Bakkalaureatsarbeit

An Executable Rewriting Logic Semantics for Concurrent Haskell

Patrick Bahr

November 2007

Abstract

Concurrent Haskell is an extension to the pure-functional language Haskell that provides primitives to construct concurrent programs. There is already an operational semantics for this extension as well as a huge number of semantics for the core language in a variety of different styles. The aim of this thesis is to present a coherent dynamic semantics of Concurrent Haskell comprising a wide range of its features including laziness, pattern matching, mutual recursion, imprecise exceptions (synchronous and asynchronous), I/O and of course concurrency. This is done using Meseguer's semantic framework of rewriting logic. Despite the algebraic formulation the style of the semantics is still operational. Moreover the resulting rewrite theory of the semantics of Concurrent Haskell is shown to meet certain requirements that makes it executable by the Maude system, an implementation of rewriting logic. This and the modularity and extensibility of the developed rewrite theory distinguishes it from previous semantic definitions for Haskell in general and makes it furthermore usefull for practical purposes. In addition the choice of the formulation of the semantic rewrite theory is justified by showing its equivalence to several existing semantics each of which only covers a different subset of the language features that are considered here.

Technische Universität Dresden
Fakultät Informatik

Verantwortlicher Hochschullehrer:	Prof. Dr. Christel Baier
Betreuer:	Dr. habil. Michael Posegga

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 27. November 2007

Patrick Bahr

Contents

List of Figures

1 Introduction

Declarative programming languages are of particular interest in theoretical computer science as they somewhat combine on the one hand implementation and on the other hand specification and verification. For the particular case of functional programming — which this thesis is concerned with — this means, a functional program is considered as a specification of the problem which is then transformed into an equivalent, but much more efficient, functional program which then serves as the implementation for solving the problem. The most common example of such an approach is the Bird-Meertens formalism [Bir84].

Unfortunately this pure functional method is too limited to cover practical computational aspects like input/output (I/O), error handling and concurrency. Traditional, i.e. imperative style, approaches can include these features in a rather straightforward manner as the order of evaluation is fixed. For functional programs the order of evaluation can be left unspecified since there are no side effects. However realising I/O and concurrency in functional programming certainly introduces side effects, hence destroying this property of referential transparency. Therefore an expression like `e + f`, where both `e` and `f` can cause and as well be affected by side effects, can surely evaluate to different results depending on the order of evaluation. The same holds true if possible errors that are encountered evaluating `e + f` should be handled. What if both `e` and `f` produce an error? Which one is propergated and eventually handed to the error handling function?

Luckily this can be solved rather gracefully by introducing monads [Mog91, JW93]. Basically this enables to specify an order of evaluation where needed, i.e. for the cases of I/O, concurrency and error handling as discussed above. The functional programming language Haskell is based on this logical foundation using monads and the lazy lambda calculus.

Specifically for this functional programming language a number of dynamic semantics were proposed covering different aspects of the language using different styles of presentation including algebraic [Ben04], denotational [JRH$^+$99] and primarily operational [BR00, MJT04, JGF96, HH92, MJMR01, MLJ99, Lau93] semantics. So the semantics of Haskell is well studied to some extent.

The aim of this thesis is to present a coherent semantical treatment of Haskell including I/O, error handling and most notably concurrency as defined by Concurrent Haskell. It is chosen to do this in an algebraic manner using the framework of rewriting logic (RL) [MR07]. This allows both aspects on the one hand the lazy functional part and on the other hand the concurrency respectively I/O part to be specified in one single semantic framework.

For this purpose the operational semantics as proposed in [MLJ99] for the pure functional part including error propagation and in [MJMR01] for the concurrent part are taken as the basis of this algebraic formulation of the Concurrent Haskell semantics. Hence also the algebraic semantics presented here will be in an operational style. In addition also the semantics of pattern matching and recursion is considered, thus offering a full-scale dynamic semantics for Haskell.

Another reason for advocating RL as the framework for the Haskell semantics is the existence of implementations for it — particularly the Maude [CDE$^+$] system. This allows getting an interpreter for the semantics' object language for free. Moreover by the availability of a large set of further tools for this particular implementation this also offers amongst others model checking and semi-automated inductive proofs inside the semantic theory. Therefore the semantic theory of Concurrent Haskell will be given in a special — executable — form that allows the use of Maude.

Needless to say, this implies that we get a hole set of tools for analysing possible future changes or extensions to the language as well. This in mind the semantic theory of Concurrent Haskell was designed in a modular manner as proposed by [MB03], which enables extensions to the object language to be incorporated into the existing semantic theory as presented here without touching the definitions of the existing language features. An additional set of sentences describing the new language features will suffice.

The presentation in this thesis is as follows: At first basic preliminaries are introduced in section 2 including a brief introduction to RL and the notion of execuability of a RL theory.

The presentation of the semantics of Concurrent Haskell is subdivided into two parts: Section 3 covers the functional part of Haskell. The semantic theory for the given sublanguage of Haskell is introduced, executability and equivalence to the semantics given in [MLJ99] are briefly shown. The concurrent extension is then considered in section 4, also including besides the semantic RL theory proofs for the executability and the equivalence to the semantics given in [MJMR01]. Furthermore some drawbacks of the restriction to an executable RL theory are mentioned and possible solutions ignoring this restriction are outlined.

2 Preliminaries

The purpose of this section is to shortly introduce some preliminaries needed to understand this thesis. Nevertheless, not everything can be mentioned here. In the following the reader is assumed to have basic knowledge of universal algebra and term rewriting. In particular the more general notion of many-sorted algebras will be used. Recent textbooks on that matter are [BN99, Wec92].

2.1 Rewriting Logic

The goal of this thesis is to present the semantics of Concurrent Haskell inside the framework of rewriting logic (RL). As this logic forms the theoretical basis of several specification languages such as ELAN, CafeOBJ and particularly Maude, it is somewhat evolving to include new features of the specification languages. Furthermore it is parametric w.r.t. the underlying equational sublogic. Hence there are several different flavours of rewriting logic, making it necessary to clearly fix the version of rewriting logic used here.

As it is very close to the implementation of the Maude specification language — and of course ultimately it is aspired getting the executable theory developed here running — a form of generalised rewriting logic (GRL) as proposed in [BM03] is used here.

Atomic formulae of rewriting logic are statements of the form $t \longrightarrow t'$ having both a logical and a computational reading. As rewriting logic is used as a semantic framework here, only the latter reading will be of interrest. t and t' are terms of some language defined by the signature of the rewriting theory in question, they represent — in a semantical setting — states of computation. So, $t \longrightarrow t'$ is read as, "During the computation the program state changes from t to t'".

As already mentioned rewriting logic includes an equational sublogic s.t. reasoning is done modulo the equational subtheory. The equational sublogic used here is membership equational logic (MEL) (cf. [Mes97]). It provides besides — of course — equational statements $t = t'$ also membership statements $t : s$, whereas s denotes a sort.

At first MEL will be introduced briefly. Those of the readers who are already familiar with MEL might notice that our definition here is different from the usual one. The variant introduced here is somewhat weaker since kinds are not defined explicitly by the signature but through an order on the sorts.

2.1.1 Syntax and Semantics of Generalised Rewriting Logic

Definition 2.1. A *membership equational signature* (or MEL signature) is a triple $\Omega = (\Sigma, S, <)$, where S is a finite set of elements called *sorts*, $<$ is a strict order on S, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$ is a $K^* \times K$-indexed family of function symbols. $K := {}^{S}/_{\equiv_<}$ is the set of kinds induced by the equivalence closure of $<$, i.e. two sorts are in the same kind if and only if they are in the same connected component of $<$. If s is a sort then $[s]_{\equiv_<}$ or more shortly $[s]$ denotes the corresponding kind of s. The notation S_k for a kind k is used to refer to the set of all sorts in that kind $\{s \in S \mid [s] = k\}$.

Please note that kinds in this context are usually referred to as sorts in the context of many-sorted algebras. In that context Σ would be called a K-sorted signature, a family $X = \{X_k\}_{k\in K}$ a K-sorted set of variables, $T_\Sigma(X)$ the term language over the signature Σ and the variables X and $T_\Sigma(X)_k$ the respective terms of sort k. In this context of MEL this notion of a sort is called kind as the notion of a sort will be used for something similar but different. Hence the term "K-kinded" will be used instead of "K-sorted" respectively "of kind k" instead of "of sort k".

Definition 2.2. Let K be a non-empty finite set of kinds, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$ a $K^* \times K$-indexed family of sets of function symbols, or shortly a K-kinded signature, and $X = \{X_k\}_{k\in K}$ a

$$\frac{t \in T_\Sigma(X)}{\mathcal{E} \vdash (\forall X)\ t = t}\ \text{(Refl)} \qquad \frac{\mathcal{E} \vdash (\forall X)\ t = t'}{\mathcal{E} \vdash (\forall X)\ t' = t}\ \text{(Symm)}$$

$$\frac{\mathcal{E} \vdash (\forall X)\ t = t' \qquad \mathcal{E} \vdash (\forall X)\ t' = t''}{\mathcal{E} \vdash (\forall X)\ t = t''}\ \text{(Trans)}$$

$$\frac{f \in \Sigma_{k_1 \cdots k_n, k} \qquad t_i, t_i' \in T_\Sigma(X)_{k_i} \qquad 0 \leq i \leq n \in \mathbb{N} \\ \mathcal{E} \vdash (\forall X)\ t_i = t_i'}{\mathcal{E} \vdash (\forall X)\ f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}\ \text{(Cong)}$$

$$\frac{\mathcal{E} \vdash (\forall X)\ t = t' \qquad \mathcal{E} \vdash (\forall X)\ t : s}{\mathcal{E} \vdash (\forall X)\ t' : s}\ \text{(Member Eq)} \qquad \frac{s < s' \qquad \mathcal{E} \vdash (\forall X)\ t : s}{\mathcal{E} \vdash (\forall X)\ t : s'}\ \text{(Member Sub)}$$

$$\frac{(\forall X)\ t = t' \Leftarrow \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j \in E \qquad \theta : X \to T_\Sigma(Y) \\ \mathcal{E} \vdash (\forall Y)\ \overline{\theta}(u_i) = \overline{\theta}(v_i) \quad 1 \leq i \leq n \qquad \mathcal{E} \vdash (\forall Y)\ \overline{\theta}(w_j) = s_j \quad 1 \leq j \leq m}{\mathcal{E} \vdash (\forall Y)\ \overline{\theta}(t) = \overline{\theta}(t')}\ \text{(MP Eq)}$$

$$\frac{(\forall X)\ t'' : s \Leftarrow \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j \in E \qquad \theta : X \to T_\Sigma(Y) \\ \mathcal{E} \vdash (\forall Y)\ \overline{\theta}(u_i) = \overline{\theta}(v_i) \quad 1 \leq i \leq n \qquad \mathcal{E} \vdash (\forall Y)\ \overline{\theta}(w_j) : s_j \quad 1 \leq j \leq m}{\mathcal{E} \vdash (\forall Y)\ \overline{\theta}(t) : s}\ \text{(MP Mb)}$$

Fig. 2.1: Deduction rules for membership equational theories.

