Diplomarbeit

Implementierung einer Abstrakten Maschine zu einem nebenläufigen Programmkalkül mit Futures und Exceptions in Haskell

Tobias Gordon Leppin

30. September 2013

Eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Lehrstuhl für Künstliche Intelligenz und Softwaretechnologie

Fachbereich Informatik und Mathematik
Johann Wolfgang Goethe Universität
Frankfurt am Main

Erklärung gemäß DPO § 11 Abs. 11

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen oder Hilfsmittel verwendet habe.

Kriftel, den 30. September 2013

Tobias Gordon Leppin

Inhaltsverzeichnis

1 Einleitung

Als Johann Wolfgang v. Goethe einst die Zeilen „Herr, die Not ist groß! | Die ich rief, die Geister | werd' ich nun nicht los“ in seinem Gedicht *Der Zauberlehrling* schrieb, beklagte er damit eine Schar von Besen, die zu der endlosen Aufgabe verzaubert waren, unheilvolle Wassermassen herbeizutragen. Erst der herbeigerufene Meister vermochte diesem Treiben wieder Einhalt zu gebieten.

Dieses Zitat hat sich seitdem im Volksmund zu einem geflügelten Wort für Situationen etabliert, die außer Kontrolle geraten sind, und oft eines externen Events bedürfen, um einem eingetretenen Fehler beizukommen.

Die unglücklich ungenaue Ausdrucksweise des Zauberlehrlings führt hier dazu, dass der verzauberte Besen nicht nur ein– oder zweimal Wasser holt, sondern dies unendlich oft fortzusetzen gedenkt. Hätte man vom Zauberlehrling verlangen müssen, diesen Fall vorherzusehen, und in seinem Zauberspruch zu berücksichtigen? Lassen sich überhaupt alle Fehler antizipieren?

Betrachtet man abseits dieses metaphysischen Beispiels die konkrete Welt der Programmiersprachen, führen Fehler dort oft zu falschen oder gar keinen Ergebnissen. Der Umgang mit Fehlern ist also wesentlich, und es ist eine Möglichkeit gefordert, sowohl auf antizipierbare, als auch auf unvorhersehbare Fehler reagieren zu können, ohne das Ergebnis eines Programms nachhaltig gefährden zu müssen.

Speziell mit Einführung der nebenläufigen Programmierung fällt es leicht, ein Programm in einzelne Prozesse zu gliedern. Diese Prozesse strukturieren und optimieren die Ausführung des Programms bestenfalls, indem sie voneinander unabhängige Aktionen z.T. parallel durchführen. Dabei können sie sich aber auch untereinander austauschen oder synchronisieren. Die Gefahr, solchen — mit verzauberten Besen vergleichbaren — Prozessen versehentlich eine fehlerhaft formulierte Berechnung mitzugeben, ist groß. Im schlimmsten Fall kann es dazu kommen, dass die Berechnung nie endet und womöglich den knappen Speicher mit unheilvoll unbrauchbaren Datenmengen bis zum „Überlaufen“ füllt. Spätestens jetzt benötigt die Programmiersprache die Fähigkeit, diesen nie endenden Prozess von außen — von Meisterhand — zu beenden.

Die Fähigkeit der Prozesse, sich zu synchronisieren, untereinander auszutauschen und auf gemeinsame Daten zuzugreifen, eröffnet dabei weitere Schwierigkeiten. Eine Programmiersprache muss nicht nur in der Lage sein, auftretende Fehler zu signalisieren oder Prozesse jederzeit beenden zu können, sondern sich außerdem der auftretenden kausalen

Probleme annehmen können. Hierbei gilt es z.B. der Korruption der Datenbasis vorzubeugen oder kaskadierte Berechnungen abzubrechen, die auf möglicherweise fehlerhaften Ergebnissen des gerade beendeten Prozesses basieren.

Ziel dieser Arbeit ist es, einer funktionalen Programmiersprache das Instrumentarium anheim zu stellen, auftretende Fehler zu signalisieren und angemessen darauf reagieren zu können. Dieses Instrumentarium kann außerdem verwendet werden, einen extern abgebrochenen Prozess derart aufzufangen, dass das Ergebnis des Programms nicht gefährdet werden muss. Um dies zu gewährleisten muss ein Kompromiss gefunden werden zwischen dem Aufwand, der einem Anwender beim sorgfältigen Programmieren zuzumuten ist, und den Möglichkeiten, die der Programmiersprache selbst zur Verfügung gestellt werden können.

In Kapitel 2 wird dafür zunächst der Lambda-Kalkül als eine sehr einfache funktionale Programmiersprache eingeführt. Stückweise erweitert um Typsystem, Datenkonstruktoren und Fallunterscheidungen, Seiteneffekte und nebenläufige Prozesse in Form von Futures, entsteht daraus die getypte funktionale Programmiersprache Concurrent Haskell mit Futures. Das Kapitel schließt mit der formalen Angabe von Syntax und Semantik dieser Sprache.

Eine spezielle Form von Fehlern, Exceptions, werden in Kapitel 3 eingeführt. Die Unterscheidung in synchrone und asynchrone Exceptions hilft bei der Betrachtung lokal auftretender Fehler und extern induzierter Events. Neben den notwendigen Erweiterungen einer Sprache zum Umgang mit Exceptions werden außerdem die entstehenden Problematiken diskutiert.

Kapitel 4 schließlich vereinigt Futures und Exceptions in der neuen Sprache Concurrent Haskell mit Futures und Exceptions. Die dazu notwendigen Anpassungen an Syntax und Typsystem werden definiert und die Semantik in Form einer Abstrakten Maschine vorgestellt. Die Verwendung Abstrakter Maschinen zur Angabe der operationalen Semantik an dieser Stelle führt zu einer höheren Genauigkeit und liefert dabei bereits direkt einen Interpreter für die entstandene Sprache. Dadurch bietet sich außerdem die Möglichkeit zu weiterführenden Analysen der Semantik.

Hinweise zur Implementierung in Haskell und Programmbeispiele folgen in Kapitel 5.

2
Concurrent Haskell mit Futures

In diesem Kapitel wird ein grober Überblick über die Grundzüge funktionalen Programmierens gegeben sowie die Kernsprache *Concurrent Haskell with Futures*, kurz *CHF*, vorgestellt.

In Abschnitt 2.1 wird dafür zunächst der klassische Lambda-Kalkül mit seiner Syntax und Semantik eingeführt. Abschnitt 2.2 beschreibt eine erste Kernsprache der funktionalen Programmiersprache Haskell, indem der Lambda-Kalkül um Namenszuweisungen, Konstruktoren, Fallunterscheidungen und die Möglichkeit der strikten Auswertung erweitert wird. Das dafür notwendige Typsystem wird in Abschnitt 2.3 informativ skizziert. Einen Überblick über nebenläufige Programmkalküle mit den dazu notwendigen Konstrukten zur Synchronisierung und Kommunikation mehrerer Prozesse wird in Abschnitt 2.4 gegeben. Insbesondere wird hier auch das Konzept sog. Futures eingeführt, dem Hauptmerkmal des *CHF*–Kalküls. Zum Schluss wird in Abschnitt 2.5 die Syntax und operationale Semantik von *CHF* eingeführt.

2.1 Funktionales Programmieren: Der Lambda-Kalkül

Neben den bekannten, größtenteils imperativen Programmiersprachen wie Java, *C*, *C#*, etc. existieren auch sog. Funktionale Programmiersprachen, deren Ursprung auf den von Alonzo Church entwickelten Lambda-Kalkül [Chu36] zurückgehen. Grundsätzliches Programmierparadigma ist hierbei die *Abstraktion*, das Definieren von Funktionen über einer Menge V von (unendlich vielen) Variablen und das, *Applikation* genannte, Anwenden von diesen Funktionen auf Argumente. Die Syntax von Ausdrücken E im Lambda-Kalkül lässt sich damit wie in [PJ87] als kontextfreie Grammatik der folgenden Form definieren:

$$E ::= V \mid (E_1\, E_2) \mid \lambda V.E$$

Hierbei bezeichnet $(E_1\, E_2)$ die Anwendung des Ausdrucks E_1 auf das Argument E_2, also die Applikation. Eine Abstraktion wird durch einen Lambda-Binder λ und die Menge der formalen Parameter der Funktion, gefolgt von einem Punkt und dem eigentlichen Funktionsterm dargestellt.

Die Identitätsfunktion, die ein Argument auf sich selbst abbildet, ließe sich damit beispielsweise durch $\lambda x.x$ mit $x \in V$ definieren. In ihrem Funktionsterm kommt nur der durch

$FV(x)$	$=$	x	$BV(x)$	$=$	$\emptyset$
$FV(\lambda x.s)$	$=$	$FV(s) \setminus \{x\}$	$BV(\lambda x.s)$	$=$	$BV(s) \cup \{x\}$
$FV(s\ t)$	$=$	$FV(s) \cup FV(t)$	$BV(s\ t)$	$=$	$BV(s) \cup BV(t)$

(a) freie Variablen (b) gebundene Variablen

Abbildung 2.1: Rekursive Definition (a) der Menge der freien Variablen FV und (b) der Menge der gebundenen Variablen BV im λ–Kalkül [Plo75].

λx gebundene, formale Parameter x vor, weswegen x auch als *gebundene* Variable bezeichnet wird. Kommen in einem Funktionsterm noch andere, nicht gebundene Variablen vor, so werden diese als *freie* Variablen bezeichnet. Eine intuitive Definition der Menge der freien und gebundenen Variablen eines Lambda-Ausdrucks findet sich in Abbildung 2.1.

Damit der Lambda-Kalkül als Programmiersprache betrachtet werden kann, bedarf es zusätzlich zur vorgestellten Syntax noch einer *operationalen Semantik*, die es ermöglicht, die Lambda-Ausdrücke Schritt für Schritt mittels definierter Regeln zu transformieren und somit auszuwerten.

2.1.1 Operationale Semantik des Kalküls

Maßgeblich zur schrittweisen Auswertung eines Lambda-Ausdrucks sind die sogenannte α– und β–Reduktion.

Möchte man ein Argument auf eine Funktion anwenden, so muss im Funktionsterm der Abstraktion jedes Vorkommen der gebundenen Variable, d.h. des formalen Parameters der Funktion, durch das einzusetzende Argument ersetzt werden. Für diese Substitution wird die Notation $t[s/x]$ verwendet, um anzudeuten, dass in einem Ausdruck t alle Vorkommen der Variable x durch den Ausdruck s ersetzt werden. Damit lässt sich die *β–Reduktion* definieren als

$$(\lambda x.t)\ s \xrightarrow{\beta} t[s/x]$$

Da im Lambda-Kalkül nicht nur Variablen, sondern auch ganze Abstraktionen als Argumente zulässig sind, kann es nun durchaus vorkommen, dass man sich nach Anwendung der β–Reduktion Variablen aus den Argumenten „einfängt“, deren Namen mit bereits frei im Ausdruck vorkommenden Variablen kollidieren. Zur Vermeidung dieses Falles sollte es möglich sein, Variablen umbenennen zu dürfen.

Die *α–Reduktion* (strenggenommen: α–Konversion) erlaubt es, gebundene Variablen in einem Ausdruck beliebig umzubenennen, solang dies konsistent geschieht, und der neue Variablenname nicht bereits frei im Ausdruck vorkommt. Formal lässt sich das durch

$$\lambda x.t \xrightarrow{\alpha} \lambda y.t[y/x] \text{ und neue Variable } y \notin FV(t)$$

ausdrücken [PJ87, S. 23]. Da die Menge V der Variablen unendlich groß ist, ist es möglich, derartige Namenskonflikte bei der Auswertung vollständig durch entsprechende Umbenennungen zu vermeiden. Im Folgenden wird daher angenommen, dass α–Umbenennungen zur Vermeidung von Namensüberschneidungen automatisch immer dann durchgeführt werden, wenn sie notwendig werden. Insbesondere bedeutet das, dass die *Distinct Variable Convention* (kurz: *DVC*), die fordert, dass in einem Ausdruck alle gebundenen Variablen unterschiedliche und stets von freien Variablen verschiedene Namen haben, immer gilt [SS11, S. 17].

2.1.2 Auswertungsstrategien

Nachdem mit der β–Reduktion ein Werkzeug zum Reduzieren von Lambda-Ausdrücken vorhanden ist, ist zu klären, an welchen Stellen sie in einem Ausdruck anzuwenden ist. Beispielsweise bietet der Ausdruck $(\lambda x.x)\,(\lambda y.(\lambda z.z)\,y)$ unterschiedliche Möglichkeiten zur Reduktion. Diese Möglichkeiten werden *Redex* (von *Reducible Expression*) genannt. Der obige Ausdruck ließe sich zu $(\lambda x.x)\,(\lambda y.y)$ oder aber zu $(\lambda y.(\lambda z.z)\,y)$ reduzieren.

Die Art und Weise, an welcher Stelle reduziert wird, wird als *Auswertungsstrategie* beschrieben. Unter anderem werden hier die Strategien *call-by-name* [Plo75] und *call-by-need* [MO98] unterschieden[1].

Die klassische Auswertungsstrategie im Lambda-Kalkül ist die, auch *Normalordnungsreduktion* genannte, *call-by-name*–Auswertung. Sie sucht immer den am weitesten links und außen stehenden Redex und reduziert diesen durch Einsetzen der Argumente. Oft wird dies durch Reduktionskontexte R dargestellt, die aus einer Grammatik mit einem „Loch“ $[\cdot]$ bestehen, und den nächsten Redex eines Ausdrucks E finden [MO98, S. 277]:

$$R ::= [\cdot] \mid (R\,E)$$

Die einzusetzenden Argumente werden bei der Normalordnungsreduktion, so wie sie sind, in den Funktionsterm eingesetzt. Betrachtet man z.B. einen Term in der Form $\big((\lambda x.x\,x\,x)\;E_2\big)$, fällt auf, dass nach Reduktion in diesem Fall das Argument E_2 mehrfach vollständig eingesetzt wurde. Handelt es sich bei E_2 selbst um einen komplexen Lambda-Term, so muss dieser nun (möglicherweise) dreimal ausgewertet werden. Da es sich allerdings um jeweils den gleichen Term handelt, ist es wünschenswert, diesen, wenn er an der Reihe ist, nur einmal auszuwerten, und das Ergebnis anschließend mit allen Stellen zu „teilen“, an denen dieser Term abermals auftaucht. Auf diese Weise können unnötige Mehrfachauswertungen vermieden werden. Dieses *Sharing* genannte Teilen ist die Grundlage der *call-by-need*–Auswertungsstrategie, die somit eine Optimierung der Normalordnungsreduktion darstellt[2].

2.1.3 Terminieren in Normalform

Mit den bisher vorgestellten Reduktionsregeln und Auswertungsstrategien lassen sich λ-Terme nun deterministisch umformen und im besten Fall so lange reduzieren, bis kein weiterer Redex mehr vorhanden ist, womit die sog. *Normalform* erreicht ist. Da der obige Reduktionskontext der Normalordnungsreduktion immer den am weitesten links und außen stehenden Redex sucht, kann es vorkommen, dass ein Ausdruck so weit reduziert wird, dass es keinen äußeren Redex mehr gibt, obwohl es noch „innere“ Redexe geben könnte, wie z.B. in $\lambda x.\big((\lambda y.y)\,a\big)$. In diesem Fall ist die sogenannte *schwache Kopfnormalform* (im Folgenden WHNF von engl. *weak head normal form* genannt) erreicht und der Ausdruck ist unter Normalordnung nicht weiter reduzibel. Die Normalordnungsreduktion terminiert in diesem Fall [PJ87, S. 23, 197–200]. Umformungen die über eine WHNF als Ergebnis hinausgehen, z.B. zu einer Normalform, werden in dieser Arbeit nicht verlangt.

Mittels der Church-Rosser–Theoreme [PJ87, S. 24 f.] kann gezeigt werden, dass jeder Ausdruck s, der über beliebige Reduktionen zu einer WHNF transformiert werden kann,

[1] Es existieren noch andere Auswertungsstrategien, wie z.B. *call-by-value*, die für diese Arbeit allerdings nicht relevant sind. Bei dieser Strategie werden zunächst die Argumente einer Funktion ausgewertet und anschließend angewandt [PJ87, S. 193].

[2] Die *call-by-need*–Auswertungsstrategie wird hier der Vollständigkeit halber bereits erwähnt, da sie die Grundlage der vorzustellenden Sprache *CHF* darstellt. Für eine verständliche Implementierung wird der λ-Kalkül in den folgenden Kapiteln zunächst noch um einige Konstrukte erweitert.

auch allein über Normalordnungsreduktionsschritte in diese überführt werden kann. Man sagt, der Ausdruck *konvergiert* und schreibt $s \Downarrow$. Außerdem wird damit ausgesagt, dass unterschiedliche Auswertungsfolgen nicht zu unterschiedlichen Normalformen führen.

Offenbar existieren auch Ausdrücke, die keine WHNF haben, in die sie überführt werden können. Betrachtet man bspw. den Ausdruck $\Omega \stackrel{\text{def}}{=} (\lambda x.x\,x)\,(\lambda x.x\,x)$, ist leicht zu sehen, dass eine β–Reduktion wieder den Ausgangsterm herleitet. Eine Reduktion im Sinn einer „Verkleinerung“ des Ursprungsterms findet nicht statt. Für Ausdrücke s, die sich (wie Ω) nicht reduzieren lassen, sondern sich schlimmstenfalls durch β–Reduktionen möglicherweise unendlich vergrößern, terminiert die Normalordnungsreduktion nicht. Derlei Ausdrücke werden als *divergierend* bezeichnet, und man schreibt $s \Uparrow$ oder auch $s = \bot$, womit angedeutet wird, dass s keine WHNF und keine Normalform hat [SS11, PJ87, S. 31].

Wenn ein Ausdruck also eine WHNF hat, konvergiert er; ob ein Ausdruck eine WHNF hat oder aber möglicherweise divergiert, ist hingegen nicht entscheidbar. Diese Tatsache wird allgemein als *Halteproblem* bezeichnet und wurde durch Alan Turing in [Tur36] nachgewiesen.

2.2 Erweiterung des Lambda-Kalküls

Obwohl der Lambda-Kalkül turingmächtig[3] ist, ist es sowohl umständlich als auch unübersichtlich, komplexere Programme allein damit zu entwickeln. Man erweitert ihn daher um weitere Primitiven, Datentypen und ein Typsystem, um eine umgänglichere Programmiersprache zu erhalten. Eine solche, mittlerweile stark vorangeschrittene Erweiterung des Lambda-Kalküls ist z.B. die Sprache Haskell [Mar10].

Abgesehen von zahlreichen Erweiterungen, wie z.B syntaktischen Vereinfachungen zum angenehmeren Programmieren, besteht Haskells Funktionsumfang, die sog. Kernsprache, allerdings aus einem erweiterten Lambda-Kalkül mit *call-by-need*–Auswertung.

Zunächst werden die für diese Arbeit notwendigen Erweiterungen des Lambda-Kalküls, wie sie auch für *CHF* benötigt werden, informativ beschrieben. Die zugehörige Syntax und Semantik folgen in Kapitel 2.5.

2.2.1 Definitionen mit `letrec`

Mithilfe von `letrec` lassen sich im erweiterten Lambda-Kalkül Namen für ganze Ausdrücke definieren. Einer Variablen $v \in V$ kann mit dem Konstrukt $\texttt{letrec}\ v = E_1\ \texttt{in}\ E$ der Ausdruck E_1 zugewiesen werden. Es entsteht eine neue Variablenbindung. Bei der Auswertung des eigentlichen Funktionsausdrucks E wird nun jedes Vorkommen von v durch E_1 ersetzt.

Eine Besonderheit von `letrec` stellt seine Rekursivität dar, d.h. für $n \in \mathbb{N}$ kommen die neu gebundenen Variablen $v_1 \dots v_n \in V$ im `letrec`–Konstrukt

$$\begin{array}{rlcl} \texttt{letrec} & v_1 & = & E_1 \\ & v_2 & = & E_2 \\ & & \dots & \\ & v_n & = & E_n \\ \texttt{in} & E & & \end{array}$$

im Bindungesbereich (dem Skopus) der $E_1 \dots E_n$ und in E gebunden vor. Daher sind

[3] Es lassen sich alle Programme im Lambda-Kalkül codieren, die auch auf einer universellen Turingmaschine programmierbar sind [Tur37].

rekursiv ineinander verschachtelte Aufrufe wie z.B.

$$\begin{array}{rrcl} \texttt{letrec} & v_1 & = & \ldots v_1 \ldots v_2 \ldots \\ & v_2 & = & \ldots v_1 \ldots \\ \texttt{in} & \ldots & & \end{array}$$

durchaus möglich. Hier referenziert v_1 sich selbst und v_2, und v_2 referenziert wieder v_1 [PJ87, S. 40–43]. Damit liefert `letrec` die Möglichkeit, wiederkehrende, gleiche Terme der Übersicht halber zu substituieren, und insbesondere implizit die Möglichkeit, benannte Funktionen, wie man sie aus anderen Programmiersprachen kennt, zu definieren. Die Identitätsfunktion $id(x) = x$ mit Parameter x lässt sich nun z.B. durch `letrec` $id = \lambda x.x$ `in` ... im erweiterten Lambda-Kalkül zur Verfügung stellen.

2.2.2 Mehr als Variablen: Konstruktoren

Der ursprüngliche ungetypte Lambda-Kalkül kennt ausschließlich Variablen und Funktionen. Bequemer und aus gängigen Programmiersprachen bekannt ist es, Daten direkt ablesen oder codieren zu können. Über sog. *Konstruktoren* können daher im erweiterten Lambda-Kalkül Daten wie z.B. die Boolean-Werte `True`, `False`, Listen, Bäume, etc. eingeführt werden. In *CHF* werden sie ähnlich einer Grammatik definiert und einem auszuwertenden Programm vorangestellt. So werden bspw. wie in [Sab11] mit

$$\begin{array}{rcl} \texttt{Bool} & = & \texttt{True} \mid \texttt{False} \\ \texttt{ListBool} & = & \texttt{ConsBool Bool ListBool} \mid \texttt{NilBool} \end{array}$$

der Datentyp `Bool` für die Booleschen Wahrheitswerte `True` und `False`, sowie eine Liste von Daten genau dieses Typs definiert. Man unterscheidet hierbei die zu definierenden *Typkonstruktoren*, die anzeigen, von welchem Typ die Daten sind, und die eigentlichen *Datenkonstruktoren*, mit denen die Daten im Programm „erzeugt“ werden. Im obigen Beispiel sind `Bool` und `ListBool` Typkonstruktoren, und `True`, `False`, `ConsBool`, `NilBool` die Datenkonstruktoren.

Datenkonstruktoren haben eine feste Stelligkeit (engl. *arity*). Die Datenkonstruktoren `True`, `False` und `NilBool` sind nullstellig, wohingegen der Konstruktor `ConsBool` zweistellig ist. Er verlangt zwingend erst ein Listenelement vom Typ `Bool` und anschließend die „Restliste“ vom Typ `ListBool`, womit dieser Datentyp rekursiv definiert ist.

Mithilfe dieser Definitionen kann man im erweiterten Lambda-Kalkül nun z.B. eine dreielementige Liste [`True`, `False`, `True`] vom Typ `ListBool` mit (ausschließlich) Elementen vom Typ `Bool` definieren durch

(`ConsBool True` (`ConsBool False` (`ConsBool True NilBool`)))

wobei `NilBool` die leere Liste darstellt, bzw. hier das Ende der Liste markiert.

Die allgemeine Syntax zum Definieren eines Datentyps T kann nach [PJ87, S. 55–56] angegeben werden mit $T ::= c_1\ T_{1,1} \ldots T_{1,\mathrm{ar}(c_1)} \mid \ldots \mid c_n\ T_{n,1} \ldots T_{n,\mathrm{ar}(c_n)}$, wobei die $T_{i,j}$ Typen und die c_i Datenkonstruktoren mit Stelligkeit $\mathrm{ar}(c_i)$ sind.

2.2.3 Fallunterscheidungen mit `case`

Arbeitet man mit den oben definierten Datentypen, ist es hilfreich, auf unterschiedliche Werte unterschiedlich reagieren zu können. Wenn bei der Reduktion einer Funktion bspw.

```
letrec
  list    = (ConsBool True (ConsBool True NilBool))
  isEmpty = λx.case_ListBool x of
                NilBool        → True
                ConsBool y ys  → False
in (isEmpty list)
```

Abbildung 2.2: Beispiel eines λ–Ausdrucks der mittels `letrec` die Liste $list = [\texttt{True}, \texttt{True}]$ und eine Funktion $isEmpty(x)$ definiert, die bei leerer Liste `True` und andernfalls `False` zurückliefert. Das Ergebnis dieses Lambda-Ausdrucks ist also insgesamt `False`, da die Liste nicht leer ist.

`False` anstatt `True` als Resultat entsteht, sollte es möglich sein, mithilfe einer Fallunterscheidung die weitere Auswertung des Programms zu beeinflussen. Diese Fallunterscheidungen werden im erweiterten Lambda-Kalkül durch das `case`–Konstrukt ermöglicht. Die allgemeine Syntax eines `case`–Ausdrucks lässt sich wie folgt beschreiben [SSS11]:

$$
\begin{array}{lcl}
\texttt{case}_T\ v\ \texttt{of} & & \\
\quad c_1\ x_1 \ldots x_{\mathrm{ar}(c_1)} & \rightarrow & E_1 \\
\quad \ldots & & \\
\quad c_{|T|}\ x_1 \ldots x_{\mathrm{ar}(c_{|T|})} & \rightarrow & E_{|T|}
\end{array}
$$

Das an das `case` angehängte T symbolisiert, dass es sich hierbei bereits um ein „getyptes `case`“ handelt. Bei v muss es sich also um eine Variable vom Typ T handeln. Nachfolgend werden alle möglichen Konstruktoren des Typs T mit entsprechend seiner Stelligkeit vielen Variablen x_i als sog. *Case-Pattern* (dt. Muster) aufgeführt.

Bei der Auswertung des `case`-Ausdrucks wird nun der Wert der Variable v mit den vorhandenen Pattern verglichen. Stimmt ein Case-Pattern mit dem Wert (dem Datenkonstruktor) von v überein, wird der zugehörige Ausdruck E_i als neuer Auswertungszweig angesehen und die zum Konstruktor gehörenden Variablen x_i werden an den Ausdruck E_i gebunden. Sie können darin weiter (z.B. über ein weiteres `case`) ausgewertet werden [PJ87, S. 74–76].

Die Abbildung 2.2 zeigt exemplarisch, dass man mit den bisher vorgestellten Konstrukten bereits intuitiv mächtige Programme im erweiterten Lambda-Kalkül formulieren kann.

2.2.4 Strikte Auswertung mit `seq`

Wie in Abschnitt 2.1.3 gezeigt, wendet die Normalordnungsreduktion Funktionen auf unausgewertete Argumente an. Sie werden erst dann ausgewertet, wenn sie auch tatsächlich benötigt werden, bzw. niemals, falls sie nie benötigt werden. Man spricht in diesem Fall auch von *nicht-strikter* Auswertung. Um dem Anwender Kontrolle über die Auswertungsreihenfolge zu geben, diese also *strikt* zu machen, wird das `seq`–Konstrukt mit der Definition aus [JV06] eingeführt:

$$
\begin{array}{lcl}
\texttt{seq}\ \bot\ b & = & \bot \\
\texttt{seq}\ a\ b & = & b, \text{ wenn } a \neq \bot
\end{array}
$$

Zuerst wird das erste Argument von `seq` bis zu einer WHNF ausgewertet, und anschließend das Ergebnis des zweiten Arguments (nach dessen Auswertung) zurückgeliefert.

Hilfreich kann `seq` z.B. beim Sparen von Speicherkapazität bei manchen rekursiven Funktionen sein. Es lässt sich mit `seq` sicherstellen, dass die Argumente direkt ausgewertet

werden, anstatt durch die Rekursion zunächst einen (möglicherweise sehr) großen Ausdruck im Speicher zu erzeugen, der anschließend als Ganzes ausgewertet werden muss.

2.3 Typsystem und Typchecker

In Abschnitt 2.2.2 wurden Typ– und Datenkonstruktoren vorgestellt. Dabei lässt sich jeder Konstruktor genau einem Typ zuordnen. Die gängige Notation hierfür lautet $t :: \tau$ und bedeutet, dass t den Typ τ hat, wobei t ein Datenkonstruktor, eine Variable und sogar eine Funktion darstellen kann. Ein Funktionstyp wird dabei durch $\tau_1 \to \tau_2$ dargestellt, womit angedeutet wird, dass die Funktion einen Parameter vom Typ τ_1 erhält, und als Ergebnis einen Wert vom Typ τ_2 generiert.

Für die in Abb. 2.2 verwendeten Konstrukte lässt sich mit dieser Notation z.B. feststellen, dass `True` :: `Bool`, `ConsBool` :: `ListBool` und *isEmpty* :: `ListBool` → `Bool` gilt.

