aus `base-4.8.2.0`[11] (Datentyp zur Darstellung von Versionsnummern), `System.FilePath` Version 1.4.1.0[12] (Operationen auf Dateipfaden), `Text.ParserCombinators.Parsec` Version 3.1.9[13] (Parsing-Bibliothek), `Data.Data`

[7] Issue #281: + binds tighter than `.&.`, `https://github.com/Frege/frege/issues/281`, abgerufen am 17.09.2016

[8] Issue #221: Library: `Data.Array`, `https://github.com/Frege/frege/issues/221`, abgerufen am 17.09.2016

[9] Pullrequest #241 (merged): `showChar` should not add redundant quotes around character `https://github.com/Frege/frege/pull/241`, abgerufen am 17.09.2016

[10] `https://hackage.haskell.org/package/dlist-0.7.1.2`, abgerufen am 17.09.2016

[11] `https://hackage.haskell.org/package/base-4.8.2.0`, abgerufen am 17.09.2016

[12] `https://hackage.haskell.org/package/filepath-1.4.1.0`, abgerufen am 17.09.2016

[13] `https://hackage.haskell.org/package/parsec-3.1.9`, abgerufen am 17.09.2016

und `Data.Generics` aus Scrap Your Boilerplate (SYB) 0.3[14] (generische Programmierung) und `Text.XML.Light.Output` Version 1.3.14[15] (Erstellen von XML-Dateien) nach Frege portiert. Bei Modulen, die vom CSP$_M$-Parser nicht in der neuesten Version benötigt werden, deren neueste Version aber viele GHC-Abhängigkeiten hat, wurde oft auf eine ältere Version mit weniger GHC-Abhängigkeiten zurückgegriffen, wie es z. B., wie später in Abschnitt 2.3 erläutert, bei SYB gemacht wurde. Das Portieren der zum Teil umfangreicheren Pakete wurde teilweise durch Fehler, die im Compiler gefunden[16] wurden, erschwert bzw. verzögert.

Von den Modulen `Data.Typeable`[17] (typsichere Casts) und `Text.PrettyPrint`[18] (Prettyprinter) gab es bereits Frege-Portierungen, die – nach kleinen syntaktischen Anpassungen an die aktuelle Frege-Version – verwendet werden konnten. Das Prettyprinter-Modul wurde so umgeschrieben, dass anstelle von Strings als interne Darstellung Character-Listen verwendet werden, da sich diese Variante bei den durchgeführten Benchmarks wegen der wegfallenden Stringkonkatenation als schneller herausgestellt hat. Der Code der Module `Data.Array` (immutable Arrays mit $\mathcal{O}(1)$-Indexzugriff) und `System.Exit` (Programmterminierung mit Exit-Code), die in der verwendeten Frege-Version nicht zur Verfügung stehen, wurde aus der aktuellen Entwicklungsversion von Frege übernommen, in der diese Module vorhanden sind.

Das Modul `System.Console.CmdArgs` zum Parsen von Kommandozeilenargumenten wurde nicht portiert, weil der verwendete Datentyp `data Any = forall a. Data a => Any a` in Frege nicht deklariert werden kann, da dieser zur Verwendung des `forall` die von Frege nicht unterstützte GHC-Erweiterung `ExistentialQuantification` benötigt. Weil das Parsen der Argumente direkt in der Hauptfunktion des CSP$_M$-Parsers geschieht und es somit keinen Frege-Code gibt, der diese Funktion aufruft, konnte diese einfach unter Verwendung der Apache Commons CLI-Bibliothek in Java neugeschrieben werden. Nach der Verarbeitung der Kommandozeilenargumente im Java-Code wird die entsprechende Frege-Funktion aufgerufen, die den gewünschten Befehl ausführt und sich ggf. um auftretende Exceptions kümmert. Somit stellt dieser Code auch direkt eine Referenz dafür dar, wie Frege-Code von Java aus aufgerufen werden kann, was für die zukünftige Integration in die Java-Umgebung relevant ist.

2.1.4 Umgang mit nicht verfügbaren GHC-Erweiterungen

Frege unterstützt eine Reihe von GHC-Erweiterungen wie `RankNTypes` (Verwendung von `forall` in Typsignaturen), `BangPatterns` (Strictnessannotationen für Funktionsparameter) und `StandaloneDeriving` (andere Syntax und bisher einzige Möglichkeit in Frege zum Deriven, vgl. Abschnitt 2.1.1). Diese sind immer aktiviert, wodurch verhindert werden soll, dass wie bei GHC mehr oder weniger unübersichtliche Abhängigkeiten und Implikationen der Erweiterungen entstehen.

[14] `https://hackage.haskell.org/package/syb-0.3`, abgerufen am 17.09.2016

[15] `https://hackage.haskell.org/package/xml-1.3.14`, abgerufen am 17.09.2016

[16] Issue #270 (closed): javac cannot find symbol with `forall`-types, `https://github.com/Frege/frege/issues/270`, abgerufen am 17.09.2016

[17] `https://github.com/rdegnan/typeable`, abgerufen am 17.09.2016

[18] `https://github.com/datatyped/pretty/`, abgerufen am 17.09.2016

```
block_comment :: Alex Token
block_comment = do
  let
    tokenId = mkTokenId cnt
    tokenStart = startPos
    tokenLen = length tokenStr
  return $ Token {..}
```

```
block_comment :: Alex Token
block_comment = do
  let
    tokenId = mkTokenId cnt
    tokenStart = startPos
    tokenLen = length tokenStr
  return $ Token {
    tokenId = tokenId,
    tokenStart = tokenStart,
    tokenLen = tokenLen
  }
```

Listing 11: Vermeidung der links verwendeten GHC-Erweiterung `RecordWildCards`

Viele im CSP$_M$-Parser verwendete Erweiterungen fügen vor allem syntaktischen Zucker hinzu, sodass sich diese durch kleine Veränderungen am Code vermeiden lassen. In Listing 11 ist z. B. dargestellt, wie die Erweiterung `RecordWildCards` vermieden werden kann.

Drei andere GHC-Erweiterungen werden im Prolog- und im XML-Prettyprinter verwendet. Im Prolog-Prettyprinter gibt es u. a. eine Instanzdeklaration `instance TERM t => PREDICATE t`, d. h. jeder Term ist ein Prädikat (vgl. Listing 12). Dadurch wird es möglich, mit `predicate` sowohl ein Prädikat als auch einen Term in ein Prädikat zu konvertieren, wie es `f3` tut. Die Instanzdeklaration ist aber so ohne Weiteres weder in Haskell noch in Frege möglich.

Gemäß Haskell-Standard kann eine Klasseninstanz nur für einen Typen `T a1 ... an` ($n \geq 0$) definiert werden, wobei `T` ein Typkonstruktor und `a1 ... an` unterschiedliche Typvariablen sind, d. h. `PREDICATE t` ist in der Instanzdeklaration nicht erlaubt. Die GHC-Erweiterung `FlexibleInstances` lockert diese Bedingung und erlaubt beliebig verschachtelte Typen; beim Kompilieren wird für einen Typen dann die konkreteste Instanzdeklaration benutzt. Weiterhin schreibt GHC vor, dass die Bedingungen einer Instanzdeklaration weniger Konstruktoren und Variablen enthalten müssen als der Typ, für den eine Instanz definiert wird; diese und weitere Bedingungen dienen dazu, die Terminierung des Typechecking-Algorithmus von GHC zu garantieren. Die Prüfung dieser Bedingungen kann mit `UndecidableInstances` deaktiviert werden. Zuletzt gibt es noch das Problem, dass die Instanzen von `Predicate` und `t` für `PREDICATE` überlappen[19]. Die Erweiterung `OverlappingInstances` erlaubt diese Überlappung, sofern es wieder eine konkreteste Instanz gibt, was hier gegeben ist.

Da Frege `FlexibleInstances` nicht unterstützt,[20] musste dieser Code angepasst werden. Auf alle drei GHC-Erweiterungen kann verzichtet werden, wenn `instance TERM t => PREDICATE t` vermieden werden kann. Beim CSP$_M$-Parser war dies möglich, da

[19] Da es möglich wäre, dass an einer anderen Stelle für `Predicate` eine `TERM`-Instanz deklariert wird, wird beim Prüfen auf Überlappung von GHC die Bedingung `TERM t` ignoriert.[GHC16]

[20] Issue #5: Flexible instances, `https://github.com/Frege/frege/issues/5`, abgerufen am 17.09.2016

```
data Term = Term {unTerm :: String}
data Predicate = Predicate {unPredicate :: String}

class TERM t where
    term :: t -> Term
instance TERM Term where
    term = id

class PREDICATE p where
    predicate :: p -> Predicate
instance PREDICATE Predicate
    where predicate = id
instance TERM t => PREDICATE t
    where predicate = Predicate . unTerm . term

f1 :: Term -> Predicate
f1 t = predicate t

f2 :: Predicate -> Predicate
f2 t = predicate t

f3 :: PREDICATE p => p -> Predicate
f3 t = predicate t

f4 :: Term -> Predicate
f4 t = (Predicate . unTerm . term) t
```

Listing 12: Verwendungsbeispiel für die GHC-Erweiterungen `FlexibleInstances`, `OverlappingInstances` und `UndecidableInstances`

es nie die allgemeine Verwendung wie in `f3` gibt, sondern der konkrete Typ immer bekannt ist, sodass explizit wie in `f4` Terme in Prädikate umgewandelt werden können.