K-indexed family of sets of variables, or shortly a K-kinded set of variables. Then for every kind $k \in K$ the set of terms over Σ and X of kind k, $T_\Sigma(X)_k$, is inductively defined as follows:

1. $X_k \subseteq T_\Sigma(X)_k$,
2. if $\sigma \in \Sigma_{k_1 \ldots k_n, k}$ for some $n \geq 0$, $k_1, \ldots, k_n, k \in K$ and $t_i \in T_\Sigma(X)_{k_i}$ for every $1 \leq i \leq k$ then $\sigma(t_1, \ldots, t_n) \in T_\Sigma(X)_k$.

Then $T_\Sigma(X) = \{T_\Sigma(X)_k\}_{k \in K}$ will denote the K-indexed family of sets of terms over Σ and X, or shortly the K-kinded set of terms over Σ and X.

Definition 2.3. Let $\Omega = (\Sigma, S, <)$ be a MEL signature, K the corresponding set of kinds, X a K-kinded set of variables, I, J some finite index sets, $t, t' \in T_\Sigma(X)_k$ for some $k \in K$, $u_i, v_i \in T_\Sigma(X)_{k_i}$ for $i \in I$ and some $k_i \in K$, $w_j \in T_\Sigma(X)_{[s_j]}, s_j \in S$ for $j \in J$ and $t'' \in T_\Sigma(X)_{[s]}$. Then the following are *membership equational sentences* (or MEL sentences):

$$(\forall X)\ t = t' \Leftarrow \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j \qquad \text{(Equation)}$$

$$(\forall X)\ t'' : s \Leftarrow \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j \qquad \text{(Membership)}$$

Definition 2.4. A *membership equational theory* (or MEL theory) $\mathcal{E}$ is a pair (Ω, E) where Ω is a MEL signature and E a is set of MEL sentences.

Now that the constitutes of a MEL theory are defined, it can be defined as well which equalities are entailed by a particular theory $\mathcal{E}$. For this purpose the operational semantics of MEL is given as a deduction calculus in figure 2.1. Note that by $\theta : X \to T_\Sigma(Y)$ a K-kinded assignment is meant, i.e. $\theta = \{\theta_k : X_k \to T_\Sigma(Y)_k\}_{k \in K}$.

With MEL syntax and semantics defined, we can approach GRL.

$$\frac{t \in T_\Sigma(X)}{\mathcal{R} \vdash (\forall X)\ t \longrightarrow t} \text{ (Refl)} \qquad \frac{\mathcal{R} \vdash (\forall X)\ t \longrightarrow t' \quad \mathcal{R} \vdash (\forall X)\ t' \longrightarrow t''}{\mathcal{R} \vdash (\forall X)\ t \longrightarrow t''} \text{ (Trans)}$$

$$\frac{\mathcal{E} \vdash (\forall X)\ t = u \quad \mathcal{R} \vdash (\forall X)\ u \longrightarrow u' \quad \mathcal{E} \vdash (\forall X)\ u' = t'}{\mathcal{R} \vdash (\forall X)\ t \longrightarrow t'} \text{ (Eq)}$$

$$\frac{f \in \Sigma_{k_1 \cdots k_n, k} \quad t_i, t'_i \in T_\Sigma(X)_{k_i} \quad \mathcal{R} \vdash (\forall X)\ t_i \longrightarrow t'_i}{\mathcal{R} \vdash (\forall X)\ f(t_1, \ldots, t_n) \longrightarrow f(t'_1, \ldots, t'_n)} \text{ (Cong)}$$

$$\frac{\begin{array}{c}(\forall X)\ r : t \longrightarrow t' \Leftarrow \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \longrightarrow t'_l \in R \\ \theta, \theta' : X \to T_\Sigma(Y) \\ \mathcal{E} \vdash (\forall Y)\ \overline{\theta}(u_i) = \overline{\theta}(v_i) \quad i \in I \qquad \mathcal{E} \vdash (\forall Y)\ \overline{\theta}(w_j) : s_j \quad j \in J \\ \mathcal{R} \vdash (\forall Y)\ \overline{\theta}(t_l) \longrightarrow \overline{\theta}(t'_l) \quad l \in L \qquad \mathcal{R} \vdash (\forall Y)\ \overline{\theta}(x) \longrightarrow \overline{\theta'}(x) \quad x \in X\end{array}}{\mathcal{R} \vdash (\forall Y)\ \overline{\theta}(t) \longrightarrow \overline{\theta'}(t')} \text{ (Nested Repl)}$$

Fig. 2.2: Deduction rules for generalised rewrite theories.

Definition 2.5. Let $\Omega = (\Sigma, S, <)$ be a MEL signature, K the corresponding set of kinds X a K-kinded set of variables, I, J, L some finite index sets, $t, t' \in T_\Sigma(X)_k$ for some $k \in K$, $u_i, v_i \in T_\Sigma(X)_{k_i}$ for $i \in I$ and some $k_i \in K$, $w_j \in T_\Sigma(X)_{[s_j]}$, $s_j \in S$ for $j \in J$ and $t_l, t'_l \in T_\Sigma(X)_{k'_l}$ for $l \in L$ and some $k'_l \in K$. Then the following is a *generalised rewrite sentence* (or *GRL sentence*):

$$(\forall X)\ t \longrightarrow t' \Leftarrow \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \longrightarrow t'_l \qquad \text{(Rewrite)}$$

Definition 2.6. A *generalised rewrite theory* (or *GRT*) is a triple $\mathcal{R} = (\Omega, E, R)$ where $\mathcal{E} = (\Omega, E)$ is a MEL theory and R is a set of GRT sentences of signature Ω .

The operational semantics of a GRT $\mathcal{R}$ is given in figure 2.2.

2.1.2 Notation

Now that we have rigorously defined syntax and semantics of generalised rewriting logic we can stipulate some notational conventions. Readers already familiar with Maude will find the notational abbreviations introduced in the following similarly familiar.

Firstly, the strict order on the sorts of a theory is usually given as a collection of statements of the form $s_0 < s_1 < \ldots < s_n$ with the intended meaning that $<$ is the smallest strict order that agrees upon those statements.

The family Σ of function symbols is given as a collection of statements of the form $f : k_1 \cdots k_n \to k$ or $c : k$ for some kinds $k, k_1, \ldots, k_n$ and $n > 0$ with the intended meaning $f \in \Sigma_{k_1 \cdots k_n, k}$ or $c \in \Sigma_{\varepsilon, k}$ respectively. Statements of this form will also be referred to as declarations. Furthermore such declarations of operator symbols will also be stated at sort level, i.e. $f : s_1 \cdots s_n \to s$ for $n > 0$ or $c : s$ for some sorts $s, s_1, \ldots, s_n$. The intended meaning of this is $f \in \Sigma_{[s_1] \cdots [s_n], [s]}$ or $c \in \Sigma_{\varepsilon, [s]}$ respectively and an additional membership sentence $(\forall \{x_i : k_i | 1 \le i \le n\})\ f(x_1, \ldots, x_n) : s \Leftarrow \bigwedge_{1 \le i \le n} x_i : s_i$ or $c : s$ respectively.

Apart from ordinary prefix operator symbols we are also interrested in mixfix operators that will become essential when defining the syntax of an object language so that words of that language become terms of MEL. A dot "$\cdot$" will be used to mark the places where an argument of a mixfix operator is expected, e.g.

$$(\backslash \cdot . \cdot) : \textbf{Var Term} \to \textbf{LambdaAbstraction}$$

could be a definition for a mixfix function symbol for the lambda abstraction.

We also want to get rid of the variable quantification and assume implicit quantification of all variables of a sentence. To do so we have to fix the kind (or the sort) of the variables used in the sentences beforehand. This is done by giving variable names in braces when introducing the sorts of a rewrite theory, e.g. $\mathbf{Exp}\{e_i\} \in S$ states that the variables e_i range over the sort **Exp** and the variables $[e]_i$ range over the kind $[\mathbf{Exp}]$.

The intended meaning of a variable ranging over a sort is the same as the variable ranging over the corresponding kind plus a membership condition stating the membership of that variable in the sort. In particular if a variable v ranging over the sort s is used in a sentence it can be replaced by $[v]$ and an additional condition $[v] : \mathsf{s}$ in that sentence.

There is another shorthand that is used by Maude that can be quite useful: The `owise` attribute. It will be used here as an additional condition "otherwise" — like atomic equational, membership and rewrite formulae. Informally the otherwise condition is fullfilled if no other sentence of the theory *not* having the otherwise condition is applicable, i.e. neither by its head nor its conditions, if any. So the sentences of a rewriting theory are — informally speaking — segregated into "ordinary" sentences and otherwise sentences where the first have their usual semantics whereas the latter are only considered if none of the first is applicable. But actually the semantics of rewriting logic does not have to be touched at all, merely a translation of the theory is necessary that introduces a predicate symbol $\mathsf{enabled}(\,\cdot\,) : [\mathsf{s}] \rightarrow [\mathbf{Bool}]$ defined to be equal to true if one of the non-otherwise sentences can be applied to its argument term. Then the otherwise condition can be replaced by $(\mathsf{enabled}(t) \neq \mathsf{true}) = \mathsf{true}$ [1] where t is the sentence's lefthand term. A more detailed discussion on this is given in [CDE$^+$, section 4.5.5].

In order to make the rewriting logic sentences more readable it is also convenient to use an inference style presentation:

$$\frac{u_i = v_i \qquad w_j : s_j}{t = t'} \tag{Equation'}$$

$$\frac{u_i = v_i \qquad w_j : s_j}{t : s} \tag{Membership'}$$

$$\frac{u_i = v_i \qquad w_j : s_j \qquad t_l \longrightarrow t'_l}{t \longrightarrow t'} \tag{Rewrite'}$$

In some cases of rather complex sentences the following presentation is chosen for rewrite sentences:

$$\frac{u_i = v_i \qquad w_j : s_j \qquad t_l \longrightarrow t'_l}{\begin{array}{c} t \\ \xrightarrow{\hspace{6em}} \\ t' \end{array}} \tag{Rewrite''}$$

2.2 Executable Theories

A considerable advantage of the rewriting logic over other semantic frameworks is the availability of an implementation that offers the possibility to "execute" a theory. In the case of a theory describing a semantics this automatically yields an interpreter for the object language.

Unfortunately not every arbitrary theory can be executed by those implementations. The purpose of this section is to discuss requirements theories have to meet to be executable. This is done here for the particular case of the Maude implementation [CDE$^+$].

As already mentioned reasoning in rewriting logic is done modulo the equational subtheory $\mathcal{E}$. One approach is to use a representative of each equivalence class in ${}^{T_\Sigma(X)}\!/\!_{\mathcal{E}}$. Those representatives should be unique, of course. Such a unique representative for a term $t \in {}^{T_\Sigma(X)}\!/\!_{\mathcal{E}}$ can be found by rewriting w.r.t. $\mathcal{E}$ — that is using the equations in $\mathcal{E}$ in an oriented form — provided that the corresponding rewriting system $\rightarrow_{\mathcal{E}}$ is confluent and terminating. Of course, as we are only interrested in its initial algebra semantics, that is ground terms (i.e. in T_Σ), $\rightarrow_{\mathcal{E}}$ is only demanded to be ground confluent and ground terminating.