In dieser Arbeit werden ausschließlich *monomorphe Typsysteme* betrachtet. Im Unterschied zu einem polymorphen Typsystem ist es hierbei nicht möglich, allgemeine Datentypen mit Typvariablen zu definieren. In einem monomorphen Typsystem muss, wie oben gezeigt, eine Liste von Booleschen Werten explizit als `ListBool` definiert werden. Diese Liste kann dann ausschließlich Werte vom Typ `Bool` enthalten. Hat man noch weitere Datentypen wie z.B. natürliche Zahlen vom Typ `Integer` o.ä., und möchte auch diese in einer Liste verwalten, so muss in gleicher Weise auch eine Liste `ListInt` definiert werden. Es ist also in einem monomorphen Typsystem nicht möglich, einen allgemeinen Listentyp der Form

$$\texttt{List}\ \alpha = \texttt{Cons}\ \alpha\ (\texttt{List}\ \alpha) \mid \texttt{Nil}$$

zu definieren, der dann polymorph durch Einsetzen der Typen `Bool` oder `Integer` für die Typvariable α zu einer Liste des entsprechenden Typs instantiiert werden kann. Die Einschränkung der Betrachtung auf ein monomorphes Typsystem wird, der Argumentation in [SSS11] folgend, aus Gründen der Überschaubarkeit vorgenommen, wobei angenommen wird, dass die vorgestellten Systeme leicht zu polymorphen Typsystemen erweitert werden könnten.

2.3.1 Typisierungsregeln

Eine der Aufgaben des *Typcheckers* ist es, zu überprüfen, ob ein Lambda-Ausdruck *wohltypisiert* ist. So wäre es z.B. bei der Auswertung eines `case`–Ausdrucks fatal, wenn in der Fallunterscheidung alle Konstruktoren des Typs `Bool` abgeprüft würden, als Argument aber eine Liste des Typs `ListBool` übergeben würde. Die Auswertung müsste an dieser Stelle zwangsläufig mit einem Laufzeitfehler scheitern.

Für einen funktionalen Ausdruck, der einen Typ hat, zeigt Robin Milner in [Mil78, S. 359], dass daraus solange Ergebnisse mit dem richtigen Typ entstehen, solange die Ausdrücke auf Argumente mit dem jeweils passenden Typ angewendet werden. Ist ein Ausdruck also wohlgetypt, wird es keinen derartigen Laufzeitfehler bei der Ausführung geben: „Well-Typed Expressions Do Not Go Wrong“ [Mil78, S. 364]. Insbesondere schützt der Typcheck den Programmierer somit davor, unsinnige Programme zu verfassen, in denen bspw. Buchstaben für Zahlen gehalten, oder Boolesche Wahrheitswerte mit Listen verwechselt werden.

Zum Überprüfen, ob ein Ausdruck wohltypisiert ist, werden auf ihn sog. Typisierungsregeln angewandt, die üblicherweise in der Form

$$\frac{\text{Voraussetzung}}{\text{Konsequenz}}$$

notiert werden. Für die Applikation $(s\ t)$ lässt sich damit bspw. als Typregel

$$\frac{s :: \tau_1 \to \tau_2 \text{ und } t :: \tau_1}{(s\ t) :: \tau_2}$$

aufstellen (vgl. [SS11]), und wie folgt lesen: Wenn s vom Funktionstyp $\tau_1 \to \tau_2$ ist und das Argument t den dazu passenden Typ τ_1 hat, dann wird das Ergebnis von $(s\ t)$ nach der Anwendung des Arguments auf die Funktion den Typ τ_2 haben.

Durch wiederholtes rekursives Anwenden der zu einem Kalkül gehörenden Typisierungsregeln lässt sich feststellen, ob eine Typisierung legal ist. Andernfalls wird eine der Regeln nicht erfüllbar sein, und einen Typfehler verursachen.

2.3.2 Typinferenz mit Unifikation

In Abb. 2.2 ist dem `case` der verlangte Typ `ListBool` des Arguments als Index beigefügt[4], bei den anderen Datenkonstruktoren fehlt hingegen diese Information. Um vom Programmierer nicht die Angabe der Typen für jeden Ausdruck und alle Unterausdrücke verlangen zu müssen, besteht eine weitere, schwierigere Aufgabe des Typcheckers in der sog. *Typinferenz*. Darunter ist das Herleiten einer legalen (d.h. wohltypisierten) Typisierung aufgrund unvollständiger Typinformationen im Ausdruck zu verstehen.

Zum Herleiten einer legalen Typisierung wird ein Typisierungsalgorithmus zunächst die Regeln rekursiv auf den Ausdruck anwenden. Die dabei zugeordneten Typen werden in einer Typumgebung Γ festgehalten. Für manche Konstrukte lässt sich zunächst kein Typ direkt ablesen, er müsste „geraten“ werden. In diesem Fall wird der Typ zunächst allgemein mit einer Typvariablen festgehalten. Zusätzlich werden an diesen Typ geknüpfte Bedingungen in Form von Gleichungen aufgestellt und in einer Menge E mitgeführt.

Die Typregel für die Anwendung lässt sich nun etwas ausführlicher definieren

$$\frac{\Gamma \vdash s :: \tau_1, E_1 \text{ und } \Gamma \vdash t :: \tau_2, E_2}{\Gamma \vdash (s\ t) :: \alpha, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}}$$

und ist wie folgt zu verstehen: Wenn aus der Typumgebung Γ für s und t jeweils die Typen τ_1 und τ_2 mit den zugehörigen Gleichungen E_1, E_2 hergeleitet werden können, dann kann der Typ der Applikation $(s\ t)$ mit Typvariable α angenommen werden, und das neue Gleichungssystem entsteht aus der Vereinigung der beiden Gleichungssysteme E_1, E_2, erweitert um die Bedingung, dass es sich bei s tatsächlich um eine Funktion handeln muss, die auch ein Ergebnis vom Typ α erzeugt.

Anschließend wird auf das Gleichungssystem E ein Unifikationsalgorithmus angewendet, dessen Ziel es ist, „Typen gleich zu machen“. Das Ergebnis stellt entweder einen *allgemeinsten Unifikator* dar, oder endet mit einem Fehler. Ein allgemeinster Unifikator ist dabei eine Menge von Substitutionen σ der Form $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$, wobei α_i eine Typvariable, und τ_i der durch die Unifikation zugeordnete Typ ist (vgl. z.B. [BS01, Sab11]).

Endet dieses Prozedere ohne Fehler, so gilt der funktionale Ausdruck als wohlgetypt, und für jeden Ausdruck und Unterausdruck ist nun sein entsprechender Typ eindeutig bekannt.

[4] In der Praxis steht die Typinformation auch dem `case` nicht zur Verfügung; sie wird in dieser Arbeit der Vollständigkeit und Übersicht halber jedoch stets mit angegeben.

2.4 Nebenläufigkeit mehrerer Prozesse

In [Mar12] unterscheidet Simon Marlow zwischen parallelen und nebenläufigen (engl. *concurrency*) Programmen. Parallele Programme nutzen die mittlerweile häufig parallel vorhandene Hardware aus, insbesondere das Vorhandensein mehrerer Prozessoren oder Prozessorkerne, um Programme durch Verteilen der Arbeit effizienter ausführen zu können.

Nebenläufige Programme hingegen liefern eine Strukturierungsmethode, mit der ein Programm die Interaktionen mit mehreren unabhängigen Agenten (z.B. Datenbank, Benutzerschnittstelle, etc.) steuern kann. Anders als z.B. der Ansatz einer Event-Schleife, die auf alle Ereignisse reagieren können muss, bieten nebenläufige Prozesse (im Folgenden auch *Threads* genannt) den Vorteil der Modularität: Der Prozess, der die Benutzerschnittstelle verwaltet, unterscheidet sich von demjenigen, der die Zugriffe auf die Datenbank ausführt. Durch die nebenläufige Programmierung wird hierbei sichergestellt, dass während Datenbankzugriffen die Benutzerschnittstelle nicht „einfriert“.

Ob ein nebenläufig strukturiertes Programm letztendlich tatsächlich parallel auf vorhandenen Prozessorkernen ausgeführt wird, oder ob die einzelnen Prozesse *interleaved* (dt. verschachtelt) auf einem einzelnen Prozessor ausgeführt werden, spielt für das Ergebnis nur eine untergeordnete Rolle. Die Prozesse werden als gleichzeitig ablaufend „angenommen“.

In Concurrent Haskell [PJGF96] werden neue Prozesse i.d.R. über einen `forkIO`-Befehl erzeugt, dem ein Ausdruck zur nebenläufigen Auswertung übergeben wird. Als Rückgabe liefert er eine dem Prozess eindeutig zugeordnete ID. Das Suffix `IO` zeigt dabei an, dass es sich bei dieser Aktion um eine Operation in der sog. I/O–Monade handelt.

2.4.1 Seiteneffekte und die I/O–Monade

Der bisher vorgestellte erweiterte Lambda-Kalkül ist in der Lage, rein funktionale Ausdrücke zu reduzieren, wobei immer die gleiche WHNF herauskommt, sofern eine existiert (vgl. Abschnitt 2.1.3). Dies funktioniert, da keinerlei *Seiteneffekte* auftreten, womit gemeint ist, dass in einem rein funktionalen Kontext die immer gleichen Argumente auf die immer gleichen Funktionen angewendet werden und das immer gleiche Ergebnis liefern.

Mit dem Auftreten von Seiteneffekten ist nicht mehr gewährleistet, dass stets das gleiche Ergebnis berechnet wird. Vereinfacht kann man sich den Unterschied z.B. durch die mathematische Auswertung von $4 + 2 - 3$ vorstellen, die immer das gleiche Ergebnis 3 liefert, unabhängig davon, ob man zuerst addiert oder subtrahiert. Eine Gleichung der Form $4 + 2 - x$, wobei für die Variable x durch einen Anwender ein beliebiger Wert eingesetzt wird, liefert hingegen zwangsläufig unterschiedliche (möglicherweise für z.B. $x =$ `text` sogar unsinnige) Ergebnisse.

Mit Programmiersprachen möchte man selten nur starre Berechnungen lösen, sondern zahlreiche Seiteneffekte, wie z.B. Setzen eines Pixels auf einem Monitor, Lesen einer Datei, Warten auf eine Benutzereingabe, usw. nutzen.

Um eine Unterscheidung zwischen rein funktionalen Ausdrücken, deren Ergebnis bei Auswertung gleich bleibt, und solchen Ausdrücken zu schaffen, deren Ergebnis abhängig von Seiteneffekten ist, wurde die I/O–Monade wie in [PJ01] eingeführt. Findet während einer Berechnung ein Seiteneffekt statt, so muss diese Funktion und auch ihr Ergebnis in die Monade gekapselt werden. Funktionen können also zusätzlich zu den definierten Basistypen (wie z.B. `Char`, `Int`, ...) nun Rückgabewerte der Form `IO Char`, `IO Int`, etc. haben.

Wesentliches Merkmal dabei ist, dass die Monade nicht mehr verlassen werden kann: Verwendet eine Funktion auch nur eine monadische Funktion, so muss auch ihr Ergebnis in die Monade gekapselt werden. Liefert eine Funktion kein Ergebnis, weil sie z.B. eine

Aktion wie das Schreiben einer Zeichenkette auf den Monitor ausführt, so kann sie den speziellen Rückgabewert `IO ()` liefern.

Monadische Aktionen können mit dem Bind-Operator `>>=` „zusammengeklebt“ werden, wodurch in gewisser Weise eine Form von imperativer Programmierung ermöglicht wird:

```
getChar >>= \x -> putChar x
wobei getChar :: IO Char und putChar :: Char -> IO ()
```

liefert bspw. eine simple Echo-Funktion in Haskell[5]. Deutlich wird hierbei auch, dass monadische Aktionen problemlos rein funktionale Werte wie das `Char` weiterverarbeiten können, solange die I/O–Monade nicht verlassen wird.

Da in einem nebenläufigen Kalkül Prozesse strukturiert eingesetzt werden sollen, um die Benutzerschnittstelle zu verwalten, auf Daten zuzugreifen etc., sind sie daher notwendigerweise nichtdeterministisch und müssen in die I/O–Monade gekapselt werden [Mar12]. Damit wird auch deutlich, warum `forkIO` zwangsläufig `IO`–Typen als Ein– und Ausgabe verlangt.

2.4.2 Synchronisation über `MVars`

Zur Kommunikation und Synchronisation führt Concurrent Haskell das Konstrukt der `MVar`s ein. Wie in [Mar12] kann man sich eine `MVar` als eine „Kiste“ vorstellen, die entweder leer, oder aber mit einem Wert gefüllt ist. Es handelt sich dabei um eine Variable, auf die von mehreren Prozessen aus zugegriffen werden kann. Die Besonderheit besteht darin, dass beim Lesen des Wertes aus einer `MVar` diese leer zurückgelassen wird. Versucht ein Prozess aus einer leeren `MVar` zu lesen, so wird dieser Vorgang verzögert, bis sie von einem anderen Prozess wieder befüllt wurde. Versucht ein Prozess einen Wert in eine bereits gefüllte `MVar` zu schreiben, so muss auch er warten, bis die `MVar` durch einen anderen Prozess geleert wurde und somit aufnahmebereit ist.

Auf diese Weise lässt sich in seiner simpelsten Form bereits ein Locking-Mechanismus erkennen. Möchte ein Prozess exklusiven Zugriff auf einen Wert, so liest er ihn aus der `MVar`, die leer zurückbleibt. Andere Prozesse müssen nun zwangsläufig warten, bis der Prozess den (veränderten) Wert zurückgeschrieben hat, ehe sie ihrerseits den Wert auslesen und ggf. verändern können. Sogenannte *Race Conditions* werden auf diese Weise vermieden, da Variablen nicht ausgelesen oder verändert werden können, solange sie in Bearbeitung eines anderen Prozesses sind [PJ01].

Zur Synchronisation lassen sich `MVar`s einsetzen, indem ein Prozess so lange (entweder durch Schreiben auf eine volle `MVar`, oder Lesen aus einer leeren) warten muss, bis ein anderer Prozess die `MVar` derart verändert, dass die Aktion durchgeführt und die weitere Auswertung des Prozesses fortgesetzt werden kann. Auch das Übermitteln eines Ergebnisses an den aufrufenden Prozess, das bei den bisher vorgestellten Threads im Unterschied zu Funktionen nicht vorgesehen ist, wird mittels einer vorher eingerichteten `MVar` möglich.

Da `MVar`s zur Kommunikation von Prozessen eingesetzt werden, es sich hierbei also um Input/Output–Aktionen handelt und ihr Inhalt nichtdeterministisch ist, müssen auch `MVar`-Werte in die I/O–Monade gekapselt werden.

2.4.3 Futures

Um, wie oben skizziert, das nebenläufig berechnete Ergebnis eines Threads zu erhalten, wird zunächst eine leere `MVar` angelegt und die Berechnung gestartet. Nach Vollenden

[5] Bei Programmlistings wird in Lambda-Abstraktionen das λ durch \ und der Punkt durch `->` ersetzt.

der Berechnung schreibt der Thread das Ergebnis in die `MVar`, wo es von einem anderen (bzw. dem aufrufenden) Prozess ausgelesen werden kann. Ist die Berechnung zum Zeitpunkt, zu dem das Ergebnis des Threads benötigt wird, noch nicht vollendet, entsteht eine Wartesituation (durch den Ausleseversuch einer noch leeren `MVar`), bis das Ergebnis vorliegt.

Die `MVar` beschreibt also eine Variable, deren Wert erst in der Zukunft feststeht. In [SSS11] werden solche Variablen als *Futures* bezeichnet. Da sie im beschriebenen Fall mit den vorhandenen Mitteln von Concurrent Haskell implementiert werden kann, wird auch von einer *expliziten Future* gesprochen.

Im Fall expliziter Futures muss das Ergebnis zur Weiterverarbeitung aus der `MVar` herausgelesen werden, was den Programmierer mit einem fehleranfälligen Mehraufwand belastet. Angenehmer wäre der Fall *impliziter Futures*, deren Ergebnis, automatisch berechnet, genau dann und dort zur Verfügung steht, wann und wo es gebraucht wird.

Ebenda wurde von David Sabel und Manfred Schmidt-Schauss der bisherige `forkIO`-Befehl derart umgebaut, dass sein Rückgabewert nicht mehr eine ThreadID, sondern eine Variable darstellt, in der das Ergebnis der Berechnung gespeichert wird. Diese Variable kann in der weiteren Programmausführung wie jede andere verwendet werden, mit dem Unterschied, dass ihr Wert nebenläufig berechnet wird und erst in der Zukunft zur Verfügung steht, wenn die Variable tatsächlich referenziert wird. Durch die Erweiterung der Sprache um diese Fähigkeit entfällt das Hantieren mit `MVar`s zu diesem Zweck.

2.5 Syntax und Semantik von *Concurrent Haskell mit Futures*

Mit den bisher eingeführten Hilfsmitteln ist es nun möglich, den CHF–Kalkül aus [SSS11, Sab11] syntaktisch und semantisch zu beschreiben und zu erklären, soweit es für diese Arbeit relevant ist.

CHF beschreibt einen auf Concurrent Haskell aufbauenden erweiterten Lambda-Kalkül, der nebenläufige Berechnungen in Form von Futures unterstützt. Neben den rein funktionalen Konstrukten wie Abstraktion, Applikation, Datenkonstruktoren, `case`–Ausdrücken, rekursiven `letrec`–Definitionen und dem aus Haskell bekannten `seq`–Operator zur strikten Auswertung, stehen außerdem monadische Aktionen in Form von `MVar`s zur Verfügung.

2.5.1 Syntax von *CHF*

Die Syntax von *CHF* ist in Form von Grammatiken in Abbildung 2.3 nach [SSS11] dargestellt und besteht aus zwei Ebenen, die zwischen Prozessen und Ausdrücken unterscheidet.

Prozesse *Proc* bestehen aus der parallelen Komposition, nebenläufigen Threads, Namensrestriktionen, gefüllten und leeren `MVar`s und Bindungen. Die Komposition $P_1 \mid P_2$ konstruiert zwei nebenläufige Threads bzw. andere Elemente aus *Proc*. Ein nebenläufiger Thread $x \Leftarrow e$, wobei x eine Variable (dies ist eine *Future*) und $e \in \mathit{Expr}$ ein beliebiger Ausdruck ist, wertet zunächst den Ausdruck e aus und bindet das Ergebnis im Anschluss an die Variable x. Es existiert genau ein ausgezeichneter Thread, der als main-Thread bezeichnet und mit $x \overset{\mathsf{main}}{\Longleftarrow} e$ beschrieben wird. Mit der Namensrestriktion kann der Bindungsbereich einer Variable beschränkt werden. So ist z.B. die Variable x in $Q \mid \nu x.P$ durch den ν–Binder ausschließlich im Bindungsbereich von P verfügbar. Wie bereits beschrieben, können `MVar`s entweder leer, dargestellt durch $x\,\mathbf{m}\,-$, oder gefüllt wie in $x\,\mathbf{m}\,e$ sein, wobei $e \in \mathit{Expr}$ ein Ausdruck ist, und das x den Namen der `MVar` darstellt. Durch Bindungen der Form $x = e$ werden Ausdrücke e über einen globalen Heap mittels einer Variablenbindung x geteilt.

$$
\begin{array}{rcll}
P, P_i \in \mathit{Proc} & ::= & P_1 \mid P_2 & \text{(parallele Komposition)} \\
& \mid & x \Leftarrow e & \text{(nebenläufiger Thread)} \\
& \mid & \nu x.P & \text{(Namensrestriktion)} \\
& \mid & x\,\mathbf{m}\,e & \text{(gefüllte } \texttt{MVar}\text{)} \\
& \mid & x\,\mathbf{m}\,- & \text{(leere } \texttt{MVar}\text{)} \\
& \mid & x = e & \text{(Bindung)} \\
e, e_i \in \mathit{Expr} & ::= & x \mid me \mid \lambda x.e \mid (e_1\ e_2) \mid c\ e_1 \ldots e_{\mathrm{ar}(c)} \mid \texttt{seq}\ e_1\ e_2 & \\
& \mid & \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e & \\
& \mid & \texttt{case}_T\ e\ \texttt{of}\ alt_{T,1} \ldots\ alt_{T,|T|} & \\
& & \quad \text{wobei } alt_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i) & \\
me \in \mathit{MExpr} & ::= & \texttt{return}\,e \mid e_1 \gg\!= e_2 \mid \texttt{future}\,e & \\
& \mid & \texttt{takeMVar}\,e \mid \texttt{newMVar}\,e \mid \texttt{putMVar}\,e_1\,e_2 & \\
\tau, \tau_i \in \mathit{Typ} & ::= & \texttt{IO}\ \tau \mid (T\ \tau_1 \ldots \tau_n) \mid \texttt{MVar}\ \tau \mid \tau_1 \to \tau_2 &
\end{array}
$$

Abbildung 2.3: Definition der Syntax von Prozessen *Proc*, Ausdrücken *Expr*, Monadischen Ausdrücken *MExpr* und Typen *Typ* der Sprache *CHF*.

Funktionale Ausdrücke *Expr* sind analog zu denen des erweiterten Lambda-Kalküls in Abschnitt 2.2 definiert, und umfassen Variablen $x \in V$, Lambda–Abstraktionen $\lambda x.e$, Applikationen $(e_1\ e_2)$, ein getyptes `case`, rekursive `letrec`-Definitionen, Datenkonstruktoren $(c\ e_1 \ldots e_{\mathrm{ar}(c)})$, sowie den `seq`–Operator zur sequentiellen Auswertung zweiter Ausdrücke e_1 und e_2. Zu jedem Typ T gehört eine Menge mit einer festen Anzahl Datenkonstruktoren $c_1 \ldots c_{|T|}$ wobei $|T|$ gerade die Anzahl der zum Typ T gehörenden Konstruktoren darstellt. Jeder dieser Datenkonstruktoren c hat eine feste Stelligkeit $\mathrm{ar}(c) \geq 0$. Innerhalb von Ausdrücken dürfen Konstruktoren $(c\ e_1 \ldots e_{\mathrm{ar}(c)})$ nur vollständig gesättigt auftreten. Das getypte $\texttt{case}_T$ muss für jeden zum Typ T gehörenden Konstruktor genau eine Alternative anbieten.

Mit $me \in \mathit{MExpr}$ werden der Syntax monadische Aktionen hinzugefügt, darunter die notwendigen Aktionen zum Umgang mit `MVar`s: Mit `newMVar` e wird eine neue `MVar` mit Namen e angelegt. Sie kann mittels `putMVar` $e\ e_2$ mit dem Wert e_2 befüllt werden, sofern sie zu diesem Zeitpunkt leer ist, bzw. falls sie gefüllt ist, kann der Wert mit `takeMVar` e ausgelesen werden. Mit `return` kann ein monadischer Wert zurückgegeben werden und monadische Aktionen können mit dem Bind-Operator `>>=` miteinander verbunden werden. Ein Ausdruck e kann als Future mit `future` e erzeugt werden, womit e nebenläufig in einem eigenen Thread berechnet wird.

Die Typen bestehen zum Einen aus bereits bekannten Typkonstruktoren $(T\ \tau_1 \ldots \tau_n)$ und Funktionstypen $\tau_1 \to \tau_2$. Zum Anderen haben monadische Aktionen die Form `IO` τ, und MVars die Form `MVar` τ, wobei τ ein beliebiger Typ ist.

2.5.2 Kontexte

Kontexte modellieren, wie in Abschnitt 2.1.2 beschrieben, die Suche nach dem nächsten Redex. Die für *CHF* notwendigen Kontexte sind in Abbildung 2.4 dargestellt. Prozesskontexte $\mathbb{D} \in \mathit{PCtxt}$ werden verwendet, um einen Thread $P \in \mathit{Proc}$, einen Thread und eine `MVar` oder einen Thread und eine Menge an ν-Bindungen des Threads auszuwählen.

Zum Finden der ersten monadische Aktion in einer mit `>>=` verknüpften Folge dient der monadische Kontext *MCtxt*.

Der *ECtxt* definiert den bereits bekannten funktionalen Auswertungskontext. Dieser ist

$$
\begin{array}{rcl}
\mathbb{D} \in \mathit{PCtxt} & ::= & [\cdot] \mid \mathbb{D} \mid P \mid P \mid \mathbb{D} \mid \nu x.\mathbb{D} \\
\mathbb{M} \in \mathit{MCtxt} & ::= & [\cdot] \mid \mathbb{M} \gg\!= e \\
\mathbb{E} \in \mathit{ECtxt} & ::= & [\cdot] \mid (\mathbb{E}\ e) \mid (\texttt{case}\ \mathbb{E}\ \texttt{of}\ \mathit{alts}) \mid (\texttt{seq}\ \mathbb{E}\ e) \\
\mathbb{F} \in \mathit{FCtxt} & ::= & \mathbb{E} \mid (\texttt{takeMVar}\ \mathbb{E}) \mid (\texttt{putMVar}\ \mathbb{E}\ e) \\
\mathbb{L} \in \mathit{LCtxt} & ::= & x \Leftarrow \mathbb{M}[\mathbb{F}] \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1 \\
 & & \quad \text{wobei } \mathbb{E}_i \neq [\cdot] \text{ für } i = 2, \ldots, n \\
\widehat{\mathbb{L}} \in \widehat{\mathit{LCtxt}} & ::= & x \Leftarrow \mathbb{M}[\mathbb{F}] \mid x \Leftarrow \mathbb{M}[\mathbb{F}[x_n]] \mid x_n = \mathbb{E}_n[x_{n-1}] \mid \ldots \mid x_2 = \mathbb{E}_2[x_1] \mid x_1 = \mathbb{E}_1 \\
 & & \quad \text{wobei } \mathbb{E}_i \neq [\cdot] \text{ für } i = 1, \ldots, n
\end{array}
$$

Abbildung 2.4: Prozess–, Monadische–, Auswertungs–, und Forcing–Kontexte von *CHF*.

um `case` und `seq` erweitert, um sicherzustellen, dass zunächst das Argument von `case` bzw. das erste Argument von `seq` bis zur WHNF ausgewertet wird.

Die Aktionen `putMVar` und `takeMVar` erwarten als erstes Argument den Namen einer existierenden `MVar`. Dieser muss allerdings nicht direkt angegeben sein, da die Syntax an der Stelle des ersten Arguments komplette Ausdrücke erlaubt. Um die monadische Aktion zum Lesen oder Schreiben einer `MVar` durchführen zu können, muss dieser evtl. vorhandene Ausdruck zunächst zum Namen der gewünschten `MVar` ausgewertet werden. Diese Auswertung wird mit dem Forcing-Kontext *FCtxt* erzwungen.

Nachdem ein Thread bereits ausgewählt wurde, wird der monadische oder funktionale Redex mit dem Kontext *LCtxt* unter Beachtung der vom Thread benötigten globalen Heap-Bindungen gesucht. Dabei modelliert der $\widehat{\mathit{LCtxt}}$ den Fall von möglichen Variable-zu-Variable-Bindungen der Form $x = y$ im Heap (vgl. dazu [SSS11, S. 105]).

2.5.3 Der Typcheck-Algorithmus

Um den Typcheck erfolgreich durchführen zu können, bedarf es der Angabe der Typisierungsregeln sowie des dazu passenden Unifikationsalgorithmus. Aus beiden lässt sich bereits implizit der Typisierungsalgorithmus ablesen, indem nichtdeterministisch jeweils die passende Regel angewendet wird, bis keine weitere Anwendung mehr möglich ist, oder ein Fehler auftritt.

2.5.3.1 Typisierungsregeln

Die Typregeln aus [Sab11] werden der Vollständigkeit halber bereits um Gleichungen E erweitert angegeben. Diese Gleichungen werden bei der Typisierung gesammelt und mittels Unifikation gelöst.

Die ersten sechs Regeln in Abbildung 2.5 beschreiben die Typisierung der monadischen Aktionen und liefern entsprechend allesamt einen `IO`-Typ. Die siebte Regel beschreibt die Typisierung von Konstruktoranwendungen, und die achte die bereits aus Abschnitt 2.3 bekannte Regel für die Applikation.

Beim Typisieren von Abstraktionen muss der Rumpf e rekursiv typisiert werden, wobei darin x als freie Variable vorkommen kann. Sie wird daher der Typumgebung Γ als Annahme hinzugefügt.

Die zehnte Regel bildet ein Axiom zur Typisierung von Variablen und die elfte Regel typisiert `seq`-Ausdrücke unter der Voraussetzung, dass das erste Argument keinen `IO`- und keinen `MVar`-Typ haben darf.