Andere GHC-Erweiterungen können nur mit großem Aufwand vermieden werden, wie beispielsweise die im Parser verwendete Erweiterung `DeriveDataTypeable`. Wie und warum diese portiert wurde, wird in Abschnitt 2.3 beschrieben.

2.1.5 Weitere kleine Unterschiede

Die Typklasse `Monad` enthält, anders als in Haskell, gemäß der mathematischen Definition einer Monade nicht die Funktion `fail`. In Frege gibt es dafür die Typklasse `MonadFail`, die der Klasse `Monad` in Haskell entspricht. Frege setzt damit das `MonadFail` Proposal[21] für Haskell um.

[21] `MonadFail` proposal (MFP), `https://prime.haskell.org/wiki/Libraries/Proposals/MonadFail`, abgerufen an 17.09.2016

Sowohl in Haskell als auch in Frege können beliebige Funktionen in Infix-Schreibweise benutzt werden, wenn sie in Backquotes (`` `f` ``) eingeschlossen werden. In Haskell haben diese Funktionen, sofern nicht anders vom Programmierer festgelegt, höchste Präzedenz und sind linksassoziativ; in Frege haben sie auch höchste Präzedenz, sind aber standardmäßig weder links- noch rechtsassoziativ, sodass entweder explizit geklammert oder die Assoziativität mit einer `infix`-Deklaration angegeben werden muss.[22]

In der verwendeten Frege-Version erlaubt es der Lexer im Gegensatz zum Haskell 2010-Standard nicht, dass ein `where`, das auf einen `do`-Block folgt, genauso weit eingerückt ist wie der Code des `do`-Blocks. Deswegen musste an einigen Stellen die Einrückung des Codes geändert werden. Nachdem dieser Fehler[23] gemeldet wurde, ist es inzwischen in der aktuellen Entwicklungsversion möglich, die ursprüngliche Einrückung zu verwenden.

Frege unterstützt beim Patternmatching keine *negative patterns*,[24] d. h. eine Definition wie `f (-1) = 0` ist nicht gültig. Als Alternative kann in diesen Fällen eine `if`-Abfrage in der Funktion benutzt werden.

In Frege müssen alle Pattern, die ein Funktionsargument sind und komplexer als ein Identifier, Literal oder Record-Konstruktor sind, geklammert werden. `f xs@(x:_) = x:xs` ist in Frege also nicht gültig, da `xs@(x:_)` geklammert werden müsste. Der Grund für diesen Unterschied ist, dass so die Syntax für Ausdrücke links vom Gleichheitszeichen und rechts vom Gleichheitszeichen gleich ist.

Konstanten primitiver Datentypen haben in Frege dieselbe Bedeutung wie in Java: In Haskell wird `'\172'` hexadzimal als ¬ interpretiert, in Frege und Java oktal als `z`.

Dokumentationskommentare werden in Frege mit `---` (einzeilig) und `{--` (mehrzeilig) eingeleitet und sind im Gegensatz zu ihren Pendants `-- |` und `{-|` von Haskells Haddock-Dokumentationskommentaren syntaktische Elemente und müssen damit auch den Layout-Regeln folgen, d. h. genauso wie der umgebene Code eingerückt werden. Das gilt auch für aus Bindestrichen bestehende „optische Trennlinien“, die als Dokumentationstext interpretiert werden und damit u. U. Syntaxfehler verursachen können.[25] Der Dokumentationstext eines Recordfeldes eines Datentyps ersetzt außerdem in Frege das Komma als Trennzeichen zwischen Feldern. Die Dokumentationskommentare des CSP$_M$-Parsers wurden an die Frege-Syntax angepasst, sodass der Frege-Compiler Moduldokumentationen erstellen kann (vgl. Abschnitt 3.2).

2.2 Portierung des Lexers

Der Lexer des CSP$_M$-Parsers wird mithilfe des in Haskell geschriebenen Lexer-Generators *Alex* aus einer Datei mit Token-Beschreibungen generiert. Da nur der generierte Haskell-Code für den Parser relevant ist und nicht der Quelltext von Alex, war es ausrei-

[22] Issue #298: functions enclosed in backquotes should be left-associative by default, `https://github.com/Frege/frege/issues/298`, abgerufen am 17.09.2016

[23] Issue #272 (closed): unexpected token `where` when indentation of `do`-block same as `where`, `https://github.com/Frege/frege/issues/272`, abgerufen am 17.09.2016

[24] Issue #26: Pattern support, `https://github.com/Frege/frege/issues/26`, abgerufen am 17.09.2016

[25] Issue #274: non-indented documentation comment is syntax error, `https://github.com/Frege/frege/issues/274`, abgerufen am 17.09.2016

chend, ersteren nach Frege zu portieren. Dabei musste darauf geachtet werden, dass die notwendigen Änderungen automatisiert werden können, sodass im Build-Prozess eine geänderte Token-Definition mit Alex nach Haskell und anschließend automatisch nach Frege übersetzt werden kann.

Die kleinen syntaktischen Anpassungen wegen der bereits beschriebenen Unterschiede werden im Build-Prozess von dem Shellskript `AlexToFrege` mittels `grep` und `sed` vorgenommen. Da es sich dabei nur um einfache Stringersetzungen handelt, ist nicht garantiert, dass Code, der mit einer beliebigen Alex-Version mit beliebigen Eingabedateien erstellt wurde, korrekt nach Frege übersetzt werden kann. Das geschriebene Skript funktioniert mit der aktuellen Alex-Version 3.1.7 und der Tokendefinition für den CSP_M-Parser.

Darüber hinaus gab es bei zwei Listen im Lexer-Code mit mehr als 10.000 Einträgen, die DFA-Zustände repräsentieren, Probleme bei der Übersetzung nach Java. Wenn versucht wird, eine Frege-Datei mit einer Liste dieser Größe zu übersetzen, kommt es bei Verwendung eines 2 GB großen Heaps nach mehreren Minuten zu einem `OutOfMemoryError`, da das Overhead-Limit der Garbage-Collection überschritten wird.

Dieses Problem wurde zunächst gelöst, indem die Liste in mehrere kleine Listen aufgeteilt wurde, die zur Laufzeit konkateniert werden. Frege kann dann Java-Code erzeugen, der aber nicht kompiliert werden kann, da der Code für die Klassenkonstanten das Limit von 65.535 Byte Bytecode der JVM[LYBB15] überschreitet: Die einzelnen Funktionen, die konstante Listen zurückgeben, werden nämlich in Klassenvariablen, deren Java-Code insgesamt zu groß wird, übersetzt. Diese Problematik konnte dadurch umgangen werden, dass ein unbenutzter Parameter zu diesen Funktionen hinzugefügt wurde, wodurch der Frege-Compiler Java-Methoden generiert, die jeweils das Bytecodelimit nicht überschreiten.

Später wurde eine alternative Möglichkeit benutzt, die im übermittelten Fehlerbericht[26] für den Compiler vorgeschlagen wurde und auch im Frege-Compiler selbst zum Einsatz kommt: Statt im Quelltext direkt eine Liste zu verwenden, wird die Liste in einem String gespeichert und zur Laufzeit mittels `parseJSON` in eine Liste umgewandelt.

2.3 Portierung der GHC-Erweiterung DeriveDataTypeable

Generische Programmierung (nicht zu verwechseln mit den Generics in Java, die in etwa den Typparametern in Haskell entsprechen) bezeichnet in Haskell eine Abstraktion, die es mit wenig Code ermöglicht, Operationen auf großen rekursiven Datentypen auszuführen, die normalerweise mit selbst geschriebenen Funktionen durchlaufen werden müssten. Zur Vermeidung von *Boilerplate*-Code gibt es Haskell-Module mit verschiedenen Ansätzen. Einer davon ist Scrap your Boilerplate[LJ03] (SYB), das im Paket `Data.Generics` die nötigen Funktionen anbietet und im CSP_M-Parser zum Einsatz kommt.

Da das Paket nicht in Frege verfügbar ist und der Portierungsaufwand vergleichsweise hoch ist, soll zunächst an einem Beispiel motiviert werden, warum die Portierung erstre-

[26] Issue #287: Compiling big list results in error: code too large, `https://github.com/Frege/frege/issues/287`, abgerufen am 17.09.2016

benswert ist, wie SYB in Haskell funktioniert und anschließend erläutert werden, wie die generische Programmierung in Frege umgesetzt wurde.