[1] Of course also an appropriate predicate $\cdot \neq \cdot : \mathbf{s}\ \mathbf{s} \rightarrow \mathbf{Bool}$ needs to be equationally defined. By the result in [BT80] this can be done, even in a way that complies with the notion of executability introduced in the upcoming section.

$$\begin{tikzcd}
t \arrow[r, "R"] \arrow[d, equal, "E"'] & t' \arrow[d, equal, "E"] \\
s \arrow[r, dashed, "R"] & s'
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
t \arrow[r, "R"] \arrow[dd, "E!"'] & t' \arrow[dr, dashed, "E!"] & \\
 & & u \\
s \arrow[r, dashed, "R"] & s' \arrow[ur, dashed, "E!"'] &
\end{tikzcd}$$

(b)

Fig. 2.3: Commutative diagrams showing the desired property for rewrite theories (a) and the sufficient property of coherence (b).

However these restrictions are quite demanding, if one considers equational sentences for associativity and commutativity of binary operators — fairly common and essential properties — which trivially destroy the termination property immediately. Fortunately there is a complete rewriting algorithm (cf. [HM07]) for rewriting systems that are, excluding their rules for associativity and commutativity, terminating and confluent. So in principal equational sentences establishing associativity and commutativity do not have to be considered when reasoning about termination of the equational theory.

Furthermore it is also needed that every term has a least sort w.r.t. the sort order $<$. A sufficient requirement for this property is the preregularity property. A MEL theory $\mathcal{E} = (\Omega, E)$ is preregular if for every operator symbol $f \in \Sigma_{k_1 \cdots k_n, k}$ for some kinds $k_1, \ldots, k_n, k \in K$, there is for every $(s_1, \ldots, s_n) \in S_{k_1} \times \cdots \times S_{k_n}$ a smallest $s \in S_k$ amongst all of those for which $\mathcal{E}' \vdash f(c_1, \ldots, c_n) : s$ is derivable. Where $\mathcal{E}' = (\Omega \cup (\{c_i\}_{k_i}), E' \cup \{c_i : s_i\})$ for E' the subset of E containing only membership sentences and those equational sentences establishing associativity or commutativity, i.e. the theory $\mathcal{E}'$ with additional fresh constants c_i of the sort s_i respectively. Or to put it more compact: Every operator has to have a smallest codomain sort. Note that the definition of E' considers only equational sentences defining associativity or commutativity. That is because as already mentioned such sentences are treated differently to maintain termination.

Moreover the rewriting system induced by a MEL theory has to be ground sort-decreasing. By preregularity every term t has among its sorts, if any, a smallest one $\mathrm{LS}(t)$. The rewriting system induced by a MEL theory $\mathcal{E} = (\Omega, E)$ is called sort-decreasing if $t \rightarrow_{\mathcal{E}} t'$ and t has a sort implies that $\mathrm{LS}(t) \geq \mathrm{LS}(t')$ where $\geq$ is the reflexive closure of the order $>$ on sorts.

So far only restrictions for the equational subtheory of a rewrite theory were mentioned, but also the set of rewrite sentences has to meet some requirements depending on its equational subtheory. When trying to proof an atomic sentence $t \longrightarrow t'$ a breadth first search strategy in the rewriting system $\rightarrow_{\mathcal{R}}$ induced by the rewrite rules (similar to $\rightarrow_{\mathcal{E}}$) is clearly complete provided that $\rightarrow_{\mathcal{R}}$ is terminating. But since reasoning in rewriting logic is done modulo the equational subtheory (see deduction rule Eq in figure 2.2), is must be guaranteed that rewriting on the canonical normal forms of the equational theory is still complete. Or to be more precise: The diagram in figure 2.3(a) has to commute.

This property can be assured if the rewrite theory is coherent. The commutative diagram in figure 2.3(b) depicts the requirements for a rewrite theory to be coherent. The "!" on the arrows is supposed to be understood as "until a normal form is reached". Clearly commutativity of the second diagram implies commutativity of the first one provided the equational rewriting system is confluent and terminating. Of course this property is also only necessary for ground terms.

3 Functional Semantics

3.1 Syntax of Pure Haskell

The goal of this section is to formulate a MEL theory that describes the functional semantics of Concurrent Haskell. In the context of this thesis this is meant to be understood as the non-concurrent semantics, that is, the pure Haskell semantics without the concurrent extension.

For the sake of conciseness of the presentation of the semantics it is preferable to concentrate on a sufficiently expressive sublanguage of Haskell. Thus it is necessary to define this sublanguage. This should be done here in form of a grammar in Backus-Naur form (bnf).

Figure 3.1 describes the basic elements of the syntax of Haskell. The basic literals, described by the nonterminals *Float*, *Int* and *Char*, as well as the identifiers for variables and data constructors, described by the nonterminals *Var* and *CustCtor* respectively, are only defined informal. For details on how those can be defined formally please refer to [Jon03].

$$\begin{array}{rcl}
\mathit{Float} & ::= & <\textit{floating point literals}> \\
\mathit{Int} & ::= & <\textit{integer point literals}> \\
\mathit{Char} & ::= & <\textit{character literals}> \\
\mathit{String} & ::= & \varepsilon \mid \mathit{Char}\ \mathit{String} \\
\mathit{Var} & ::= & <\textit{variables}> \\
\mathit{BinOp} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>} \\
 & & \mid \texttt{/=} \mid \texttt{==} \mid \texttt{!!} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{**} \mid \texttt{++} \mid \texttt{\$} \mid \texttt{\$!} \\
\mathit{BoolConst} & ::= & \texttt{True} \mid \texttt{False} \\
\mathit{PDCtor} & ::= & \mathit{BoolConst} \mid \mathit{Float} \mid \mathit{Int} \mid \texttt{'}\ \mathit{Char}\ \texttt{'} \mid \texttt{"}\ \mathit{String}\ \texttt{"} \mid \texttt{()} \mid \texttt{[]} \\
\mathit{CustCtor} & ::= & <\textit{data constructor}> \\
\mathit{Ctor} & ::= & \mathit{CustCtor} \mid \mathit{PDCtor} \\
\mathit{Primitive} & ::= & \texttt{seq} \mid \texttt{not} \mid \texttt{raise} \mid \texttt{(}\ \mathit{BinOp}\ \texttt{)} \\
\mathit{AtExp} & ::= & \mathit{Ctor} \mid \mathit{Var} \mid \mathit{Primitive}
\end{array}$$

Fig. 3.1: Basic syntactic elements of Haskell.

Please note that in the following syntactic variables are confused with their respective syntactic category. Thus, by writing *BinOp* outside a bnf definition the set $\{\texttt{+}, \texttt{-}, \texttt{*}, \ldots\}$ is meant for example.

Now that the basic syntactic elements are defined, Haskell expressions can be defined. For the sublanguage of Haskell that is investigated here, this is done in figure 3.2. Note that `let` expression are not included, but as a matter of fact their most common usage for non-recursive pattern binding can be simulated by a `case` expression whereas recursive bindings can be simulated by function bindings that will be introduced later. Furthermore patterns are defined that will be used for pattern matching in lambda abstractions, `case` expressions and function bindings. Also note that lambda expressions are defined here — inconsistently with [Jon03] — as having its argument patterns separated by commas. This is done to make the semantic definitions a bit easier.

Finally figure 3.3 on the next page gives the syntax of a program in our Haskell sublanguage. Note that Haskell programs in this sense only contain function bindings, and therefore particularly no type signatures or type declarations. As this thesis is only concerned about the dynamic semantics, type information typically given in a Haskell program is completely ignored here. Whereas

$$\begin{array}{rcl}
\mathit{Exp} & ::= & \mathit{AtExp} \\
 & & \mid \mathit{Exp}\ \mathit{Exp} \\
 & & \mid\ \texttt{\textbackslash}\,\mathit{NePatList}\,\texttt{->}\,\mathit{Exp} \\
 & & \mid \texttt{case}\,\mathit{Exp}\,\texttt{of}\ \texttt{\{}\,\mathit{Cases}\,\texttt{\}} \\
 & & \mid \texttt{if}\,\mathit{Exp}\,\texttt{then}\,\mathit{Exp}\,\texttt{else}\,\mathit{Exp} \\
 & & \mid \mathit{Exp} : \mathit{Exp} \\
 & & \mid \mathit{Exp}\ \mathit{BinOp}\ \mathit{Exp} \\
 & & \mid \texttt{[}\,\mathit{ExpList}\,\texttt{]} \\
 & & \mid \texttt{(}\,\mathit{ExpList}\,\texttt{)} \\
\mathit{AtPat} & ::= & \mathit{Var} \mid \texttt{\_} \mid \mathit{Ctor} \\
\mathit{Pat} & ::= & \mathit{AtPat} \\
 & & \mid \mathit{Pat}\ \mathit{Pat} \\
 & & \mid \mathit{Pat} : \mathit{Pat} \\
 & & \mid \texttt{[}\,\mathit{PatList}\,\texttt{]} \\
 & & \mid \texttt{(}\,\mathit{PatList}\,\texttt{)} \\
\mathit{ExpList} & ::= & \mathit{Exp} \mid \mathit{ExpList}\ \texttt{,}\ \mathit{Exp} \\
\mathit{PatList} & ::= & \mathit{Pat} \mid \mathit{PatList}\ \texttt{,}\ \mathit{Pat} \\
\mathit{Case} & ::= & \mathit{Pat}\,\texttt{->}\,\mathit{Exp} \\
\mathit{Cases} & ::= & \mathit{Case} \mid \mathit{Cases}\ \texttt{;}\ \mathit{Case}
\end{array}$$

Fig. 3.2: Expression syntax of Haskell.

the Haskell syntax as described in [Jon03] allows omitting type signatures as well, for they can be inferred, this is — syntactically — of no further impact, type declarations are crucial for the static semantics, as they define amongst others the type signature of each data constructor.

Therefore the dynamic semantics that will be given here as a rewrite theory is only meaningfull for a subset of those programs defined by the preceding syntax definitions that, enriched with the "right" type declarations, fullfills certain context sensitive properties as typability and closedness, i.e. that every occurrence of a variable is a bound one. A static semantics that addresses these concerns very thoroughly can be found in [JW92].

Another context sensitive property, besides typing related ones, generously ignored by these definitions for the sake of a concise presentation is the repeated occurrence of variables in a pattern which is not allowed.

$$\begin{array}{rcl}
\mathit{FuncBindLhs} & ::= & \mathit{Var} \mid \mathit{FuncBindLhs}\ \mathit{Pat} \\
\mathit{FuncBind} & ::= & \mathit{FuncBindLhs}\,\texttt{=}\,\mathit{Exp} \\
\mathit{Program} & ::= & \mathit{FuncBind} \mid \mathit{Program}\ \texttt{;}\ \mathit{FuncBind}
\end{array}$$

Fig. 3.3: Program syntax of Haskell.