Die Fallunterscheidung wird in der zwölften Regel definiert, wobei verlangt wird, dass alle Konstruktor-Pattern $(c_i\ x_1 \ldots x_n)$ zum Typ des Ausdrucks e passen, und alle rechten

$$\frac{\Gamma \vdash e :: \tau, E}{\Gamma \vdash \texttt{return}\ e :: \texttt{IO}\ \tau, E} \qquad \frac{\Gamma \vdash e_1 :: \tau_1, E_1 \text{ und } \Gamma \vdash e_2 :: \tau_2, E_2}{\Gamma \vdash e_1 \gg\!= e_2 :: \texttt{IO}\ \alpha_2, E_1 \cup E_2 \cup \{\tau_1 \doteq \alpha \to \texttt{IO}\ \alpha_2\}}$$

$$\frac{\Gamma \vdash e :: \tau, E}{\Gamma \vdash \texttt{future}\ e :: \tau, E \cup \{\tau \doteq \texttt{IO}\ \alpha\}} \qquad \frac{\Gamma \vdash e :: \tau, E}{\Gamma \vdash \texttt{takeMVar}\ e :: \texttt{IO}\ \alpha, E \cup \{\tau \doteq \texttt{MVar}\ \alpha\}}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1, E_1 \text{ und } \Gamma \vdash e_2 :: \tau_2, E_2}{\Gamma \vdash \texttt{putMVar}\ e_1\ e_2 :: \texttt{IO}\ (), E_1 \cup E_2 \cup \{\tau_1 \doteq \texttt{MVar}\ \tau_2\}} \qquad \frac{\Gamma \vdash e :: \tau, E}{\Gamma \vdash \texttt{newMVar}\ e :: \texttt{IO}\ (\texttt{MVar}\ \tau), E}$$

$$\frac{\forall i : \Gamma \vdash e_i :: \tau_i, E_i}{\Gamma \vdash (c\ e_1 \dots e_{\mathrm{ar}(c)}) :: \tau_{n+1}, \bigcup_i E_i \cup \{\mathrm{Typ}(c) \doteq \tau_1 \to \dots \to \tau_n \to \tau_{n+1}\}}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1, E_1 \text{ und } \Gamma \vdash e_2 :: \tau_2, E_2}{\Gamma \vdash (e_1\ e_2) :: \alpha, E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}}$$

$$\frac{\Gamma \cup \{x :: \alpha\} \vdash e :: \tau, E}{\Gamma \vdash (\lambda x \to e) :: \alpha \to \tau, E} \qquad \frac{}{\Gamma \cup \{x :: \tau\} \vdash x :: \tau, \emptyset} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 :: \tau_1, E_1 \quad \Gamma \vdash e_2 :: \tau_2, E_2 \\ \tau_1 \doteq \tau_3 \to \tau_4 \text{ oder } \tau_1 \doteq T\end{array}}{\Gamma \vdash (\texttt{seq}\ e_1\ e_2) :: \tau_2, E_1 \cup E_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash e :: \tau, E \\ \forall i : \Gamma \cup \{x_{1,i} :: \alpha_{1,i}, \dots, x_{n_i,i} :: \alpha_{n_i,i}\} \vdash (c_i\ x_{1,i} \dots x_{n_i,i}) :: \tau_i, E_i \\ \forall i : \Gamma \cup \{x_{1,i} :: \alpha_{1,i}, \dots, x_{n_i,i} :: \alpha_{n_i,i}\} \vdash e_i :: \tau_i', E_i'\end{array}}{\Gamma \vdash \begin{pmatrix}\texttt{case}\ e\ \texttt{of}(c_1\ x_{1,1} \dots x_{n_1,1} \to e_1) \\ \dots \\ (c_m\ x_{1,m} \dots x_{n_m,m} \to e_m)\end{pmatrix} :: \alpha, E \cup \bigcup_i E_i \cup \bigcup_i E_i' \cup \bigcup_i \alpha \doteq \tau_i' \cup \bigcup_i \tau \doteq \tau_i}$$

$$\frac{\forall i : \Gamma \cup \{x_1 :: \alpha_1, \dots, x_n :: \alpha_n\} \vdash e_1 :: \tau_i, E_i \quad \Gamma \cup \{x_1 :: \alpha_1, \dots, x_n :: \alpha_n\} \vdash e :: \tau, E}{\Gamma \vdash (\texttt{letrec}\ x_1 = e_1, \dots, x_n = e_n\ \texttt{in}\ e) :: \tau, E \cup \bigcup_i E_i \cup \bigcup_i \{\alpha_i \doteq \tau_i\}}$$

Abbildung 2.5: Typisierungsregeln von *CHF* mit Gleichungen aus [Sab11].

Seiten der Alternativen e_i von jeweils gleichem Typ sind.

Das `letrec` wird mit der letzten Typisierungsregel beschrieben. Aufgrund seiner Rekursivität können die definierten Bindungen verschachtelt in den Ausdrücken e_i auftreten. Durch Hinzufügen der x_i als Annahme zur Typumgebung Γ wird sichergestellt, dass sie sowohl bekannt, als auch mit jeweils dem selben Typ rekursiv typisiert werden.

2.5.3.2 Unifikationsregeln

Die beim Anwenden der Typisierungsregeln entstandenen Gleichungen werden mittels Unifikation gelöst. Das Ergebnis ist wie in Abschnitt. 2.3.2 entweder ein allgemeinster Unifikator, der jedem Element des Ausdrucks seinen Typ zuordnet, oder ein Fehler, wenn der Ausdruck nicht typisierbar ist. Die zum CHF-Kalkül gehörenden, für die Unifikation notwendigen Regeln sind in Abbildung 2.6 dargestellt.

Der Unifikationsalgorithmus basiert auf einer Gleichungsmenge E_u für ungelöste und einer Gleichungsmenge E_g für gelöste Gleichungen. Sein Ziel ist das Gleichmachen von Typen.

Die Regeln Decomp.T, Decomp.IO, Decomp.MVar und Decomp.Fn eliminieren die

$$\textsc{Solve}\ \frac{E_g, \{\alpha \doteq \tau\} \cup E_u}{E_g[\tau/\alpha] \cup \{\alpha \doteq \tau\}, E_u[\tau/\alpha]},\ \text{falls } \alpha \text{ nicht in } \tau \text{ vorkommt.}$$

$$\textsc{Elim}\ \frac{E_g\{\alpha \doteq \alpha\} \cup E_u}{E_g, E_u} \qquad \textsc{Decomp.IO}\ \frac{E_g, \{\texttt{IO}\ \tau_1 \doteq \texttt{IO}\ \tau_2\} \cup E_u}{E_g, \{\tau_1 \doteq \tau_2\} \cup E_u}$$

$$\textsc{FailIO}\ \frac{E_g, \{\texttt{IO}\ \tau_1 \doteq \tau_2\} \cup E_u}{\mathit{Fehler}},\ \text{falls } \tau_2 = T,\ \tau_2 = \texttt{MVar}\ \tau' \text{ oder } \tau_2 = \tau_1 \to \tau_2$$

$$\textsc{Occ.Check}\ \frac{E_g, \{\alpha \doteq \tau\} \cup E_u}{\mathit{Fehler}},\ \text{falls } \alpha \text{ in } \tau \text{ vorkommt und } \tau \neq \alpha$$

$$\textsc{Orient}\ \frac{E_g, \{\tau \doteq \alpha\} \cup E_u}{E_g, \{\alpha \doteq \tau\} \cup E_u},\ \text{wenn } \tau \text{ keine Typvariable ist}$$

$$\textsc{Decomp.T}\ \frac{E_g, \{T \doteq T\} \cup E_u}{E_g, E_u}$$

$$\textsc{FailT}\ \frac{E_g, \{T_1 \doteq \tau\} \cup E_u}{\mathit{Fehler}},\quad \begin{array}{c}\text{falls } \tau = T_2 \text{ und } T_1 \neq T_2,\ \tau = \texttt{IO}\ \tau',\\ \tau = \texttt{MVar}\ \tau' \text{ oder } \tau = \tau_1 \to \tau_2\end{array}$$

$$\textsc{Decomp.MVar}\ \frac{E_g, \{\texttt{MVar}\ \tau_1 \doteq \texttt{MVar}\ \tau_2\} \cup E_u}{E_g, \{\tau_1 \doteq \tau_2\} \cup E_u}$$

$$\textsc{FailMVar}\ \frac{E_g, \{\texttt{MVar}\ \tau_1 \doteq \tau_2\} \cup E_u}{\mathit{Fehler}},\ \text{falls } \tau_2 = T,\ \tau_2 = \texttt{IO}\ \tau' \text{ oder } \tau_2 = \tau_1 \to \tau_2$$

$$\textsc{Decomp.Fn}\ \frac{E_g, \{\tau_1 \to \tau_2 \doteq \tau_1' \to \tau_2'\} \cup E_u}{E_g, \{\tau_1 \doteq \tau_1', \tau_2 \doteq \tau_2'\} \cup E_u}$$

$$\textsc{FailFn}\ \frac{E_g, \{\tau_1 \to \tau_2 \doteq \tau\} \cup E_u}{\mathit{Fehler}},\ \text{falls } \tau = T,\ \tau = \texttt{IO}\ \tau' \text{ oder } \tau = \texttt{MVar}\ \tau'$$

Abbildung 2.6: Unifikationsregeln für den Typchecker von *CHF* aus [Sab11], wobei α stets eine Typvariable ist, und T, T_1, T_2 Typkonstruktoren darstellen. Die Menge der ungelösten Gleichungen wird mit E_u, und die Menge der gelösten Gleichungen mit E_g bezeichnet.

jeweils äußeren Typkonstruktoren, falls auf beiden Seiten einer Gleichung gleiche Typen stehen. Ist dies nicht der Fall und keine andere Regel anwendbar, geben die zugehörigen Regeln FailT, FailIO, FailMVar und FailFn entsprechend einen *Fehler* aus.

Die Regel Orient dient zum Justieren der Gleichungen und die Regel Elim entfernt überflüssige Gleichungen aus der Menge. Die Regel Occ.Check stellt sicher, dass keine Variable durch einen Term ersetzt wird, der diese Variable selbst enthält. Wurde für ein α ein Typ τ bestimmt (und kommt α nicht mehr in τ vor), so führt die Regel Solve die entsprechende Substitution in beiden Gleichungsmengen E_u und E_g durch.

2.5.4 Auswertungsregeln

Die operationale Semantik in Form von Auswertungsregeln unterteilt sich in diejenigen zur Auswertung funktionaler Ausdrücke und solche zur Berechnung monadischer Aktionen. Die Funktionalen Auswertungsregeln sind in Abbildung 2.7(a) dargestellt. Wenn innerhalb des angegebenen Kontexts ein Redex gefunden wird, transformiert die passende Regel den Ausdruck, was durch $\xrightarrow{CHF}$ dargestellt wird.

(cp) $\widehat{\mathbb{L}}[x] \mid x = v \xrightarrow{CHF} \widehat{\mathbb{L}}[v] \mid x = v$, sofern v eine Abstraktion oder Variable

(cpcx) $\widehat{\mathbb{L}}[x] \mid x = c\ e_1 \ldots e_n$ falls c ein Konstruktor oder Monadischer Operator
$\xrightarrow{CHF} \nu y_1, \ldots y_n.(\widehat{\mathbb{L}}[c\ y_1 \ldots y_n] \mid x = c\ y_1 \ldots y_n \mid y_1 = e_1 \mid \ldots \mid y_n = e_n)$

(mkbinds) $\mathbb{L}[\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e]$
$\xrightarrow{CHF} \nu x_1, \ldots, x_n.(\mathbb{L}[e] \mid x_1 = e_1 \mid \ldots \mid x_n = e_n)$

(lbeta) $\mathbb{L}[((\lambda x.e_1)\ e_2)] \xrightarrow{CHF} \nu x.(\mathbb{L}[e_1] \mid x = e_2)$

(case) $\mathbb{L}[\texttt{case}_T\ (c\ e_1\ \ldots\ e_n)\ \texttt{of}\ \ldots(c\ y_1\ \ldots\ y_n \to e)\ldots]$
$\xrightarrow{CHF} \nu y_1, \ldots, y_n.(\mathbb{L}[e] \mid y_1 = e_1 \mid \ldots \mid y_n = e_n])$

(seq) $\mathbb{L}[(\texttt{seq}\ v\ e)] \xrightarrow{CHF} \mathbb{L}[e]$, falls v ein funktionaler Wert (also eine Abstraktion oder Konstruktorapplikation) ist

(a) Funktionale Auswertungsregeln

(lunit) $y \Leftarrow \mathbb{M}[\texttt{return}\ e_1 \gg\!= e_2] \xrightarrow{CHF} y \Leftarrow \mathbb{M}[e_2\ e_1]$

(tmvar) $y \Leftarrow \mathbb{M}[\texttt{takeMVar}\ x] \mid x\,\mathbf{m}\,e \xrightarrow{CHF} y \Leftarrow \mathbb{M}[\texttt{return}\ e] \mid x\,\mathbf{m} -$

(pmvar) $y \Leftarrow \mathbb{M}[\texttt{putMVar}\ x\ e] \mid x\,\mathbf{m} - \xrightarrow{CHF} y \Leftarrow \mathbb{M}[\texttt{return ()}] \mid x\,\mathbf{m}\,e$

(nmvar) $y \Leftarrow \mathbb{M}[\texttt{newMVar}\ e] \xrightarrow{CHF} \nu x.(y \Leftarrow \mathbb{M}[\texttt{return}\ x] \mid x\,\mathbf{m}\,e)$

(fork) $y \Leftarrow \mathbb{M}[\texttt{future}\ e] \xrightarrow{CHF} \nu z.(y \Leftarrow \mathbb{M}[\texttt{return}\ z] \mid z \Leftarrow e)$
wobei z frisch, und der neue Thread nich der main-Thread ist

(unIO) $y \Leftarrow \texttt{return}\ e \xrightarrow{CHF} y = e$
falls der Thread nicht der main-Thread ist

(b) Monadische Auswertungsregeln

Abbildung 2.7: Funktionale (a) und Monadische (b) Reduktionsregeln zur Normalordnungsreduktion im *CHF*–Kalkül

Die bisher vorgestellte β–Reduktion zur Anwendung einer Abstraktion auf ihr Argument ist durch die Regel (lbeta) modelliert. Anders als in Abschnitt 2.1.1 wird hierbei allerdings keine direkte Substitution durchgeführt, sondern eine neue, mit ν namensbeschränkte Variablenbindung im Heap angelegt. Dadurch wird gerade das Sharing implementiert und mögliche Mehrfachauswertungen des Arguments selbst werden vermieden. In ähnlicher Form legt die Auswertungsregel (mkbinds) für `letrec`-Konstrukte Heap-Bindungen für die neu definierten Ausdrücke an. Die Regel (case) reduziert auf denjenigen Ausdruck e, für den ein Pattern gepasst hat. Außerdem macht sie die Argumente des Pattern-Konstruktors als neue Heap-Bindungen, mit den in der Alternative gebundenen Variablennamen, in e verfügbar. Ist bei einem `seq`-Ausdruck das erste Argument v ausgewertet, reduziert die Regel (seq) den Ausdruck auf das zweite Argument e. Die Regeln (cp) und (cpcx) dienen dazu, Abstraktionen, Variablen oder Konstruktoren aus dem Heap in den Ausdruck zu kopieren. Im Fall eines Konstruktors $(c\ e_1 \ldots e_n)$ werden die möglicherweise noch nicht ausgewerteten Ausdrücke e_i zwecks Sharings als eigene Heapbindungen hinzugefügt.

Die Monadischen Auswertungsregeln sind in Abbildung 2.7(b) dargestellt. Die Verkettung zweier monadischer Aktionen durch den Bind-Operator `>>=` wird durch die Regel (lunit) abgedeckt, wobei das Ergebnis des ersten Ausdrucks als Argument auf den zweiten angewendet wird. Eine neue, mit einem übergebenen Ausdruck e bereits befüllte `MVar` wird durch die Regel (nmvar) bereitgestellt, wobei das Ergebnis dieser Reduktion gerade den Namen dieser `MVar` liefert. Da es sich um eine monadische Aktion handelt, muss dies mittels `return` x einen `IO`-Wert ergeben. Die Regeln (tmvar) und (pmvar) modellieren den

Zugriff auf eine existierende `MVar`, wobei nur der jeweils erfolgreiche Fall des Lesens einer vollen, bzw. des Schreibens einer leeren `MVar` abgedeckt ist. Das explizite Fehlen entsprechender Regeln zum Lesen von leeren bzw. Schreiben auf volle `MVar`s modelliert hierbei gerade die verlangte Unmöglichkeit dieser Aktionen.

Ähnlich wie beim Erstellen einer neuen `MVar` erstellt die Regel (fork) einen neuen Thread, der das übergebene Argument weiter auswertet. Das Ergebnis der Regel ist der, mittels `return` als monadischer Wert verpackte, neue Name dieses Threads. Ist ein Wert fertig ausgewertet, d.h. sind keine weiteren Reduktionen mehr möglich, so entfernt die Regel (unIO) den Thread, und fügt dem globalen Heap den berechneten Wert als neue normale Variablenbindung unter dem Namen des Threads hinzu.

3

Fehlerbehandlung in funktionalen Programmiersprachen

In diesem Kapitel wird die Notwendigkeit von Exceptions in nebenläufigen Programmkalkülen wie Concurrent Haskell erörtert. Zusätzlich wird die notwendige Semantik vorgestellt, um bei Auftreten von Exceptions entsprechend darauf reagieren zu können.

In Abschnitt 3.1 werden Exceptions informativ eingeführt und von allgemeinen Fehlern unterschieden. Dabei wird sich auf die ausschließliche Betrachtung von synchronen und asynchronen Exceptions beschränkt.

Wie der Umgang mit synchronen und asynchronen Exceptions in funktionalen Programmiersprachen gehandhabt werden sollte, beschreibt Abschnitt 3.2. Dabei wird insbesondere auf die besonderen Gegebenheiten bei zugelassenen asynchronen Exceptions hingewiesen.

Die als Erweiterung von Concurrent Haskell bereits etablierte Syntax und Semantik zum Umgang mit Exceptions wird in Abschnitt 3.3 vorgestellt, wobei sich auf die für diese Arbeit relevanten Inhalte beschränkt wird.

3.1 Einführung und Unterscheidung von Exceptions

Sowohl bei Concurrent Haskell, als auch beim vorgestellten *CHF*–Kalkül treten durch das vorhandene Typsystem keine Typfehler zur Laufzeit auf, falls das Programm durch den Typchecker als wohlgetypt erkannt worden ist. Dennoch können selbst in wohlgetypten Programmen gewisse Fehler auftreten. Eine bestimmte Klasse dieser Fehler wird im Folgenden vorgestellt, wobei dafür der englische Begriff *Exception* verwendet wird, um sie von allen anderen möglichen Fehlerarten (Tippfehler, Hardwarefehler, Stromausfall, etc.) explizit zu unterscheiden.

Tritt ein derartiger Fehler auf, so ist in den bisher vorgestellten Kalkülen nicht definiert, wie sie sich zu verhalten haben. In den meisten Fällen werden solche Fehler daher semantisch wie $\perp$ behandelt, und die Auswertung terminiert nicht (vgl. [PJ01]).

3.1.1 Synchrone Exceptions

Fehler, die unmittelbar durch die direkte Ausführung eines Programms auftreten, werden *synchrone Exceptions* genannt. Bekannte Beispiele wären hierfür die Division–durch–Null,

oder der Schreibversuch auf einen bereits vollen Datenträger. Konkret für den *CHF*-Kalkül mag man sich z.B. auch den Versuch vorstellen, das erste Element (*head*) einer leeren Liste zu extrahieren. In all diesen Situationen kann die Auswertung eines Programms jeweils nicht wie vorgesehen fortgesetzt werden.

Die Besonderheit synchroner Exceptions ist, dass man sie häufig antizipieren kann, indem man sich für alle Aktionen Gedanken macht, in welcher Weise sie „schiefgehen" könnten. Beim Schreiben einer Datei kann das Medium bereits voll sein, oder eine zu lesende Datei nicht existieren. Listen könnten leer, bzw. ein Divisor Null werden, etc.

Ein Programmierer könnte nun für die möglicherweise auftretenden Exceptions einer jeden Aktion Sicherheitsabfragen implementieren und der Aktion voranstellen. Vor dem Extrahieren eines Listenelements würde dann z.B. überprüft, dass die Liste nicht leer ist, und jede Division nur dann durchgeführt, wenn der zuvor ausgewertete Divisor von Null verschieden ist. Dies erfordert allerdings eine hohe Aufmerksamkeit und bedingt zusätzliche (das eigentliche Programm unübersichtlicher machende) Programmierarbeit.

Ob bei diesem Ansatz am Ende ein robustes Programm entsteht, bei dem wirklich alle synchronen Exceptions abgefangen wurden, ist fraglich. Eine einzige unbedachte Situation reicht aus, um das Programm durch eine Exception nicht terminieren zu lassen. Es ist daher wünschenswert, dass der Kalkül selbst die Möglichkeit erhält, festzustellen, ob eine Exception eintritt und diese ggf. signalisieren kann. Auf diese Weise kann der Kalkül eine $\perp$–Situation vermeiden und die Auswertung des Programms kann sich mit der eingetretenen Exception (entsprechend den Vorgaben des Programmierers) beschäftigen, indem Exceptions an bestimmten Stellen des Programms „gefangen" (engl. *catch*) werden.

3.1.2 Asynchrone Exceptions

Im Unterschied zu synchronen Exceptions, die bei der Ausführung eines Programms entstehende Auswertungsfehler signalisieren, existieren außerdem sog. *asynchrone Exceptions*. Diese werden durch externe Events induziert, und können daher unvorhersehbar zu jedem beliebigen Zeitpunkt auftreten. Insbesondere im Fall nebenläufiger Threads liefern asynchrone Exceptions dadurch die Möglichkeit, Threads „von außen" zu beenden. Wünschenswert ist dies z.B. im Falle eines Timeouts, eines User–Interrupts oder zum Auflösen eines Deadlocks (dt. Verklemmung).

Ein User–Interrupt tritt bspw. dann auf, wenn ein Anwender die Ausführung eines Programms mit der Tastenkombination Ctrl + C beenden möchte. Das entsprechende Programm (bzw. der entsprechende Thread) werden über dieses Event benachrichtigt und sollten, sobald es ihnen möglich ist, darauf reagieren.

Die Gefahr eines unvorhergesehenen Deadlocks entsteht insbesondere bei Vorhandensein mehrerer nebenläufiger Threads. Ein prominentes Beispiel für solch eine Deadlock-Situation ist das von Edsger W. Dijkstra in [Dij71] formulierte *Dining Philosophers Problem*. Hierbei sitzen fünf Philosophen an einem runden Tisch, die abwechselnd nachdenken oder essen. Zum Essen hat jeder Philosoph vor sich einen Teller und links und rechts liegt eine Gabel. Insgesamt liegen damit fünf Gabeln auf dem Tisch. Um essen zu können, benötigt ein Philosoph in jeder Hand eine Gabel, weswegen keine zwei nebeneinander sitzenden Philosophen gleichzeitig essen können, da sie sich die Gabel zwischen ihnen teilen müssen. Es wird angenommen, das jeder Philosoph, wenn er Hunger bekommt, zunächst die linke und dann die rechte Gabel in die Hand nimmt, bzw. auf sie wartet, falls sie nicht verfügbar ist. Sollten nun alle Philosophen zum gleichen Zeitpunkt Hunger bekommen, nimmt jeder Philosoph die linke Gabel in die Hand und wartet auf das Freiwerden der rechten Gabel. Diese hat allerdings der jeweils rechte Nachbarphilosoph bereits in der Hand und wartet seinerseits auf das Freiwerden seiner rechten Gabel. Da keiner der Philo-

sophen in diesem Szenario die linke Gabel wieder aus der Hand legt, ohne zuvor die rechte aufgenommen und etwas gegessen zu haben, müssen alle Philosophen verhungern.

Es fällt leicht, sich an Stelle der Gabeln in diesem Beispiel gedanklich `MVar`s, und Threads statt der Philosophen vorzustellen. In nebenläufigen Programmkalkülen entsteht bspw. genau dann ein Deadlock, wenn zwei Prozesse A, B jeweils Zugriff auf zwei `MVar`s a, b erhalten wollen. Prozess A hat dabei bereits den Wert aus a gelesen und Prozess B den Wert aus b. Prozess A schreibt erst nach Lesen von b wieder Werte in a,b und Prozess B erst nach Lesen von a. Beide Prozesse warten darauf, dass die jeweils vom anderen Prozess gelesene `MVar` wieder befüllt wird, bevor sie ihre Ausführung fortsetzen können. Sie warten für immer.

Da diese Verklemmung i.d.R. durch eine ungünstige Ausführungsreihenfolge der Threads entsteht, kann eine mögliche Auflösung dieser Situation durch „Abschießen“ eines der beteiligten Threads mithilfe einer asynchronen Exception erfolgen. Die Hoffnung dabei ist, dass nun die benötigten `MVar`s freiwerden und die anderen Threads weiterarbeiten können. Der unterbrochene Thread sollte dann anschließend erneut aufgerufen werden.

Entscheidend bei dieser Idee ist, dass bei Zulassen asynchroner Exceptions dafür Sorge getragen wird, dass durch das unmittelbar erzwungene Beenden eines Threads das Programm in keinen inkonsistenten Zustand gerät: Hat ein Thread bspw. den Inhalt einer `MVar` gelesen und wird extern durch eine Exception beendet, so muss gewährleistet werden, dass die `MVar` nicht leer zurückgelassen wird. Stattdessen sollte sie mit ihrem ursprünglichen Wert überschrieben werden.

3.2 Exceptions in funktionalen Programmiersprachen

Simon Peyton Jones stellt in [PJ01, S. 29] fest, dass robuste Programme in unvorhergesehenen Situationen nicht zusammenbrechen ($\bot$) sollten und Programmierer selbstverständlich darauf bedacht sind, Programme derart zu formulieren, dass diese nicht fehlschlagen. Da Programmierer allerdings fehlbar, und manche Fehler selbst durch sorgfältiges Programmieren nicht vermieden werden können, erscheint dieser Ansatz allein unzureichend. Als Konsequenz erweitern Simon Marlow et al. in [MJMR01] das ursprüngliche Concurrent Haskell um ein Exception-Handling-System zum Umgang mit synchronen und asynchronen Exceptions.

3.2.1 Synchrone Exceptions generieren und verarbeiten

Um im Fall eines auftretenden Fehlers nicht in eine $\bot$–Situation zu geraten, wird der Sprachumfang syntaktisch um das Konstrukt `throw` q erweitert, wobei q die Beschreibung der Exception (als speziell definierten Datentyp $q \in$ `Exception`) darstellt. Erkennt die Sprache einen Fehler (wie z.B. Division–durch–Null), ist sie nun in der Lage, diesen zu signalisieren, indem sie den auszuwertenden Term durch ein (`throw` *divideByZero–Exception*) ersetzt, bzw. dies vom Programmierer aufgerufen wird.

Das Gegenstück zu `throw` bildet das ebenfalls hinzugefügte Konstrukt `catch` e_1 e_2, womit eine Art Weiche für den Programmfluss etabliert wird: Tritt in einem Unterausdruck $e_1 \in \mathit{Expr}$ keine Exception auf, so wird nach dessen Auswertung das `catch`–Konstrukt und insbesondere der Exception-Handler $e_2 \in \mathit{Expr}$ nicht weiter beachtet. Tritt hingegen im Unterausdruck e_1 bei der Auswertung eine Exception auf (d.h. trifft die Auswertung auf ein `throw` q), so soll e_1 nicht weiter ausgewertet werden (denn dies scheitert ja gerade), sondern stattdessen der Exception-Handler e_2 aufgerufen werden. Dieser ist von der Form $\lambda q.e_2'$, und erhält als Parameter gerade die Exeption-Beschreibung q, die `throw` q mitliefert.

Auf diese Weise ist es möglich, an bestimmten Stellen im Programm auf eintretende Exceptions reagieren zu können. Innerhalb von e_2' kann z.B. eine Fallunterscheidung stattfinden, die entsprechend der im Unterausdruck e_1 aufgetretenen Exception q reagiert. Ihr jeweiliges Ziel soll gerade die Vermeidung inkonsistenter Zustände des Programms (wie z.B. gelesene und nicht zurückgeschriebene `MVar`s, etc.) sein. Hierbei ist Sorgfalt des Programmierers notwendig.

Die gerade in funktionalen Programmiersprachen besonders gewünschte Modularität wird durch dieses Sprachdesign wesentlich unterstützt. Auftretende Fehler können so lange weitergereicht werden, bis ein zuständiges `catch` sich ihrer annimmt. Insbesondere muss die Fehlerbehandlung also nicht direkt dort angegeben sein, wo der Fehler auftritt.

An dieser Stelle wird bereits deutlich, dass es sich beim Exception–Handling um Operationen handeln muss, die in die I/O–Monade gekapselt werden müssen. Wie erwähnt, können und werden Exceptions als Werte definiert und interpretiert. Obwohl es also durchaus möglich ist, im rein funktionalen Kontext eine Exception zu generieren, kann diese rein funktional nicht aufgefangen werden. Welche Exception z.B. bei einem Ausdruck der Form $((1/0) + (\texttt{throw}\ \textit{UserError}))$ tatsächlich auftritt, hängt von der verwendeten Auswertungsstrategie zur Ausführungszeit ab, und ist infolgedessen nichtdeterministisch. Der in [PJRH+99] vorgestellte Lösungsansatz vermeidet die mögliche Idee des Fixierens der Ausführungsreihenfolge zugunsten „unpräziserer“ Exceptions, indem ein Ausdruck nicht eine designierte Exception generiert, sondern semantisch als eine *Menge von Exceptions* interpretiert wird. Erst beim Auswerten durch die monadische `catch`–Operation wird eine beliebige Exception nichtdeterministisch aus dieser Menge ausgewählt und behandelt. Auf diese Weise werden keine allzu schwerwiegenden Eingriffe in die Semantik der Sprache notwendig, um die Erweiterung für Exceptions einzuführen.

3.2.2 Erweiterung um asynchrone Exceptions

Die Erweiterung von Concurrent Haskell um asynchrone Exceptions gestaltet sich etwas umfangreicher und bedarf einiger zusätzlicher Überlegungen.

3.2.2.1 Exceptions mit `throwTo` versenden

Asynchrone Exceptions unterscheiden sich von synchronen nur in der Hinsicht, dass sie extern induziert sind, und jederzeit unvorhergesehen an einen Thread gesendet werden und auftreten können. Hierzu wird in Concurrent Haskell der Befehl `throwTo` t q eingeführt, der eine Exception q an den Thread mit ThreadID t „sendet“

Wurde eine solche Exception an einen Thread t gesendet, so manifestiert sie sich dort „so bald wie möglich“ (denn der Thread könnte sich z.B. gerade in einer Wartesituation auf eine `MVar` befinden) als ein `throw` q, welches mit den bereits eingeführten Konstrukten zum Behandeln synchroner Exceptions (`catch` e_1 e_2) problemlos weiterverarbeitet werden kann.