2.3.1 Motivation: Vereinfachung von Ausdrücken

Es seien folgende Datentypen gegeben, die ein Programm repräsentieren können, das aus einer Liste von einfachen Anweisungen besteht. Eine Anweisung ist entweder eine If-Anweisung oder eine Zuweisung:

```
data Program = Program [Statement]

data Statement
    = If  Expression [Statement]
    | Let Identifier Expression

data Expression
    = Plus Expression Expression
    | Mult Expression Expression
    | Lt   Expression Expression
    | Const Int
    | Var Identifier

type Identifier = String
```

Listing 13: Die algebraischen Datentypen zur Repräsentation eines Programms

Es soll nun eine einfache Funktion zur Konstantenfaltung geschrieben werden, die einfache Ausdrücke wie `4+2` oder `6*7` zu Konstanten vereinfacht. Dieser Code würde üblicherweise wie folgt aussehen:

```
simplify :: Program -> Program
simplify (Program e2) = Program $ map simplStmt e2
  where
    simplStmt :: Statement -> Statement
    simplStmt (If  e s) = If (simplExpr e) (map simplStmt s)
    simplStmt (Let i e) = Let i $ simplExpr e
    simplExpr :: Expression -> Expression
    simplExpr (Plus (Const n) (Const m)) = Const $ n + m
    simplExpr (Mult (Const n) (Const m)) = Const $ n * m
    simplExpr (Plus e1 e2) = Plus (simplExpr e1) (simplExpr e2)
    simplExpr (Mult e1 e2) = Mult (simplExpr e1) (simplExpr e2)
    simplExpr (Lt   e1 e2) = Lt   (simplExpr e1) (simplExpr e2)
    simplExpr e = e
```

Listing 14: Konstantenfaltung mit Boilerplate-Code

Auffällig ist hier, dass nur die ersten beiden Fälle der Funktion `simplExpr` die eigentliche Arbeit verrichten, und sämtlicher anderer Code nur den Zweck hat, die Datenstruktur abzulaufen. Bei umfangreicheren Datenstrukturen, wie sie z. B. bei einem vollständi-

gen Compiler aufträten, fiele noch mehr Boilerplate-Code an, der von den eigentlich relevanten Fällen ablenkt und beim Erweitern des `Program`-Typs ebenfalls erweitert werden müsste.

Deshalb wäre es wünschenswert, wenn man sich auf die beiden relevanten Fälle und einen allgemeinen „Standardfall“ beschränken könnte. Genau dies ist mithilfe generischer Programmierung – hier konkret SYB – möglich, sodass sich der Code wie folgt verkürzen lässt:

```
simplify :: Program -> Program
simplify = everywhere $ mkT simplExpr
  where
    simplExpr :: Expression -> Expression
    simplExpr (Plus (Const n) (Const m)) = Const $ n + m
    simplExpr (Mult (Const n) (Const m)) = Const $ n * m
    simplExpr e = e
```

Listing 15: Konstantenfaltung mit SYB

Das Ablaufen der Datenstruktur wird jetzt von den Funktionen `everywhere` und `mkT` aus `Data.Generics` übernommen. `mkT` (*make transformation*) erweitert dabei den Typen von `simplExpr`, sodass die Funktion auf alle Typen angewendet werden kann: Angewendet auf eine `Expression` verhält sie sich wie definiert, angewendet auf jeden anderen Typen wie die Identitätsfunktion. Die Funktion `everywhere` wendet dann diese Funktion auf jedes Element in der Datenstruktur an.

2.3.2 Die Typklassen Typeable und Data

Damit das Ganze funktioniert, müssen die beteiligten Datentypen den Typklassen `Typeable` und `Generic` angehören. GHC kann automatisch mittels `deriving` Code dafür erzeugen, wenn die Spracherweiterung `DeriveDataTypeable` verwendet wird.

`Typeable` ermöglicht es, typsichere Casting-Operationen zu definieren: Datentypen, die `Typeable` implementieren, können nach ihrem Typen in `String`-Form „gefragt“ werden, sodass – vorausgesetzt, der Typ „lügt“ nicht – zur Laufzeit sicher festgestellt werden kann, ob zwei Typen gleich sind. Diese Funktionalität kann dann benutzt werden, um in `mkT` festzustellen, ob das übergebene Element denselben Typen hat wie das erste Argument der Funktion.

Die Typklasse `Data` definiert die generische Transformationsfunktion `gmapT :: Data a => (forall b. Data b => b -> b) -> a -> a`, die eine generische Transformation (z. B. mittels `mkT` erstellt) auf alle direkten Kinder eines Datentypen anwendet, sich also ähnlich wie die von Listen bekannte Funktion `map` verhält. Die Transformationsfunktion ist eine mittels `forall` angegebene polymorphe Funktion, da die Kinder von unterschiedlichen Typen sein können und eine „einfache“ Funktion `b -> b` dann nicht auf alle Kinder angewendet werden könnte, weil `b` kein eindeutiger Typ zugeordnet werden kann. `everywhere :: Data a => (forall b. Data b => b -> b) -> a -> a` kann nun mittels Rekursion und `gmapT` eine Transformation auf *alle* (nicht nur direkte) Kinder anwenden.

Im CSP$_M$-Parser wird `everywhere` u. a. zum Vereinfachen des ASTs verwendet. Darüber hinaus kommt die generische Funktion `listify` zum Einsatz, die es ermöglicht, alle Daten eines bestimmten Typs als Liste zu erhalten, was z. B. zur Bestimmung von Use- und Def-Mengen verwendet wird.

Eine minimale Instanzdefinition für `Data` für einen nicht-trivialen Datentypen `a` enthält im Wesentlichen die Funktion `gfoldl :: (forall d b. Data d => c (d -> b) -> d -> c b) -> (forall g. g -> c g) -> a -> c a`[27]. `gfoldl` ist eine Verallgemeinerung der `fold`-Funktion (welche der Methode `Stream.collect` in Java ähnelt) für die Anwendung von Konstruktoren, mit der alle `gmap`-Operationen in SYB definiert werden können.

2.3.3 DataTypeable-Implementierung in Frege

Frege unterstützt die GHC-Erweiterung `DeriveDataTypeable` nicht und kennt auch nicht die Typklassen `Typeable` und `Data`, weshalb einerseits die Module `Data.Typeable` und `Data.Data` portiert werden mussten und andererseits eine Möglichkeit gefunden werden musste, um `Typeable`- und `Data`-Instanzen von algebraischen Datentypen – am besten automatisch – erzeugen zu können. Da über den Haskell 2010-Standard herausgehend dafür theoretisch nur Rank-2-Types (von Frege unterstützt) und eine Cast-Operation benötigt werden, schien eine Portierung möglich zu sein.

`Data.Typeable` wurde bereits vor zwei Jahren von Ryland Degnan nach Frege portiert (vgl. Abschnitt 2.1.3). Dieser Code konnte nach kleinen Syntax-Anpassungen mit der aktuellen Frege-Version verwendet werden. Die Funktion `unsafeCoerce :: a -> b`, die in GHC für die Cast-Operation verwendet wird, wird hier mit der Java-Funktion `T requireNonNull(T obj)` nachgebildet.

Von `Data.Data` wurde der Code aus dem Paket `syb` in der Version 0.3 nach Frege portiert. Dies ist die älteste Version, die mit dem CSP$_M$-Parser kompatibel ist und die im Gegensatz zur aktuellen Version, die Bestandteil von GHC ist, weniger Abhängigkeiten von weiteren GHC-Erweiterungen (wie z. B. `PolyKinds`) hat. Die Portierung des Codes verlief grundsätzlich nach dem oben beschriebenen Schema, hinderlich dabei waren aber verschiedene Fehler, die dabei im Typechecker von Frege in Zusammenhang mit Rang-n-Polymorphie gefunden wurden[28][29].

Außerdem kommt Frege vor Version 3.24 mit den Typen der Funktionen, die generische Operationen auf Typen mit Typparametern durchführen, nicht zurecht. Deswegen mussten im XML-Prettyprinter einige generische Funktionen auf dem AST durch mehrere konkrete Funktionen ersetzt werden, wie in Listing 16 gezeigt wird. Dadurch entsteht zwar wieder mehr Boilerplate-Code, aber die ursprüngliche Funktionalität konnte beibehalten werden.

Damit `Data.Data` genutzt werden kann, müssen für jeden algebraischen Datentypen,

[27] „Trying to understand the type of `gfoldl` directly can lead to brain damage.“[LJ03]

[28] Issue #273: Type error when defining instance for function with `forall`, `https://github.com/Frege/frege/issues/273`, abgerufen am 17.09.2016

[29] Issue #277: Two `forall a` in a type signature get confused, `https://github.com/Frege/frege/issues/277`, abgerufen am 17.09.2016

```
astToXML :: Data a => a -> Element
astToXML
  = genericCase
        `ext1Q` listToXML

  where
    genericCase :: Data a => a -> Element
    genericCase n =
        unode (showConstr $ toConstr n) $
                        gmapQ astToXML n

    listToXML :: Data a => [a] -> Element
    listToXML = unode "list" . map astToXML
```

```
astToXML :: Data a => a -> Element
astToXML
  = genericCase
        `extQ` listToXML_FunCase
        `extQ` listToXML_LDecl
  where
    genericCase :: Data a => a -> Element
    genericCase n =
        unode (showConstr $ toConstr n) $
                        gmapQ astToXML n

    listToXML :: Data a => [a] -> Element
    listToXML = unode "list" . map astToXML

    listToXML_FunCase :: [FunCase] -> Element
    listToXML_FunCase = listToXML
    listToXML_LDecl :: [LDecl] -> Element
    listToXML_LDecl = listToXML
```

Listing 16: Ersetzen der in Frege 3.23 nicht verwendbaren Query-Funktion `ext1Q` durch `extQ`

der Funktionen des Moduls verwendet, Instanzen der Typklassen `Data` und `Typeable` geschrieben werden. Da dies einerseits arbeitsaufwändig und andererseits fehleranfällig ist (GHC erlaubt seit Version 7.8 aus diesem Grund gar nicht mehr, `Typeable`-Implementierungen selbst zu schreiben), wurde ein (Frege-)Programm geschrieben, welches die Instanzen automatisch aus den Datentypdefinitionen erzeugt.