3.2 Formulating the Syntax as a MEL Theory

Now the Haskell syntax given in the previous section has to be translated into an appropriate MEL signature and a set of membership and equational sentences such that Haskell programs are represented as terms of MEL. But note that we will stick to the context-free characterisation that was introduced in the previous section using bnf. Hence only the subset of terms that corresponds to a typable program resp. expression of Haskell will be considered.

For our purposes it is assumed that there are already defined the sorts $\mathbf{FloatConst}\{fc_i\}$, $\mathbf{IntConst}\{ic_i\}$, $\mathbf{CharConst}\{cc_i\}$, $\mathbf{StringConst}\{sc_i\}$, $\mathbf{Var}\{v_i\}$ and **CustCtor** having the obvious respective meaning. Whereas the sorts **FloatConst**, **IntConst**, **CharConst** and **StringConst** mentioned above are supposed to be literals only, i.e. they are pure syntax not having any operation defined on them, there is a need for their "semantic" counterparts. Hence the sorts $\mathbf{Float}\{f_i\}$, $\mathbf{Int}\{i_i\}$, $\mathbf{Char}\{c_i\}$ and $\mathbf{String}\{str_i\}$ are assumed to be properly defined along with operators to switch between the pure literal sorts and these semantic sorts, that is, for example $\mathsf{fromFloat} : \mathbf{Float} \rightarrow \mathbf{FloatConst}$ and $\mathsf{toFloat} : \mathbf{FloatConst} \rightarrow \mathbf{Float}$. It is also assumed that there is an appropriate set of operators defined on these semantic sorts to calculate with $(+, *, \ldots)$, manipulate $(\mathsf{substring}, \mathsf{concat}, \ldots)$ and compare $(<, \leq, \ldots)$ elements of these sorts. For truth values that might be the result of some of these operators a sort $\mathbf{Bool}\{b_i\}$ is assumed, also with an appropriate set of operators defined on it. Later such operators will become necessary to define the semantics of those elements.

Having these different sorts, there is a clear distinction between the syntactic elements and their intended meaning. But these semantic sorts are not to be confused with a possible (flat) domain of some kind. They are merely a tool to define built-in functions of Haskell.

With this defined only few basic syntactic elements are left to be translated into MEL syntax. In the following the sorts $\mathbf{BoolConst}\{bc_i\}$, **PDCtor**, $\mathbf{Ctor}\{ct_i\}$, **Primitive** and **AtExp** are defined.

Some of the syntax is only a sublanguage relation, e.g. all literals are also pre-defined constructors, pre-defined constructors are constructors etc. This can be expressed as a subsort relation:

$$\begin{aligned}
\mathbf{BoolConst}, \mathbf{FloatConst}, \mathbf{IntConst}, \mathbf{CharConst}, \mathbf{StringConst} &< \mathbf{PDCtor}\\
\mathbf{CustCtor}, \mathbf{PDCtor} &< \mathbf{Ctor}\\
\mathbf{Ctor}, \mathbf{Var}, \mathbf{Primitive} &< \mathbf{AtExp}\\
\mathbf{CtorVar} &< \mathbf{AtPat}
\end{aligned}$$

The rest can be expressed as operator declarations:

$$\begin{aligned}
b &: \mathbf{BoolConst} && (b \in \mathit{BoolConst})\\
\texttt{()}\,,\ \texttt{[]} &: \mathbf{PDCtor}\\
\texttt{seq}\,,\ \texttt{not}\,,\ \texttt{raise} &: \mathbf{Primitive}\\
\texttt{(}\ \circ\ \texttt{)} &: \mathbf{Primitive} && (\circ \in \mathit{BinOp})\\
\texttt{\_} &: \mathbf{AtPat}
\end{aligned}$$

But note that — since the above operator declarations are at sort level — this implicitly includes, as described in section 2.1.2, a set of membership sentences.

Now these newly defined sorts can be used to define the sorts $\mathbf{Exp}\{e_i\}$ and $\mathbf{Pat}\{pt_i\}$ for Haskell expressions and patterns respectively. But beforehand we need some auxiliary sorts like $\mathbf{Case}\{ca_i\}$ and $\mathbf{Cases}\{cas_i\}$ for `case` expressions:

$$\mathbf{Case} < \mathbf{Cases}$$

$$\begin{aligned}
\cdot\ \texttt{->}\ \cdot &: \mathbf{Pat}\ \ \mathbf{Exp} \rightarrow \mathbf{Case}\\
\cdot\ \texttt{;}\ \cdot &: \mathbf{Cases}\ \ \mathbf{Cases} \rightarrow \mathbf{Cases}
\end{aligned}$$

Also the sorts $\mathbf{ExpList}\{l_i\}$ and $\mathbf{PatList}\{pl_i\}$ for (possibly empty) lists of expressions and patters respectively as well as their non-empty correspondents $\mathbf{NeExpList}\{nl_i\}$ and $\mathbf{NePatList}\{npl_i\}$ are needed

$$\mathbf{Pat} < \mathbf{NePatList} < \mathbf{PatList}$$

$$\begin{aligned}
\mathsf{nil} &: \textbf{PatList}\\
\cdot\ \texttt{,}\ \cdot &: \textbf{PatList}\ \ \textbf{PatList}\ \rightarrow\ \textbf{PatList}\\
\cdot\ \texttt{,}\ \cdot &: \textbf{PatList}\ \ \textbf{NePatList}\ \rightarrow\ \textbf{NePatList}\\
\cdot\ \texttt{,}\ \cdot &: \textbf{NePatList}\ \ \textbf{PatList}\ \rightarrow\ \textbf{NePatList}
\end{aligned}$$

, and similar for **ExpList** plus the following obvious relation between them:

$$\textbf{PatList} < \textbf{ExpList}$$
$$\textbf{NePatList} < \textbf{NeExpList}$$

It is not necessary to have empty lists of expressions to define the Haskell syntax but the adjoined empty list nil will help to formulate some of the sentences for the semantics quite concisely.

Additionally we have to give some equational sentences establishing the operator $\cdot$, $\cdot$ as associative and having nil as its identity:

$$\frac{}{[l]_1\ \texttt{,}\ [l]_2 = [l]_2\ \texttt{,}\ [l]_1}\ \textbf{assoc} \qquad \frac{}{[l]_1\ \texttt{,}\ \mathsf{nil} = [l]_1}\ \textbf{id1} \qquad \frac{}{\mathsf{nil}\ \texttt{,}\ [l]_1 = [l]_1}\ \textbf{id2}$$

Please remember that the variables $[l]_1$, $[l]_2$ are of kind [**ExpList**] which contains amongst others the sorts **ExpList**, **NeExpList**, **PatList** and **NePatList** due to the subsort order $<$.

Now all necessary preparations are made to define the sorts **Exp** and **Pat**, this is done in figure 3.4.

$$\begin{aligned}
\textbf{AtExp} &< \textbf{Exp}\\
\textbf{AtPat} &< \textbf{Pat}\\
\textbf{Pat} &< \textbf{Exp}
\end{aligned}$$

Operator declarations for expressions:

$$\begin{aligned}
\cdot\ \ \cdot &: \textbf{Exp}\ \ \textbf{Exp}\ \rightarrow\ \textbf{Exp}\\
\texttt{\textbackslash}\ \cdot\ \texttt{->}\ \cdot &: \textbf{NePatList}\ \ \textbf{Exp}\ \rightarrow\ \textbf{Exp}\\
\texttt{case}\ \cdot\ \texttt{of}\ \cdot &: \textbf{Exp}\ \ \textbf{Cases}\ \rightarrow\ \textbf{Exp}\\
\texttt{if}\ \cdot\ \texttt{then}\ \cdot\ \texttt{else}\ \cdot &: \textbf{Exp}\ \ \textbf{Exp}\ \ \textbf{Exp}\ \rightarrow\ \textbf{Exp}\\
\cdot\ \texttt{:}\ \cdot &: \textbf{Exp}\ \ \textbf{Exp}\ \rightarrow\ \textbf{Exp}\\
\cdot\ \circ\ \cdot &: \textbf{Exp}\ \ \textbf{Exp}\ \rightarrow\ \textbf{Exp} \qquad\qquad (\circ \in \mathit{BinOp})\\
\texttt{[}\ \cdot\ \texttt{]} &: \textbf{ExpList}\ \rightarrow\ \textbf{Exp}\\
\texttt{(}\ \cdot\ \texttt{)} &: \textbf{ExpList}\ \rightarrow\ \textbf{Exp}
\end{aligned}$$

Operator declarations for patterns:

$$\begin{aligned}
\texttt{\_} &: \textbf{Pat}\\
\cdot\ \ \cdot &: \textbf{Pat}\ \ \textbf{Pat}\ \rightarrow\ \textbf{Pat}\\
\cdot\ \texttt{:}\ \cdot &: \textbf{Pat}\ \ \textbf{Pat}\ \rightarrow\ \textbf{Pat}\\
\texttt{[}\ \cdot\ \texttt{]} &: \textbf{PatList}\ \rightarrow\ \textbf{Pat}\\
\texttt{(}\ \cdot\ \texttt{)} &: \textbf{PatList}\ \rightarrow\ \textbf{Pat}
\end{aligned}$$

Fig. 3.4: Operator and subsort declarations for patterns and expressions.

String literals are only treated as syntactic sugar for the corresponding list of characters. Similarly list expressions of the form `[` e_1 `,` ... `,` e_n `]` are just a shorthand for e_1 `:` ... `:` e_n `:` `[]` . Here are the corresponding equational sentences describing this:

$$\frac{\mathsf{length}(\mathsf{toString}(sc_1)) = 0}{sc_1 = \texttt{[]}}\ \textbf{str1}$$

$$\frac{str_1 = \mathsf{toString}(sc_1) \qquad cc_1 = \mathsf{fromChar}(\mathsf{head}(str_1)) \qquad sc_2 = \mathsf{fromString}(\mathsf{tail}(str_1))}{sc_1 = cc_1 \texttt{ : } sc_2}\ \textbf{str2}$$

$$\frac{}{\texttt{[}\,\mathsf{nil}\,\texttt{]} = \texttt{[]}}\ \textbf{list1} \qquad \frac{}{\texttt{[}\,e_1\,\texttt{,}\,l_1\,\texttt{]} = e_1 \texttt{ : [}\,l_1\,\texttt{]}}\ \textbf{list2}$$

where $\mathsf{head}(\cdot) : [\textbf{String}] \rightarrow [\textbf{Char}]$ is supposed to evaluate to the first character of the argument string if it is nonempty and $\mathsf{tail}(\cdot) : [\textbf{String}] \rightarrow [\textbf{String}]$ to the rest string of the argument string if it is nonempty. Note that sentence **str2** does not need the precondition $\mathsf{length}(\mathsf{toString}(sc_1)) = 0$ as this is implicitly stated by $cc_1 = \mathsf{fromChar}(\mathsf{head}(str_1))$. Remember that cc_1 ranges over sort **CharConst**. Therefore implicitly the precondition cc_1 : **CharConst** is added to this sentence such that $cc_1 = \mathsf{fromChar}(\mathsf{head}(str_1))$ can only be fullfilled if $\mathsf{fromChar}(\mathsf{head}(str_1))$: **CharConst** witch is only true for non-empty str_1.