Da asynchrone Exceptions allerdings jederzeit unvorhergesehen auftreten können, bieten sie nun die Möglichkeit für Race Conditions. Stellt man sich z.B. vor, dass in einem Ausdruck (`catch` e_1 e_2) während der Auswertung von e_1 eine `MVar` eingelesen wird, und eine (synchrone oder asynchrone) Exception auftritt, wird infolgedessen der Handler e_2 aufgerufen, dessen Aufgabe es ist, den Zustand der `MVar` zur Wahrung der Konsistenz wiederherzustellen. Fatalerweise wird bisher allerdings nicht ausgeschlossen, dass nicht eine weitere asynchrone Exception den Handler unterbricht und infolgedessen daran hindert, eben diese Konsistenz wieder herzustellen.

3.2.2.2 Zulassen und Unterdrücken von Exceptions mit `block` und `unblock`

Aus diesem Grund wird der Sprache mit den Konstrukten `block` und `unblock` zusätzlich die Möglichkeit eingeräumt, das Empfangen asynchroner Exceptions in bestimmten Situationen zu *unterbinden*[6].

Dabei wird bei `block` e der Ausdruck e in einem Threadzustand ausgewertet, in dem keinerlei Exceptions zugelassen werden. Erst wenn dieser Zustand wieder verlassen wird, werden evtl. zwischenzeitlich eingetroffene asynchrone Exceptions wieder durchgelassen und zugestellt. Der Zustand blockierter Exceptions wird genau dann verlassen, wenn der von `block` umschlossene Ausdruck ausgewertet wurde, und eine evtl. mit `>>=` auf den block–Kontext folgende monadische Aktion begonnen wird, die sich nicht mehr im block–Kontext befindet. Eine weitere Möglichkeit besteht darin, innerhalb eines block–Kontexts einen Ausdruck mit `unblock` zu umschließen. Die beiden Konstrukte sind beliebig ineinander verschachtelbar.

Die Race Condition des Handlers im obigen Beispiel kann nun durch einen Ausdruck der Form `block` (`catch` (`unblock` e_1) e_2) vermieden werden, wobei der Ausdruck e_1 in einem unblock–Kontext ausgewertet wird, in dem asynchrone Exceptions ausdrücklich zugelassen werden. Tritt eine Exception auf, so wird die Auswertung abgebrochen, der unblock–Kontext verlassen und dem aufgerufenen Handler e_2 die Exceptionbeschreibung übergeben, der sich seinerseits im umschließenden block–Kontext befindet. Während nun der Handler den konsistenten Zustand des Programms sicherstellt, können ihn diesmal kein Exceptions dabei unterbrechen. Zum Gewährleisten dieser Funktionalität muss sich `catch` zusätzlich „merken“, in welchem Zustand (block– oder unblock–Kontext) sich der Thread vor der Auswertung des ersten `catch`–Arguments befunden hat, und diesen vor der Auswertung des Handlers wiederherstellen.

3.2.2.3 Anpassung der Semantik bestimmter kritischer Operationen

Mit den bisher vorgestellten Möglichkeiten können nun zum einen Threads durch ein externes Signal beendet werden, und zum anderen kritische Ausdrücke als nicht–unterbrechbar (blocked–Kontext) gekennzeichnet werden. Bei sorgfältiger Programmierung kann damit theoretisch die Konsistenz eines Programms nachhaltig geschützt werden.

Unter ungünstigen Umständen kann es praktisch allerdings vorkommen, dass z.B. ein Deadlock einer `MVar` stattfindet, und sich gerade alle daran beteiligten `takeMVar`– und `putMVar`–Operationen innerhalb eines blocked–Kontext befinden. Die Auswertungen der jeweiligen Threads geraten in eine Wartesituation und stocken dort für immer, wodurch die block–Kontexte niemals mehr verlassen werden. Die Versuche, die beteiligten Threads mithilfe asynchroner Exceptions zu beenden, würden in diesem Fall allesamt scheitern.

Da damit allerdings gerade der Nutzen asynchroner Exceptions ad absurdum geführt würde, müssen in diesen speziellen Fällen trotz der Blockierung dennoch asynchrone Exceptions zugelassen werden können. Um dies zu gewährleisten, ohne gleichfalls den Zweck von `block` und `unblock` wieder zu konterkarieren, wird in [MJMR01] eine sorgfältig formulierte Anpassung der Semantik der Sprache vorgenommen:

> Any operation which may need to wait indefinitly for a resource (e.g. `takeMVar`) may receive asynchronous exceptions even within an enclosing `block`, but only while the resource is unavailable. Such operations are termed *interruptible operations*.

[6] Das Auftreten synchroner Exceptions bei der Auswertung wird dadurch allerdings nicht beeinflusst. Synchrone Exceptions werden bei Auftreten weiterhin unabhängig des block/unblock–Zustandes des Threads bis zum nächsten umschließenden `catch` weitergereicht.

Hiermit wird gewährleistet, dass bei ausgeschalteten Exceptions Operationen, die möglicherweise in Wartesituationen geraten können, als „unterbrechbar“ markiert werden. Insbesondere bedeutet es allerdings, dass bei Auswertung von Operationen wie `takeMVar`, `putMVar`, etc. innerhalb eines blocked–Kontext keinerlei asynchrone Exceptions zugelassen werden, solange die Auswertung normal fortgesetzt werden kann. Erst wenn eine der genannten Operationen fehlschlägt (schreiben auf volle, bzw. lesen einer leeren `MVar`), werden für die Dauer dieser Wartesituation externe Exceptions *temporär* zugelassen.

3.3 Syntax und Semantik für synchrone und asynchrone Exceptions in Concurrent Haskell

Bei den im Folgenden vorgestellten Erweiterungen von Syntax und Semantik handelt es sich um Ergänzungen für ein Exception–Handlung–System von Concurrent Haskell aus [PJGF96]. Da der bereits vorgestellte *CHF*–Kalkül seinerseits selbst eine Erweiterung von Concurrent Haskell darstellt, wird an dieser Stelle darauf verzichtet, den Vorgängerkalkül im Detail wiederzugeben.

Der bisher vorgestellte *CHF*–Kalkül implementiert alle, für diese Arbeit notwendigen Konstrukte von Concurrent Haskell und unterscheidet sich nur wesentlich bzgl. der Generierung von Threads wie folgt: Anstelle des in Abbildung 2.3 eingeführten `future`–Befehls zum Erzeugen eines Threads, kennt Concurrent Haskell noch den Vorgänger

```
forkIO :: IO a -> IO ThreadID
```

In gleicher Weise wird hiermit ein neuer Thread erstellt, der allerdings keine Future (d.h. eine Variable, in der am Ende ein Wert steht) als Ergebnis liefert, sondern eine den Thread bezeichnende ThreadID eigenen Typs.

Zur effektiven Erweiterung um synchrone und asynchrone Exceptions wird in [MJMR01] die Syntax der monadischen Ausdrücke von Concurrent Haskell zunächst um die bisher neu vorgestellten Konstrukte `throw` q | `catch {`e_1`} {`e_2`}` | `throwTo` t q | `block {`e`}` | `unblock {`e`}` erweitert. Hierbei stellen e, e_i beliebige Ausdrücke dar. Das q steht für eine Exception-Beschreibung und das t für eine ThreadID, zur eindeutigen Adressierung des Zielthreads. Die ThreadID ist dabei gerade das Ergebnis eines mit `forkIO` erstellten Threads.

Die angegebenen geschweiften Klammern `{`$\cdots$`}` zeigen an, dass an dieser Stelle eine explizite Bereichseingrenzung innerhalb der Ausdrücke ausdrücklich gefordert wird.

Zum Andeuten, dass eine asynchrone Exception q zu einem Thread t gesendet wurde, wird auf der Prozess–Ebene P das Konstrukt *ExceptionInFlight* als $[\![t \not\downarrow q]\!]$ hinzugefügt.

Die in [MJMR01] zur Auswertung benötigten Kontexte werden im Folgenden mit leicht an diese Arbeit angepasster Notation angegeben:

$$\begin{array}{lcl} \mathbb{B} & ::= & \mathbb{M} \mid \mathbb{M}[\texttt{block}\ \mathbb{B}] \mid \mathbb{M}[\texttt{unblock}\ \mathbb{B}] \\ \mathbb{M} & ::= & [\cdot] \mid \mathbb{M} \gg\!= e \mid \texttt{catch}\ \mathbb{M}\ e \end{array}$$

Hierbei definiert $\mathbb{M}$ den monadischen Kontext, erweitert um `catch`. Diese Anpassung sorgt dafür, dass nicht der Handler e, sondern der vom `catch` umschlossene eigentliche Ausdruck (das erste Argument) ausgewertet wird.[7] Zusätzlich wird in [PJGF96] verlangt, dass bei den Operationen `takeMVar` und `putMVar` zunächst immer jeweils das erste Argument ausgewertet wird. Durch Einführen einer zweiten Kontext–Ebene $\mathbb{B}$ zum Abbilden

[7] Die syntaktische Einschränkung von `throwTo :: ThreadID -> Exception -> ()` entstammt [MJMR01], und wird unverändert wiedergegeben, da andernfalls auch eine Anpassung des monadischen Kontextes $\mathbb{M}$ notwendig würde, um bei einem allgemeinen `throwTo` e_1 e_2 zunächst e_1 bis zu einer ThreadID auszuwerten, an die dann anschließend Exception e_2 gesendet werden kann.

$$
\begin{array}{ll}
\text{(Propagate)} & (\!|\mathbb{B}[\texttt{throw}\ q \gg\!= e]|\!)_t \longrightarrow (\!|\mathbb{B}[\texttt{throw}\ q]|\!)_t \\
\text{(Catch)} & (\!|\mathbb{B}[\texttt{catch}\,\{\texttt{return}\ e_1\}\,\{e_2\}]|\!)_t \longrightarrow (\!|\mathbb{B}[\texttt{return}\ e_1]|\!)_t \\
\text{(Handle)} & (\!|\mathbb{B}[\texttt{catch}\,\{\texttt{throw}\ q\}\,\{e_2\}]|\!)_t \longrightarrow (\!|\mathbb{B}[(e_2\ q)]|\!)_t
\end{array}
$$

(a) Reduktionsregeln für rein synchrone Exceptions

$$
\begin{array}{ll}
\text{(Block Return)} & (\!|\mathbb{B}[\texttt{block}\,\{\texttt{return}\ e\}]|\!)_t \longrightarrow (\!|\mathbb{B}[\texttt{return}\ e]|\!)_t \\
\text{(Unblock Return)} & (\!|\mathbb{B}[\texttt{unblock}\,\{\texttt{return}\ e\}]|\!)_t \longrightarrow (\!|\mathbb{B}[\texttt{return}\ e]|\!)_t \\
\\
\text{(Block Throw)} & (\!|\mathbb{B}[\texttt{block}\,\{\texttt{throw}\ q\}]|\!)_t \longrightarrow (\!|\mathbb{B}[\texttt{throw}\ q]|\!)_t \\
\text{(Unblock Throw)} & (\!|\mathbb{B}[\texttt{unblock}\,\{\texttt{throw}\ q\}]|\!)_t \longrightarrow (\!|\mathbb{B}[\texttt{throw}\ q]|\!)_t \\
\\
\text{(ThrowTo)} & (\!|\mathbb{B}[\texttt{throw}\ t\ q]|\!)_u \longrightarrow (\!|\mathbb{B}[\texttt{return()}]|\!)_u \mid [\![t \lightning q]\!] \\
\text{(Receive)} & (\!|\mathbb{B}[\texttt{unblock}\,\{\mathbb{M}[e_1]\}]|\!)_t \mid [\![t \lightning q]\!] \longrightarrow (\!|\mathbb{B}[\texttt{unblock}\,\{\mathbb{M}[\texttt{throw}\ q]\}]|\!)_t \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{und } e_1 \neq \texttt{block}\,\{e_2\} \\
\text{(Interrupt)} & (\!|\mathbb{B}[e]|\!)^{\bullet}_t \mid [\![t \lightning q]\!] \longrightarrow (\!|\mathbb{B}[\texttt{throw}\ q]|\!)^{\circ}_t \\
\\
\text{(Stuck PutMVar)} & (\!|\mathbb{B}[\texttt{putMVar}\ x\ e_1]|\!)^{\circ}_t \mid x\,\mathbf{m}\,e_2 \longrightarrow (\!|\mathbb{B}[\texttt{putMVar}\ x\ e_1]|\!)^{\bullet}_t \mid x\,\mathbf{m}\,e_2 \\
\text{(Stuck TakeMVar)} & (\!|\mathbb{B}[\texttt{takeMVar}\ x]|\!)^{\circ}_t \mid x\,\mathbf{m}\,- \longrightarrow (\!|\mathbb{B}[\texttt{takeMVar}\ x]|\!)^{\bullet}_t \mid x\,\mathbf{m}\,-
\end{array}
$$

(b) Reduktionsregeln für asynchrone Exceptions

Abbildung 3.1: Reduktionsregeln synchroner (a) und asynchroner (b) Exceptions zur Ergänzung von Concurrent Haskell aus [MJMR01]. Hierbei bezeichnen e, e_i Ausdrücke, x den Namen einer `MVar`, q eine Exception und t, u ThreadIDs

der blocked/unblocked–Zustände lässt sich leicht angeben, ob der innerste Kontext in einem blocked– oder unblocked–Zustand ist. Ein Zustand, in dem asynchrone Exceptions nicht blockiert werden, kann durch $\mathbb{B}[\texttt{unblock}\ \mathbb{M}]$ dargestellt werden.

3.3.1 Übergangsregeln synchroner Exceptions

Die zu Concurrent Haskell zusätzlich hinzuzufügenden Auswertungsregeln zum Umgang mit synchronen Exceptions sind in Abbildung 3.1(a) dargestellt. Hierbei bezeichnet $(\!|\cdots|\!)_t$ einen Thread mit ThreadID t.

Tritt in einem Ausdruck eine Exception durch `throw` q auf, sorgt die Regel (Propagate) dafür, dass sie so lange „weitergereicht“ wird, bis sie schließlich ein umschließendes `catch`–Konstrukt erreicht. Mittels $\gg\!=$ miteinander verbundene monadische Aktionen werden übergangen, und nicht weiter ausgeführt.

Erreicht solch eine Exception ein umschließendes `catch`, so ruft die Regel (Handle) den zugehörigen Handler e_2 auf und wendet ihn auf die Exceptionbeschreibung q an. Zuvor sollte allerdings der block/unblock-Zustand des Threads, wie in Abschnitt 3.2.2.2 skizziert, auf den Zustand zurückgesetzt werden, in dem er vor Auswertung des von `catch` umschlossenen Arguments war. Möglicherweise bei der Auswertung (durch weitere verschachtelte `block`/`unblock`–Aufrufe) eingetretene Veränderungen dieses Zustandes wurden durch das Weiterleiten der (Propagate)–Regel nicht in gewohnter Weise rückgängig gemacht.

Tritt hingegen keine Exception auf, so wird das Ergebnis der monadischen Auswertung irgendwann als ein Ausdruck der Form (`return` e) vorliegen. In diesem Fall ist alles gut gegangen und der Handler e_2 wird nicht benötigt. Das umschließende `catch`–Konstrukt samt Handler kann durch die Regel (Catch) verworfen werden.

3.3.2 Übergangsregeln asynchroner Exceptions

Zur Behandlung asynchroner Exceptions wird jeder Thread um ein *Stuck-Flag* erweitert, das entweder gesetzt $\bullet$ oder ungesetzt $\circ$ sein kann. Mit $(\!|\cdots|\!)^{\bullet}_{t}$ kann auf diese Weise ein Thread beschrieben und von anderen unterschieden werden, der in eine Wartesituation eingetreten ist. Derartige Threads werden *stuck* genannt, und sollen gemäß den Überlegungen in Abschnitt 3.2.2.3 für die Dauer dieses Zustandes auch bei ausdrücklich blockierten asynchronen Exceptions unterbrechbar sein. In den im Folgenden erklärten Notationen von Abb. 3.1(b) zur Beschreibung der Übergangsregeln bei asynchronen Exceptions, wird das *Stuck-Flag* der Übersicht halber nur angegeben, wenn es tatsächlich von Relevanz ist. Falls nicht anders angegeben, beziehen sich die Regeln daher auf Threads, deren Flag nicht gesetzt $\circ$ ist, und somit normal ausgewertet werden.

Die beiden Regeln (Block Return) und (Unblock Return) dienen dazu, bei erfolgreicher Auswertung eines von `block`/`unblock` umschlossenen Ausdrucks, den jeweiligen Kontext zu verlassen. Nur das eigentliche Ergebnis `return` e des Ausdrucks wird weitergereicht.

Einer innerhalb von `block`/`unblock` umschlossenen Ausdrücken entstandenen Exception `throw` q wird mit den Regeln (Block Throw) und (Unblock Throw) in gleicher Weise ermöglicht, zur weiteren Auswertung weitergereicht zu werden. Dabei ist es nebensächlich, ob es sich bei der aufgetretenen Exception um eine synchrone oder asynchrone Variante handelt, da die Regeln ausschließlich dazu dienen, eine bereits aufgetretene synchrone oder bereits zugestellte asynchrone Exception (ähnlich der (Propagate)–Regel) weiterzureichen, bis ein `catch` erreicht wird.

Asynchrone Exceptions q werden von einem Thread u an einen anderen Thread mit ThreadID t mittels `throwTo` t q gesendet. Die Regel (ThrowTo) modelliert diesen Vorgang. Das Ergebnis einer `throwTo`–Aktion ist der leere monadische Wert `return ()`, wodurch die Auswertung des Threads u direkt fortgesetzt werden kann. Der adressierte Thread t erhält von der an ihn gesendeten Exception zunächst keine unmittelbare Kenntnis. Stattdessen wird sie als separates *ExceptionInFlight*–Element $[\![t \lightning q]\!]$ auf Prozessebene hinzugefügt, und dem Thread t in einem „passenden“ Moment zugestellt.

Dieser passende Moment wird von den beiden Regeln (Receive) und (Interrupt) bestimmt.

Befindet sich ein Thread t in einem Zustand, in dem asynchrone Exceptions ausdrücklich zugelassen werden (d.h. die Auswertung findet innerhalb eines unblock–Kontextes statt, woran die direkt nächste auszuführende Aktion auch nichts ändert), und existiert eine an diesen Thread adressierte *ExceptionInFlight*, dann kann sie problemlos „empfangen“ werden. Die Regel (Receive) modelliert genau diesen Vorgang. Das $[\![t \lightning q]\!]$–Element wird auf Prozessebene entfernt, und in ein `throw` q umgewandelt, das in den auszuwertenden Ausdruck des Threads t injiziert wird.

Befindet sich ein Thread in einem Zustand, in dem asynchrone Exceptions blockiert werden (block–Kontext), dürfen keine Exceptions zugestellt werden. Entsprechend ist für solch einen Fall auch keine Übergangsregel notiert. Lediglich in denjenigen Momenten, in denen ein Thread *stuck* wird, was durch ein gesetztes *Stuck-Flag* angezeigt wird, dürfen Exceptions zugestellt werden. Die Regel (Interrupt) injiziert eine Exception unabhängig vom block/unbock–Zustand, sofern das *Stuck-Flag* gesetzt ist, und entfernt das *ExceptionInFlight*–Element.

Threads werden durch das *Stuck-Flag* als *stuck* gekennzeichnet, wenn Aktionen in eine Wartesituation geraten, die sich womöglich nie mehr ändert. Für die bisher in dieser Arbeit vorgestellten funktionalen Kalküle sind dies die Operationen `putMVar` und `takeMVar`. Die Regeln (Stuck PutMVar) und (Stuck TakeMVar) modellieren genau jene Situationen, die bisher nicht vorgesehen waren. Wird versucht, mit `putMVar` in eine bereits gefüllte `MVar` zu

schreiben, so ändert (Stuck PutMVar) zwar nicht den auszuwertenden Ausdruck, setzt aber das *Stuck-Flag*. In gleicher Weise wird es bei dem Versuch gesetzt, mittels `takeMVar` einen Wert aus einer leeren `MVar` auszulesen. Konsequenterweise muss bei jedem Erfolg einer `takeMVar`/`putMVar`–Operation ein gesetztes *Stuck-Flag* wieder gelöscht werden, sofern in der Zwischenzeit keine Exception eingetreten ist.

$$\text{(UnStuck TakeMVar)} \quad (\!|\mathbb{M}[\texttt{takeMVar}\ x]|\!)^{\bullet}_t \mid x\,\mathbf{m}\,e \longrightarrow (\!|\mathbb{M}[\texttt{return}\ e]|\!)^{\circ}_t \mid x\,\mathbf{m}\,-$$

$$\text{(UnStuck PutMVar)} \quad (\!|\mathbb{M}[\texttt{putMVar}\ x\ e]|\!)^{\bullet}_t \mid x\,\mathbf{m}\,- \longrightarrow (\!|\mathbb{M}[\texttt{return}\ ()]|\!)^{\circ}_t \mid x\,\mathbf{m}\,e$$

Die bisher verwendeten Übergangsregeln für `putMVar` und `takeMVar` werden also um die Verwendung des *Stuck-Flag* erweitert. Da der Thread normal weiter ausgewertet werden kann, hängt das Zustellen von asynchronen Exceptions nun wieder allein vom jeweiligen block/unblock–Zustand des Threads ab.

4

Concurrent Haskell mit Futures und Exceptions

Da es sich sowohl beim *CHF*–Kalkül aus Kapitel 2, als auch bei der Einführung des Exception-Handlings aus Kapitel 3 jeweils um eine Erweiterung von Concurrent Haskell handelt, liegt es nahe, das eine mit dem anderen zu verbinden. Das Ergebnis sollte ein Programmkalkül sein, der nebenläufige Berechnungen in Form von Futures unterstützt und zusätzlich in der Lage ist, mit synchronen und insbesondere asynchronen Exceptions umzugehen.

Ziel dieses Kapitels ist es, die dafür notwendigen Anpassungen der Sprache in Abschnitt 4.1 zu diskutieren. Anschließend wird die gemeinsame Syntax für *Concurrent Haskell mit Futures und Exceptions* (kurz: *CHFE*) in Abschnitt 4.2 vorgestellt. Der Typcheck wird entsprechend in Abschnitt 4.3 erweitert.

Für die Operationale Semantik, die in diesem Kapitel erstmalig in Form Abstrakter Maschinen angegeben wird, wird das *CHFE*–Programm in einer spezielle Maschinensyntax benötigt. Deren Syntax, sowie die dafür notwendige Transformation wird in Abschnitt 4.4.1 gegeben.

Es folgt die Definition der Abstrakten Maschine in einer dreigliedrigen Architektur. Rein funktionale Auswertungen auf der Mark 1 werden in Abschnitt 4.4.2 vorgestellt. Diese Maschine wird in Abschnitt 4.4.3 um monadische Auswertungen zur I/O–Mark 1 erweitert. Nebenläufige Auswertung mehrerer Threads schließlich, findet auf der Concurrent–Mark 1 in Abschnitt 4.4.4 statt.

4.1 Zusammenführung von Futures und Exceptions

Die Erweiterung des *CHF*–Kalküls um synchrone Exceptions sollte im Wesentlichen keine Schwierigkeiten bereiten, da beide Kalküle auf einem ähnlichen Kalkülfundament aufbauen. Da aber in *CHF* der ursprüngliche `forkIO`–Befehl zugunsten eines `future`–Befehls zum Erzeugen von Futures ersetzt wurde, fehlt an dieser Stelle die zum Senden von asynchronen Exceptions notwendige ThreadID zur eindeutigen Identifikation des Zielthreads. Ergebnis eines in *CHF* mit `future` e erzeugten Threads ist eine Variable x, die nach Auswertung des Ausdrucks e dessen Ergebniswert enthält.

Obwohl die Future mit x zwar eindeutig referenzierbar ist, scheitert der naive Ansatz, einem `throwTo`–Befehl zu erlauben, als erstes Argument die Future-Variable zuzulassen. Betrachtet man dazu das konstruierte Beispiel

$$\texttt{future}\ e \gg\!= \lambda x.\texttt{return}(x) \gg\!= \lambda y.\texttt{throwTo}\ y\ q$$

lässt sich erkennen, dass es durchaus Situationen geben kann, in denen das erste Argument von `throwTo` (hier y) noch soweit ausgewertet werden muss, bis die zu adressierende Future x vorliegt. Das Wesen der Futures bedingt gerade, dass für die Sprache nicht erkennbar ist, ab wann eine Variable „normal“, oder aber eine nebenläufig ausgewertete Future ist. Für die Sprache wäre in diesem Fall nicht erkennbar, dass sie die Auswertung des ersten Arguments y bei Erreichen von x anhalten, mit der `throwTo`–Aktion beginnen, und die Exception q an die Future x senden müsste. Tatsächlich würde sie die Aktion erst ausführen, wenn sie das mit x referenzierte, nebenläufig berechnete Ergebnis des Ausdrucks e erreicht hat.

Falls aber $e \Uparrow$, terminiert die Auswertung der Future nicht. Als Konsequenz terminiert nun auch die Auswertung von `throwTo` x q nicht, und die gerade wünschenswerte Möglichkeit, eine nicht-terminierende Future mittels einer asynchronen Exception beenden zu können, scheitert zwangläufig.

4.1.1 Futures mit ThreadID

Als Ausweg aus dieser Situation muss die bereits von `forkIO` bekannte ThreadID, zur eindeutigen Kennzeichnung eines Threads, auch zur Kennzeichnung einer Future wieder eingeführt werden. Der in *CHF* eingeführte `future`–Befehl wird dazu derart angepasst, dass sein Ergebnis nunmehr ein Paar darstellt, dessen erste Komponente weiterhin die Future an sich liefert. Die zweite Komponente stellt die bekannte ThreadID dar. Zur Darstellung dieses Paars wird ein Konstruktor *FIDCons* der Form $\langle \mathit{Future}; \mathit{ThreadID} \rangle$ eingeführt, der in *CHFE* als Ergebnis eines `future`–Aufrufs zurückgegeben werden soll.

Die daraus resultierende bedingte Unterscheidung zwischen Future und ThreadID macht zusätzliche Anpassungen am Typsystem notwendig. Dazu werden zwei zusätzliche Typkonstruktoren `IDTyp` und `TFID` α eingeführt, wodurch *FIDCons* wie folgt typisiert werden kann: $\langle a :: \alpha; b :: \texttt{IDTyp} \rangle :: \texttt{TFID}\ \alpha$. Die ThreadID b erhält den Typ `IDTyp`, wodurch die Sprache in die Lage versetzt wird, ThreadIDs durch ihren Typ eindeutig von Ausdrücken oder Variablen unterscheiden zu können. Futures a erhalten fortan den Typ α, das gerade denjenigen Typ darstellt, den das Ergebnis des Ausdrucks e einer Future annehmen wird.

Zum korrekten Typisieren von Programmen wird dem *FIDCons*–Konstruktor selbst der Typ der Future `TFID` α gegeben, womit Folgeausdrücken der Typ α der Future mitgeteilt werden kann.

Auf diese Weise kann mit den bereits in der Sprache vorhandenen Mitteln problemlos auf die eigentliche Future mit *future* x, oder mit *futureID* x auf die TreadID zugegriffen werden, wobei x ein *FIDCons* ist. Eine simple Fallunterscheidung genügt dazu:

$$\begin{array}{l}\texttt{letrec}\\ \quad \mathit{future} \quad = \lambda m.(\texttt{case}_{\texttt{TFID}\,\alpha}\ m\ \texttt{of}\ \{\langle a;b\rangle \rightarrow a\});\\ \quad \mathit{futureId} \ = \lambda m.(\texttt{case}_{\texttt{TFID}\,\alpha}\ m\ \texttt{of}\ \{\langle a;b\rangle \rightarrow b\});\\ \texttt{in}\ \ldots\end{array}$$

Für das obige Beispiel genügt nun eine Zeile der Form

$$\texttt{future}\ e \gg\!= \lambda x.\texttt{return}(x) \gg\!= \lambda y.\texttt{throwTo}\ (\mathit{futureID}\ y)\ q$$

um eine Exception q an die Future x zu senden.

4.1.2 Behandlung einer asynchronen Exception in einer Future

Mit den vorgestellten Anpassungen ist es nun möglich, asynchrone Exceptions auch an Futures zu senden.

Anders als bei Concurrent Haskell findet die Kommunikation mit Futures allerdings nicht ausschließlich über Strukturen wie z.B. `MVars` statt. Besonderheit der Futures ist gerade die Möglichkeit, an weiteren Stellen im Ausdruck als Variablen referenziert werden zu können.

Zusätzlich zur Wahrung des konsistenten Zustandes beteiligter `MVars` muss nun auch sichergestellt werden, dass eine Future bei Auftreten einer Exception dennoch ein Ergebnis liefert, mit dem ein darauf zugreifender Ausdruck sinnvoll umgehen kann. Trotz Auftretens einer Exception muss die Auswertung einer Zeile der Form

$$\texttt{future}\ e \gg\!= \lambda x.\texttt{throwTo}\ (\mathit{futureId}\ x)\ q \gg\!= \texttt{return}(\mathit{future}\ x) \gg\!= \ldots$$

also dennoch fehlerfrei möglich sein. Die Verantwortung hierfür liegt abermals beim Programmierer, der diesen Fall beim Entwerfen des Exception-Handlers sorgfältig bedenken muss.