Das Programm *DataDeriver* geht dabei wie folgt vor:

1. Lese die übergebene Frege-Datei ein und behalte daraus alle Data-Blöcke, die mit dem Pragma `{-# derive DataTypeable #-}` versehen sind.
2. Parse die Data-Deklarationen gemäß Grammatik 1 in einen abstrakten Syntaxbaum (AST) nach Grammatik 2.
3. Gib für jede Data-Deklaration `Data`- und `Typeable`-Implementierungen aus.

Die Einführung eines Pragmas wurde gegenüber eines einfachen `derive Data`, `derive Typeable` bevorzugt, damit der Code auch ohne *DataDeriver* mit Frege kompiliert werden kann. Außerdem wird so deutlich, dass es (noch) keinen Support seitens des Frege-Compilers gibt.

Grammatik 1 ist eine vereinfachte Version der Grammatik für Definitionen von algebraischen Datentypen in Frege[Wec14], bei der beispielsweise die Schlüsselwörter `abstract`, `private` und `forall` nicht unterstützt werden. Für die Anwendung auf den Code des CSP_M-Parser ist die Grammatik aber vollkommen ausreichend. Langfristig sollte der Frege-Compiler `derive Data` unterstützen, weshalb ein umfangreicherer Parser unnötige Arbeit gewesen wäre. Da nur der Name des Datentyps, die Anzahl der Typparameter, die Namen der Konstruktoren und die Anzahl deren Parameter wichtig sind, werden, wie in Grammatik 2 gezeigt, vom Parser nur diese relevanten Informationen in einem AST gespeichert.

⟨*topdecl*⟩ ::= ‘`data`’ ⟨*typeName*⟩ ⟨*typeVar*⟩* ‘`=`’ ⟨*constructors*⟩

⟨*constructors*⟩ ::= ⟨*constructor*⟩ (‘`|`’ ⟨*constructor*⟩)*

⟨*constructor*⟩ ::= ‘`!`’? ⟨*typeName*⟩ ⟨*simpleType*⟩*
| ‘`!`’? ⟨*typeName*⟩ ‘`{`’ ⟨*fieldLine*⟩ (‘`,`’ ⟨*fieldLine*⟩)* ‘`}`’

⟨*fieldLine*⟩ ::= ‘`!`’? ⟨*identifier*⟩ ‘`::`’ ⟨*typeApp*⟩

⟨*simpleType*⟩ ::= ⟨*identifier*⟩
| ‘`(`’ ⟨*typeApp*⟩ (‘`,`’ ⟨*typeApp*⟩)* ‘`)`’
| ‘`[`’ ⟨*typeApp*⟩ ‘`]`’

⟨*typeApp*⟩ ::= ⟨*simpleType*⟩
| ⟨*typeApp*⟩ ⟨*simpleType*⟩

⟨*typeName*⟩ ::= ⟨*identifier*⟩

⟨*typeVar*⟩ ::= ⟨*identifier*⟩

⟨*identifier*⟩ ::= [`0-9A-Za-z_'`]+

Grammatik 1: Die vom Parser des *DataDerivers* benutzte Grammatik

⟨*topdecl*⟩ ::= ⟨*typeName*⟩ ⟨*numberOfTypeVars*⟩ ⟨*constructor*⟩*

⟨*constructor*⟩ ::= ⟨*typeName*⟩ ⟨*numberOfParameters*⟩

⟨*typeName*⟩ ::= String

⟨*numberOfTypeVars*⟩ ::= Int

⟨*numberOfParameters*⟩ ::= Int

Grammatik 2: Die Grammatik für den AST

Der AST wird dann in Frege-Code, der die entsprechenden Instanzen implementiert, übersetzt. Dabei wurde sich an dem Beispielcode aus der Dokumentation von `Data.Data` orientiert. Exemplarisch ist in Listing 17 die Ausgabe des *DataDerivers* für den Datentyp `Expression` aus Listing 13 zu sehen.

```
tc_Expression :: TyCon
tc_Expression = mkTyCon3 "HHU" "ConstantFolding" "Expression"
instance Typeable (Expression ) where
    typeOf _ = mkTyConApp tc_Expression []
con_Expression_Plus :: Constr
con_Expression_Plus = mkConstr ty_Expression "Plus" [] Prefix
con_Expression_Mult :: Constr
con_Expression_Mult = mkConstr ty_Expression "Mult" [] Prefix
con_Expression_Lt :: Constr
con_Expression_Lt = mkConstr ty_Expression "Lt" [] Prefix
con_Expression_Const :: Constr
con_Expression_Const = mkConstr ty_Expression "Const" [] Prefix
con_Expression_Var :: Constr
con_Expression_Var = mkConstr ty_Expression "Var" [] Prefix
ty_Expression :: DataType
ty_Expression = mkDataType "HHU.ConstantFolding.Expression" [con_Expression_Plus,
        con_Expression_Mult, con_Expression_Lt, con_Expression_Const, con_Expression_Var]
instance Data (Expression ) where
    toConstr (Plus _ _) = con_Expression_Plus
    toConstr (Mult _ _) = con_Expression_Mult
    toConstr (Lt _ _) = con_Expression_Lt
    toConstr (Const _) = con_Expression_Const
    toConstr (Var _) = con_Expression_Var
    dataTypeOf _ = ty_Expression
    gunfold k z c = case constrIndex c of
                        1 -> k (k (z Plus))
                        2 -> k (k (z Mult))
                        3 -> k (k (z Lt))
                        4 -> k (z Const)
                        5 -> k (z Var)
                        _ -> error "gunfold(Expression)"
    gfoldl f z x = case x of
                        (Plus a1 a2) -> ((z Plus) `f` a1) `f` a2
                        (Mult a1 a2) -> ((z Mult) `f` a1) `f` a2
                        (Lt a1 a2) -> ((z Lt) `f` a1) `f` a2
                        (Const a1) -> (z Const) `f` a1
                        (Var a1) -> (z Var) `f` a1
```

Listing 17: Automatisch erzeugte `Data` und `Typeable`-Implementierung für `Expression`

2.4 Portierung von Exceptions

In Haskell gibt es für Funktionen grundsätzlich zwei Möglichkeiten, Fehler, wie z. B. Lexing-Fehler im Lexer des CSP$_M$-Parsers, zu melden: über einen speziellen Rückgabedatentyp (z. B. `Either a b`) oder – wie von Java bekannt – durch das Werfen einer Exception.[OGS09] Im CSP$_M$-Parser wird eine Mischung verwendet: Beispielsweise hat die Hauptfunktion des Lexers den Typ `Either LexError [Token]` und die Funktion, die die Lexerfunktion aufruft, wirft im Falle eines `LexError`s eine Exception. Da Frege nicht den Exception-Mechanismus von Haskell unterstützt, musste dieser, wie im Folgenden beschrieben, nachgebildet werden. Dabei wurde darauf geachtet, möglichst wenig an der bisher verwendeten Syntax ändern zu müssen.

In Haskell wird im Paket `Control.Exception` die Typklasse `Exception` definiert. Alle Datentypen, die Instanz dieser Klasse sind, können mit `throw :: Exception e`

`=> e -> a` als Exception geworfen werden und mit `catch :: Exception e => IO a -> (e -> IO a) -> IO a` abgefangen werden.

In Frege wird auf das Exception-System von Java zurückgegriffen. Ein Wert vom nativen Java-Typ `Throwable` kann mit der Funktion `throwST :: Throwable -> ST s a` als Exception geworfen werden und mittels `catch :: Exceptional e => ST s a -> (e -> ST s a) -> ST s a` abgefangen werden. `Exceptional` ist eine Typklasse, die zur Zeit nur von nativen Java-Typen implementiert werden kann.[Wec14] Es ist also nicht möglich, eigene Exception-Typen im Frege-Code zu definieren.

Deshalb wurde bei der Portierung für alle Datentypen, die eine Instanz von `Exception` waren, jeweils eine Wrapper-Klasse in Java geschrieben, die von der Java-Klasse `Exception` erbt und als Attribut den Frege-Datentyp speichert, der als Exception geworfen werden soll. Instanzen der Wrapper-Klassen können dann im Frege-Code als Exception geworfen und abgefangen werden.

Da nun z. B. das Modul `Language.CSPM.Token` sowohl den Datentypen `LexError` definiert, als auch die Wrapper-Klasse `LexErrorException`, die den Typen `LexError` kennen muss, verwendet, wurde die Definition von `LexError` in ein separates Modul ausgelagert, das vor `LexErrorException` kompiliert wird und in `Language.CSPM.Token` öffentlich importiert wird.

Weiterhin gibt es im Parser die Funktion `eitherToExc`, die eine Exception wirft, wenn ihr `Left e` übergeben wird, wobei `e` eine Instanz der Haskell-Klasse `Exception` ist. Weil diese in Frege fehlt, wurde als Ersatz eine Klasse `CspmException` eingeführt, die die Funktion `throwCspm` enthält. Diese Funktion verpackt einen Datentypen in seine native Java-Exception-Klasse und wirft die Exception.