Defining the sort **Program**$\{p_i\}$ describing Haskell programs is easy now after having defined **FuncBindLhs**$\{fl_i\}$ — left-hand-sides of function bindings — and **FuncBind**$\{fb_i\}$ — function bindings:

$$\begin{aligned} \textbf{FuncBind} &< \textbf{Program} \\ \textbf{Var} &< \textbf{FuncBindLhs} \end{aligned}$$

$$\begin{aligned} \cdot\ \cdot &: \textbf{FuncBindLhs}\ \ \textbf{Pat} \rightarrow \textbf{FuncBindLhs} \\ \cdot \texttt{ = } \cdot &: \textbf{FuncBindLhs}\ \ \textbf{Exp} \rightarrow \textbf{FuncBind} \\ \cdot \texttt{ ; } \cdot &: \textbf{Program}\ \ \textbf{Program} \rightarrow \textbf{Program} \end{aligned}$$

3.3 Semantic Expressions

Before being able to describe the semantics of the sublanguage of Haskell defined in the previous section, the notion of an erroneous computation is needed to be defined. In Haskell this is the *exception*. In principle an exception is only an expression of the type `Exception`. Since static semantics is beyond the scope of this treatment this type has to be explicitly captured by an appropriate syntax definition. For this purpose **ExcCtor** — exception constructors — a subsort of **Ctor** along with **ExcExp**$\{ece_i\}$ — exception expressions — a subsort of **Exp** have to be introduced to describe precisely expressions of that type[1]. Furthermore a sort **ExcPat** of exception patterns that includes patterns of type `Exception` is needed to keep the theory preregular:

$$\begin{aligned} \textbf{ExcCtor} &< \textbf{PDCtor} \\ \textbf{ExcCtor} &< \textbf{ExcExp} < \textbf{Exp} \\ \textbf{ExcPat} &< \textbf{ExcExp}, \textbf{Pat} \end{aligned}$$

$$\begin{aligned} e &: \textbf{ExcCtor} && (e \in \{ArithException, ArrayException, \ldots\}) \\ \cdot\ \cdot &: \textbf{ExcExp}\ \ \textbf{Exp} \rightarrow \textbf{ExcExp} \\ \cdot\ \cdot &: \textbf{ExcExp}\ \ \textbf{Pat} \rightarrow \textbf{ExcPat} \end{aligned}$$

[1]If the expression is well typed in the sense given in section 3.1.

Additionally "real" exceptions are introduced using the sort **Exception**$\{ec_1\}$ and an constructor symbol $\langle\,\cdot\,\rangle_{\lightning}$ along with a operation to retrieve the underlying exception expression:

$$\begin{aligned} \langle\,\cdot\,\rangle_{\lightning} &: \textbf{ExcExp} \rightarrow \textbf{Exception} \\ \mathsf{exp}(\,\cdot\,) &: \textbf{Exception} \rightarrow \textbf{ExcExp} \end{aligned}$$

$$\frac{}{\mathsf{exp}(\langle ece_1\rangle_{\lightning}) = ece_1}\ \textbf{exp}$$

So exception handling inside the Haskell program is of course done with terms of the sort **ExcExp** whereas the exception handling in the semantic definitions is done on the level of the sort **Exception**. In that way there is a clear distinction between "syntactic" and "semantic" exception. Nevertheless the rôle of exceptions in this sense is different from the one of sorts like **Float**, **Int** etc. as exceptions can be the denotation of a Haskell expression whereas elements of the latter cannot.

We also need (finite) sets of exceptions as an equational defined data structure. Since expressions may have different exceptional behaviour depending on the evaluation order, exceptional behaviour will be denoted by a set of exceptions as proposed in [JRH$^+$99]. So let us assume we have sorts **Exceptions**$\{ecs_i\}$ and **NeExceptions**$\{necs_i\}$ of finite sets and nonempty finite sets of exceptions respectively (i.e. elements of sort **Exception**) with an appropriate set of operations like $\cdot\cup\cdot$, $\cdot\cap\cdot$, $\cdot\setminus\cdot$ and $\cdot\in\cdot$ and a usual syntax like e.g. $\{\langle$`ArithException Overflow`$\rangle_{\lightning}, \langle$`IOException`$\rangle_{\lightning}\}$ for a set having two exceptions.

Still not all preparations are made to define the semantics of Haskell properly. Beforehand it is necessary to enrich the syntax with a few expressions such that we are able to express partially applied function bindings. For example if we have a binary function defined in Haskell, say `add`, the correct result of `add 1` should be the unary successor function.

For this purpose the sorts **DefMatch**$\{m_i\}$, which is in principle a lambda abstraction having a different syntax plus an exceptional case (having the particular subsort **ExcDefMatch**$\{em_i\}$ for this), **DefMatches**$\{ms_i\}$, a list of those, and **NeDefMatches**$\{nms_i\}$, its nonempty correspondent, are introduced:

$$\textbf{ExcDefMatch} < \textbf{DefMatch} < \textbf{NeDefMatches} < \textbf{DefMatches}$$

$$\begin{aligned} \cdot :\!\rightarrow \cdot &: \textbf{PatList}\ \textbf{Exp} \rightarrow \textbf{DefMatch} \\ :\!\rightarrow \cdot &: \textbf{Exceptions} \rightarrow \textbf{ExcDefMatch} \\ \mathsf{nil} &: \textbf{DefMatches} \\ \cdot\,;\,\cdot &: \textbf{DefMatches}\ \textbf{DefMatches} \rightarrow \textbf{DefMatches} \\ \cdot\,;\,\cdot &: \textbf{NeDefMatches}\ \textbf{DefMatches} \rightarrow \textbf{NeDefMatches} \\ \cdot\,;\,\cdot &: \textbf{DefMatches}\ \textbf{NeDefMatches} \rightarrow \textbf{NeDefMatches} \\ \{\,\cdot\,\} &: \textbf{DefMatches} \rightarrow \textbf{Exp} \end{aligned}$$

The necessary equational sentences establishing the associativity of $\cdot\,;\,\cdot$ and its identity nil are trivial and are not explicitly mentioned here.

Of course the reader might argue that this could instead be expressed with a combination of a lambda abstraction and a `case` expression rendering the above syntax extension unnecessary. But this would be cumbersome and would yield a quite cluttered semantics definitions as well. For example the following expressions are supposed to be equivalent:

```
{1 , y :→ y;
 2 , y :→ 4 * y;
 3 , y :→ 8 * y}
```

```
(\a b -> case (a,b) of
         1 , y -> y;
         2 , y -> 4 * y;
         3 , y -> 8 * y)
```

In addition, when choosing the second approach, the semantic definition has to introduce *fresh* variables, in this example `a` and `b` and hence spawning the need for a syntax extension anyway, not mentioning a syntax for explicit exceptional behaviour. At the end the choice remains a matter of taste, of course.

The inclusion of an empty list of defined matches in the above operator declaration is intentionally. It allows a concise formulation of the semantics if the following equality is included:

$$\frac{}{\{\mathsf{nil} :\!\rightarrow e_1\} = e_1}$$

The same holds true for the following extension of the lambda abstraction that allows an empty list of argument patterns:

$$\texttt{\textbackslash}\ \cdot\ \texttt{->}\ \cdot : \textbf{PatList}\ \textbf{Exp} \rightarrow \textbf{Exp}$$

$$\frac{}{\texttt{\textbackslash}\,\mathsf{nil}\ \texttt{->}\ e_1 = e_1}$$

To describe function application, matching etc. it is required to have substitutions and means to manipulate them and to apply them to Haskell expressions. However it is not necessary to get too much into the details on how to define them equationally, for our purposes it may suffices to informally introduce the required sorts **SubstEntry** and **SubstEntrys**$\{pss_i\}$ for entries and collections of entries respectively and **SimpSubst**$\{ss_1\}$ for simple substitutions and to show how they are supposed to work on examples only:

For example $\{$ `x` $\mapsto$ `1` $,$ `y` $\mapsto$ `(+)` $\}$ is of sort **SimpSubst** whereas `y` $\mapsto$ `(+)` is of sort **SubstEntry** and `x` $\mapsto$ `1` $,$ `y` $\mapsto$ `(+)` of sort **SubstEntrys**, $ss_1(v_1)$ yields the image of v_1 if it is in the domain of ss_1 and $\perp$ otherwise and $ss_1 \overleftarrow{\cup} ss_2$ is the substitution that maps every variable v_1 being in the domain of ss_2 to $ss_2(v_1)$ and every other variable v_2 to $ss_1(v_2)$. ε will be assumed to denote the empty substitution.

We also need finite sets of variables as an equational defined data structure. So let us assume having the sorts **Vars**$\{vs_i\}$ and **NeVars**$\{nvs_i\}$ of finite sets and nonempty finite sets of variables (i.e. elements of sort **Var**) with an appropriate set of operations like $\cdot \cup \cdot$, $\cdot \cap \cdot$, $\cdot \setminus \cdot$ and $\cdot \in \cdot$ and a usual syntax like $\{v_1, v_2, v_3\}$ for a set having three variables.

Now we are able to define some operations concerning both simple substitutions and sets of variables: $\mathsf{dom}(\,\cdot\,)$, the domain of a simple substitution, $\cdot_{|\cdot}$ the restriction of simple substitution to a domain and $\cdot_{\setminus\cdot}$ the subtraction of a domain:

$$\begin{aligned} \mathsf{dom}(\,\cdot\,) &: \textbf{SimpSubst} \rightarrow \textbf{Vars} \\ \cdot_{|\cdot} &: \textbf{SimpSubst}\ \textbf{Vars} \rightarrow \textbf{SimpSubst} \\ \cdot_{\setminus\cdot} &: \textbf{SimpSubst}\ \textbf{Vars} \rightarrow \textbf{SimpSubst} \end{aligned}$$

$$\frac{}{\mathsf{dom}(\varepsilon) = \emptyset}\ \textbf{dom1} \qquad \frac{}{\mathsf{dom}(\{v_1 \mapsto e_1, pss_1\}) = v_1 \cup \mathsf{dom}(\{pss_1\})}\ \textbf{dom2}$$

$$\frac{v_1 \in vs_1 = true}{\{v_1 \mapsto e_1, pss_1\}_{|vs_1} = \{pss_1\}_{|vs_1} \overleftarrow{\cup} \{v_1 \mapsto e_1\}}\ \textbf{restr1} \qquad \frac{\neg(v_1 \in vs_1) = true}{\{v_1 \mapsto e_1, pss_1\}_{|vs_1} = \{pss_1\}_{|vs_1}}\ \textbf{restr1'}$$

$$\frac{}{\varepsilon_{|vs_1} = \varepsilon}\ \textbf{restr2}$$

$$\frac{v_1 \in vs_1 = true}{\{v_1 \mapsto e_1, pss_1\}_{\setminus vs_1} = \{pss_1\}_{\setminus vs_1}}\ \textbf{rem1} \qquad \frac{\neg(v_1 \in vs_1) = true}{\{v_1 \mapsto e_1, pss_1\}_{\setminus vs_1} = \{pss_1\}_{\setminus vs_1} \overleftarrow{\cup} \{v_1 \mapsto e_1\}}\ \textbf{rem1'}$$

$$\frac{}{\varepsilon_{\setminus vs_1} = \varepsilon}\ \textbf{rem2}$$

Before defining how substitutions are applied to Haskell expression, we need another kind of substitution: A fixed point substitution. Fixed point substitutions are — well — fixed points of simple substitutions, and hence are able to express mutual recursion thus being the perfect tool to define the semantics of function bindings. Here is the definition of the respective sort **FixpSubst**:

$$\begin{aligned} fix(\,\cdot\,) &: \textbf{SimpSubst} \;\rightarrow\; \textbf{FixpSubst} \\ \langle\!\langle \cdot \,|\, \cdot \rangle\!\rangle &: \textbf{SimpSubst}\;\;\textbf{Vars} \;\rightarrow\; \textbf{FixpSubst} \end{aligned}$$

$$\frac{}{fix(ss_1) = \langle\!\langle ss_1 | \mathsf{dom}(ss_1) \rangle\!\rangle}\;\textbf{fix}$$