Die in Kapitel 3 angegebenen Regeln zeigen, dass bei Auftreten einer Exception, der aktuell auszuwertende Ausdruck unterbrochen wird, und die Exception mit der (Propagate)-Regel bis zum nächsten umschließenden `catch` weitergeleitet wird. Die durch die Unterbrechung der monadischen Aktionen möglicherweise auftretenden Inkonsistenzen müssen vom Programmierer explizit im Handler bereinigt werden.

Die (Propagate)–Regel „überspringt“ monadische Aktionen. Zu klären bleibt, was mit der Auswertung eines funktionalen Ausdrucks bei Eintreten einer asynchronen Exception zu geschehen hat. Für sich in der Auswertung befindliche unterbrochene funktionale Berechnungen stellt bspw. [MJMR01] zwei Möglichkeiten der Behandlung vor:

1. Einfrieren des Ausdrucks: Handelte es sich um eine asynchrone Exception, ist ungewiss, ob diese bei erneutem Auswerten des Ausdrucks abermals auftritt. In [Rei98] wird beispielhaft diskutiert, wie in solch einem Fall der Ausdruck „eingefroren“ und die Auswertung bei erneutem Aufruf an der gleichen Stelle (diesmal möglicherweise ununterbrochen) fortgesetzt werden kann.

2. Zurücksetzen auf Initialzustand: Eine erneute Auswertung des gleichen Ausdrucks beginnt mit dessen Initialzustand. Alle vorherigen Auswertungsschritte wurden bei Eintreten einer asynchronen Exception rückgängig gemacht.

4.2 Die Syntax von *CHFE*–Ausdrücken und Typen

Mit den oben bereits angestellten Überlegungen kann nun die Syntax des neuen *CHFE*–Kalküls in Form einer Grammatik angegeben werden:

$$
\begin{array}{rcl}
e, e_i \in \mathit{Expr}_{\mathit{CHFE}} & ::= & x \mid me \mid \lambda x.e \mid (e_1\ e_2) \mid (c\ e_1 \ldots e_{\mathrm{ar}(c)}) \mid \texttt{seq}\ e_1\ e_2 \\
& \mid & \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e \text{ wobei } n \geq 1 \\
& \mid & \texttt{case}_T\ e\ \texttt{of}\ \mathit{alt}_{T,1} \ldots\ \mathit{alt}_{T,|T|} \\
& & \quad \text{mit } \mathit{alt}_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i) \\
& \mid & \langle x; t \rangle, \text{ wobei } x \text{ eine Future} \\
& & \quad \text{und } t \text{ die zugehörige FutureID ist.} \\
\\
\mathit{me} \in \mathit{MExpr}_{\mathit{CHFE}} & ::= & \texttt{return}\ e \mid e_1 \gg\!= e_2 \mid \texttt{future}\ e \\
& \mid & \texttt{takeMVar}\ e \mid \texttt{newMVar}\ e \mid \texttt{putMVar}\ e_1\ e_2 \\
& \mid & \texttt{block}\ \{e\} \mid \texttt{unblock}\ \{e\} \mid \texttt{catch}\ \{e_1\}\ \{e_2\} \\
& \mid & \texttt{throw}\ e \mid \texttt{throwTo}\ e_1\ e_2
\end{array}
$$

Die Syntax der Ausdrücke $\mathit{Expr}_{\mathit{CHFE}}$ wird von *CHF* übernommen und um den speziellen Konstruktor *FIDCons* erweitert. Den monadischen Ausdrücken $\mathit{MExpr}_{\mathit{CHFE}}$ werden die benötigten Primitiven `throw`, `throwTo`, `block {·}`, `unblock {·}` und `catch {·} {·}` mit explizit ausgewiesener Klammerung hinzugefügt.

Die Syntax der Typen wird um die in Abschnitt 4.1.1 eingeführten Typkonstruktoren `TFID` und `IDTyp` erweitert, und der $\mathit{Proc}_{\mathit{CHFE}}$–Syntax das *ExceptionInFlight*–Konstrukt $[\![t \lightning e]\!]$ zum Senden asynchroner Exceptions an andere Prozesse hinzugefügt.

$$
\begin{array}{rcl}
\tau, \tau_i \in \mathit{Typ}_{\mathit{CHFE}} & ::= & \texttt{IO}\ \tau \mid (T\ \tau_1 \ldots \tau_n) \mid \texttt{MVar}\ \tau \mid \tau_1 \to \tau_2 \mid \texttt{TFID}\ \tau \mid \texttt{IDTyp} \\
P, P_i \in \mathit{Proc}_{\mathit{CHFE}} & ::= & P_1 \mid P_2 \mid x \Leftarrow e \mid \nu x.P \mid x\,\mathbf{m}\,e \mid x\,\mathbf{m}- \mid x = e \mid [\![t \lightning e]\!]
\end{array}
$$

Hierbei sind τ, τ_i Typen, T ein Typkonstruktor, P, P_i Prozesse, x ist eine Variable, e ein beliebiger Ausdruck und t eine FutureID.

4.3 Erweiterung des Typchecks für die neuen Konstrukte

Die Erweiterung und Anpassung der Syntax machen Anpassungen am Typsystem und dem Unifikationsalgorithmus notwendig. Hierzu wird das bereits in Kapitel 2 vorgestellte Typsystem von *CHF* entsprechend erweitert und dem Unifikationsalgorithmus die fehlenden Regeln hinzugefügt.

4.3.1 Zusätzliche Typisierungsregeln

Zum Typisieren eines *CHFE*–Ausdrucks müssen für die neu hinzugenommenen Konstrukte `block`, `unblock`, `catch`, `throw`, `throwTo` und den Konstruktor *FIDCons* jeweils eigene Typisierungeregeln, zu den bereits bei *CHF* in Abbildung 2.5 vorgestellten Regeln, hinzugefügt werden. Außerdem muss die dort angegebene Typisierungsregel für den Befehl `future` derart angepasst werden, dass ihr Rückgabewert nunmehr ein Konstruktor mit speziellem Typ ist.

Im Folgenden werden die notwendigen neuen Typisierungeregeln vorgestellt und im einzelnen erläutert. Da es sich hierbei definitionsgemäß nur um monadische Aktionen handelt, verlassen sie die I/O–Monade nicht und liefern allesamt durch `IO` gekennzeichnete Typen.

Der leere Typ `()` (auch `Unit` genannt) ist fest (z.B. mit `data Unit = Unit`) implementiert.

4.3.1.1 Typisierungsregel (*Block*) und (*Unblock*)

$$(Block)\ \frac{\Gamma \vdash e :: \texttt{IO}\ \tau, E}{\Gamma \vdash \texttt{block}\,\{e\} :: \texttt{IO}\ \tau, E} \qquad (Unblock)\ \frac{\Gamma \vdash e :: \texttt{IO}\ \tau, E}{\Gamma \vdash \texttt{unblock}\,\{e\} :: \texttt{IO}\ \tau, E}$$

Da `block` und `unblock` für die eigentliche Auswertung eines Ausdrucks unwesentlich sind, sondern nur dazu verwendet werden, den internen Zustand eines Threads umzuschalten, werden sie beim Typcheck derart „übergangen“, dass ihnen gerade der Typ ihres eingeschlossenen Ausdrucks e zugewiesen wird.

4.3.1.2 Typisierungsregel (*Throw*)

$$(Throw)\ \frac{\Gamma \vdash e :: \tau, E}{\Gamma \vdash (\texttt{throw}\ e) :: \texttt{IO}\ \alpha, E \cup \{\tau \doteq \texttt{Exception}\}}\ \text{wobei } \alpha \text{ eine neue Typvariable ist}$$

Das Auftreten einer synchronen Exception kann mit `throw` im Programm angezeigt werden, wobei eine Exception e dabei den speziellen Typ `Exception` hat. Eine synchrone Exception wird durch die neue Typvariable α mit einem allgemeinen Typ versehen, da sie meist dem umschließenden Ausdruck angepasst werden muss. Oft geht dem Generieren einer synchronen Exception z.B. eine Fallunterscheidung voraus:

$$\begin{array}{l}\texttt{case}\ e\ \texttt{of}\ \{\\ \quad Alt_1 \quad \rightarrow \texttt{return}\ True;\\ \quad Alt_2 \quad \rightarrow \texttt{return}\ False;\\ \qquad \ldots\\ \quad Alt_n \quad \rightarrow \texttt{throw}\ SomeError;\\ \}\end{array}$$

wobei die Typisierung von `case` verlangt, dass alle Ergebnisausdrücke vom gleichen Typ sein müssen. Bei normaler Ausführung liefert der Ausdruck ein Ergebnis vom Typ `Bool`, im Fehlerfall aber vom Typ `Exception`. Durch den allgemeinen Typ von $\texttt{throw} :: \texttt{IO}\ \alpha$ kann dieser Ausdruck dennoch richtig typisiert werden. Der inhärent zugelassene Typfehler kann ignoriert werden, da sowieso eine synchrone Exception aufgetreten ist, die die Auswertung an dieser Stelle abbricht, und bis zum nächsten umschließenden `catch` weitergeleitet wird.

4.3.1.3 Typisierungsregel (*ThrowTo*)

$$(ThrowTo)\ \frac{\Gamma \vdash e_1 :: \tau_1, E_1 \text{ und } \Gamma \vdash e_2 :: \tau_2, E_2}{\Gamma \vdash (\texttt{throwTo}\ e_1\ e_2) :: \texttt{IO}\ (), E_1 \cup E_2 \cup \{\tau_1 \doteq \texttt{IDTyp}, \tau_2 \doteq \texttt{Exception}\}}$$

Die Typisierung von $\texttt{throwTo}\ e_1\ e_2$ zum Senden einer asynchronen Exception ist genau dann erfolgreich, wenn e_1 eine FutureID vom Typ `IDTyp`, und e_2 eine Exception vom Typ `Exception` ist. Diese monadische Operation liefert kein Ergebnis und ist damit insgesamt vom Typ `()`.

4.3.1.4 Typisierungsregel (*Catch*)

$$(Catch)\ \frac{\Gamma \vdash e_1 :: \tau_1, E_1 \text{ und } \Gamma \vdash e_2 :: \tau_2, E_2}{\Gamma \vdash \texttt{catch}\,\{e_1\}\,\{e_2\} :: \tau_1, E_1 \cup E_2 \cup \{\tau_2 \doteq \texttt{Exception} \rightarrow \texttt{IO}\ \tau_1, \tau_1 \doteq \texttt{IO}\ \alpha\}}$$

Der Typ eines `catch`–Ausdrucks entspricht dem des von ihm umschlossenen Ausdrucks e_1, da `catch` ohne Auswirkung bleibt, solange keine Exception auftritt. Im Fall einer Exception vom Typ `Exception` wird diese an den Handler übergeben, der dementsprechend eine Funktion sein muss, die ein Argument dieses Typs bekommt, und als Ergebnis den gleichen

Typ τ_1 wie der Ausdruck e_1 liefern muss. Damit wird für die weitere Typisierung sichergestellt, dass `catch` den Typ τ_1 liefert, unabhängig davon, ob e_1 fehlerfrei ausgewertet werden kann, oder aber eine Exception auftritt.

4.3.1.5 Typisierungsregel (*FIDCons*)

$$(\mathit{FIDCons})\ \frac{\Gamma \vdash e_1 :: \tau_1, E_1 \text{ und } \Gamma \vdash e_2 :: \tau_2, E_2}{\Gamma \vdash (\langle e_1; e_2 \rangle) :: \alpha, E_1 \cup E_2 \cup \{\alpha \doteq \texttt{TFID}\ \tau_1, \tau_2 \doteq \texttt{IDTyp}\}}$$

Der neu eingeführte Konstruktor *FIDCons* umschließt ein Paar, bestehend aus Future und FutureID. Die Gleichungen erzwingen, dass die FutureID e_2 vom Typ `IDTyp` ist, und der Konstruktor selbst mit dem monadischen Typ `TFID` α typisiert wird, wobei α gerade den allgemeinen Typ der Future darstellt.

4.3.1.6 Typisierungsregel (*Fork*)

$$(\mathit{Fork})\ \frac{\Gamma \vdash e :: \tau, E}{\Gamma \vdash (\texttt{future}\ e) :: \texttt{IO}\ (\texttt{TFID}\ \alpha), E \cup \{\tau \doteq \texttt{IO}\ \alpha\}}$$

Die Typregel für `future` lieferte in *CHF* den Typ, den der nebenläufig auszuwertende Ausdruck innehatte (vgl. Abb. 2.5). Da in *CHFE* die Aktion `future` jedoch ein Datenpaar, bestehend aus ursprünglicher Future und FutureID, in Form des Konstruktors *FIDCons* liefert, wird sie gemäß des Konstruktors mit `TFID` α typisiert. Die zusätzliche Gleichung stellt sicher, dass nur monadische Aktionen als Futures zugelassen werden.

4.3.2 Anpassung des Unifikationsalgorithmus

Zum korrekten Unifizieren der entstehenden Gleichungen, mit dem für *FIDCons* neu eingeführten Typkonstruktor `TFID`, muss der Unifikationsalgorithmus aus Abbildung 2.6 um die entsprechenden Regeln erweitert werden.

$$\textsc{Decomp.TFID}\ \frac{E_g, \{\texttt{TFID}\ \tau_1 \doteq \texttt{TFID}\ \tau_2\} \cup E_u}{E_g, \{\tau_1 \doteq \tau_2\} \cup E_u}$$

$$\textsc{FailTFID}\ \frac{E_g, \{\texttt{TFID}\ \tau_1 \doteq \tau_2\} \cup E_u}{\mathit{Fehler}}, \quad \begin{array}{c}\text{falls } \tau_2 = T,\ \tau_2 = \texttt{MVar}\ \tau',\\ \tau_2 = \tau_1 \to \tau_2 \text{ oder } \tau_2 = \texttt{IO}\ \tau'\end{array}$$

Damit wird sichergestellt, dass Gleichungen nur dann mit (Decomp.TFID) unifiziert werden köennen, wenn ihre Typen in diesem Schritt bis auf die τ_i gleich sind, andernfalls muss mit (FailTFID) ein Typfehler angezeigt werden.

4.4 Semantik in Form Abstrakter Maschinen

Anders als in den Kapiteln 2 und 3 wird die Semantik der neuen Sprache *CHFE* etwas präziser angegeben. Als Vorlage dient eine von David Sabel in [Sab12] für *CHF* vorgestellte *Abstrakte Maschine*. Dabei handelt es sich um ein formales Automatenmodell, das von einem Anfangszustand aus, für jeden möglichen Folgezustand einen Übergang definiert.

Als Vorteile Abstrakter Maschinen werden in [Sab08, S. 241] deren kleinere, nachvollziehbarere Auswertungsschrittfolge und die direkt implementierte Suche des nächsten Redexes durch Zustandsübergänge unter Zuhilfenahme eines Stacks genannt. Auswertungskontexte sind also inhärent und nicht explizit angegeben. Zusätzlich wird durch eine Abstrakte Maschine direkt ein Interpreter für die durch den Kalkül definierte Sprache geliefert.

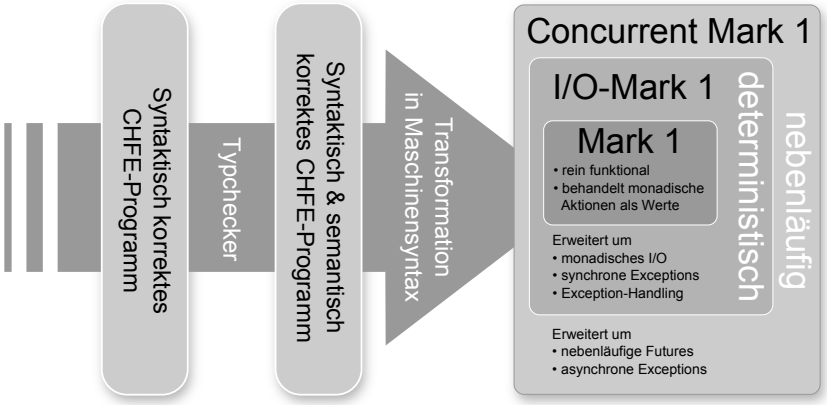

Abbildung 4.1: Schematische Darstellung der dreigliedrigen Architektur der Abstrakten Maschinen Concurrent–Mark 1, I/O–Mark 1 und Mark 1.

Die Abstrakte Maschine zur Auswertung von *CHFE* wird dabei in der gleichen dreigliedrigen Architektur beschrieben wie die Abstrakte Maschine für *CHF* aus [Sab12] (vgl. Abb. 4.1): Auf einer angepassten, ursprünglich von Peter Sestoft in [Ses97] vorgestellten Maschine Mark 1 werden rein funktionale Berechnungen eines erweiterten Lambda–Kalküls durchgeführt. Monadische Aktionen zur Verwendung von `MVar`s werden in einem zweiten Schritt hinzugefügt und bilden die Maschine I/O–Mark 1. Im letzten Schritt wird der Maschine Concurrent–Mark 1 die Möglichkeit gegeben, mehrere nebenläufige Threads auszuwerten und zu verwalten.

Abstrakte Maschinen auf Basis von Sestofts Architektur benötigen eine leicht modifizierte Syntax. Die dazu notwendige Transformation wird im Folgenden vorgestellt. Im Anschluss werden die drei Abstrakten Maschinen im Detail beschrieben.

4.4.1 Transformation in vereinfachte Syntax

Zum Verarbeiten eines Programms auf der Abstrakten Maschine wird eine leicht veränderte Syntax benötigt. Dazu werden die Ausdrücke derart transformiert, dass als Argumente in Anwendungen und Konstruktoren nur noch Variablen x, x_i vorkommen. Die Argumente monadischer Operationen und das zweite Argument von `seq` dürfen ebenfalls nur Variablen sein.

4.4.1.1 Syntax der Maschinenausdrücke

Die Syntax der *Maschinenausdrücke* $ExprM_{CHFE}$ und monadischen Maschinenausdrücke $MExprM_{CHFE}$ wird, der Notation in [Sab12, Sab11] folgend, formal wie folgt definiert:

$$
\begin{array}{rcl}
e, e_i \in ExprM_{CHFE} & ::= & x \mid me \mid \lambda x.e \mid (e\ x) \mid c\ x_1 \ldots x_{\mathrm{ar}(c)} \mid \texttt{seq}\ e\ x \\
& \mid & \texttt{letrec}\ x_1 = e_1\ \ldots\ x_n = e_n\ \texttt{in}\ e \\
& \mid & \texttt{case}_T\ e\ \texttt{of}\ alt_{T,1} \ldots\ alt_{T,|T|} \\
& & \quad \text{wobei } alt_{T,i} = (c_{T,i}\ x_1 \ldots x_{\mathrm{ar}(c_{T,i})} \to e_i) \\
& \mid & \texttt{TID}\ x \\
me \in MExprM_{CHFE} & ::= & \texttt{return}\ x \mid x_1\ \texttt{>>=}\ x_2 \mid \texttt{future}\ x \\
& \mid & \texttt{takeMVar}\ x \mid \texttt{newMVar}\ x \mid \texttt{putMVar}\ x_1\ x_2 \\
& \mid & \texttt{block}\ x \mid \texttt{unblock}\ x \\
& \mid & \texttt{throw}\ x \mid \texttt{throwTo}\ x_1\ x_2 \mid \texttt{catch}\ x_1\ x_2
\end{array}
$$

Vereinfachte Prozesse $ProcM_{CHFE}$ werden wie *Proc* definiert, wobei alle Ausdrücke Maschinenausdrücke sind, und alle `MVar`s nur Variablen als Inhalt haben.

Das neu hinzugekommene `TID` ist ein internes Konstrukt, das zur expliziten Kennzeichnung einer FutureID verwendet wird, und ausschließlich dann erzeugt wird, wenn eine neue Future erstellt wird.

4.4.1.2 Transformationsalgorithmus

Die eigentliche Transformation in diese Syntax wird durch Einführen neuer `letrec`–Ausdrücke bewerkstelligt, wobei $[\![e]\!]$ den zu einem Ausdruck e gehörenden Maschinenausdruck bezeichnet.[8]

- $[\![(e_1\ e_2)]\!] = \texttt{letrec}\ x = [\![e_2]\!]\ \texttt{in}\ ([\![e_1]\!]\ x)$, wobei x eine neue Variable ist.
- $[\![(c\ e_1 \ldots e_n)]\!] = \texttt{letrec}\ x_1 = [\![e_1]\!] \ldots x_n = [\![e_n]\!]\ \texttt{in}\ (c\ x_1 \ldots x_n)$, wobei $x_1 \ldots x_n$ neue Variablen sind.

  Da *FIDCons* als normaler Konstruktor behandelt wird, greift diese Transformation gleichfalls auch für $[\![\langle e_1; e_2\rangle]\!] = \texttt{letrec}\ x_1 = [\![e_1]\!], x_2 = [\![e_2]\!]\ \texttt{in}\ \langle x_1; x_2\rangle$.
- $[\![\texttt{seq}\ e_1\ e_2]\!] = \texttt{letrec}\ x = [\![e_2]\!]\ \texttt{in}\ (\texttt{seq}\ [\![e_1]\!]\ x)$, wobei x eine neue Variable ist.
- $[\![\texttt{return}\ e]\!] = \texttt{letrec}\ x = [\![e]\!]\ \texttt{in return}\ x$, wobei x eine neue Variable ist.
- $[\![e_1 \gg\!= e_2]\!] = \texttt{letrec}\ x_1 = [\![e_1]\!], x_2 = [\![e_2]\!]\ \texttt{in}\ x_1 \gg\!= x_2$, wobei x_1 und x_2 neue Variablen sind.
- $[\![\texttt{future}\ e]\!] = \texttt{letrec}\ x = [\![e]\!]\ \texttt{in future}\ x$, wobei x eine neue Variable ist.
- $[\![\texttt{takeMVar}\ e]\!] = \texttt{letrec}\ x = [\![e]\!]\ \texttt{in takeMVar}\ x$, wobei x eine neue Variable ist.
- $[\![\texttt{putMVar}\ e_1\ e_2]\!] = \texttt{letrec}\ x_1 = [\![e_1]\!], x_2 = [\![e_2]\!]\ \texttt{in putMVar}\ x_1\ x_2$, wobei x_1 und x_2 neue Variablen sind.
- $[\![\texttt{newMVar}\ e]\!] = \texttt{letrec}\ x = [\![e]\!]\ \texttt{in newMVar}\ x$, wobei x eine neue Variable ist.
- $[\![\texttt{block}\,\{e\}]\!] = \texttt{letrec}\ x = [\![e]\!]\ \texttt{in block}\ x$, wobei x eine neue Variable ist.
- $[\![\texttt{unblock}\,\{e\}]\!] = \texttt{letrec}\ x = [\![e]\!]\ \texttt{in unblock}\ x$, wobei x eine neue Variable ist.
- $[\![\texttt{catch}\,\{e_1\}\,\{e_2\}]\!] = \texttt{letrec}\ x_1 = [\![e_1]\!], x_2 = [\![e_2]\!]\ \texttt{in catch}\ x_1\ x_2$, wobei x_1 und x_2 neue Variablen sind.
- $[\![\texttt{throw}\ e]\!] = \texttt{letrec}\ x = [\![e]\!]\ \texttt{in throw}\ x$, wobei x eine neue Variable ist.
- $[\![\texttt{throwTo}\ e_1\ e_2]\!] = \texttt{letrec}\ x_1 = [\![e_1]\!], x_2 = [\![e_2]\!]\ \texttt{in throwTo}\ x_1\ x_2$, wobei x_1 und x_2 neue Variablen sind.

Für alle anderen Konstrukte wird $[\![\cdot]\!]$ homomorph über die Termstruktur gezogen:

- $[\![x]\!] = x$
- $[\![\lambda x \to e]\!] = \lambda x.[\![e]\!]$
- $[\![\texttt{letrec}\ x_1 = e_1 \ldots x_n = e_n\ \texttt{in}\ e]\!] = \texttt{letrec}\ x_1 = [\![e_1]\!] \ldots x_n = [\![e_n]\!]\ \texttt{in}\ [\![e]\!]$
- $[\![\texttt{case}_T\ e\ \texttt{of}\ \{pat_1 \to e_1; \ldots; pat_{|T|} \to e_{|T|}\}]\!]$
  $= \texttt{case}_T\ [\![e]\!]\ \texttt{of}\ \{pat_1 \to [\![e_1]\!]; \ldots; pat_{|T|} \to [\![e_{|T|}]\!]\}$

[8] Es sei an dieser Stelle nochmals daran erinnert, dass standardmässig angenommen wird, dass die *DVC* immer gilt, und alle gebundenen Variablen stets paarweise disjunkte Bezeichner haben.

4.4.2 Abstrakte Maschine Mark 1

Die in [Sab12] vorgestellte Mark 1 (kurz: *M1*) implementiert eine, um getyptes `case` und sequentielle Auswertung mit `seq` erweiterte Abstrakte Maschine aus [Ses97] zur Auswertung des rein funktionalen, erweiterten Lambda-Kalküls.

Ein Zustand dieser Maschine ist ein Tupel der Form $(\mathcal{H}, e, \mathcal{S})$. Hierbei beschreibt $\mathcal{H}$ einen Heap, e den aktuell auszuwertenden Ausdruck (dieser wird *Control* genannt) und $\mathcal{S}$ einen Stack.

Ein Heap $\mathcal{H}$ definiert eine Menge von Heapbindungen der Form $\{x \mapsto e\}$, wobei x eine eindeutige (d.h. nur einmal im Heap vorkommende) Heapvariable und e einen Ausdruck darstellt.

Zur Darstellung eines Heaps wird die Mengenschreibweise verwendet. Ein leerer Heap wird durch $\emptyset$ dargestellt, und mit $\mathcal{H}_1 \dot\cup \mathcal{H}_2$ wird die disjunkte Vereinigung zweier Heaps $\mathcal{H}_1$ und $\mathcal{H}_2$ bezeichnet.

Ein Stack $\mathcal{S}$ ist eine LIFO–Datenstruktur (L*ast*-I*n*-F*irst*-O*ut*) mit zwei Operationen *push* und *pop*. Ein Element x kann mit *push* x auf den Stack gespeichert werden. Mit der Operation *pop* wird das zuletzt auf dem Stack abgelegte Element wieder vom Stack entfernt und als Ergebnis zurückgegeben.

Ein Stack wird in Form einer Liste dargestellt. Die Notation $x : \mathcal{S}$ wird verwendet, um anzudeuten, dass ein Element x als oberstes Element auf den Stack $\mathcal{S}$ gespeichert wurde. Ein leerer Stack wird durch eine leere Liste [] gekennzeichnet.

Der Stack wird verwendet, um sich bei der Auswertung eines Ausdrucks jene Teile zu merken, deren Auswertung erst später an der Reihe sind. Dies modelliert gerade die Auswertungskontexte, die dadurch implizit angegeben werden. Auf dem Stack $\mathcal{S}$ der Mark 1 können nur eine zuvor bestimmte Menge von Elementen gespeichert werden. Diese Elemente können sein:

- $\#_{\mathsf{app}}(x)$ bei der Auswertung von einer Applikation $(e\ x)$ wird x auf dem Stack gespeichert, bis der Ausdruck e ausgewertet wurde. Anschließend kann x wieder vom Stack geholt und die Funktion darauf angewendet werden.
- $\#_{\mathsf{case}}(alts)$ bei der Auswertung von `case` e `of` $alts$ werden zunächst die unterschiedlichen Alternativen $alts$ als $\#_{\mathsf{case}}(alts)$ auf dem Stack gesichert. Nach Auswertung des Ausdrucks e werden sie zum Vergleichen wieder vom Stack geholt.
- $\#_{\mathsf{seq}}(x)$ sichert zur vorrangigen Auswertung des ersten Arguments e in einem Ausdruck `seq` $e\ x$ das zweite Argument x auf dem Stack.
- $\#_{\mathsf{heap}}(x)$ merkt sich den Namen einer Heapvariablen x, wenn diese aus dem Heap $\mathcal{H}$ zur Auswertung geholt wurde. Ist die Auswertung abgeschlossen, kann das Ergebnis, unter dem auf dem Stack gesicherten Namen x zurück in den Heap geschrieben werden.

Der Anfangszustand der Maschine besteht aus einem Tupel $(\emptyset, e, [])$ mit leerem Heap, leerem Stack und dem auszuwertenden Ausdruck $e \in ExprM_{CHFE}$ als Control. Die Auswertung auf der Maschine endet, wenn ein Endzustand der Form $(\mathcal{H}, v, [])$ erreicht ist. Dabei ist v ein Wert. Für die Mark 1 bestehen Werte entweder aus einer Abstraktion, Konstruktorapplikation, einem monadischen Ausdruck oder einer (mit `TID` gekennzeichneten) FutureID. Eine WHNF ist entweder ein Wert, oder ein Ausdruck der Form

(pushApp) $(\mathcal{H}, (e\ x), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{app}}(x) : \mathcal{S})$

(pushSeq) $(\mathcal{H}, (\texttt{seq}\ e\ x), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{seq}}(x) : \mathcal{S})$

(pushAlts) $(\mathcal{H}, (\texttt{case}_T\ e\ \texttt{of}\ alts), \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e, \#_{\mathsf{case}}(alts) : \mathcal{S})$

(takeApp) $(\mathcal{H}, \lambda x.e, \#_{\mathsf{app}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e[y/x], \mathcal{S})$

(takeSeq) $(\mathcal{H}, v, \#_{\mathsf{seq}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, y, \mathcal{S})$,
sofern v eine Abstraktion oder ein Konstrukor ist.