Wie sich die Umsetzung in Frege von dem bisherigen Haskell-Code unterscheidet, kann Listing 18 entnommen werden.

```
import Control.Exception

data LexErr = LexErr {
    lexEPos :: !Int
   ,lexEMsg :: !String
} deriving Show

instance Exception LexErr

eitherToExc :: Exception a => Either a b -> IO b
eitherToExc (Right r) = return r
eitherToExc (Left e)  = throw e

translate :: String -> IO ()
translate str = do
    res <- mainWork str `catch` lexErrHandler
    putStrLn res

mainWork :: String -> IO String
mainWork str = do
    tokenList <- lexer str >>= eitherToExc
    return tokenList

lexer :: String -> IO (Either LexErr String)
lexer str = return $ Left $ LexErr 0 "Err"

lexErrHandler :: LexErr -> IO String
lexErrHandler err = do
    let loc = lexEPos err
    evaluate $ "lexErr␣" ++ show loc
```

```
import Language.CSPM.CspmException

data LexErrException = pure native LexErrException where
    pure native new new :: LexErr -> LexErrException
    pure native get getLexErr :: LexErrException -> LexErr
derive Exceptional LexErrException

data LexErr = LexErr {
    !lexEPos :: Int
   ,!lexEMsg :: String
}
derive Show LexErr

instance CspmException LexError where
    throwCspm :: LexError -> IO a
    throwCspm = throwIO . LexErrorException.new

eitherToExc :: CspmException a => Either a b ->  IO b
eitherToExc (Right r) = return r
eitherToExc (Left e)  = throwCspm e

translate :: String -> IO ()
translate str = do
    res <- mainWork str `catch` lexErrHandler
    putStrLn res

mainWork :: String -> IO String
mainWork str = do
    tokenList <- lexer str >>= eitherToExc
    return tokenList

lexer :: String -> IO (Either LexErr String)
lexer str = return $ Left $ LexErr 0 "Err"

lexErrHandler :: LexErrException -> IO String
lexErrHandler err = do
    let loc = err.get.lexEPos
    evaluate $ "lexErr␣" ++ show loc
```

Listing 18: Exception-Handling in Haskell (links) und die gewählte Umsetzung in Frege (rechts) im Vergleich

3 Integration

Nach der Portierung des CSP$_M$-Parsers wurde Code geschrieben, um die Integration in ProB, die bestehende Build- und Test-Umgebung und die zukünftige JVM-Umgebung, in der alle Parser laufen sollen, zu ermöglichen.

3.1 Optimierungen für den Interaktivmodus

Bei Benutzung des Interaktivmodus, um eine Anfrage auf die aktuell geladene CSP$_M$-Spezifikation durchzuführen, wurden bisher immer folgende Schritte durchgeführt:

1. Füge die Anfrage am Ende der CSP$_M$-Datei an.
2. Parse diesen Quelltext.
3. Führe den Renaming-Prozess auf dem AST durch.
4. Gib die letzte Deklaration im AST als Prolog-Fakt aus.

Das erneute Parsen des kompletten Quelltextes ist aber unnötig, sofern sich dieser seit der letzten Anfrage nicht geändert hat. Wenn die JVM, in der der CSP$_M$-Parser ausgeführt wird, im Hintergrund weiterläuft, kann der beim ersten Laden der Datei erstellte AST gespeichert und beim Ausführen einer interaktiven Anfrage wiederverwendet werden.

Um dies zu ermöglichen, wurde das Modul `TranslateToProlog` um in Wesentlichen zwei Funktionalitäten erweitert: `translateToAst :: String -> IO ModuleFromParser` erhält eine CSP$_M$-Spezifikation als String und gibt den AST zurück. Um Speicherplatz zu sparen werden aus dem AST Informationen, die für den Interaktivmodus nicht relevant sind, entfernt; dazu zählen u. a. die Listen von Tokens und Pragmas und die Positionen der Tokens im Quelltext.

Weiterhin wurde das Funktionenpaar `translateExpToPrologTerm, translateDeclToPrologTerm :: Maybe FilePath -> String > IO ()` um analoge Funktionen `translateExpToPrologTerm', translateDeclToPrologTerm' :: ModuleFromParser -> String -> IO String` ergänzt, die statt einer Eingabedatei und einer CSP$_M$-Expression bzw. Deklaration einen AST erhalten und ihr Ergebnis nicht auf die Standardausgabe schreiben, sondern als `IO String` zurückgeben. Die gestellte Anfrage wird jetzt separat geparst und die eine Deklaration des resultierenden AST wird an den übergebenen AST der restlichen Spezifikation angehängt. Danach werden die Schritte 3 und 4 wie gehabt ausgeführt.

Um die Funktionen aus Java-Code heraus aufzurufen, müssen grundsätzlich die Funktionen aus der von Frege erstellten Java-Klasse aufgerufen werden, dabei sind aber folgende Punkte zu beachten:

- Ein `'` im Frege-Funktionsnamen wird zu `$tick`.
- Funktionsargumente und Rückgabetypen, die keine primitiven Typen sind, sind, da das Typsystem von Java nicht dieselbe Ausdrucksstärke hat wie das von Frege, generisch und müssen explizit gecastet werden.

- Der Rückgabewert ist i. d. R. lazy, d. h. das Ausführen des eigentlichen Codes muss erzwungen werden, um das konkrete Ergebnis zu erhalten.
- Ein im Frege-Module `M` deklarierter Datentyp `D` wird zu einer Klasse `M.TD`.
- Alle auftretenden Exceptions werden in eine `WrappedCheckedException` gewrappt; die Original-Exception erhält man mit `getCause`.

Listing 19 zeigt exemplarisch, wie die Frege-Funktionen `translateToAst` und `translateDeclToPrologTerm'` von Java aus aufgerufen werden können.

```
import frege.language.CSPM.AST.TModule;
import frege.language.CSPM.TranslateToProlog;

class CallFregeExample {

  static void translateDeclExample() {
    try {
      TModule ast = (TModule)evaluateIOFunction(
        TranslateToProlog.translateToAst("spec.csp")
      );
      String term = (String)evaluateIOFunction(
        TranslateToProlog.translateDeclToPrologTerm$tick(ast, "N")
      );
    } catch(frege.runtime.WrappedCheckedException e) {
        System.out.println(e.getCause().getMessage());
    }
  }

  static Object evaluateIOFunction(frege.runtime.Lambda res) {
    return frege.runtime.Delayed.forced(
      frege.prelude.PreludeBase.TST.performUnsafe(
        res
      )
    );
  }

}
```

Listing 19: Aufruf von Frege-Funktionen in einer Java-Klasse

3.2 Buildprozess

Der Frege-Compiler bietet mit der Option `-make` einen Modus an, in dem die gegenseitigen Abhängigkeiten aller übergebenen Module geprüft werden und alle Module in der richtigen Reihenfolge neu kompiliert werden, wenn das Modul oder eine seiner Abhängigkeiten seit dem letzten Übersetzen geändert wurden. Da beim Kompilieren des CSP_M-Parsers aber auch Java-Klassen kompiliert werden müssen, die von Frege-Modulen und

von denen Frege-Module abhängig sind, können nicht einfach alle Frege-Dateien in einem Compileraufruf übersetzt werden. Außerdem erfordern einige Dateien den Aufruf des `DataDerivers`. Zur Automatisierung des Buildprozesses wurde daher ein Makefile geschrieben, welches die Compiler und Tools in der richtigen Reihenfolge aufruft.

Entgegen der üblichen Variante, für jede Quelldatei ein eigenes Target im Makefile zu haben, was einerseits automatische Parallelisierung und andererseits eine Neukompilation nur von geänderten Dateien und Dateien mit geänderten Abhängigkeiten durch `make` ermöglichen würde, wurde nur für jedes Paket ein Target angelegt. Parallelisierung und Vermeidung von unnötigen Neukompilationen werden bereits vom Make-Modus des Frege-Compilers unterstützt und der gesamte Kompilierprozess wird schneller, wenn der Frege-Compiler möglichst viele Dateien gleichzeitig kompiliert, da so Start-up- und Warm-up-Zeiten der JVM, in der der Compiler läuft, wegfallen.

Das Standardtarget ist `cspmf`, was alle Dateien übersetzt und die generierten Dateien im `build`-Verzeichnis speichert. Der CSP$_M$-Parser kann dann mit `./cspmf.built.sh` gestartet werden, was äquivalent zum bisherigen Aufruf von `cspmf` ist.

Weiterhin gibt es ein Target `jar`, welches mithilfe der Anwendung ProGuard eine kompakte jar-Datei erstellt, die alle zur Laufzeit benötigten Klassen enthält, sodass nicht die komplette `frege.jar`, die neben der zur Laufzeit benötigten Klassen auch den Frege-Compiler enthält, zur Ausführung benötigt wird. Der Parser in der jar-Datei wird mit `./cspmf.sh` gestartet.

Schließlich ist es noch möglich, mit `make doc` eine Dokumentation der Module im HTML-Format zu erstellen.

3.3 Überarbeitung der Funktionstests

Die Funktionstests des CSP$_M$-Parsers haben seit einiger Zeit nicht mehr funktioniert, da Funktionen benutzt wurden, die nicht mehr im Code existieren. Daher wurde der Testcode komplett neu geschrieben.