So a fixed point is, syntactically speaking, only a pair containing a simple substitution and a set of variables. The set of variables is intended to denote the domain of the fixed point substitution. The reader might argue that the simple substitution has already a defined domain. So why the need for this explicit indication? Well, the intended inductive semantics of a fixed point substitution is the same as for simple substitutions except that, if they are applied to a variable they do not only yield the image expression for the variable but the expression after having applied the substitution to it again. So we get — in principle — the fixed point of the original simple substitutions. The crux now are the operations on the domain, which need to be defined for fixed point substitutions as well. Those operations will only remove elements from the set of variables carried by the fixed point substitution instead of removing elements from the simple substitution. If entries of the underlying substitution would be removed, this could clearly affect the image expression of the variables that are still in the domain as well:

$$\begin{aligned} \mathsf{dom}(\,\cdot\,) &: \textbf{FixpSubst} \;\rightarrow\; \textbf{Vars} \\ \cdot_{|\,\cdot} &: \textbf{FixpSubst}\;\;\textbf{Vars} \;\rightarrow\; \textbf{FixpSubst} \\ \cdot_{\setminus\,\cdot} &: \textbf{FixpSubst}\;\;\textbf{Vars} \;\rightarrow\; \textbf{FixpSubst} \end{aligned}$$

$$\frac{}{\mathsf{dom}(\langle\!\langle ss_1 | vs_1 \rangle\!\rangle) = vs_1}\;\textbf{fixdom} \qquad \frac{}{\langle\!\langle ss_1 | vs_1 \rangle\!\rangle_{|vs_2} = \langle\!\langle ss_1 | vs_1 \cap vs_2 \rangle\!\rangle}\;\textbf{fixrestr}$$

$$\frac{}{\langle\!\langle ss_1 | vs_2 \rangle\!\rangle_{\setminus vs_1} = \langle\!\langle ss_1 | vs_2 \setminus vs_1 \rangle\!\rangle}\;\textbf{fixrem}$$

Still this is not enough for defining fixed points of substitutions. If the semantics of fixed point substitutions would be defined as outlined above, the application of recursive substitutions, that is, substitutions for which there is a variable whose (fixed point) image expression has a free occurrence of that variable, would yield an expression of infinite length (or operationally speaking: It would not terminate). For example consider the following substitution defining the multiplication:

$$\left\{ \texttt{mult} \;\mapsto\; \begin{array}{l} \{\texttt{1 y :}\!\rightarrow \texttt{y;} \\ \;\;\texttt{x y :}\!\rightarrow \texttt{y + (mult (x - 1) y)}\;\; \} \end{array} \right\}$$

If the application of the fixed point of this simple substitution to the variable `mult` would yield — as proposed above — the image expression with the substitution applied to it, the variable `mult` in it would also be replaced by this expression and so on. Thus every such application of a fixed point of a recursive substitution would have to yield an infinite expression.

Therefore we need a means to defer the repeated substitution application until the variable is "needed", i.e. a lazy variant of substitution application is needed. To do so the syntax of expressions has to be extended once more by recursive closures **RecCl**$\{rc_i\}$:

$$\textbf{RecCl} < \textbf{Exp}$$

$$\cdot \vdash \cdot : \textbf{Env}\;\;\textbf{Var} \;\rightarrow\; \textbf{RecCl}\,,$$

with the intended meaning of $ss_1 \vdash v_1$ as being the image expression of v_1 of the fixed point of ss_1. The unfolding of this closure can be given right away:

$$\mathsf{unfold}(\,\cdot\,) : \textbf{RecCl} \;\rightarrow\; \textbf{Exp}$$

$$\frac{e_1 = ss_1(v_1)}{\mathsf{unfold}(ss_1 \vdash v_1) = e_1[\mathsf{fix}(ss_1)]}\ \textbf{unfold}$$

The use of $\mathsf{fix}(ss_1)$ restores the original fixed point substitution.

To properly define the application of substitutions the notion of free variables in Haskell expressions must be defined. This is done by equationally defining a function symbol $\mathsf{fv}(\,\cdot\,)$, denoting the set of free variables of an expression, in figure 3.5.

$$\mathsf{fv}(\,\cdot\,) : \textbf{ExpList} \rightarrow \textbf{Vars}$$
$$\mathsf{fv}(\,\cdot\,) : \textbf{Cases} \rightarrow \textbf{Vars}$$
$$\mathsf{fv}(\,\cdot\,) : \textbf{DefMatches} \rightarrow \textbf{Vars}$$
$$\mathsf{fv}(\,\cdot\,) : \textbf{SimpSubst} \rightarrow \textbf{Vars}$$

$$\frac{}{\mathsf{fv}(\mathsf{nil}) = \emptyset}\ \textbf{list1} \qquad \frac{}{\mathsf{fv}(e_1\ \mathtt{,}\ nl_1) = \mathsf{fv}(e_1) \cup \mathsf{fv}(nl_1)}\ \textbf{list2} \qquad \frac{}{\mathsf{fv}(v_1) = \{v_1\}}\ \textbf{var}$$

$$\frac{}{\mathsf{fv}(ct_1) = \emptyset}\ \textbf{ctor} \qquad \frac{}{\mathsf{fv}(pr_1) = \emptyset}\ \textbf{primitive} \qquad \frac{}{\mathsf{fv}(e_1\ e_2) = \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)}\ \textbf{app}$$

$$\frac{}{\mathsf{fv}(e_1 \circ e_2) = \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)}\ \circ \qquad \text{for all } \circ \in \mathit{Binop} \cup \{\ \mathtt{:}\ \}$$

$$\frac{}{\mathsf{fv}(\ \mathtt{(}\ l_1\ \mathtt{)}\ ) = \mathsf{fv}(l_1)}\ \textbf{tuple} \qquad \frac{}{\mathsf{fv}(\ \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) = \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \cup \mathsf{fv}(e_3)}\ \textbf{if}$$

$$\frac{}{\mathsf{fv}(\ \mathtt{case}\ e_1\ \mathtt{of}\ cas_1) = \mathsf{fv}(e_1) \cup \mathsf{fv}(cas_1)}\ \textbf{case} \qquad \frac{}{\mathsf{fv}(\ \backslash\ pl_1\ \mathtt{->}\ e_1) = \mathsf{fv}(e_1) \backslash \mathsf{fv}(pl_1)}\ \textbf{lambda}$$

$$\frac{}{\mathsf{fv}(pt_1\ \mathtt{->}\ e_1) = \mathsf{fv}(e_1) \backslash \mathsf{fv}(pt_1)}\ \textbf{cases1} \qquad \frac{}{\mathsf{fv}(ca_1\ \mathtt{;}\ cas_1) = \mathsf{fv}(ca_1) \cup \mathsf{fv}(cas_1)}\ \textbf{cases2}$$

$$\frac{}{\mathsf{fv}(rc_1) = \emptyset}\ \textbf{closure} \qquad \frac{}{\mathsf{fv}(\{ms_1\}) = \mathsf{fv}(ms_1)}\ \textbf{match} \qquad \frac{}{\mathsf{fv}(\mathsf{nil}) = \emptyset}\ \textbf{matches1}$$

$$\frac{}{\mathsf{fv}(pl_1 :\rightarrow e_1; ms_1) = (\mathsf{fv}(e_1) \backslash \mathsf{fv}(pl_1)) \cup \mathsf{fv}(ms_1)}\ \textbf{matches2} \qquad \frac{}{\mathsf{fv}(:\rightarrow ecs_1; ms_1) = \emptyset}\ \textbf{matches3}$$

$$\frac{}{\mathsf{fv}(\varepsilon) = \emptyset}\ \textbf{subst1} \qquad \frac{}{\mathsf{fv}(\{v_1 \mapsto e_1, pss_1\}) = \mathsf{fv}(e_1) \cup \mathsf{fv}(\{pss_1\})}\ \textbf{subst2}$$

Fig. 3.5: Definition of free variables.

Now all preparations are made to define the application of a substitution to an expression. To be able to do this more concisely a super sort **Subst**$\{s_i\}$ of **SimpSubst** and **FixpSubst** is introduced. Figure 3.6 on the facing page contains all necessary definitions.

$$\textbf{SimpSubst}, \textbf{FixpSubst} < \textbf{Subst}$$

$$\begin{aligned}
\cdot\,[\,\cdot\,] &: \textbf{ExpList}\ \ \textbf{Subst}\ \rightarrow\ \textbf{ExpList}\\
\cdot\,[\,\cdot\,] &: \textbf{Cases}\ \ \textbf{Subst}\ \rightarrow\ \textbf{Cases}\\
\cdot\,[\,\cdot\,] &: \textbf{DefMatches}\ \ \textbf{Subst}\ \rightarrow\ \textbf{DefMatches}
\end{aligned}$$

$$\frac{}{\mathsf{nil}[s_1] = \mathsf{nil}}\ \textbf{explist1} \qquad \frac{}{(e_1\ ,\ nl_1)[s_1] = e_1[s_1]\ ,\ nl_1[s_1]}\ \textbf{explist2} \qquad \frac{}{rc_1[s_1] = rc_1}\ \textbf{closure}$$

$$\frac{ss_1(v_1) = e_1}{v_1[ss_1] = e_1}\ \textbf{var} \qquad \frac{ss_1(v_1) = \bot}{v_1[ss_1] = v_1}\ \textbf{var'}$$

$$\frac{v_1 \in vs_1 = true}{v_1[\langle\!\langle ss_1 | vs_1 \rangle\!\rangle] = ss_1 \vdash v_1}\ \textbf{varfix} \qquad \frac{\neg(v_1 \in vs_1) = true}{v_1[\langle\!\langle ss_1 | vs_1 \rangle\!\rangle] = v_1}\ \textbf{varfix'} \qquad \frac{}{ct_1[s_1] = ct_1}\ \textbf{ctor}$$

$$\frac{}{pr_1[s_1] = pr_1}\ \textbf{primitive} \qquad \frac{}{(\ l_1\ )\ [s_1] =\ (\ l_1[s_1]\ )}\ \textbf{tuple} \qquad \frac{}{(e_1 e_2)[s_1] = (e_1[s_1])(e_2[s_1])}\ \textbf{app}$$