(branch) $(\mathcal{H}, (c\ x_1 \ldots x_n), \#_{\mathsf{case}}(\ldots (c\ y_1 \ldots y_n \to e) \ldots) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H}, e[x_i/y_i]_{i=1}^n, \mathcal{S})$

(nobranch) $(\mathcal{H}, (c\ x_1 \ldots x_n), \#_{\mathsf{case}}(alts) : \mathcal{S}) \xrightarrow{M1}$ **Laufzeitfehler**,
falls $alts$ keine Alternative für Konstruktor c enthält

(enter) $(\mathcal{H} \cup \{y \mapsto e\}, y, \mathcal{S}) \xrightarrow{M1} (\mathcal{H} \cup \{y \mapsto e\}, e, \#_{\mathsf{heap}}(y) : \mathcal{S})$

(update) $(\mathcal{H} \cup \{y \mapsto v\}, v', \#_{\mathsf{heap}}(y) : \mathcal{S}) \xrightarrow{M1} (\mathcal{H} \cup \{y \mapsto v'\}, v', \mathcal{S})$,
sofern v eine Abstraktion, ein Konstruktor, ein monadischer Operator, eine mit `TID` gekennzeichnete FutureID oder eine Variable mit $v \neq y$ ist.

(mkBinds) $(\mathcal{H}, \texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e, \mathcal{S}) \xrightarrow{M1} (\mathcal{H} \cup \bigcup_{i=1}^n \{x_i \mapsto e_i\}, e, \mathcal{S})$

Abbildung 4.2: Zustandsübergangsregeln der Mark 1. Hierbei besteht jedes Tupel aus drei Elementen: $\mathcal{H}$ bezeichnet den Heap, gefolgt von einem Ausdruck und $\mathcal{S}$ beschreibt den Stack

`letrec` $x_1 = e_1, \ldots, x_n = e_n$ `in` v, wobei v ein *M1*-Wert ist. Ein äußeres `letrec` wird dabei durch den Heap dargestellt.

Die Auswertung eines Ausdrucks findet in Form von Zustandsübergängen $\xrightarrow{M1}$ statt. In Abbildung 4.2 sind die Zustandsübergangsregeln aus [Sab12, Sab11] für die Mark 1 dargestellt.

Die Regeln (pushApp), (pushSeq), (pushAlts) finden den neuen Redex, indem sie die später auszuwertenden Teile des Ausdrucks, in entsprechend gekennzeichneten Stackelementen, auf den Stack speichern. Der verbleibende Ausdruck wird bis zu einem Wert ausgewertet, und die Auswertung anschließend durch Betrachten des obersten Stackelements fortgesetzt. Für Applikationen und `seq` wird mit (pushApp) und (pushSeq) das Argument als $\#_{\mathsf{app}}(x)$ bzw. $\#_{\mathsf{seq}}(x)$ gesichert und zunächst e ausgewertet. Bei einem `case` werden die Alternativen mit $\#_{\mathsf{case}}(alts)$ auf den Stack gespeichert, und die Auswertung konzentriert sich auf den zu unterscheidenden Ausdruck.

Nachdem eine Auswertung mit Erreichen eines Wertes beendet ist, wird mit den Regeln (takeApp), (takeSeq), (branch) und (nobranch) das jeweils passende Element vom Stack geholt. Die gekennzeichneten Stackelemente ($\#_{\mathsf{app}}, \#_{\mathsf{case}}, \ldots$) signalisieren dabei, um was für eine Auswertung es sich ursprünglich gehandelt hat, und damit auch, wie fortzufahren ist. Bei einer Applikation endete die Auswertung mit einer Abstraktion, und das Stackelement kann mit (takeApp) vom Stack geholt, und mittels β–Reduktion für den formalen Parameter eingesetzt werden. Bei sequentieller Auswertung mit `seq` wird die Auswertung, nach Erreichen eines Wertes im ersten Argument, einfach mit dem zweiten, mittels (takeSeq) vom Stack geholten, Argument fortgesetzt. Der zu einem Konstruktor ausgewertete Vergleichswert des `case`-Ausdrucks wird mit den, auf dem Stack gespeicherten, Alternativen verglichen. Passt keine der vorhandenen Alternativen, wird die Regel (nobranch) einen Laufzeitfehler anzeigen, und die Maschine anhalten. Für eine passende Alternative wird die Regel (branch) den zugehörigen Ausdruck $e[x_i/y_i]_{i=1}^n$ als neuen Control einsetzen, und alle Variablen y_i des Ausdrucks e durch die Variablen x_i des zu vergleichenden

Konstruktors ($c\ x_1 \ldots x_n$) ersetzen.

Zur Verarbeitung eines `letrec`-Ausdrucks legt die Regel (mkBinds) neue Bindungen im Heap an. Dabei werden diese sowohl im Heap, als auch im `in`–Ausdruck mit frischen Variablennamen umbenannt.

Im Heap angelegte Bindungen können mit der Regel (enter) vom Heap genommen, und als Control ausgewertet werden. Der Heap-Name des gebundenen Ausdrucks wird auf dem Stack in einem $\#_{\mathsf{heap}}$–Element gespeichert. Ist die Auswertung fertiggestellt, kann der Ergebnisausdruck mit der Regel (enter) unter dem gleichen Namen zurück in den Heap geschrieben werden. Bei diesen beiden Regeln handelt es sich um die einzigen *M1*-Regeln, die zur Implementierung von *CHFE* nicht unangepasst aus [Sab12] übernommen werden können. Sie wurden in zweierlei Hinsicht geändert:

1. Die (update)-Regel wurde in die Lage versetzt, auch das neue interne Konstrukt `TID` zur Identifikation einer FutureID als Wert anzuerkennen und zurück in den Heap zu schreiben.

2. Bei den ursprünglich in [Sab12] definierten Regeln wurde eine Heapbindung bei (enter) aus dem Heap entfernt und erst bei einem späteren (update) wieder eingefügt. In einer nebenläufigen Umgebung mit mehreren Threads wurde damit ein redundantes Auswerten des gleichen funktionalen Ausdrucks verhindert.

   Bei den hier definierten Regeln verbleiben die Bindungen im Heap und werden bei erfolgreicher Auswertung dort mit dem neuen Wert überschrieben. In der später definierten nebenläufigen Umgebung kann es dadurch zu ungewollten Mehrfachauswertungen des gleichen Ausdrucks in unterschiedlichen Threads kommen. Dies wird in dieser Arbeit allerdings in Kauf genommen, da es sich zum einen um rein funktionale Berechnungen handelt, die jeweils mit dem gleichen Ergebnis (vgl. Kapitel 2.1.3) enden müssen, es zum anderen aber die Behandlung asynchroner Exceptions (in Abschnitt 4.4.4.6) deutlich vereinfachen wird.

Eine Konsequenz daraus ist, dass die ursprünglich in [Sab11] definierte Regel

$$(\mathit{blackhole}) \qquad (\mathcal{H}, y, \mathcal{S}) \rightarrow (\mathcal{H}, y, \mathcal{S}), \text{falls keine Bindung für } y \text{ im Heap}$$

nicht mehr anwendbar wird. Der konstruierte Fall, dass z.B. bei einem sich selbst referenzierenden Ausdruck `letrec` $x = x$ `in` x keine Heapbindung existiert, tritt durch die vorgenommenen Anpassungen nun nicht mehr ein. Stattdessen gerät der Ausdruck in eine Endlosschleife bei stetig wachsendem Stack (vgl. Abb. 4.3). Da in beiden Fällen (ohne Veränderung des Wertes im Heap) jeweils kein Halten der Maschinen zu erwarten ist, wird diese Tatsache an dieser Stelle vernachlässigt und die (blackhole)–Regel weggelassen.

4.4.3 Abstrakte Maschine I/O–Mark 1

Die erste Erweiterung der vorgestellten Mark 1 bereichert sie in [Sab12] um monadische Operationen, d.h. monadisches I/O in Form von `MVars`. Diese Erweiterungen werden im Folgenden erläutert und um die Fähigkeit synchroner Exceptions und deren Behandlung ergänzt.

Ein Zustand der Maschine I/O–Mark 1 (kurz: *IOM1*) für die Sprache *CHFE* wird durch ein Tupel der Form $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I}, \binom{s}{b})$ beschrieben. Zu den bereits von der Mark 1 bekannten Strukturen Heap $\mathcal{H}$, Control e und Stack $\mathcal{S}$ wurden ein I/O–Stack $\mathcal{I}$, eine Menge $\mathcal{M}$ von `MVars` sowie ein *Stuck-Flag* s und ein *Block-Flag* b hinzugefügt.

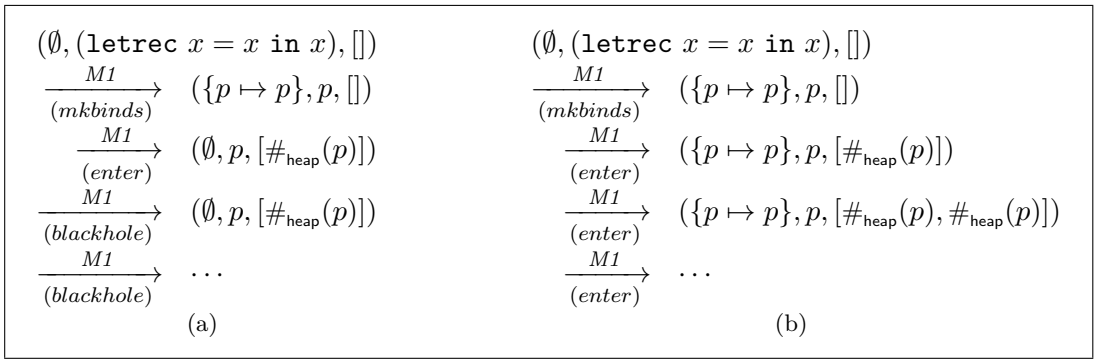

Abbildung 4.3: *M1*–Beispielauswertung eines sich selbst referenzierenden Ausdrucks. (a) Ursprüngliche Auswertung mit Endlosschleife durch die Regel (blackhole). (b) Neue Auswertung ohne (blackhole)-Regel führt zu Endlosschleife mit stetigem Stackwachstum.

Eine Menge $\mathcal{M}$ von `MVars` modelliert die `MVar`–Speicherzellen und stellt dazu Abbildungen von Variablen auf Ausdrücke dar. Eine `MVar` x mit Inhalt y wird als $\{x \,\mathbf{m}\, y\}$ notiert. Da `MVar`s auch leer sein können, wird eine leere `MVar` x als $\{x \,\mathbf{m}\, -\}$ dargestellt.

Das *Block-Flag* wird durch Ausführen einer monadischen `block`–Aktion gesetzt $b = \bullet$ und bei Verlassen des umschlossenen Bereichs oder durch Ausführen einer monadischen `unblock`–Aktion wieder gelöscht $b = \circ$.

Der Zustand des Flags wird in den Notationen der Übersicht halber nur dann explizit angegeben, wenn er relevant ist, oder verändert wird.

Das *Stuck-Flag* wurde in Kapitel 3.3.2 eingeführt, und wird von monadischen Operationen genau dann gesetzt $s = \bullet$, wenn diese scheitern. Im Fall von *CHFE* sind dies die Versuche, eine volle `MVar` zu beschreiben oder aus einer leeren `MVar` zu lesen. Sobald die betreffende Operation allerdings erfolgreich durchgeführt werden konnte, wird es wieder gelöscht $s = \circ$. Dieses Flag signalisiert in der späteren nebenläufigen Maschine, dass der Thread unabhängig vom *Block-Flag* unterbrochen werden darf.

Auch der Zustand dieses Flags wird in den Notationen der Übersicht halber nur dann explizit angegeben, wenn er relevant ist, oder verändert wird.

Der I/O–Stack $\mathcal{I}$ merkt sich, ähnlich wie der funktionale Stack $\mathcal{S}$ die jeweiligen „Rücksprungadressen“ des auszuwertenden Ausdrucks. Dabei ist er allerdings auf rein monadische Aktionen beschränkt, und kann dementsprechend nur $\#_{\gg=}$, $\#_{\mathsf{put}}$, $\#_{\mathsf{take}}$, $\#_{\mathsf{block}}$, $\#_{\mathsf{unblock}}$ und $\#_{\mathsf{catch}}$ als Stackelemente für die entsprechenden monadischen Aktionen beinhalten.

Hierbei merkt sich $\#_{\gg=}(y)$ den nach einem $x \gg= y$ folgenden Ausdruck y. Das zweite Argument von `putMVar` x y, und damit der in eine `MVar` zu schreibende Wert y, wird auf dem I/O–Stack in einem $\#_{\mathsf{put}}(y)$ gesichert. Dass die Aktion `takeMVar` x, nach Auswertung von x zu einem `MVar`–Bezeichner, noch auszuführen ist, wird sich mit $\#_{\mathsf{take}}$ auf dem I/O–Stack gemerkt. Für *CHFE* neu hinzugekommen sind die Stackelemente $\#_{\mathsf{block}}$ und $\#_{\mathsf{unblock}}$, die jeweils das Ende eines mit `block` bzw. `unblock` umschlossenen Ausdrucks, und damit das Umschalten des *Block-Flags* signalisieren. Mit $\#_{\mathsf{catch}}(y)$ wird der Exception-Handler eines `catch`–Ausdrucks auf den Stack geschoben, und verbleibt dort, bis der umschlossene Ausdruck ausgewertet oder innerhalb des Ausdrucks eine Exception aufgetreten ist.

Für einen Ausdruck e wird ein Startzustand der I/O–Mark 1 durch ein Tupel der Form $(\emptyset, \emptyset, e, [], [], (\begin{smallmatrix}s=\circ\\ b=\circ\end{smallmatrix}))$ definiert. Ein Endzustand der Maschine ist erreicht, wenn er von der Form $(\mathcal{H}, \mathcal{M}, \texttt{return}\ x, [], [], (\begin{smallmatrix}s\\ b\end{smallmatrix}))$, oder von der Form $(\mathcal{H}, \mathcal{M}, v, [], [], (\begin{smallmatrix}s\\ b\end{smallmatrix}))$ ist, wobei v eine Abstraktion, Konstruktoranwendung der Name einer `MVar` oder eine mit `TID` gekennzeichnete FutureID ist.

In Abbildung 4.4 werden die aus [Sab12] übernommenen Zustandsübergangsregeln $\xrightarrow{IOM1}$ dargestellt. Die Notation der Tupel ist dabei bereits um die beiden Flags erweitert worden.

Handelt es sich bei dem aktuell betrachteten Ausdruck Control um rein funktionale Auswertungen, so werden die Zustandsübergangsregeln der Mark 1 angewandt. Sie werden an dieser Stelle nicht wiederholt, sondern durch die Regel (liftM1) modelliert, die e (ungeachtet des I/O–Stacks $\mathcal{I}$ und der `MVar`-Menge $\mathcal{M}$) bis zu einem Wert auswertet. Entsprechend der Definition eines *M1*–Endzustandes lässt sich das Erreichen eines *M1*–Wertes an einem leeren Stack $\mathcal{S}$ erkennen. Handelt es sich bei dem *M1*–Wert um eine monadische Aktion, wird diese mit den verbleibenden Regeln der I/O–Mark 1 durchgeführt. Für alle monadischen Aktionen ist gleichsam charakteristisch, dass der funktionale Stack $\mathcal{S}$ leer ist und daher jeweils mit [] notiert wurde. Lediglich die *M1*-(update)–Regel wird dahingehend ergänzt, dass sie einen Wert v auch dann in den Heap zurückschreibt, wenn es sich dabei um den Namen einer in $\mathcal{M}$ vorkommenden `MVar` handelt.

Die Regel (pushBind) dient dazu, bei einem monadischen Ausdruck der Form $x \gg\!= y$ zunächst das Folgeargument y auf dem I/O–Stack zu sichern und die Aktion x auszuführen. Endet die Aktion mit einem `return` x' wird Nachfolgeaktion mit der Regel (lunit) wieder vom I/O–Stack geholt und auf das Ergebnis x' angewendet.

Die Regel (newMVar) fügt der Menge $\mathcal{M}$ eine neue `MVar` mit Namen y hinzu, die direkt mit dem übergebenen Ausdruck x befüllt wurde. Ihr monadischer Rückgabewert ist der neue Name der erzeugten `MVar`.

4.4.3.1 Anpassung der Regeln für MVars

Die in Abb. 4.4 angegebenen Regeln (pushTake) und (pushPut) können unverändert aus [Sab12] übernommen werden.

Trifft die Auswertung auf die monadischen Aktionen `takeMVar` x oder `putMVar` $x\ y$, kann es sein, dass x erst noch soweit ausgewertet werden muss, bis damit eine `MVar` bezeichnet wird. Hierfür wird jeweils ein entsprechendes Element (im Fall von `putMVar` auch die zu schreibende Variable y) auf den I/O–Stack gespeichert. Dies spiegelt gerade die in Abschnitt 2.5.2 vorgestellten Forcing–Kontexte wider.

Liegt nach Auswertung von x der Bezeichner einer `MVar` vor, kann die Aktion ausgeführt werden. Dies wird durch die Regeln (takeMVar) und (putMVar) dargestellt, die jeweils das Stackelement entfernen, die Aktion durchführen und entweder den gewünschten Wert der `MVar` oder `()` zurückgeben. Da mit diesen Regeln die Aktionen erfolgreich abgeschlossen werden können, werden sie dahingehend erweitert, dass sie das *Stuck-Flag* löschen. Dabei ist es unerheblich, in welchem Zustand es sich vorher befunden hat. Damit wird die – nun vorerst tatsächlich stattfindende – weitere Ausführung der Auswertung angezeigt.

$$\begin{array}{ll}
\text{(fullPut)} & (\mathcal{H}, \mathcal{M} \dot\cup \{x\ \mathbf{m}\ z\}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I}, (\begin{smallmatrix}s=\circ\\ b\end{smallmatrix})) \\
& \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \dot\cup \{x\ \mathbf{m}\ z\}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I}, (\begin{smallmatrix}s=\bullet\\ b\end{smallmatrix})) \\
\\
\text{(emptyTake)} & (\mathcal{H}, \mathcal{M} \dot\cup \{x\ \mathbf{m}\ -\}, x, [], \#_{\mathsf{take}}(y) : \mathcal{I}, (\begin{smallmatrix}s=\circ\\ b\end{smallmatrix})) \\
& \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \dot\cup \{x\ \mathbf{m}\ -\}, x, [], \#_{\mathsf{take}}(y) : \mathcal{I}, (\begin{smallmatrix}s=\bullet\\ b\end{smallmatrix}))
\end{array}$$

Neu hinzugekommen sind die Regeln (fullPut) und (emptyTake). Sie finden genau dann Anwendung, wenn eine `MVar`-Operation nicht erfolgreich durchgeführt werden kann. Im

(liftM1) $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I}) \xrightarrow{IOM1} (\mathcal{H}', \mathcal{M}', e', \mathcal{S}', \mathcal{I}', \binom{s}{b})$
falls $(\mathcal{H}, e, \mathcal{S}) \xrightarrow{M1} (\mathcal{H}', e', \mathcal{S}', \binom{s}{b})$ auf Maschine *M1*

(newMVar) $(\mathcal{H}, \mathcal{M}, \texttt{newMVar}\ x, [], \mathcal{I}, \binom{s}{b}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{y\ \mathbf{m}\ x\}, \texttt{return}\ y, [], \mathcal{I}, \binom{s}{b})$
wobei y eine neue Variable ist.

(takeMVar) $(\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ y\}, x, [], \#_{\mathsf{take}} : \binom{s}{b}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ -\}, \texttt{return}\ y, [], \mathcal{I}, \binom{s=\circ}{b})$

(putMVar) $(\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ -\}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I}, \binom{s}{b})$
$\xrightarrow{IOM1} (\mathcal{H}, \mathcal{M} \cup \{x\ \mathbf{m}\ y\}, \texttt{return}\ (), [], \mathcal{I}, \binom{s=\circ}{b})$

(pushTake) $(\mathcal{H}, \mathcal{M}, \texttt{takeMVar}\ x, [], \mathcal{I}, \binom{s}{b}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{take}} : \mathcal{I}, \binom{s}{b})$

(pushPut) $(\mathcal{H}, \mathcal{M}, \texttt{putMVar}\ x\ y, [], \mathcal{I}, \binom{s}{b}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{put}}(y) : \mathcal{I}, \binom{s}{b})$

(pushBind) $(\mathcal{H}, \mathcal{M}, x\ \gg\!=\ y, [], \mathcal{I}, \binom{s}{b}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\gg\!=}(y) : \mathcal{I}, \binom{s}{b})$

(lunit) $(\mathcal{H}, \mathcal{M}, \texttt{return}\ x, [], \#_{\gg\!=}(y) : \mathcal{I}, \binom{s}{b}) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (y\ x), [], \mathcal{I}, \binom{s}{b})$

Abbildung 4.4: Zustandsübergangsregeln der I/O–Mark 1-Maschine. Jedes Tupel besteht dabei aus den Elementen Heap $\mathcal{H}$, `MVar`-Menge $\mathcal{M}$, Control, I/O–Stack $\mathcal{I}$ und den neu eingeführten Flags s und b.

Wesentlichen findet keine Veränderung des Zustandes der Maschine statt. Beide markieren die Auswertung als *stuck* durch Setzen des *Stuck-Flags* und wiederholen den Vorgängerzustand. Eine asynchrone Exception dürfte nun unabhängig vom *Block-Flag* zugelassen werden. Kann im Folgenden auf die gewünschte Ressource ungehindert zugegriffen werden, bevor eine asynchrone Exception eingetreten ist, greifen die Regeln (takeMVar) oder (putMVar), die das *Stuck-Flag* ihrerseits wieder löschen.

4.4.3.2 Block– und Unblock–Zustand kontrollieren

Der Zustand des *Block-Flag* wird über die Aktionen `block` x und `unblock` x gesteuert. Ist das *Block-Flag* gesetzt $b = \bullet$, befindet sich die Auswertung in einem *blocked*–Zustand, und die Zustellung asynchroner Exceptions, d.h. solcher Exceptions, die von einem anderen Thread gesendet wurden, ist unterbunden. Exceptions, die während eines gesetzten *Block-Flags* eintreffen, müssen auf ihre Zustellung solange warten, bis entweder das *Block-Flag* wieder gelöscht $b = \circ$ wird oder das *Stuck-Flag* gesetzt wird.

(blockPush) $(\mathcal{H}, \mathcal{M}, (\texttt{block}\ x), [], \mathcal{I}, \binom{s}{b=\circ})$
$\xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{unblock}} : \mathcal{I}, \binom{s}{b=\bullet})$

(blockPop) $(\mathcal{H}, \mathcal{M}, (\texttt{return}\ x), [], \#_{\mathsf{unblock}} : \mathcal{I}, \binom{s}{b=\bullet})$
$\xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (\texttt{return}\ x), [], \mathcal{I}, \binom{s}{b=\circ})$

(blockBlock) $(\mathcal{H}, \mathcal{M}, (\texttt{block}\ x), [], \mathcal{I}, \binom{s}{b=\bullet})$
$\xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \mathcal{I}, \binom{s}{b=\bullet})$

(blockUnblock) $(\mathcal{H}, \mathcal{M}, (\texttt{block}\ x), [], \#_{\mathsf{block}} : \mathcal{I}, \binom{s}{b=\circ})$
$\xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \mathcal{I}, \binom{s}{b=\bullet})$

Die Regel (blockPush) verändert den aktuellen Status des Threads auf *blocked*, indem das *Block-Flag* gesetzt wird. Zusätzlich wird $\#_{\mathsf{unblock}}$ auf den I/O–Stack gelegt und signalisiert

das Ende des block-Bereichs. Ist die Auswertung des in einem block–Kontext befindlichen Ausdrucks zu einem Wert abgeschlossen, entfernt die Regel (blockPop) das Stackelement und löscht das *Block-Flag*, womit der block–Kontext wieder verlassen wurde.

Auf diese Weise ist es möglich, `block` und `unblock` beliebig ineinander zu verschachteln und das *Block-Flag* zu kontrollieren.

Die Regel (blockBlock) dient dazu, ein unnötiges I/O–Stackwachstum zu vermeiden, falls bei bereits gesetztem *Block-Flag* durch ein im Ausdruck vorkommendes `block` das Flag erneut gesetzt werden soll. Dem Fall `unblock {block {...}}` mit direkt aufeinander folgenden, sich gegenseitig aufhebenden Aktionen, wird mit der Regel (blockUnblock) begegnet. Auch hier wird unnötiges I/O–Stackwachstum vermieden.

Die Regeln (unblockPush), (unblockPop), (unblockBlock) und (unblockUnblock) zum expliziten Löschen des *Block-Flags* mit `unblock` sind vice versa definiert:

$$\begin{array}{ll}
\text{(unblockPush)} & (\mathcal{H}, \mathcal{M}, (\texttt{unblock}\ x), [], \mathcal{I}, (^{s}_{b=\bullet})) \\
& \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \#_{\mathsf{block}} : \mathcal{I}, (^{s}_{b=\circ})) \\
\text{(unblockPop)} & (\mathcal{H}, \mathcal{M}, (\texttt{return}\ x), [], \#_{\mathsf{block}} : \mathcal{I}, (^{s}_{b=\circ})) \\
& \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (\texttt{return}\ x), [], \mathcal{I}, (^{s}_{b=\bullet})) \\
\text{(unblockBlock)} & (\mathcal{H}, \mathcal{M}, (\texttt{unblock}\ x), [], \mathcal{I}, (^{s}_{b=\circ})) \\
& \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \mathcal{I}, (^{s}_{b=\circ})) \\
\text{(unblockUnblock)} & (\mathcal{H}, \mathcal{M}, (\texttt{unblock}\ x), [], \#_{\mathsf{unblock}} : \mathcal{I}, (^{s}_{b=\bullet})) \\
& \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x, [], \mathcal{I}, (^{s}_{b=\circ}))
\end{array}$$

4.4.3.3 Exceptions mit (propThrow) weiterleiten

Tritt im Ausdruck ein `throw` x durch eine synchrone (oder asynchrone) Exception auf, muss die Exception x bis zum nächsten `catch` weitergeleitet werden. Dazu wird mit der Regel (propThrow) der I/O–Stack sukzessive geleert und die Stackelemente $\#_{\gg=}$, $\#_{\mathsf{take}}$, $\#_{\mathsf{put}}$, $\#_{\mathsf{block}}$ und $\#_{\mathsf{unblock}}$ ohne ausgeführte Aktionen „übergangen“.

$$\begin{array}{ll}
\text{(propThrow)} & (\mathcal{H}, \mathcal{M}, (\texttt{throw}\ x), [], top : \mathcal{I}, (^{s}_{b})) \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (\texttt{throw}\ x), [], \mathcal{I}, (^{s}_{b})) \\
& \text{mit } top \in \{\#_{\gg=}, \#_{\mathsf{take}}, \#_{\mathsf{put}}, \#_{\mathsf{block}}, \#_{\mathsf{unblock}}\}
\end{array}$$

Beim Weiterleiten über $\#_{\mathsf{put}}$ und $\#_{\mathsf{take}}$ werden keine `MVar`s verändert. Es liegt in der Verantwortung des Programmierers, deren Konsistenz im Handler einer umschließenden `catch`–Anweisung zu gewährleisten. Auch über die Stackelemente $\#_{\mathsf{block}}$ und $\#_{\mathsf{unblock}}$ kann eine Exception problemlos weitergeleitet werden. Der Zustand des *Block-Flags* wird von einem umschließenden `catch` gespeichert und im Fall einer Exception automatisch wiederhergestellt.[9]

4.4.3.4 Abfangen von Exceptions mit `catch`

Das letzte neue Konstrukt von *CHFE* ist der umschließende `catch`–Ausdruck mit seinem Handler. Trifft die Auswertung im Ausdruck auf `catch` x_1 x_2, sichert sie mit der Regel (pushCatch) den Handler x_2 auf den Stack. Das zugehörige Stackelement $\#_{\mathsf{catch}}(x_2, \diamond)$ speichert in $\diamond \in \{\bullet, \circ\}$ dabei zusätzlich noch den aktuellen Zustand des *Block-Flags* b, notiert

[9] Das *Stuck-Flag* ist nur bei asynchronen Exceptions auf einer nebenläufigen Maschine von Bedeutung. Asynchrone Exceptions werden ausschließlich über eine eigene Regel (Interrupt) eingeleitet, die ihrerseits ein möglicherweise gesetztes *Stuck-Flag* wieder löscht.

als $\diamond = b$. Damit wird bei Ausführen des Handlers später sichergestellt, dass der Zustand dieses Flags wiederhergestellt werden kann.

Abhängig davon, ob die nachfolgende Auswertung des Ausdrucks x_1 in einem `return` x oder einem `throw` x endet, werden zum Auflösen des I/O–Stackelements entweder die Regel (catchPop) oder (handlePop) verwendet.

$$\begin{aligned}
&\text{(catchPush)} \quad (\mathcal{H}, \mathcal{M}, (\texttt{catch}\ x_1\ x_2), [], \mathcal{I}, \binom{s}{b}) \\
&\qquad \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, x_1, [], \#_{\mathsf{catch}}(x_2; \diamond = b) : \mathcal{I}, \binom{s}{b}) \\
&\text{(catchPop)} \quad (\mathcal{H}, \mathcal{M}, (\texttt{return}\ x_1), [], \#_{\mathsf{catch}}(x_2; \diamond) : \mathcal{I}, \binom{s}{b}) \\
&\qquad \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (\texttt{return}\ x_1), [], \mathcal{I}, \binom{s}{b}) \\
&\text{(handlePop)} \quad (\mathcal{H}, \mathcal{M}, (\texttt{throw}\ x_1), [], \#_{\mathsf{catch}}(x_2; \diamond) : \mathcal{I}, \binom{s}{b}) \\
&\qquad \xrightarrow{IOM1} (\mathcal{H}, \mathcal{M}, (x_2\ x_1), [], \mathcal{I}, \binom{s}{b=\diamond})
\end{aligned}$$

Lässt sich der Ausdruck ohne Auftreten einer Exception bis zu einem `return` x auswerten, kommt dem umschließenden `catch` keinerlei Bedeutung mehr zu. In der Regel (catchPop) wird daher das Stackelement wieder vom I/O–Stack genommen und die Auswertung mit der nächsten monadischen Aktion fortgesetzt.