Zum einen gibt es jetzt Tests, die für jede CSP$_M$-Testdatei im Repository Prolog-Code erzeugen und diesen mit einer Referenzdatei, die vom Originalparser erzeugt wurde, vergleichen. Ferner wird nun nicht mehr nur getestet, dass das Einlesen der Ausgabe des Prettyprinters zum ursprünglichen AST führt, sondern auch, ob die Ausgabe des Prettyprinters gleich bleibt, wenn zusätzlich ASCII- durch Unicodesymbole ersetzt (z. B. == durch ≡) und dann wieder zurückersetzt werden. Dazu wurde auch eine zusätzliche Testdatei angelegt, die bereits Unicodesymbole enthält; durch diesen Test konnte ein Fehler im Lexer des CSP$_M$-Parsers in Frege gefunden werden, der auf die in Abschnitt 2.1.3 erwähnten vertauschten Operatorpräzedenzen von `.&.` und + zurückzuführen war. Weiterhin wurde ein Test für das `include`-Statement hinzugefügt, um sicherzustellen, dass das portierte `FilePath`-Modul, dass sich unter Unix und Windows unterschiedlich verhält, auf beiden Plattformen korrekt funktioniert.

Außerdem werden nun auch die Ausgaben der Parser-Optionen `--expressionToPrologTerm` und `--declarationToPrologTerm` getestet. Die Tests werden mit `make`

`test` durchgeführt. Der Code im Github-Repository der Frege-Version[30] wird automatisch auf Travis CI[31] kompiliert und getestet.

Da vor der Fertigstellung der Portierung noch nicht der gesamte Parser getestet werden konnte, wurden während des Portierungsvorgangs bereits einzelne Teile, wie z. B. die Ausgabe des Lexers, mithilfe der Testdateien auf korrekte Funktionsfähigkeit geprüft, sobald diese fertig portiert waren. Da sich die Darstellung von Record-Datentypen in Frege und Haskell unterscheidet, wurde ein Skript geschrieben, welches die beiden Darstellungen vereinheitlicht, sodass die Ausgaben anschließend normal mit `diff` verglichen werden konnten.

[30] `https://github.com/mabre/cspmf/tree/frege`, abgerufen am 17.09.2016
[31] `https://travis-ci.org/mabre/cspmf`, abgerufen am 17.09.2016

4 Ergebnis

Nach der Portierung ist zu betrachten, ob der gesamte Funktionsumfang des CSP$_M$-Parsers in Frege beibehalten werden konnte und wie die Performance im Vergleich zur Haskell-Version ausfällt.

4.1 Funktionalität

Es konnte der gesamte Funktionsumfang der Haskell-Version, inklusive des Command-Line-Interfaces, portiert werden. Das beinhaltet das Parsen von CSP- und FDR2-Dateien und die Ausgabe als Prolog-Code oder XML-Datei. Der einzige sichtbare Unterschied ist die Formatierung und Formulierung von Fehlerausgaben, da z. B. beim Versuch, eine nicht vorhandene Datei zu lesen, die Fehlernachricht der Java-Exception verwendet wird, die anders formuliert ist als die entsprechende Meldung von Haskell. Es ist also möglich, in ProB den Aufruf Programmdatei des Haskell-CSP$_M$-Parsers durch den Aufruf der jar-Datei der Frege-Version zu ersetzen, sofern ProB keine Annahme über das exakte Aussehen von Fehlermeldungen macht.

4.2 Laufzeit

Für den praktischen Einsatz ist nicht nur das gleiche Verhalten in Form des gleichen Funktionsumfangs relevant, sondern vor allem auch die Laufzeit des Parsers, weshalb diese ausführlich verglichen wurden.

Es wurde die Laufzeit des Parser-Aufrufs mit den Parametern `translate --prologOut` getestet. Dabei wurden die Frege- und Haskell-Version jeweils 100-mal mit allen verfügbaren Testdateien aufgerufen und der Mittelwert der Gesamtlaufzeiten der letzten 50 Aufrufe gebildet. Um Class-Loadingzeiten und die Vorteile des Just-in-Time-Compilers der JVM mitzuerfassen, wurden die Wiederholungen bei der Frege-Version in derselben JVM-Instanz ausgeführt. Der erste Durchlauf der Frege-Version ist je nach verwendeter Eingabedatei mehr als 10-mal langsamer als spätere Durchläufe. Die gemessenen Zeiten sind für den vorgesehenen Anwendungsfall realistisch, da die JVM mit dem CSP$_M$-Parser ständig im Hintergrund laufen soll und nicht immer beendet wird, wie es bei der bisherigen Haskell-Version der Fall war.

Das Benchmark wurde mit Frege in Version 3.23.423-gb587ccb und Oracle Java in der Version 1.8.0_101 auf einem 64-Bit Ubuntu-System durchgeführt; beim Kompilieren wurde die Option `-O` verwendet, um Codeoptimierungen zu aktivieren, die die Laufzeit um etwa 10 % verkürzt haben; Java wurde mit 16 MB maximaler Stackgröße und 2 GB maximaler Heapgröße gestartet (das Testsystem hatte 4 GiB Arbeitsspeicher). Die Haskell-Version wurde mit GHC 7.6.3 kompiliert.

An zwei Stellen musste von diesem Vorgehen abgewichen werden: Zunächst konnte die Datei `Rename.fr` aufgrund eines Fehlers im Frege-Compiler[32] nicht mit der Option `-O`

[32] Issue #297: DOUBLE CASTING FATAL error when compiling higher ranked functions with -O, `https://github.com/Frege/frege/issues/297`, abgerufen am 17.09.2016

kompiliert werden. Darüber hinaus stürzte der Parser bei der größten Eingabedatei schedueler0_6.csp mit einem `OutOfMemoryError` der Garbage-Collection ab. Bei Verwendung von Java 9-ea lief der Parser aber fehlerfrei durch, allerdings wurde festgestellt, dass diese Java-Version bei allen anderen Testdateien 10 % bis 15 % langsamer ist.

Es hat sich gezeigt, dass die Frege-Version im Durchschnitt sechsmal langsamer ist als die Haskell-Version. Abbildung 1 zeigt exemplarisch für einige Dateien die Laufzeiten der beiden Programme, wobei die Längen nach Lexing-, Parsing- und Renaming-Zeit (inkl. Aufruf des Prettyprinters) aufgeschlüsselt sind, so wie es bereits im alten, bereits vorhandenen Benchmark-Code gemacht wurde. Unter „Rest“ fallen u. a. die Zeiten bis zum Aufruf des Lexers und das Schreiben des Ergebnisses in die Ausgabedatei. Auffällig ist die verhältnismäßig lange Zeit, die die Frege-Version für das Renaming braucht. Wie eine genauere Betrachtung gezeigt hat, wird ein großer Teil dieser Zeit im Prolog-Prettyprinter verbracht, der direkt nach dem Renaming aufgerufen wird.

Während der Portierung und Benchmarks erster Versionen hat sich herausgestellt, dass vor allem ungünstige Konvertierungen zwischen Strings und Character-Listen Performanceeinbußen zur Folge haben. Zu Beginn der Portierung hat beispielsweise allein das Lexen von abp.csp länger als eine Minute gedauert, was daran lag, dass `toList` sehr oft verwendet wurde. Grundsätzlich sollte Haskell-Code, der viele Listenoperationen wie Konkatenation und List-Patternmatching auf Strings ausführt, so umgeschrieben werden, dass zu Beginn nur einmal mittels `toList` (lazy) oder `unpacked` (nicht lazy, aber meistens schneller) der String in eine Liste und diese dann am Ende wieder per `packed` in einen String konvertiert wird. Durch Umschreiben des Prettyprinters nach eben diesem Muster wurde z. B. die Renaming-Zeit von abp.csp um 60 % verkürzt.

Es wurde weiterhin untersucht, ob Strings, Character-Listen oder Differenzlisten zur internen Darstellung von Zwischenergebnissen im Prettyprinter die beste Performance bieten. Character-Listen waren zwar im Durchschnitt über alle Testdateien nur 3 % schneller als die beiden anderen Varianten, bei größeren Dateien mit mehr als 10.000 Zeichen, bei denen sich der Parser länger im Prettyprinter aufhält, aber mehr als 40 % schneller, weshalb schließlich diese Variante verwendet wurde.

Für Eingabedateien bis zu einer Größenordnung von 30.000 Zeichen mag die gemessene Verarbeitungszeit von etwa 2 Sekunden vertretbar sein. Sollten die in der Praxis verwendeten Dateien aber regelmäßig wesentlich größer sein, so ist es wahrscheinlich sinnvoller, die bisherige Variante zu verwenden, um den Benutzer nicht unnötig lange warten lassen zu müssen. schedueler0_6.csp, die einzige Testdatei, die diese Größe wesentlich überschreitet, ist ein Sonderfall, da es sich um eine automatisch generierte Datei handelt und somit keine typische, handgeschriebene CSP-Spezifikation darstellt, d. h. in der Praxis liegt die Laufzeit des Parsers bei maximal zwei Sekunden, im Durchschnitt bei nicht mehr als einer halben Sekunde.

Es soll darauf hingewiesen sein, dass die Laufzeit nicht direkt mit der Anzahl der Zeichen in einer Datei zusammenhängt, sondern vor allem von der Größe des AST. Die Frege-Version braucht beispielsweise zum Verarbeiten einer Datei, die nur einen 30.000 Zeichen langen Kommentar enthält, durchschnittlich 100 Millisekunden. Da die Zeichenzahl aber in der Regel proportional zur „Komplexität“ der Datei und damit der AST-Größe ist, wurde diese als Kennwert gewählt.