$$\frac{}{(\ \backslash\ pl_1\ \texttt{->}\ e_1)[s_1] =\ \backslash\ pl_1\ \texttt{->}\ e_1[s_{1 \backslash \mathsf{fv}(pl_1)}]}\ \textbf{lambda} \qquad \frac{}{\{ms_1\}[s_1] = \{ms_1[s_1]\}}\ \textbf{match}$$

$$\frac{}{(\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)[s_1] =\ \texttt{if}\ e_1[s_1]\ \texttt{then}\ e_2[s_1]\ \texttt{else}\ e_3[s_1]}\ \textbf{if}$$

$$\frac{}{(\ \texttt{case}\ e_1\ \texttt{of}\ cas_1)[s_1] =\ \texttt{case}\ e_1[s_1]\ \texttt{of}\ (cas_1[s_1])}\ \textbf{case}$$

$$\frac{}{(e_1 \circ e_2)[s_1] = (e_1[s_1]) \circ (e_2[s_1])}\ \circ \qquad \text{for every } \circ \in \mathit{BinOp} \cup \{\ :\ \}$$

$$\frac{}{\mathsf{nil}[s_1] = \mathsf{nil}}\ \textbf{matches1} \qquad \frac{}{(pl_1 :\longrightarrow e_1; ms_1)[s_1] = pl_1 :\longrightarrow (e_1[s_{1 \backslash \mathsf{fv}(pl_1)}]); ms_1[s_1]}\ \textbf{matches2}$$

$$\frac{}{(:\longrightarrow ecs_1)[s_1] = :\longrightarrow ecs_1}\ \textbf{matches3}$$

$$\frac{}{(pt_1\ \texttt{->}\ e_1)[s_1] = pt_1\ \texttt{->}\ e_1[s_{1 \backslash \mathsf{fv}(pt_1)}]}\ \textbf{cases1} \qquad \frac{}{(ca_1\ ;\ cas_1)[s_1] = ca_1[s_1]\ ;\ cas_1[s_1]}\ \textbf{cases2}$$

Fig. 3.6: Definition of the application of substitutions to expressions.

3.4 Semantics of Pure Haskell

Before discussing the detailed semantics of Haskell there is still some work to be done. At first some auxiliary functions are needed to be able to formulate the semantic rules appropriately. As the left hand side of a function binding is syntactically only a cascaded expression application there is a need for a function that transforms this into the function variable and a list of the argument patterns and vice versa:

$$\begin{aligned}
\mathsf{toList}(\,\cdot\,) &: \textbf{Exp}\ \rightarrow\ \textbf{NeExpList}\\
\mathsf{toExp}(\,\cdot\,) &: \textbf{NeExpList}\ \rightarrow\ \textbf{Exp}
\end{aligned}$$

$$\frac{}{\mathsf{toList}(e_1\ e_2) = \mathsf{toList}(e_1)\ ,\ toList(e_2)}\ \textbf{toList1} \qquad \frac{\mathsf{otherwise}}{\mathsf{toList}(e_1) = e_1}\ \textbf{toList2}$$

$$\frac{}{\mathsf{toExp}(e_1) = e_1}\ \textbf{toExp1} \qquad \frac{}{\mathsf{toExp}(e_1\ ,\ l_1) = e_1\ \mathsf{toExp}(l_1)}\ \textbf{toExp2}$$

To define the semantics properly we also need a means to decide whether the head of an expression is a constructor. This could have been done defining an appropriate subsort of **Exp**, but to avoid troubles that could destroy the preregularity of the theory, it is chosen to define a predicate instead:

$$\mathsf{ctorApp?}(\,\cdot\,) : \mathbf{Exp} \;\to\; \mathbf{Bool}$$

$$\frac{}{\mathsf{ctorApp?}(ct_1) = \mathsf{true}}\ \textbf{ctorApp ctor} \qquad \frac{}{\mathsf{ctorApp?}(e_1\, e_2) = \mathsf{ctorApp?}(e_1)}\ \textbf{ctorApp app}$$

$$\frac{\mathsf{otherwise}}{\mathsf{ctorApp?}(e_1) = \mathsf{false}}\ \textbf{ctorApp fail}$$

The same holds for the following predicate deciding whether the head of an expression is the list constructor $\cdot : \cdot$:

$$\mathsf{topCons?}(\,\cdot\,) : \mathbf{Exp} \;\to\; \mathbf{Bool}$$

$$\frac{}{\mathsf{topCons?}(e_1 : e_2) = \mathsf{true}}\ \textbf{topCons1} \qquad \frac{\mathsf{otherwise}}{\mathsf{topCons?}(e_1) = \mathsf{false}}\ \textbf{topCons2}$$

Furthermore a predicate deciding whether a pattern is trivial — i.e. any expression matches it — will be useful:

$$\mathsf{trivPat?}(\,\cdot\,) : \mathbf{Pat} \;\to\; \mathbf{Bool}$$

$$\frac{}{\mathsf{trivPat?}(v_1) = \mathsf{true}}\ \textbf{trivPat1} \qquad \frac{}{\mathsf{trivPat?}(\,\_\,) = \mathsf{true}}\ \textbf{trivPat1} \qquad \frac{\mathsf{otherwise}}{\mathsf{trivPat?}(pt_1) = \mathsf{false}}\ \textbf{trivPat2}$$

As already mentioned there is made a distinction between mere syntactic literals (sorts **FloatConst**, **IntConst**, etc.) and their semantic correspondents (sorts **Float**, **Int**, etc.). The same was done for boolean expressions, so here are the necessary conversion functions:

$$\mathsf{fromBool}(\,\cdot\,) : \mathbf{Bool} \;\to\; \mathbf{BoolConst}$$
$$\mathsf{toBool}(\,\cdot\,) : \mathbf{BoolConst} \;\to\; \mathbf{Bool}$$

$$\frac{}{\mathsf{fromBool}(\mathsf{true}) = \mathtt{True}}\ \textbf{fromBool true} \qquad \frac{}{\mathsf{fromBool}(\mathsf{false}) = \mathtt{False}}\ \textbf{fromBool false}$$

$$\frac{}{\mathsf{toBool}(\,\mathtt{True}\,) = \mathsf{true}}\ \textbf{toBool true} \qquad \frac{}{\mathsf{toBool}(\,\mathtt{False}\,) = \mathsf{false}}\ \textbf{toBool false}$$

Now the semantics of Haskell expression can be given. This is done by equationally defining the operators $\mathcal{F}[\![\cdot]\!]$ transforming expressions to its weak head normal form (whnf) and $\mathcal{M}[\![\cdot \overset{?}{:\doteq} \cdot]\!]$ matching an expression with a pattern by giving a simple substitution. Furthermore exceptional behaviour has to be considered. As the order of evaluation of an expression will not be fixed, an expression might have different exceptional behaviour depending on the evaluation order. Hence $\mathcal{F}[\![\cdot]\!]$ and $\mathcal{M}[\![\cdot \overset{?}{:\doteq} \cdot]\!]$ will evaluate to sets of exceptions instead of a single exception in case of exceptional behaviour. This is the same approach as taken in [MLJ99] where the reader can find a more detailed discussion of this approach using imprecise exceptions; as a matter of fact the operation $\mathcal{F}[\![\cdot]\!]$ is an amalgamation of the relations $\nearrow$ and $\Downarrow$ defined there. To properly define $\mathcal{F}[\![\cdot]\!]$ and $\mathcal{M}[\![\cdot \overset{?}{:\doteq} \cdot]\!]$ two new sorts are needed: **ExpUExc**$\{ee_i\}$, the "union" of **Exp** and **Exceptions** as well as **SimpSubstUExc**$\{sse_i\}$, the "union" of **SimpSubst** and **Exceptions**. This is necessary to maintain sort-decreasingness as both symbols may evaluate also to a term of sort **Exception**. Moreover additional constants are needed to denote that no matching was possible:

$$\mathbf{Exceptions}, \mathbf{SimpSubst} < \mathbf{SimpSubstUExc}$$
$$\mathbf{Exceptions}, \mathbf{Exp} < \mathbf{ExpUExc}$$

$$\begin{aligned}\mathsf{noMatching} &: [\mathbf{ExpUExc}]\\ \mathsf{noMatching} &: [\mathbf{SimpSubstUExc}]\end{aligned}$$

To define the propagation of exceptions more concisely it is also useful to extend the union of sets of exceptions to expressions as well:

$$\cdot \cup \cdot : [\mathbf{ExpUExc}]\ \ [\mathbf{ExpUExc}] \ \rightarrow\ [\mathbf{Exceptions}]$$

$$\frac{}{\mathit{ecs_1} \cup \mathit{e_1} = \mathit{ecs_1}}\ \mathbf{excs{\cup}exp} \qquad\qquad \frac{}{\mathit{e_1} \cup \mathit{ecs_1} = \mathit{ecs_1}}\ \mathbf{excs{\cup}exp}$$

So if $\mathcal{E}_H \vdash \mathit{ee_1} \cup \mathit{ee_2} : \mathbf{Exceptions}$ for the MEL theory $\mathcal{E}_H$ defined so far at least one of $\mathit{ee_1}$ and $\mathit{ee_2}$ is exceptional and $\mathit{ee_1} \cup \mathit{ee_2}$ is the respective set of exceptions or the union of both if both of them are exceptional.

To easily create exception terms some shortcuts are introduced where an appropriately defined symbol $\mathsf{toString}(\,\cdot\,) : \mathbf{Exp} \rightarrow \mathbf{StringConst}$ is assumed that converts expressions into its string representation:

$$\begin{aligned}\mathsf{typeExc}(\,\cdot\,) &: \mathbf{Exp} \ \rightarrow\ \mathbf{Exception}\\ \mathsf{patternExc}(\,\cdot\,) &: \mathbf{Exp} \ \rightarrow\ \mathbf{Exception}\\ \mathsf{divZeroExc}(\,\cdot\,) &: \mathbf{Exp} \ \rightarrow\ \mathbf{Exception}\end{aligned}$$

$$\frac{}{\mathsf{typeExc}(\mathit{e_1}) = \langle\, \texttt{TypeError}\,\rangle_{\lightning}}\ \mathbf{type}$$

$$\frac{}{\mathsf{patternExc}(\mathit{e_1}) = \langle\, \texttt{PatternMatchFail}\, \mathsf{toString}(\mathit{e_1})\rangle_{\lightning}}\ \mathbf{pat}$$

$$\frac{}{\mathsf{divZeroExc}(\mathit{e_1}) = \langle\, \texttt{ArithException DivideByZero}\,\rangle_{\lightning}}\ \mathbf{div}$$