Bei Auftreten einer Exception wurde dieses `throw` x mit (Propagate) so lange weitergeleitet, bis sich ein $\#_{\mathsf{catch}}(x_2, \diamond)$ als oberstes Element auf dem I/O–Stack befindet. Darin befindet sich gerade der aufzurufende Exceptionhandler, sowie der wiederherzustellende Zustand $\diamond$ des *Block-Flags*, den dieses innehatte, bevor `catch` ausgewertet wurde. Die Regel (handlePop) stellt nun den Zustand des *Block-Flags* wieder her und wendet den Handler auf die aufgetretene Exception an.

4.4.4 Abstrakte Maschine Concurrent–Mark 1

Die bisher vorgestellte Maschine *IOM1* ist in der Lage, einen monadischen und funktionalen Ausdruck auszuwerten. Es liegt daher nahe, bei der Entwicklung der nebenläufigen Maschine Concurrent–Mark 1 (kurz: *CIOM1*) jeweils für die Auswertung eines Threads[10] darauf zurückzugreifen. Es fehlen für die Implementierung der *CIOM1* daher nur die Strukturen zur Verwaltung und Steuerung dieser Threads.

4.4.4.1 Modellierung der Nebenläufigkeit

Bei nebenläufiger Auswertung mehrerer Threads erscheint es einem Benutzer so, als würden diese *gleichzeitig* ausgewertet. Im Unterschied zu einer sequentiellen Auswertung entstehen also die jeweiligen Ergebnisse einzelner Threads ineinander verschachtelt (engl. *interleaved*). In wieweit die Auswertung der Threads tatsächlich parallel (d.h. auf mehreren physischen Prozessoren) und damit gleichzeitig stattfindet, oder durch verschachtelte Auswertung der Threads auf nur einem Prozessor geschieht, ist eine Frage der Implementierung (vgl. [Mar12]).

Für die konkrete Implementierung der Nebenläufigkeit existieren also unterschiedliche Möglichkeiten. Die hier vorgestellte Abstrakte Maschine Concurrent–Mark 1 verwendet der Einfachheit halber das, auch in [Sab11] verwendete Modell der „Zeitscheiben“ in einem *Round-Robin*-Verfahren.

Die unterschiedlichen Threads werden dabei in einer FIFO-Warteschlange (F*irst*-I*n*-F*irst*-O*ut*) $\mathcal{Q}$ verwaltet. Jedem Thread wird zusätzlich eine natürliche Zahl als Ressource

[10] Da die *CIOM1* nur noch Futures als Threads kennt, werden die Begriffe in diesem Kapitel synonym verwendet.

zugeteilt. Diese Zahl definiert die Anzahl der Zustandsübergänge, die der Thread hintereinander durchführen darf. Für jeden Zustandsübergang, den ein Thread vollzieht, wird die Ressourcenzahl um eins vermindert. Hat ein Thread seine Ressourcen verbraucht, wird er vom Kopf der Warteschlange $\mathcal{Q}$ entfernt, und mit einer neuen Ressourcenzahl am Ende wieder eingefügt. Die Auswertung wird mit dem nächsten Thread der Warteschlange fortgesetzt. Die neue Ressourcenzahl muss dabei von Null verschieden sein und kann zufällig gewählt werden. Um anzudeuten, dass ein Element x an das Ende der Warteschlange $\mathcal{Q}$ eingefügt wird, wird die Notation $\mathcal{Q}$++x verwendet.

Dieses Verfahren bietet eine gewisse Form von Fairness, da gewährleistet wird, dass jeder Thread nach endlich vielen Schritten rechnen darf.

4.4.4.2 Definition der Zustände

Ein Zustand der Concurrent–Mark 1 wird durch ein Tupel der Form $(\mathcal{H}, \mathcal{M}, \mathcal{T}, \mathcal{Q})$, mit einem Heap $\mathcal{H}$, der Menge der `MVars` $\mathcal{M}$, der Menge der Threads $\mathcal{T}$ und der Thread-Warteschlange $\mathcal{Q}$ beschrieben. Der Heap $\mathcal{H}$ und die Menge der `MVars` $\mathcal{M}$ ist dabei global erreichbar für alle Threads.

Eine Future wird eindeutig durch einen Bezeichner x identifiziert, mit der sie sowohl im *CHFE*–Programm referenziert als auch in der Menge der Threads $\mathcal{T}$ identifiziert werden kann. Jede Future (d.h. jeder Thread) $x \in \mathcal{T}$ wird durch ein Tupel in der Form $(x, e_x, \mathcal{S}_x, \mathcal{I}_x, \binom{s}{b}, \mathcal{E}_x)$ beschrieben. Das erste Element stellt dabei den eindeutigen Bezeichner dar, gefolgt vom auszuwertenden Control-Ausdruck e_x der Future. Die beiden nächsten Elemente stellen den Threadeigenen Stack $\mathcal{S}_x$ und I/O–Stack $\mathcal{I}_x$ dar. Ob sich der Thread in einem stuck– oder block–Zustand befindet, wird durch die Flags s und b signalisiert. Mit dem letzten Element wird eine weitere FIFO-Warteschlange $\mathcal{E}_x$ beschrieben, in der asynchrone Exceptions (*ExceptionInFlight*) vorgehalten werden, bis sie dem Thread zugestellt werden können (wozu entweder das *Block-Flag* $b = \circ$ oder das *Stuck-Flag* $s = \bullet$ sein muss).[11]

Für die *CIOM1* wird genau ein Thread als Main-Thread ausgewiesen und mit `_main_` bezeichnet. Dabei handelt es sich um den initialen Thread, von dem aus erst weitere Threads erzeugt werden können. Terminiert dieser Thread, so terminiert das *CHFE*-Programm. Per Definition kann dieser Thread keine asynchronen Exceptions empfangen. Terminiert hingegen eine Future x, indem sie bis zu einem *IOM1*–Wert y ausgewertet wurde, so wird sie dem globalen Heap $\mathcal{H}$ als normalen Heap-Bindung der Form $\{x \mapsto y\}$ hinzugefügt und aus der Menge $\mathcal{T}$ der Threads entfernt.

Ein Startzustand der *CIOM1* für einen Ausdruck e ist demnach von der Form

$$\begin{aligned}&(\emptyset, \emptyset, \mathcal{T}, \mathcal{Q}) \text{ mit}\\ &\quad \mathcal{T} = \{(\texttt{\_main\_}, e, [], [], \binom{s=\circ}{b=\circ}, [])\},\\ &\quad \mathcal{Q} = [(1, \texttt{\_main\_})]\end{aligned}$$

womit angedeutet wird, dass sowohl $\mathcal{H}$ als auch $\mathcal{M}$ der *CIOM1* leer sind. Die Menge der Threads enthält den ausgezeichneten Hauptthread `_main_`, mit dem Ausdruck e als Control. Sowohl der funktionale Stack $\mathcal{S}$ als auch der I/O–Stack $\mathcal{I}$ sind leer, und er befindet sich weder in einem stuck– noch in einem block–Zustand. Auch die Exception-Queue $\mathcal{E}$ ist leer. Die alleinige Existenz des Hauptthreads bedingt auch nur einen einzigen Eintrag in der Thread-Warteschlange $\mathcal{Q}$. Die erste Komponente des dargestellten Tupels bezeich-

[11] Asynchrone Exceptions sind also tatsächlich asynchron in der Hinsicht, dass sie nicht instantan nach ihrem Senden im Zielthread ankommen, sondern ihre Wirkung erst dann eintritt, wenn der Zielthread in die Lage versetzt wurde, sie zu empfangen.

net die Ressourcenzahl, die dem, durch die zweite Komponente bezeichneten Thread zur Verfügung steht.

Ein Endzustand der *CIOM1* ist erreicht, wenn der Hauptthread `_main_` von der Form $(\texttt{\_main\_}, \texttt{return}\ x, [], [], (\begin{smallmatrix}s\\b\end{smallmatrix}), [])$, oder von der Form $(\texttt{\_main\_}, v, [], [], (\begin{smallmatrix}s\\b\end{smallmatrix}), [])$ ist, wobei v eine Abstraktion, Konstruktoranwendung, der Name einer `MVar` oder eine mit `TID` gekennzeichnete FutureID ist.

4.4.4.3 Erzeugen neuer Futures

Eine neue Future wird im aufrufenden Thread durch die Auswertung von `future` x erzeugt.

$$\text{(ForkIO)}\quad (\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, (\texttt{future}\ x), []_u, \mathcal{I}_u, (\begin{smallmatrix}s\\b\end{smallmatrix}), []_u)\}, (n, u) : \mathcal{Q}) \xrightarrow{CIOM1} \left(\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \left\{ \begin{array}{l} (u, (\texttt{return}\ \langle v; \texttt{TID}\ v\rangle), []_u, \mathcal{I}_u, (\begin{smallmatrix}s\\b\end{smallmatrix}), []_u), \\ (v, x, []_v, []_v, (\begin{smallmatrix}s=\circ\\b=\circ\end{smallmatrix}), []_v) \end{array} \right\}, \mathcal{Q}'\right)$$

$$\text{mit } \mathcal{Q}' := \begin{cases} (n-1, u) : \mathcal{Q}\texttt{++}[(m, v)] & \text{falls } n > 1, \\ \mathcal{Q}\texttt{++}[(m, v), (n', u)] & \text{falls } n = 1. \end{cases}$$

Hierbei ist u eine Future, v eine neue Variable und m, n' zufällig gewählte natürliche Zahlen.

Die Regel (ForkIO) erzeugt eine neue Variable v. Der Menge der Threads $\mathcal{T}$ wird diese Future als ein neues Element $(v, x, [], [], (\begin{smallmatrix}s=\circ\\b=\circ\end{smallmatrix}), [])$ hinzugefügt, dass durch v eindeutig identifiziert wird. Es enthält den Ausdruck x als Control und wird mit leeren Stacks $\mathcal{S}, \mathcal{I}$ initialisiert. Per Definition befindet sich eine neu erzeugte Future weder in einem block– noch in einem stuck–Zustand. Auch die Exception-Warteschlange $\mathcal{E}$ kann bisher zwangsläufig nur leer sein.

Damit der aufrufende Thread u auf die neu erzeugte Future zugreifen kann, wird sie ihm als Rückgabe mithilfe des eingeführten *FIDCons* in der Form $\langle v; \texttt{TID}\ v\rangle$ bekannt gemacht. Dessen erste Komponente stellt gerade die Future v dar, die damit an anderen Stellen eines *CHFE*–Programms referenziert werden darf. Die zweite Komponente verwendet der Einfachheit halber den gleichen Wert v, allerdings in Verbindung mit dem nur in $MExprM_{CHFE}$ existierenden Konstruktor `TID`. Dadurch wird eine FutureID für das Senden asynchroner Exceptions dargestellt, da die Auswertung mit Erreichen von `TID` v stoppt und nicht auf den Ergebniswert der Future zugegriffen wird. Die Regel (ForkIO) stellt während der Auswertung somit die einzige Quelle für `TID`–Konstrukte als Teil des *FIDCons* dar.

Auch in die Thread-Warteschlange $\mathcal{Q}$ wird für die neue Future ein neuer Eintrag in Form des Tupels (m, v) eingefügt. Sobald die Future v an die vorderste Stelle der Schlange gerückt ist, darf sie m Auswertungsschritte vollziehen. Da mit der Regel (ForkIO) ein Auswertungsschritt des aktuellen Threads u ausgeführt wurde, wird dessen Ressource ebenfalls angepasst. Sofern diese Ressource noch nicht erschöpft ist, wird sie um eins vermindert, und der Thread verbleibt an vorderster Stelle. Andernfalls wird ein neuer Ressourcenwert zufällig bestimmt, und der Thread wird an das Ende der Warteschlange $\mathcal{Q}$ verschoben.

4.4.4.4 Beenden ausgewerteter Futures

Erreicht eine Future einen *IOM1*–Endzustand, gilt der Thread als ausgewertet. Der Thread wird entfernt und das Ergebnis als normale Heap-Bindung weiterhin verfügbar gehalten.

$$\begin{aligned}(\text{UnIO}) \quad & (\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, (\texttt{return}\ x), []_u, []_u, (^s_b), \mathcal{E}), (n, u) : \mathcal{Q}\}) \\ & \xrightarrow{CIOM1} (\mathcal{H} \dot\cup \{u \mapsto x\}, \mathcal{M}, \mathcal{T}, \mathcal{Q}) \\ & \text{mit } u \neq \texttt{\_main\_}\end{aligned}$$

Die Regel (UnIO) entfernt den Thread aus der Thread-Menge $\mathcal{T}$, und fügt ihn unter dem Namen der ehemaligen Future als Heapbindung $\{u \mapsto x\}$ in den Heap $\mathcal{H}$ ein. Konsequente Einhaltung der *DVC* gewährleistet, dass es dabei zu keinerlei Namensüberschneidungen kommt. Mit Beenden des Threads u wird auch dessen Eintrag in der Thread-Warteschlange $\mathcal{Q}$ obsolet und kann entfernt werden.

4.4.4.5 Senden asynchroner Exceptions

Das Senden einer Exception von einem Thread an einen Anderen geschieht in zwei Schritten. Zunächst wird das erste Argument einer `throwTo`–Anweisung bis zu einer FutureID ausgewertet. Dazu wird die Liste der Elemente, die auf dem I/O–Stack abgelegt werden können, um das Stackelement $\#_{\mathsf{throwTo}}$ erweitert.

$$\begin{aligned}(\text{PushTT}) \quad & (\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, (\texttt{throwTo}\ x\ y), []_u, \mathcal{I}_u, (^s_b), []_u)\}, (n, u) : \mathcal{Q}) \\ & \xrightarrow{CIOM1} (\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, x, []_u, \#_{\mathsf{throwTo}}(y) : \mathcal{I}_u, (^s_b), []_u)\}, \mathcal{Q}')\end{aligned}$$

$$\begin{aligned}(\text{ThrTo}) \quad & \left(\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \left\{ \begin{array}{l} (u, (\texttt{TID}\ v), []_u, \#_{\mathsf{throwTo}}(y) : \mathcal{I}_u, (^s_b), []_u), \\ (v, e, \mathcal{S}_v, \mathcal{I}_v, (^s_b), \mathcal{E}_v) \end{array} \right\}, (n, u) : \mathcal{Q}\right) \\ & \xrightarrow{CIOM1} \left(\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \left\{ \begin{array}{l} (u, (\texttt{return}\,()), []_u, \mathcal{I}_u, (^s_b), []_u), \\ (v, e, \mathcal{S}_v, \mathcal{I}_v, (^s_b), \mathcal{E}_v \texttt{++} y) \end{array} \right\}, \mathcal{Q}'\right)\end{aligned}$$

$$\begin{aligned}(\text{ThrToX}) \quad & (\mathcal{H} \dot\cup \{v \mapsto e\}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, (\texttt{TID}\ v), []_u, \#_{\mathsf{throwTo}}(y) : \mathcal{I}_u, (^s_b), []_u)\}, (n, u) : \mathcal{Q}) \\ & \xrightarrow{CIOM1} (\mathcal{H} \dot\cup \{v \mapsto e\}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, (\texttt{return}\,()), []_u, \mathcal{I}_u, (^s_b), []_u)\}, \mathcal{Q}')\end{aligned}$$

$$\text{wobei jeweils } \mathcal{Q}' := \left\{ \begin{array}{ll} (n-1, u) : \mathcal{Q} & \text{falls } n > 1 \\ \mathcal{Q}\texttt{++}[(n', u)] & \text{falls } n = 1 \end{array} \right.$$

und n' eine zufällig gewählte natürliche Zahl ist.

Die Regel (PushTT) sichert zunächst die eigentliche Exception im neuen Stackelement $\#_{\mathsf{throwTo}}$ auf den I/O–Stack. Es folgt die Auswertung des ersten Arguments, die mit Erreichen einer FutureID `TID` v endet und den Zielthread identifiziert.

Es sind nun zwei Fälle denkbar. Existiert der Thread v noch in der Menge der Threads $\mathcal{T}$, platziert die Regel (ThrTo) die vom Stack genommene Exception am Ende von dessen Exception-Warteschlange $\mathcal{E}_v$. Falls der Zielthread v zwischenzeitlich fertig ausgewertet wurde, so ist er nicht mehr Teil der Threadmenge $\mathcal{T}$, sondern bereits zu einer Heap-Bindung in $\mathcal{H}$ geworden. Das Zustellen einer Exception ist überflüssig, und die Regel (ThrToX) übergeht diesen Schritt ohne weitere Aktion. In beiden Fällen kann der aufrufende Thread mit einem `return ()` direkt weiterarbeiten.

Bei allen drei Regeln wird außerdem jeweils die vom aktuell ausgewerteten Thread u verbrauchte Ressource in $\mathcal{Q}'$ angepasst.

Das Senden einer Exception an den Hauptthread `_main_` wurde an dieser Stelle zwar nicht formal ausgeschlossen, ist aber praktisch nicht möglich. Es existiert keine Möglich-

keit, an einen adressierenden Bezeichner in der Form (`TID _main_`) zu gelangen, da der Hauptthread nicht als Future durch (ForkIO) erzeugt werden musste.

4.4.4.6 Empfangen asynchroner Exceptions

An einen Thread gesendete Exceptions werden zunächst nur in der Exception-Warteschlange $\mathcal{E}$ vorgehalten. Für das tatsächliche Zustellen der ersten Exception aus $\mathcal{E}$ muss eine von zwei Bedingungen erfüllt sein. Entweder signalisiert der Thread durch ein gelöschtes *Block-Flag*, dass Exceptions ausdrücklich zugelassen werden, oder aber das *Stuck-Flag* ist gesetzt. Letzteres ist genau dann der Fall, wenn eine weitere Auswertung des Threads (temporär) scheitert, wodurch insbesondere auch das *Block-Flag*, falls gesetzt, vorläufig nicht gelöscht werden könnte.

$$\begin{aligned}
&\text{(Receive)} \quad (\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, e, \mathcal{S}_u, \mathcal{I}_u, (^{s}_{b=\circ}), x : \mathcal{E}_u)\}, (n, u) : \mathcal{Q}) \\
&\qquad \xrightarrow{CIOM1} (\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, (\texttt{throw}\ x), [], \mathcal{I}_u, (^{s}_{b=\circ}), \mathcal{E}_u)\}, \mathcal{Q}') \\
&\text{(Interrupt)} \quad (\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, e, \mathcal{S}_u, \mathcal{I}_u, (^{s=\bullet}_{b}), x : \mathcal{E}_u)\}, (n, u) : \mathcal{Q}) \\
&\qquad \xrightarrow{CIOM1} (\mathcal{H}, \mathcal{M}, \mathcal{T} \dot\cup \{(u, (\texttt{throw}\ x), [], \mathcal{I}_u, (^{s=\circ}_{b}), \mathcal{E}_u)\}, \mathcal{Q}')
\end{aligned}$$

$$\text{wobei jeweils } \mathcal{Q}' := \begin{cases} (n-1, u) : \mathcal{Q} & \text{falls } n > 1 \\ \mathcal{Q}\texttt{++}[(n', u)] & \text{falls } n = 1 \end{cases}$$

und n' eine zufällig gewählte natürliche Zahl ist.
Außerdem muss $e \not\equiv (\texttt{throw}\ y)$ gelten.

Die Regel (Receive) modelliert den regulären Fall, in dem das *Block-Flag* gelöscht ist und Exceptions nicht unterbunden werden. Ist das *Stuck-Flag* gesetzt und eine Exception eingetroffen, greift die Regel (Interrupt). Das Ergebnis beider Regeln ist identisch: Der aktuell betrachtete Control-Ausdruck e wird verworfen und mit `throw` x durch die erste Exception aus $\mathcal{E}$ ersetzt. Außerdem wird der funktionale Stack $\mathcal{S}$ gelöscht, ein möglicherweise gesetztes *Stuck-Flag* gelöscht und die verbrauchte Ressource des Threads in $\mathcal{Q}'$ angepasst.

Beide Regeln werden nicht angewendet, wenn der Control-Ausdruck des Threads bereits einer anderen Exception `throw` y entspricht. Damit wird verhindert, dass eine bereits zu propagierende Exception direkt von der nächsten, sich in der Warteschlange $\mathcal{E}$ befindlichen Exception überlagert wird.

Das Setzen des Stacks $\mathcal{S}$ auf [] kommt einem simplifizierten „Zurücksetzen auf Initialzustand“ des funktionalen Ausdrucks, wie in 4.1.2 beschrieben gleich. Gefahrlos möglich wird es an dieser Stelle, da die Auswertungsregeln der Mark 1 in Abbildung 4.2 entsprechend angepasst wurden, keine Bindungen aus dem zu Heap entfernen. Bei einem erneuten Auswerten der selben Heapbindung liegt diese noch immer in ihrem Initialzustand auf dem Heap. Dies geschieht zwar auf Kosten von möglichen (unperformanten) Mehrfachauswertungen des gleichen Ausdrucks, führt allerdings aufgrund seines rein funktionalen Charakters nicht zu falschen Ergebnissen.

4.4.4.7 Auswerten einer Future

Falls keine der bisher vorgestellten *CIOM1*–Regeln angewendet werden konnte, handelt es sich beim nächsten Auswertungsschritt des aktuellen Threads offenbar um einen monadi-

schen oder funktionalen Schritt. Diese werden auf der *IOM1* durchgeführt.

(LiftIOM) $(\mathcal{H}, \mathcal{M}, \mathcal{T} \uplus \{(u, e, \mathcal{S}_u, \mathcal{I}_u, \binom{s}{b}, \mathcal{E}_u)\}, (n, u) : \mathcal{Q})$
$\xrightarrow{CIOM1} (\mathcal{H}', \mathcal{M}', \mathcal{T} \uplus \{(u, e', \mathcal{S}'_u, \mathcal{I}'_u, \binom{s'}{b'}, \mathcal{E}_u)\}, \mathcal{Q}')$
falls $(\mathcal{H}, \mathcal{M}, e, \mathcal{S}, \mathcal{I}, \binom{s}{b}) \xrightarrow{IOM1} (\mathcal{H}', \mathcal{M}', e', \mathcal{S}', \mathcal{I}', \binom{s'}{b'})$

Dabei ist $\mathcal{Q}' := \begin{cases} (n-1, u) : \mathcal{Q} & \text{falls } n > 1 \\ \mathcal{Q}\texttt{++}[(n', u)] & \text{falls } n = 1 \end{cases}$
und n' eine zufällig gewählte natürliche Zahl.

Diese Regel wird nur angewendet, falls keine der Regeln (ForkIO), (UnIO), (Receive), (Interrupt), (PshTT), (ThrTo) oder (ThrToX) für den Thread u anwendbar ist.

Die Regel (LiftIOM) führt den Zustandsübergang des Threads durch, indem sie die Auswertung auf die *IOM1* „lifted“ und passt die verbrauchte Ressource in gewohnter Weise an.

5 Implementierung in Haskell

Für eine tatsächliche Implementierung von *Concurrent Haskell mit Futures und Exceptions* bietet sich Haskell als funktionale Programmiersprache und als eine maßgebliche Orientierung die *CHF*–Implementierung aus [Sab11] an.

Im Folgenden wird ein grober Überblick gegeben, wobei insbesondere die verwendeten Datenstrukturen vorgestellt werden.

Die Gliederung dieses Kapitels orientiert sich an Abbildung 4.1 und beschreibt zunächst die syntaktische und semantische Analyse eines *CHFE*–Quelltextes in Abschnitt 5.1. Die skizzenhafte Beschreibung der Implementierung der Abstrakten Maschinen folgt in Abschnitt 5.2. In Abschnitt 5.3 folgen schließlich einige Beispielprogramme.

5.1 Syntaktische und semantische Analyse

Für die Ausführung eines durch einen Anwender geschriebenen *CHFE*–Quelltextes muss dieser erst in eine maschinenlesbare Form gebracht werden. Ein solcher Quelltext besteht dabei aus zwei Teilen. Optional ist die Angabe von Konstruktordefinitionen am Beginn des Quelltextes. Jede Definition wird dabei durch das Schlüsselwort `data` eingeleitet. Gefolgt werden diese Definitionen von einem obligatorischen auszuwertenden Ausdruck, der mit dem Schlüsselwort `expression` eingeleitet wird. Das in Abbildung 2.2 dargestellte Programm sieht als *CHFE*–Quelltext z.B. wie folgt aus:

```
data Bool     = True | False
data ListBool = ConsBool Bool ListBool | NilBool
expression
  letrec
    list    = (ConsBool True (ConsBool True NilBool));
    isEmpty = \x -> case x of {
                      NilBool       -> True;
                      ConsBool y ys -> False;
                    };
  in (isEmpty list)
```

5.1.1 Lexen und Parsen

Die Vorverarbeitung eines so aufgebauten Quelltextes besteht aus der lexikalischen Analyse durch einen *Lexer*, der den Quelltext in *Token* unterteilt. Diese Token gliedern sich in Schlüsselwörter und Symbole (vorgegeben durch die *CHFE*–Syntax), Konstruktornamen (definiert durch vorangegangene Konstruktordefinitionen) und Variablennamen. Der so erzeugte Tokenstrom des Lexers stellt die Eingabe für den im nächsten Schritt folgenden Parser dar.

Aufgabe des Parsers ist die Überprüfung der Gültigkeit des Ausdrucks und der `data`–Definitionen anhand der *CHFE*–Grammatik. Dabei stellt der Parser außerdem sicher, dass folgende Bedingungen erfüllt sind:

- Für alle verwendeten Konstruktoren ist eine entsprechende `data`–Definition vorhanden.

- Alle Bindungsvariablen eines `letrec`–Ausdrucks sind verschieden.
  Ein Ausdruck der Form `letrec` x = `True`; $x = \ldots$ `in` $\ldots$ führt zu einem Fehler.

- Alle Alternativen eines `case`–Ausdrucks sind verschieden.
  Ein Ausdruck der Form `case` x `of` {`True` $\rightarrow$ `True`; `True` $\rightarrow \ldots$} führt zu einem Fehler.

- Alle Variablen eines Patterns einer `case`–Alternative sind verschieden.
  Ein Ausdruck der Form `case` x `of` {(`ConsBool` y y) $\rightarrow$ `False`; `NilBool` $\rightarrow$ `True`} führt zu einem Fehler.

Ein solcher Parser lässt sich z.B. als Shift/Reduce-Parser mithilfe des *Happy Parser Generators für Haskell*[12] erstellen. Dieser benötigt als Eingabe eine Datei mit der in *Backus–Naur–Form* modellierten eindeutigen *CHFE*–Grammatik.

Falls keine Fehler auftreten, besteht das Ergebnis des Parsers aus einem syntaktisch korrekten *CHFE*–Programm in Form eines *Syntaxbaumes*. Dieser ist im Wesentlichen ein Paar, bestehend aus einer Liste von Typdefinitionen und einer Datenstruktur, die den eigentlichen Syntaxbaum des Ausdrucks rekursiv modelliert. Der dazu verwendete Haskell-Datentyp ist in seiner Abbildung sehr nah an der *CHFE*–Grammatik gehalten und wird ähnlich wie in [Sab11] beschrieben:

[12] Happy Parser Generator für Haskell: `http://www.haskell.org/happy`.

```
data Expr label cname vname =
    Var    vname
  | Lam    vname                        (Expr label cname vname)
  | App    (Expr label cname vname)     (Expr label cname vname)
  | Seq    (Expr label cname vname)     (Expr label cname vname)
  | Cons   (Either MAction cname)       [Expr label cname vname]
  | Case   (Expr label cname vname)     [CAlt label cname vname]
  | Letrec [Binding label cname vname] (Expr label cname vname)
  | Label  label                        (Expr label cname vname)

data MAction = Fork | Return | Take | Put | New | Bind
             | ThrowTo | Throw | Catch | Unblock | Block

data CAlt label cname vname =
  CAlt cname [vname] (Expr label cname vname)

data Binding label cname vname =
  vname :=: (Expr label cname vname)
```

Der Datentyp ist polymorph definiert, so dass die Typen für die Konstruktornamen `cname`, die Variablennamen `vname` und ein beliebiges Label `label` beim Instantiieren vorgegeben werden können. Hervorzuheben ist die Definition des Konstruktors `Cons`, der mithilfe des Haskell–Datentyps `data Either a b = Left a | Right b` definiert ist. Dadurch lassen sich normale *CHFE*–Konstruktoranwendungen $(c\ e_1 \ldots e_n)$ in dieser Datenstruktur durch `Cons (Right "c") [`$e_1, \ldots, e_n$`]` beschreiben (sofern für `cname` der Typ `String` gewählt wurde). Monadische Aktionen hingegen können ebenfalls durch `Cons` beschrieben werden. Die Behandlung als Konstruktor ist hilfreich, da die Mark 1 monadische Aktionen als Werte behandeln soll. Ein monadisches Bind der Form $e_1 \gg\!= e_2$ lässt sich hiermit z.B. darstellen als `Cons (Left Bind) [`e_1, e_2`]`.