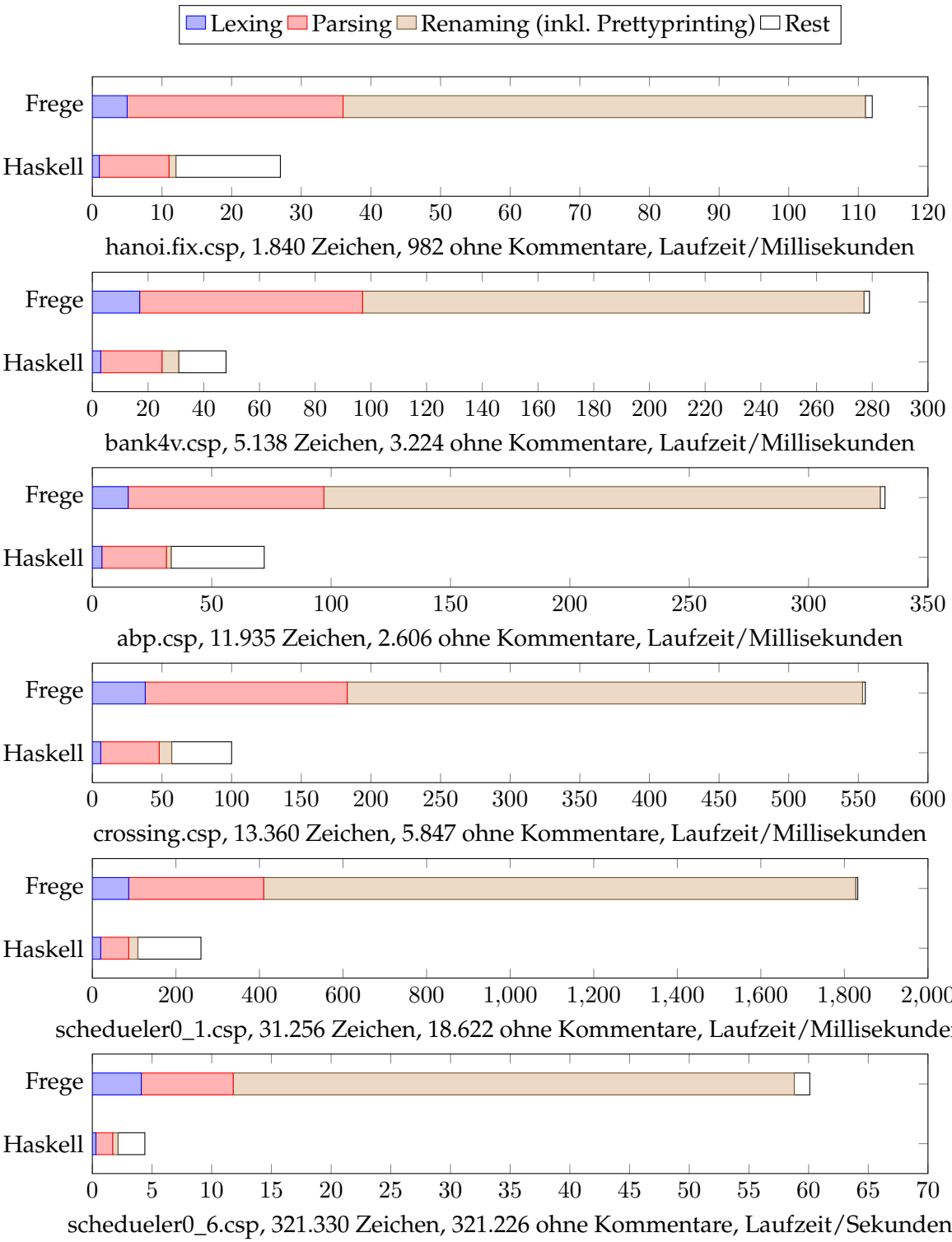

Abbildung 1: Mittlere Laufzeiten des CSP_M-Parsers in Frege und Haskell

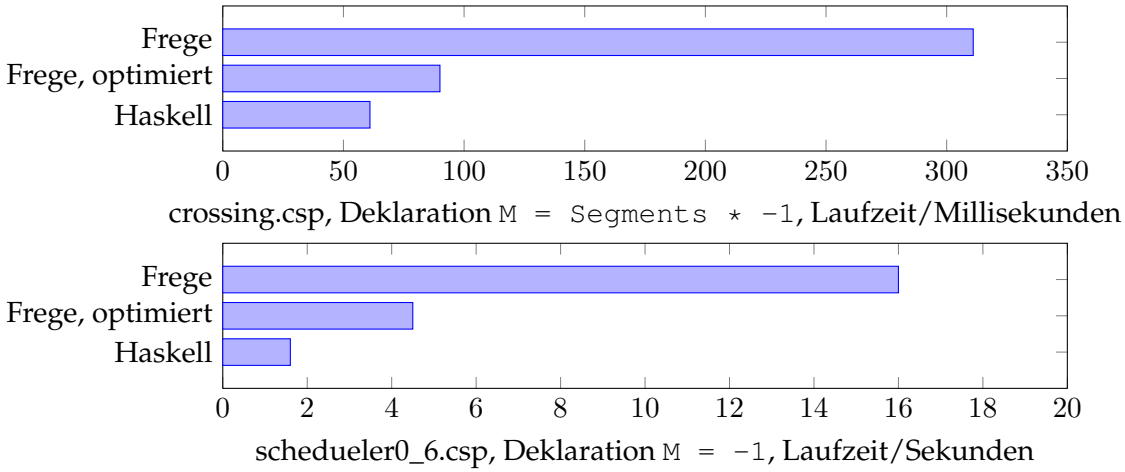

Abbildung 2: Mittlere Laufzeiten im Interaktivmodus

Die Laufzeit der neuen Funktion, eine Deklaration bezüglich eines bereits erstellten AST zu parsen, wurde ebenfalls untersucht. Abbildung 2 zeigt die durchschnittlichen Laufzeiten für das Parsen einer Deklaration bzgl. eines aus einer CSP-Datei erstellten AST, einmal für wiederholtes, vollständiges Parsen der Datei mit Frege, dann für nur einmaliges Parsen der Datei und nur wiederholtes Parsen der Deklaration und schließlich für wiederholtes, vollständiges Parsen der Datei mit der Haskell-Version, jeweils mit Ausgabe des Prolog-Prädikates. Wie man sieht, ist die Frege-Version bei dieser Anwendung nur noch etwa halb so langsam wie die Haskell-Version und, da die Laufzeiten für Dateien üblicher Größen unter 100 ms liegen, in einem Bereich, der von einem Benutzer kaum wahrgenommen wird.

4.3 Speicherverbrauch

Für den praktischen Einsatz ist nicht nur die Laufzeit, sondern auch der Speicherverbrauch relevant. Zum Vergleichen des Speicherbrauchs wurde mithilfe von GNU time der maximale Speicherverbrauch (`maxresident`) für den oben verwendeten Aufruf mit `translate --prologOut` (ohne Wiederholungen) gemessen. Bei Eingabedateien in der Größenordnung 10.000 Zeichen ist der Speicherverbrauch der Frege-Version mit ca. 250 MB etwa 25 mal größer als der der Haskell-Version, die nur 10 MB braucht. Bei der größten Testdatei schedueler0_6.csp lag der Speicherverbrauch der Frege-Version bei etwas mehr als 2 GB (limitiert durch die maximale Größe des Heaps), bei der Haskell-Version bei ca. 250 MB. Im Allgemeinen lässt sich der Speicherverbrauch durch das Verringern der maximalen Heap-Größe der JVM verkleinern, was aber wegen mehr Aufwand für die Garbage-Collection je nach Eingabegröße eine längere Laufzeit zur Folge hat.

Abhängig von der Größe der „üblichen“ CSP-Dateien und der Größe des Arbeitsspeichers der verwendeten Rechner wäre dieser Mehrverbrauch an Speicher nicht hinnehmbar. Insbesondere steigt dann auch durch mehr Aufwand für die Garbage-Collection die Laufzeit. Darüber hinaus wäre es auch nicht tolerierbar, wenn das Programm wegen

zu wenig Arbeitsspeichers nach über einer Minute abstürzt, wohingegen die Haskell-Version nach wenigen Sekunden eine Ausgabe erzeugt hätte. Weil aber weit überdurchschnittlich große, automatisch generierte Eingabedateien die Ausnahme sind und der Speicherverbrauch von 250 MB auf modernen Computer nicht allzu sehr auffällt, ist ein praktischer Einsatz möglich.

Es sei angemerkt, dass diese Ergebnisse sowohl von der verwendeten Frege- als auch Java-Version abhängen.

Die Dateigröße der cspmf.jar ist mit 3,8 MB kleiner als die der von GHC erzeugten Binärdatei cspmf, die auf dem Testsystem 5,2 MB groß ist, weshalb die Frege-Variante nicht mehr Festplattenspeicher benötigt als die Haskell-Version, sofern man den erforderlichen Speicherplatz für die JRE, die sowieso bereits auf vielen Computer verfügbar ist, nicht mitzählt.

5 Fazit

Insgesamt ist das Ergebnis zufriedenstellend: Es war möglich, den gesamten bestehenden Funktionsumfang des in Haskell geschriebenen CSP$_M$-Parsers nach Frege zu portieren, sodass dieser über Java-Schnittstellen eingebunden werden kann. Allerdings steigen in Frege der Speicherverbrauch und vor allem die Laufzeit, insbesondere wenn ungewöhnlich große Dateien verarbeitet werden. Gegenüber der Haskell-Version ist damit die Zuverlässigkeit bei der Verarbeitung großer Dateien geringer, da der Parser wegen zu wenig Stack- oder Heap-Speicher abstürzen könnte.

Sofern auf die zuverlässige Verarbeitung sehr großer Dateien verzichtet werden kann, ist der Einsatz der Frege-Version in ProB sehr wohl möglich. Kann darauf aber nicht verzichtet werden, sollte die bisherige Version weiterverwendet werden, oder, falls eine Java-Lösung bevorzugt wird, der Parser nach Java portiert werden.

5.1 Ausblick

Sofern die Frege-Version in ProB integriert werden soll, müssen als nächstes einfache Java-Methoden geschrieben werden, die die bisher über das CLI bereitgestellten Optionen anbieten und Ergebnisse als String zurückliefern. Dazu müssen z. T. neue Frege-Funktionen geschrieben werden, die die Ergebnisse des Parsers als `IO String` zurückgeben, statt diese in eine Datei oder die Standardausgabe zu schreiben. Anschließend sollten native Java-Methoden geschrieben werden, die diese Frege-Funktionen aufrufen und einen Java-String zurückgeben; als „Referenzimplementierung“ kann dafür der für das CLI und die Benchmarks geschriebene Code dienen.

Nach Behebung einiger Fehler in der Entwicklungsversion von Frege wurde am Ende der Arbeit auch kurz ein Blick darauf geworfen, ob sich die mit Frege 3.24 versprochenen Performance-Verbesserungen durch Nutzung von Java 8 Funktionalitäten und Umbau einiger Frege-Klassen positiv auf den CSP$_M$-Parser auswirken. Leider hat sich herausgestellt, dass die neuere Frege-Version sehr viel langsameren Code für den Parser erzeugt, was beispielsweise für die Testdatei abp.csp bedeutet, dass die durchschnittliche Verarbeitungszeit nicht mehr im Bereich von 300 ms, sondern 2,7 s liegt.

Frege möchte weiter die Kompatibilität zu Haskell steigern, was schon in jüngeren Syntax-Änderungen zum Ausdruck gebracht wurde. Es gibt Überlegungen, einen Haskell-Kompatibilitätsmodus einzuführen,[**?**] der in der Lage ist, unveränderten Haskell-Code zu übersetzen, sodass z. B. Feld-Getter nicht in einem eigenen Namensraum sind (vgl. Abschnitt 2.1.1). Von daher lohnt sich ein Blick auf die weitere Entwicklung von Frege. Auch wurde Frege durch diese Arbeit um einige bekannte Haskell-Module wie SYB und Parsec und nicht zuletzt auch einen Parser für CSP$_M$ reicher, und durch die übermittelten Fehlerberichte um einige Fehler ärmer.

5.2 Arbeit mit Frege

Die kleinen Unterschiede zwischen Haskell und Frege lernt man sehr schnell, sodass das Schreiben von Frege-Code einem erfahrenen Haskell-Programmierer sehr leicht fällt,

und man auch Features wie Regexp-Patterns und den Zugriff auf Java-Standardfunktionen zu schätzen lernt. Dagegen ist die Anpassung von bereits existierendem Code z. T. mit langwierigen Syntaxanpassungen, die sich nicht einfach automatisieren lassen (z. B. Ändern der Syntax zum Zugriff auf Record-Felder, die sich in Haskell nicht von der eines normalen Funktionsaufrufs unterscheidet), verbunden.

Die Frege-Sprachdefinition und das Wiki auf der Projektseite helfen sehr beim Einstieg und dokumentieren die größten Unterschiede zu Haskell, wobei in der Praxis auch nicht oder nicht gut dokumentiere Unterschiede auffallen, auf die man dann, wie bereits erwähnt, mit teilweise eher verwirrenden Fehlermeldungen aufmerksam gemacht wird.[33] Die Meldungen werden vor allem dann kryptisch, wenn der von Frege generierte Java-Code fehlerhaft ist oder der Compiler oder gar das Programm zur Laufzeit abstürzen, was oft wegen der Fehler im Typechecker vor allem beim Portieren von SYB und Parsec passiert ist.[34]

Wenn beim Kompilieren der Java-Compiler eine Fehlermeldung ausgibt, so gibt der Frege-Compiler den allgemeinen Hinweis, dass der Fehler wahrscheinlich von falscher Verwendung von nativen Java-Methoden im Frege-Code ausgelöst wird; diese Aussage war im Verlauf der Portierung nur ein einziges Mal richtig, in allen anderen Fällen handelte es sich um Fehler im Frege-Compiler. Ich möchte mich an dieser Stelle bei Ingo Wechsung, dem Erfinder von Frege bedanken, der auf der Mailingliste und im Bugtracker immer schnell dabei geholfen hat, Probleme des Frege-Compilers zu umgehen und deren Hintergründe zu klären.

Trotz der guten Dokumentation der in Frege verfügbaren Funktionen fällt bei der Suche nach Problemen mit Frege auf, dass Frege eine junge und im Vergleich zu Haskell nicht viel genutzte Sprache ist. Viele Informationen, die man z. B. auf `stackoverflow.com` findet, sind bereits veraltet und zu Problemen mit aktuellen Versionen findet man wenig.

Beim Arbeiten kann man sich auch leicht durch den Frege-Compiler ausgebremst fühlen, da dieser spürbar langsamer als `ghc` oder `javac` ist; für ein Hello-World-Programm muss man bereits etwa drei Sekunden Compilezeit einplanen, wodurch auch ein Herumexperimenten mit der Frege-REPL im Gegensatz zum Arbeiten mit dem Interpreter GHCi eher langsam vonstattengeht. Dies galt insbesondere auch dann, als für Fehlerberichte minimale Testcases erstellt wurden. Wenn man z. B. Code von GHC-Erweiterungen befreit, kann die Arbeit „flüssiger“ sein, wenn man zunächst reinen Haskell-Code ohne GHC-Erweiterungen schreibt und mit GHC testet und erst danach den Quelltext an Frege anpasst.

Nichtsdestoweniger konnte mit dieser Arbeit gezeigt werden, dass sich auch umfangreiche Haskell-Programme nach Frege portieren lassen, sofern man einplant, dass zusätzliche Module portiert und einige GHC-Zusatzfunktionen nachgebildet werden müssen.

[33] Die „schönste“ Meldung, die mir im Laufe der Arbeit angezeigt wurde, war `FATAL: Cant find context for Typeable.Typeable t27487#a. This is a compiler error. Sorry.`, als der Compiler nicht in der Lage war, den Typen einer Funktion korrekt zu inferieren.

[34] Ein Beispiel für eine für Nicht-Frege-Entwickler eher unverständliche Meldung ist `frege.runtime.Undefined: Can't adapt Bind a → a, Func.U<`$\mathcal{A}$`, `$\mathcal{A}$`>, RunTM.<Func.U<CData<`$\mathcal{A}$`>, Func.U<`$\mathcal{A}$`,`$\mathcal{A}$`>>>cast(arg$1).apply(Thunk.<CData<`$\mathcal{A}$`>>lazy(ctx$1)).call() to Func.U<CData<Object>, Func.U<Object, Object>> because `$\mathcal{A}$` does not match CData<Object>`, was laut Erklärung auf der Mailingliste ein ausführliches „cannot happen“ ist und auf schwere Fehler in der Java-Codegenerierung hindeutet.

Literatur

[GHC16] GHC TEAM: *GHC Users Guide Documentation*, Mai 2016. (8.0.1)

[LB08] LEUSCHEL, Michael ; BUTLER, Michael: ProB: An Automated Analysis Toolset for the B Method. In: *Software Tools for Technology Transfer (STTT)* 10 (2008), Nr. 2, S. 185–203

[Lip11] LIPOVACA, Miran: *Learn You a Haskell for Great Good!* San Francisco : No Starch Press, 2011

[LJ03] LÄMMEL, Ralf ; JONES, Simon P.: Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In: *ACM SIGPLAN international workshop on Types in languages design and implementation* TLDI '03 (2003)

[LYBB15] LINDHOLM, Tim ; YELLIN, Frank ; BRACHA, Gilad ; BUCKLEY, Alex: *The Java® Virtual Machine Specification Java SE 8 Edition*. `https://docs.oracle.com/javase/specs/jvms/se8/html/index.html`. Version: Februar 2015, Abruf: 25.05.2016

[Mar10] MARLOW, Simon: *Haskell 2010 Language Report*. `https://www.haskell.org/onlinereport/haskell2010/`. Version: April 2010, Abruf: 25.05.2016

[OGS09] O'SULLIVAN, Bryan ; GOERZEN, John ; STEWART, Donald B.: *Real World Haskell*. Sebastopol : O'Reilly Media, Inc., 2009

[RHB97] ROSCOE, A. W. ; HOARE, C. A. R. ; BIRD, Richard: *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1997. – ISBN 0136744095

[Wec14] WECHSUNG, Ingo: *The Frege Programming Language (Draft)*. `http://www.frege-lang.org/doc/Language.pdf`. Version: Mai 2014, Abruf: 23.05.2016

Abbildungsverzeichnis

Tabellenverzeichnis

Listings