Der Typ `CAlt` modelliert die `case`–Alternativen der Form $(c\ y_1 \ldots y_n) \rightarrow e$, wobei zu jeweils einem Konstruktornamen c eine Menge von Variablennamen y_i und ein Ausdruck e gespeichert werden kann. In ähnlicher Weise wird der Typ `Binding` verwendet, um bei `letrec`–Bindungen jeweils einer Variable einen Ausdruck zuzuordnen. Der zusätzliche Typ `Label` wird später gebraucht, um einem Unterausdruck eine beliebige Markierung zuzuweisen. Dies wird u.A. beim folgenden Typcheck verwendet, um die jeweilige Typisierung für jede Stufe zwischenspeichern zu können.

5.1.2 Typcheck und Unifikation

Die semantische Analyse wird durch den Typcheck durchgeführt. Für die Typisierung, die an der Stelle des `Label`s gespeichert wird, wird die Datenstruktur `Type` verwendet. Typkonstruktoren können durch `TC`, Funktionstypen durch `:->:`, I/O-Typen durch `TIO` und `MVar`s durch `TMVar` beschrieben werden. Der Konstruktor `TVar` stellt Typvariablen dar, wie sie in Kapitel 2.3.2 (wegen des monomorphen Typsystems von *CHFE* ausschließlich) für den Typcheck eingeführt werden. Der Konstruktor `TFID` beschreibt eine FutureID und damit die zweite Komponente eines *FIDCons*.

```
data Type tname =
    TC tname
  | (Type tname) :->: (Type tname)
  | TVar String
  | TIO (Type tname)
  | TMVar (Type tname)
  | TFID (Type tname)
```

Damit lässt sich der Typcheck nach den angegebenen Regeln definieren, indem der Syntaxbaum rekursiv abgearbeitet und typisiert wird. Die notwendigen Typinformationen für Γ können aus den `data`–Definitionen extrahiert werden. Die Typen für `Unit`, `Exception` und `IDTYP` wurden dabei fest vorgegeben. Beim Typisieren entstandene Gleichungen werden in einer Liste der Form `[(Type tname, Type tname)]` gesammelt, die für jede Gleichung ein Paar mit jeweils der linken und rechten Seite der Gleichung enthält. Das Unifizieren geschieht analog zu den angegebenen Regeln, wobei lediglich für jede Gleichung die Regeln solange rekursiv angewendet werden müssen, bis sie entweder gelöst ist, oder ein Fehler auftritt.

Nach Anwenden des allgemeinsten Unifikators auf den Ausdruck liegt ein syntaktisch und semantisch korrektes *CHFE*–Programm vor.

5.2 Implementierung der Abstrakten Maschinen

Das syntaktisch und semantisch korrekte *CHFE*–Programm muss zunächst transformiert werden um es anschließend auf der Concurrent–Mark 1 auszuführen, die sich zusätzlich aus den Maschinen I/O–Mark 1 und Mark 1 zusammensetzt.

Für die Auswertung auf den entsprechenden Maschinen werden eine `Heap`– und eine `Stack`–Datenstruktur benötigt. Letztere lässt sich bequem mit Haskells Listen modellieren, wohingegen zur Implementierung des Heaps Haskells Wörterbuch–Datenstruktur `Data.Map` Verwendung findet. Damit lassen sich die notwendigen Schlüssel/Wert–Bindungen effizient bewerkstelligen.

5.2.1 Transformation

Zur Anwendung der Übergangsregeln der verschiedenen Abstrakten Maschinen muss das *CHFE*–Programm in vereinfachter Maschinensyntax vorliegen. Die dafür notwendige Datenstruktur unterscheidet sich nur unwesentlich von der bereits definierten, modelliert aber gerade die benötigte Eigenschaft, dass Argumente nur noch aus Variablen bestehen.

```
data MExpr cname vname =
    VarM    vname
  | LamM    vname                   (MExpr cname vname)
  | AppM    (MExpr cname vname)     vname
  | SeqM    (MExpr cname vname)     vname
  | ConsM   (Either MAction cname)  [vname]
  | CaseM   (MExpr cname vname)     [MCAlt cname vname]
  | LetrecM [MBinding cname vname]  (MExpr cname vname)
  | TID     vname
```

Zusätzlich hinzugekommen ist das interne `TID`–Konstrukt, dass zum Erkennen einer FutureID benötigt wird. Die eigentliche Transformation kann durch die in Kapitel 4.4.1.2 definierten Umbenennungen rekursiv vollzogen werden. Die Typinformationen in den `Labels` sind dabei nicht mehr notwendig und können entfernt werden. Um sicherzustellen, dass

die *DVC* weiterhin erfüllt ist, wird der Ausdruck vor der Transformation von oben nach unten mit frischen Variablennamen rekursiv umbenannt.

5.2.2 Mark 1

Die rein funktionale Auswertung auf der Mark 1 lässt sich mithilfe einer Heap– und einer Stack–Datenstruktur und dem vorliegenden Syntaxbaum implementieren. Haskells Record–Syntax [Mar10, Kapitel 3] hilft dabei, den *M1*–Zustand übersichtlich zu definieren. So lassen sich ein Heap, ein funktionaler Stack und Control etwa wie folgt vereinfacht zusammenfassen:

```
data Mark1State = Mark1State {
  heap :: Heap, control :: Mark1Expr, stack :: Stack
}
```

Der Startzustand besteht aus einem leeren Heap, einem leeren Stack und dem auszuwertenden Ausdruck als Control.

Die eigentliche Implementierung besteht aus einer Funktion `nextState`, die als Parameter einen solchen Zustand übergeben bekommt und genau einen Auswertungsschritt der Maschine vollzieht. Die angegebenen Übergangsregeln können angewendet werden, indem mittels Haskells Patternmatching über den Ausdruck Control festgestellt wird, welcher Redex vorliegt.

Liegt bspw. eine Anwendung $(e\ v)$ in Form von `Mark1State{control=AppM e v}` vor, greift die Regel (pushApp) (vgl. Abb. 4.2), legt das Element $\#_{\mathsf{app}}(v)$ auf den Stack und gibt als Zustand den veränderten Stack sowie ein Control wieder, für den e eingesetzt wurde. Liegt als Control-Ausdruck dann später `Mark1State{control=LamM v e}` vor, wird geprüft, ob das oberste Stackelement $\#_{\mathsf{app}}(v)$ ist. Falls dies der Fall ist, wird die β–Reduktion durchgeführt und v für den formalen Parameter `v` in `e` eingesetzt.

Eine Funktion `finalState` bekommt als Parameter einen *M1*–Startzustand übergeben und führt solange `nextState` durch, bis ein *M1*–Endzustand erreicht ist. Dieser ist daran erkennbar, dass der Stack leer ist, und als Control entweder eine Abstraktion oder eine Konstruktoranwendung vorliegt. Da die `Either`-Datenstruktur für `ConsM` sowohl normale Konstruktoranwendungen als auch monadische Aktionen ausweist, werden damit auch monadische Aktionen als Werte behandelt. Das passt gerade zur geforderten Definition der *M1*–Werte.[13]

5.2.3 I/O-Mark 1

Die Implementierung der I/O–Mark 1 erweitert die Zustands–Datenstruktur um einen I/O–Stack, die Menge der `MVar`s und die beiden Flags `stuck` und `blocked`. Für die Menge der `MVar`s kann ein weiterer `Heap` eingesetzt werden. Ein Startzustand ergänzt den der Mark 1 um eine leere `MVar`–Menge, einen leeren I/O–Stack und die jeweils ungesetzten Flags.

Das Patternmatching von `nextState` wird um das Abfragen von monadischen Aktionen (`ConsM (Left ...)`) erweitert. Der Ausdruck wird solange auf der Mark 1 ausgewertet, bis der funktionale Stack leer ist. Besteht Control dann aus einer monadischen Aktion, wird diese unter Verwendung des I/O–Stacks ausgeführt.

Eine Funktion `finalState` bekommt als Parameter einen *IOM1*-Startzustand und führt wiederum solange Auswertungsschritte mit der Funktion `nextState` durch, bis ein *IOM1*–Endzustand erreicht ist.

[13] Theoretisch kann ein *M1*–Wert auch aus einem `TID` bestehen. Dabei handelt es sich allerdings um ein internes Konstrukt, dass ausschließlich durch die Concurrent–Mark 1 erzeugt wird, und daher praktisch auf der Mark 1 und der im nächsten Abschnitt folgenden I/O–Mark 1 noch nicht abgefragt werden muss.

5.2.4 Concurrent Mark 1

Die Datenstruktur zur Repräsentation eines Zustandes der Concurrent–Mark 1 entspricht ihrer in Kapitel 4.4.4 gegebenen Definition und modelliert den globalen Heap, die `MVar`–Menge und die Menge der Futures jeweils als `Heap`. Die Thread–Warteschlange $\mathcal{Q}$ wird durch eine Liste `queue :: [(Int,Mark1Var)]` abgebildet, wobei `Mark1Var` eine Variable zur Bezeichnung einer Future darstellt. Dieser Bezeichner stellt gleichfalls den Schlüssel zum Finden einer Future in der Future–Menge dar. Eine dort abgelegt Future besteht aus eine Datenstruktur, die die weiteren Komponenten der Future enthält. Es sind dies der Bezeichner, der Stack, der I/O–Stack, die beiden Flags, Control und eine weitere Liste als Darstellung der Exception–Warteschlange $\mathcal{E}$. Da Exception–Beschreibungen, aufgrund der Maschinensyntax, Heapbindungen sind, muss auch diese Liste nur `Mark1Var`–Variablen vorhalten können.

Die Auswertung auf der Concurrent–Mark 1 gliedert sich abermals in zwei Funktionen `finalState` und `nextState`. Dabei überprüft `finalState`, ob der aktive Thread (der vorderste in der Wartschlange) der `_main_`–Thread ist und testet, ob dieser einen Endzustand erreicht hat. In diesem Fall endet die Auswertung. Andernfalls wird `nextState` einen Auswertungsschritt ausführen.

Bevor ein Auswertungsschritt mit `nextState` durchgeführt wird, wird zunächst der aktive Thread aus der Menge geholt und überprüft, ob sich eine Exception in der Exceptionliste befindet. Ist dies der Fall und entweder das *Block-Flag* gelöscht oder das *Stuck-Flag* gesetzt, greift die Regel (Receive) bzw. (Interrupt).

Im nächsten Schritt wird überprüft, ob der aktive Thread einen Endzustand erreicht hat. Ist dies der Fall und handelt es sich nicht um den `_main_`-Thread (denn dessen Endzustand wird in `finaleState` abgefragt), greift die Regel (UnIO), erstellt eine Heapbindung mit dem Ergebnis und entfernt den Thread.

Die auf der Concurrent–Mark 1 auszuwertenden Konstrukte `future` und `throwTo` werden wie gewohnt über Haskells Patternmatching bearbeitet.

Ist keine der bisher beschriebenen Konditionen erfüllt, handelt es sich um einen *IOM1*–Auswertungsschritt und die Regel (LiftIOM) greift. Dazu wird ein *IOM1*–Zustand „zusammengebaut“, der aus dem globalen Heap, der globalen `MVar`–Menge sowie den Stacks, Flags und Control besteht. Dieser Zustand wird mit der Funktion `IOM1.nextState` auf der I/O–Mark 1 ausgewertet und die veränderten Rückgabewerte wieder „auseinandergebaut“.

Nachdem der aktive Thread einen Auswertungsschritt durchgeführt hat, wird seine Ressourcenzahl verringert und er verbleibt als erstes Element in der Warteschlange. Verringert sich die Ressourcenzahl auf Null, wird eine neue Zahl erzeugt, und der Thread an das Ende der Schlange gestellt. Dies entspricht dem vorgeschriebenen *Scheduling* der Threads im Zeitscheibenverfahren.

5.3 Beispiele in *CHFE*

Im Folgenden werden an Beispielen die Funktion `catch`, `throwTo`, `block` und `unblock` demonstriert. Dabei wird angenommen, dass die folgenden Typen in der Implementierung von *CHFE* fest definiert sind:

```
data Unit      = Unit
data Peano     = Zero | Succ Peano
data Exception = Error Peano
```

Damit lässt sich `throwTo` als Exception–Beschreibung eine Fehlernummer als Peanozahl mitgeben, auf die im Exception–Handler mit einem `case` reagiert werden könnte.

5.3.1 Beispiel eines Threads mit Endlosschleife

Das in Abbildung 5.1 dargestellte *CHFE*–Programm besteht im Wesentlichen aus einer Future und einem Hauptprogramm. Um insgesamt vier unterschiedliche Fälle abdecken zu können, wurde das Hauptprogramm selbst abstrahiert und kann als `main1`, `main2a`, `main2b` oder `main3` im äußersten `in`–Ausdruck eingesetzt werden. Diese Hauptprogramme unterscheiden sich nur durch den erzeugten Thread (die Nummer hinter `main` korrespondiert mit der Nummer eines `thread`s) und ob die erzeugte Future mit `throwTo` abgebrochen wird (dies ist bei `main2b` und `main3` der Fall).

Bei Ausführung des Programms wird zunächst eine leere Synchronisierungsvariable mit `emptySynch` angelegt und dann ein nebenläufiger Thread erzeugt. Die Synchronisierungsvariable wird von der Future wieder befüllt. Durch das, im aufrufenden Prozess wartende `takeMVar` wartet dieser Prozess also, bis die Future fertig ausgewertet wurde.[14] Dieses Verhalten zeigt die Funktion `main1` exemplarisch. Ihre Auswertung terminiert entsprechend mit `(return Finish)`.

Die Funktion `main2a` terminiert nicht. Die Future `thread2` wertet zunächst die funktionale Endlosschleife `endlosPeano` aus, bevor sie die Synchronisierungsvariable befüllen kann. Dies geschieht nie. Obwohl auf den Wert der Future nie mit `futureValue` zugegriffen wird, kann die Auswertung dennoch nicht fortgesetzt werden, da die Auswertung bei `takeMVar sync` in `main2a` stehen bleibt.

Eine solche Future kann nun im Idealfall mit `throwTo` von außen beendet werden. Die Funktion `main2b` modelliert diesen Fall und terminiert erwartungsgemäß mit `(return Finish)`, da die Synchronisierungsvariable in diesem Fall durch den Handler wieder befüllt wird. Dies gelingt, da in `thread2` die komplette `catch`–Anweisung in einem block–Kontext, der von `catch` umschlossene Ausdruck aber in einem unblock–Kontext steht. Nach Auswerten des `throwTo`–Befehls wird dieser unblock–Kontext verlassen und der Handler im äußeren block–Kontext unterbrechungsfrei ausgeführt.

Falls der Programmierer diesen inneren `unblock`–Aufruf weglässt, lässt sich die Future nicht durch `throwTo` beenden. Dieser Fall ist als `main3` aufgeführt und terminiert nicht.

5.3.2 Beispiel eines stuck Threads

Die Semantik von *CHFE* wurde angepasst, um in bestimmten Situationen zu gewährleisten, das Threads auch dann Exceptions empfangen können, wenn ihre aktuelle Auswertung sich in einem block–Kontext befindet. Das *CHFE*–Programm in Abbildung 5.2 zeigt einen solchen Fall.

Das Programm ist dabei dem vorangegangenen Beispiel sehr ähnlich. Anstelle der Endlosschleife wird hier allerdings versucht, einen weiteren Wert in eine bereits gefüllte `MVar` zu schreiben. Da diese bestimmte `MVar` durch keinen Teil des Programms jemals geleert wird, muss die Auswertung an dieser Stelle für immer warten und die Synchronisierungsvariable erteilt dem aufrufenden Prozess nie die Freigabe zum Fortsetzen. Dementsprechend terminiert die Funktion `main1a` nicht.

Zusätzlich befindet sich die Future in einem block–Zustand (ähnlich wie `thread3` aus Abbildung 5.1) und dürfte die in `main1b` gesendete Exception daher eigentlich nicht empfangen. Dass die Funktion `main1b` dennoch mit `(return Finish)` terminiert liegt gerade daran, dass der Thread sein *Stuck-Flag* gesetzt hat und für Exceptions empfangsbereit ist.

[14] In diesem Beispiel spielt der Ergebniswert (`True` oder `False`) der Future keine Rolle.

```
data Bool = True | False
data Data = Finish
expression letrec
  futureValue = \m -> (case m of {[a;b] -> a});
  futureId    = \m -> (case m of {[a;b] -> b});
  endlosPeano = letrec
                  zaehlen  = \p -> ((zaehlen (Succ p)))
                in zaehlen Zero;
  emptySynch  = (newMVar Unit) >>= \s ->
                (takeMVar s) >>= \u ->
                (return s);
  thread1     = \sync -> block{ catch
                             { ((putMVar sync Unit) >>= \u ->
                               (return True)) }
                             { \exc -> (putMVar sync Unit) >>= \u ->
                                       (return False) }
                           };
  thread2     = \sync -> block{ catch
                              {unblock{
                                 seq endlosPeano
                                     ((putMVar sync Unit) >>= \u ->
                                     (return True)) }
                              }
                              {\exc -> (putMVar sync Unit) >>= \u ->
                                       (return False) }
                            };
  thread3     = \sync -> block { catch
                              {seq endlosPeano
                                   ((putMVar sync Unit) >>= \u ->
                                   (return True))
                              }
                              {\exc -> (putMVar sync Unit) >>= \u ->
                                       (return False)}
                            };
  main1       = emptySynch >>= \sync -> (future(thread1 sync))
                >>= \f -> (takeMVar sync) >>= \u -> (return Finish);
  main2a      = emptySynch >>= \sync -> (future(thread2 sync))
                >>= \f -> (takeMVar sync) >>= \u -> (return Finish);
  main2b      = emptySynch >>= \sync -> (future(thread2 sync))
                >>= \f -> throwTo (futureId f) (Error Z) >>= \v ->
               (takeMVar sync) >>= \u -> (return Finish);
  main3       = emptySynch >>= \sync -> (future(thread3 sync))
                >>= \f -> throwTo (futureId f) (Error Z) >>= \v ->
               (takeMVar sync) >>= \u -> (return Finish)
in main1
```

Abbildung 5.1: Beispielprogramm in *CHFE* zur Demonstration von `throwTo` in Verbindung mit `block` und `unblock`. Für den untersten `in`–Ausdruck kann `main1` bis `main3` eingesetzt werden. Es terminieren: `main1`, `main2b`; es terminieren nicht: `main2a`, `main3`.

```
data Bool = True | False
data Data = Finish
expression
letrec
  future      = \m -> (case m of {[a;b] -> a});
  futureId    = \m -> (case m of {[a;b] -> b});
  stuck       = (newMVar Unit) >>= \u -> (putMVar u Unit)
  emptySynch  = (newMVar Unit) >>= \s ->
                (takeMVar s) >>= \u ->
                (return s);
  thread      = \sync -> block {
                             catch{
                               stuck >>= \v ->
                               (putMVar sync Unit) >>= \u ->
                               (return True)
                             }
                             {\exc -> (putMVar sync Unit) >>= \u ->
                                       (return False)}
                         };
  main1a       = emptySynch                      >>= \sync ->
                 (forkIO(thread sync))           >>= \f ->
                 (takeMVar sync)                 >>= \u ->
                 (return Finish);
  main1b       = emptySynch                      >>= \sync ->
                 (forkIO(thread sync))           >>= \f ->
                  throwTo (futureId f) (Error Z) >>= \v ->
                 (takeMVar sync)                 >>= \u ->
                 (return Finish)
in main1a
```

Abbildung 5.2: *CHFE*–Beispielprogramm eines stuck Threads. Die Funktion `main1a` terminiert nicht. Die Funktion `main1b` terminiert. Sie sendet eine Exception an den Thread, der dadurch beendet wird, obwohl er in einem block-Kontext ist.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Zur *Implementierung einer Abstrakten Maschine zu einem nebenläufigen Programmkalkül mit Futures und Exceptions in Haskell* wurden zunächst die Grundlagen funktionalen Programmierens und der Lambda-Kalkül vorgestellt. Dieser wurde sukzessive erweitert um Datenkonstuktoren, ein monomorphes Typsystem, Fallunterscheidungen, `letrec`–Ausdrücke, Haskells `seq`–Operator und Seiteneffekte in Form monadischen I/Os. Nebenläufige Prozesse wurden in Form von Futures eingeführt. Dabei handelt es sich um die Möglichkeit, Berechnungen zwar nebenläufig auszuführen, auf das Ergebnis aber in Form einer Variablen an beliebigen Stellen des übrigen Programms referenzieren zu können. Für diesen erweiterten Lambda-Kalkül mit Futures wurden die Syntax, ein Typisierungsalgorithmus und die zugehörige Semantik formal definiert.

Im nächsten Schritt wurden Exceptions als synchron oder asynchron unterschieden. Synchrone Exceptions treten als direkte Folge der Programmausführung auf (z.B. Division–durch–Null) und signalisieren einen Fehler, bei dem der ursprüngliche Programmkalkül meist nicht mehr terminierte. Asynchrone Exceptions sind häufig Folge externer Events und Abbruchkriterium eines Prozesses. Eine aufgetretene Exception kann im Programm aufgefangen und mit einer speziell dafür programmierten Fehlerroutine bearbeitet werden. Es obliegt damit weiterhin dem Programmierer, etwa die Konsistenz der Datenbasis durch umsichtiges Entwickeln diese Fehlerroutine zu gewährleisten. Mithilfe dieses Instrumentariums kann allerdings gewährleistet werden, dass ein Programm trotz des Auftretens eines Fehlers terminieren kann. Das asynchrone, also unvorhersehbare Auftreten von Exceptions machte zusätzlich die Möglichkeit notwendig, definierbare Programmteile temporär gegen das Eintreten asynchroner Exceptions zu schützen. Hier waren vorsichtige Anpassungen an der Semantik notwendig, um in bestimmten Situationen trotz dieses Schutzes, dennoch in der Lage sein zu können, asynchrone Exceptions empfangen zu können. Die entsprechenden syntaktischen und semantischen Anpassungen zur Unterstützung synchroner und asynchroner Exceptions eines erweiterten Lambda-Kalküls mit nebenläufigen Threads wurden eingeführt.

Die Zusammenführung des erweiterten Lambda-Kalküls mit Futures und Exceptions fand im letzten Kapitel statt. Die Syntax und der Typisierungsalgorithmus wurden erweitert und angepasst, um Exceptions generieren, temporär blockieren und verarbeiten zu

können. Die Angabe der Abstrakten Maschine gliederte sich in drei aufeinander aufbauende Abstrakte Maschinen mit jeweils erweitertem Funktionsumfang. Eine erste Maschine lieferte die Semantik für einen rein funktionalen erweiterten Lambda-Kalkül, der monadische Aktionen als Werte behandelt und nicht weiter auswertet. Darauf aufsetzend erweiterte eine zweite Maschine den Kalkül um die Auswertungen dieser monadischen Aktionen. In einer dritten Maschine wurden Futures als nebenläufige, auf der zweiten Maschine auszuwertende Prozesse eingeführt.

Damit ist ein Interpreter für die neu entstandene Sprache gegeben, die synchrone und asynchrone Exceptions auch bei Prozessen in Form von Futures unterstützt. Bei Auswertungen, bei denen der ursprüngliche Programmkalkül nicht terminierte, existiert durch das neue Instrumentarium nun die Möglichkeit, dies durch umsichtiges Programmieren zu vermeiden. Zwar bleibt der Programmierer weiterhin in der Verantwortung, ist in seiner Programmgestaltung durch die neuen Funktionen aber flexibler.

6.2 Ausblick

Im Zuge der Entwicklung asynchroner Exceptions in dieser Arbeit, wurden in der Sprache *CHFE* ungewollte redundante funktionale Auswertungen möglich. In zukünftigen Versionen der Sprache sollten diese aus Performancegründen nach Möglichkeit unterbunden werden. Dafür wäre es notwendig, beim Zustellen einer asynchronen Exception den funktionalen Stack rückwärts wieder „aufzurollen“, und auf diese Weise den Initialzustand (oder auch den partiell ausgewerteten Zustand) zurück in den Heap zu schreiben.

Ähnlich wie in [SSS11] kann die Semantik der vorliegenden Sprache in weiterführenden Arbeiten nun bzgl. geltender Programmgleichheiten unter Verwendung einer kontextuellen Äquivalenz untersucht werden.

Literaturverzeichnis

[BS01] Baader, Franz und Wayne Snyder: *Unification Theory.* In: Robinson, Alan und Andrei Voronkov (Herausgeber): *Handbook of Automated Reasoning*, Seiten 445–532. Elsevier Science Publishers B.V., 2001.

[Chu36] Church, Alonzo: *An unsolvable problem of elementary number theory.* In: *American Journal of Mathematics*, Band 58, 1936.

[Dij71] Dijkstra, Edsger Wybe: *Hierarchical ordering of sequential processes.* In: *Acta Informatica*, Band 1, Seiten 115–138. 1971.

[JV06] Johann, Patricia und Janis Voigtländer: *The Impact of `seq` on Free Theorems-Based Program Transformations.* In: *Fundamenta Informaticae*, Band 69, Seiten 63–102, 2006.

[Mar10] Marlow, Simon (Herausgeber): *Haskell 2010 Language Report.* `http://www.haskell.org/onlinereport/haskell2010`, 2010. Zuletzt geprüft am 02.08.2013.

[Mar12] Marlow, Simon: *Parallel and Concurrent Programming in Haskell.* In: Zsók, V., Z. Horváth und R. Plasmeijer (Herausgeber): *CEFP 2011*, Band 7241 der Reihe *LNCS*, Seiten 339–401. 2012.

[Mil78] Milner, Robin: *A theory of type polymorphism in programming.* In: *Journal of Computer and System Sciences*, Band 17, Seiten 348–375, 1978.

[MJMR01] Marlow, Simon, Simon Peyton Jones, Andrew Moran und John Reppy: *Asynchronous Exceptions in Haskell.* In: *ACM Conference on Programming Languages Design and Implementation (PLDI'01)*, Seiten 274–285. ACM, 2001.

[MO98] Maraist, John und Martin Odersky: *The call-by-need lambda calculus.* In: *Journal of Functional Programming*, Band 8, Seiten 275–317, 1998.

[PJ87] Peyton Jones, Simon: *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[PJ01] Peyton Jones, Simon: *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.* In: Hoare, Tony, Manfred Broy und Ralf Steinbruggen (Herausgeber): *Engineering theories of software construction*, Seiten 47–96. IOS Press, 2001. Version vom 07.04.2010, online verfügbar unter `http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf`, zuletzt geprüft am 15.8.2013.

[PJGF96] Peyton Jones, Simon, Andrew Gordon und Sigbjorn Finne: *Concurrent Haskell.* In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, Seiten 295–308, New York, NY, USA, 1996. ACM.

[PJRH+99] Peyton Jones, Simon, Alastair Reid, Fergus Henderson, Tony Hoare und Simon Marlow: *A semantics for imprecise exceptions.* In: *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, Seiten 25–36, New York, NY, USA, 1999. ACM.

[Plo75] Plotkin, Gordon D.: *Call-by-Name, Call-by-Value and the λ-Calculus.* In: *Theoretical Computer Science*, Band 1, Seiten 125–159, 1975.

[Rei98] Reid, Alastair: *Putting the Spine back in the Spineless Tagless G-Machine: An Implementation of Resumable Black-Holes.* In: *Proc. IFL'98 (selected papers), volume 1595 of LNCS*, Seiten 186–199. Springer-Verlag, 1998.

[Sab08] Sabel, David: *Semantics of a Call-by-Need Lambda Calculus with McCarthy's amb for Program Equivalence.* Dissertation, Johann Wolfgang Goethe-Universität Frankfurt, Institut für Informatik. Fachbereich Informatik und Mathematik, November 2008.

[Sab11] Sabel, David: *Anleitung zum Praktikum „Funktionale Programmierung" im Sommersemester 2011.* Johann Wolfgang Goethe Universität Frankfurt/Main, 2011. `http://www.ki.informatik.uni-frankfurt.de/lehre/SS2011/FP-PR/`. Zuletzt geprüft am 07.08.2013.

[Sab12] Sabel, David: *An Abstract Machine for Concurrent Haskell with Futures.* In: Jähnichen, Stefan, Bernhard Rumpe und Holger Schlingloff (Herausgeber): *Software Engineering 2012 Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar – 2. März 2012 in Berlin*, Band 199 der Reihe *GI Edition - Lecture Notes in Informatics*, Seiten 29–44, February 2012. (5. Arbeitstagung Programmiersprachen (ATPS'12)).

[Ses97] Sestoft, Peter: *Deriving a Lazy Abstract Machine.* In: *Journal of Functional Programming*, Band 7, Seiten 231–264, 1997.

[SS11] Schmidt-Schauss, Manfred: *Skript zur Vorlesung „Einführung in die Funktionale Programmierung".* Johann Wolfgang Goethe Universität Frankfurt/Main, 2011. `http://www.ki.informatik.uni-frankfurt.de/lehre/WS2011/EFP`. Zuletzt geprüft am 07.08.2013.

[SSS11] Sabel, David und Manfred Schmidt-Schauss: *A contextual semantics for concurrent Haskell with futures.* In: *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP'11, Seiten 101–112, New York, NY, USA, July 2011. ACM.

[Tur36] Turing, Alan M.: *On Computable Numbers, with an Application to the Entscheidungsproblem.* In: *Proceedings of the London Mathematical Society*, Band 2, Seiten 230–265, 1936.

[Tur37] Turing, Alan M.: *Computability and λ-Definability.* In: *The Journal of Symbolic Logic, Vol. 2, No. 4*, Seiten 153–163, 1937.

Abbildungsverzeichnis

Index

W

Z