Moreover a "symmetric" union $\cdot \uplus \cdot$ for simple substitutions is useful that evaluates to a simple substitution only if both argument substitutions have disjoint domains. Otherwise it is supposed to yield a set containing the appropriate exceptions:

$$\begin{aligned}\cdot \uplus \cdot &: \mathbf{SimpSubst}\ \ \mathbf{SimpSubst} \ \rightarrow\ \mathbf{SimpSubstUExc}\\ \mathsf{patExc}(\,\cdot\,) &: \mathbf{Vars} \ \rightarrow\ \mathbf{Exceptions}\end{aligned}$$

$$\frac{\mathsf{dom}(\mathit{ss_1}) \cap \mathsf{dom}(\mathit{ss_2}) = \emptyset}{\mathit{ss_1} \uplus \mathit{ss_2} = \mathit{ss_1} \overleftarrow{\cup} \mathit{ss_2}}\ \mathbf{disj} \qquad\qquad \frac{\mathsf{dom}(\mathit{ss_1}) \cap \mathsf{dom}(\mathit{ss_2}) = \mathit{nvs_1}}{\mathit{ss_1} \uplus \mathit{ss_2} = \mathsf{patVarExc}(\mathit{nvs_1})}\ \mathbf{confl}$$

$$\frac{}{\mathsf{patExc}(\emptyset) = \emptyset}\ \mathbf{exc1}$$

$$\frac{}{\begin{aligned}&\mathsf{patVarExc}(\{\mathit{v_1}, \mathit{pvs_1}\})\\ &= \mathsf{patVarExc}(\{\mathit{pvs_1}\}) \cup \{\langle\, \texttt{PatternVariableException}\, \mathsf{toString}(\mathit{v_1})\rangle_{\lightning}\,\}\end{aligned}}\ \mathbf{exc2}$$

It is also useful extending the operator $\cdot \uplus \cdot$ to exceptions to easily propagate them:

$$\cdot \uplus \cdot : \mathbf{SimpSubstUExc}\ \ \mathbf{SimpSubstUExc} \ \rightarrow\ \mathbf{SimpSubstUExc}$$

$$\frac{}{\mathit{ecs_1} \uplus \mathit{ss_2} = \mathit{ecs_1}}\ \mathbf{left} \qquad \frac{}{\mathit{ss_1} \uplus \mathit{ecs_2} = \mathit{ecs_2}}\ \mathbf{right} \qquad \frac{}{\mathit{ecs_1} \uplus \mathit{ecs_2} = \mathit{ecs_1} \cup \mathit{ecs_2}}\ \mathbf{both}$$

The same is easily done for the application of a substitution to an expression:

$$\cdot\,[\,\cdot\,] : \mathbf{Exp}\ \ \mathbf{SimpSubstUExc} \ \rightarrow\ \mathbf{ExpUExc}$$

$$\frac{}{\mathit{e_1}[\mathit{ecs_1}] = \mathit{ecs_1}}$$

$$\begin{array}{rcll}
\mathit{SimpSubst} & ::= & \ldots & \text{(cf. MEL signature)}\\
\mathit{DefMatches} & ::= & \ldots & \text{(cf. MEL signature)}\\
\mathit{Exp} & ::= & \ldots & \text{(cf. previous bnf definition)}\\
& & \mid \mathit{SimpSubst} \vdash \mathit{Var} & \\
& & \mid \{\mathit{DefMatches}\} & \\
\mathit{BinPrim} & ::= & \texttt{(}\,\mathit{BinOp}\,\texttt{)} & \\
\mathit{CtorApp} & ::= & \mathit{Ctor} \mid \mathit{Var} \mid \mathit{CtorApp}\ \mathit{Exp} & \\
\mathit{WhnfExp} & ::= & \mathit{AtExp} & \\
& & \mid \mathit{BinPrim}\ \mathit{Exp} & \\
& & \mid \mathit{CtorApp} & \\
& & \mid \backslash\, \mathit{PatList}\ \texttt{->}\ \mathit{Exp} & \\
& & \mid \{\mathit{DefMatches}\} & \\
& & \mid \mathit{Exp} : \mathit{Exp} & \\
& & \mid \texttt{[}\,\mathit{ExpList}\,\texttt{]} & \\
& & \mid \texttt{(}\,\mathit{ExpList}\,\texttt{)} &
\end{array}$$

Fig. 3.7: Syntax of Haskell expressions in whnf.

Before describing the transformation into its whnf it is or course necessary to define when a Haskell expression is in whnf. That is done in figure 3.7. Defining a predicate $\cdot \Downarrow : \mathbf{Exp} \rightarrow \mathbf{Bool}$ that decides whether an expression is in whnf is an easy translation of the given bnf grammar. The details are left to the reader.

Now everything is prepared to describe the semantics of Haskell expressions. On the following pages the transformation in whnf is given in figures 3.8 and 3.9 the matching in figures 3.10, 3.12 and 3.11.

For the definition of the built-in primitives only a selection is given. Though a particular interesting case is missing: The equality operator `==`. Of course, one problem here is that this operator is defined for the members of the type class `Eq`, and since the type calculus of Haskell is not considered in this thesis this has to be omitted. But the goal is to give the correct semantics for all types assuming they are all member of the `Eq` class. The semantics for the built-in types is clear then. For the user defined types it is assumed that their membership in `Eq` is defined by the `deriving Eq` statement delivering a standard implementation for the equality test.

That is, values of those types are equal if and only if they are literally equal. Unfortunately these values are not the whnf as defined above as arguments of constructors remain unevaluated. So an additional notion for an normal form is needed here. A constructor head normal form (chnf). An appropriate syntax definition for expressions in chnf is given below:

$$\begin{array}{rcl}
\mathit{ChnfExpList} & ::= & \mathit{ChnfExp} \mid \mathit{ChnfExpList}\ \texttt{,}\ \mathit{ChnfExp}\\
\mathit{ChnfExp} & ::= & \mathit{AtExp} \mid \mathit{BinPrim}\ \mathit{Exp} \mid \mathit{Ctor}\ \mathit{ChnfExp} \mid \mathit{Var}\ \mathit{ChnfExp} \mid \backslash\, \mathit{PatList}\ \texttt{->}\ \mathit{Exp}\\
& & \mid \{\mathit{DefMatches}\} \mid \mathit{ChnfExp} : \mathit{ChnfExp} \mid \texttt{[}\,\mathit{ChnfExpList}\,\texttt{]} \mid \texttt{(}\,\mathit{ChnfExpList}\,\texttt{)}
\end{array}$$

So another operator symbol along with set of equational sentences is needed to describe this new normal form. Fortunately the chnf differs from the whnf only in the treatment of constructor application. So most of the equational sentences for the whnf above can be reused. They do not even need to be copied. The operator for the normal form transformation $\mathcal{F}[\![\cdot]\!]$ as well as the predicate operator $\cdot \Downarrow$ only need to be changed to take another parameter (of sort $\mathbf{NfType}\{\mathit{nf}_i\}$)

$$\mathcal{F}[\![\,\cdot\,]\!] : [\mathbf{Exp}] \;\rightarrow\; [\mathbf{ExpUExc}]$$

$$\frac{\mathcal{F}[\![e_1]\!] = \texttt{True}}{\mathcal{F}[\![\,\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = \mathcal{F}[\![e_2]\!]}\ \textbf{if1} \qquad \frac{\mathcal{F}[\![e_1]\!] = \texttt{False}}{\mathcal{F}[\![\,\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = \mathcal{F}[\![e_3]\!]}\ \textbf{if2}$$

$$\frac{\mathcal{F}[\![e_1]\!] = ecs_1}{\mathcal{F}[\![\,\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = ecs_1}\ \textbf{if}\ \lightning$$

$$\frac{\mathcal{M}\,[\![cas_1 \mathrel{:\overset{?}{=}} e_1]\!] = e_2}{\mathcal{F}[\![\,\mathsf{case}\ e_1\ \mathsf{of}\ cas_1]\!] = \mathcal{F}[\![e_2]\!]}\ \textbf{case}$$

$$\frac{\mathcal{M}\,[\![cas_1 \mathrel{:\overset{?}{=}} e_1]\!] = \mathsf{noMatching}}{\mathcal{F}[\![\,\mathsf{case}\ e_1\ \mathsf{of}\ cas_1]\!] = \{\mathsf{patternExc}(\,\mathsf{case}\ e_1\ \mathsf{of}\ cas_1)\}}\ \textbf{case fail}$$

$$\frac{\mathcal{M}\,[\![cas_1 \mathrel{:\overset{?}{=}} e_1]\!] = ecs_1}{\mathcal{F}[\![\,\mathsf{case}\ e_1\ \mathsf{of}\ cas_1]\!] = ecs_1}\ \textbf{case}\ \lightning$$

$$\frac{\mathcal{F}[\![e_1]\!] = \ \backslash\, pt_1\ \texttt{,}\ pl_1\ \texttt{->}\ e_3 \qquad \mathcal{M}\,[\![pt_1 \mathrel{:\overset{?}{=}} e_2]\!] = ss_1}{\mathcal{F}[\![e_1\ e_2]\!] = \mathcal{F}[\![\,\backslash\, pl_1\ \texttt{->}\ e_3[ss_1]]\!]}\ \textbf{lambda}$$

$$\frac{\mathcal{F}[\![e_1]\!] = \ \backslash\, pt_1\ \texttt{,}\ pl_1\ \texttt{->}\ e_3 \qquad \mathcal{M}\,[\![pt_1 \mathrel{:\overset{?}{=}} e_2]\!] = \mathsf{noMatching}}{\mathcal{F}[\![e_1\ e_2]\!] = \{\mathsf{patternExc}(e_1\ e_2)\}}\ \textbf{lambda fail}$$

$$\frac{\mathcal{F}[\![e_1]\!] = \ \backslash\, pt_1\ \texttt{,}\ pl_1\ \texttt{->}\ e_3 \qquad \mathcal{M}\,[\![pt_1 \mathrel{:\overset{?}{=}} e_2]\!] = ecs_1}{\mathcal{F}[\![e_1\ e_2]\!] = ecs_1}\ \textbf{lambda}\ \lightning$$

$$\frac{\mathcal{F}[\![e_1]\!] = \{ms_1\} \qquad \mathcal{M}\,[\![ms_1 \mathrel{:\overset{?}{=}} e_2]\!] = nms_1}{\mathcal{F}[\![e_1\ e_2]\!] = \mathcal{F}[\![\{nms_1\}]\!]}\ \textbf{match}$$

$$\frac{\mathcal{F}[\![e_1]\!] = \{ms_1\} \qquad \mathcal{M}\,[\![ms_1 \mathrel{:\overset{?}{=}} e_2]\!] = \mathsf{nil}}{\mathcal{F}[\![e_1\ e_2]\!] = \{\mathsf{patternExc}(\{ms1\}\ e_1)\}}\ \textbf{match fail} \qquad \frac{}{\mathcal{F}[\![\{:\rightarrow ecs_1; ms_1\}]\!] = ecs_1}\ \textbf{match}\ \lightning$$

$$\frac{}{\mathcal{F}[\![\ \texttt{(}\ \circ\ \texttt{)}\ e_1 e_2]\!] = \mathcal{F}[\![e_1 \circ e_2]\!]}\ \texttt{(}\ \circ\ \texttt{)} \qquad \text{for every } \circ \in \mathit{BinOp}$$

$$\frac{\mathcal{F}[\![e_1]\!] = ecs_1 \qquad \mathcal{F}[\![e_2]\!] : \mathbf{Exp}}{\mathcal{F}[\![e_1\ e_2]\!] = ecs_1}\ \textbf{app}\ \lightning \qquad \frac{\mathcal{F}[\![e_1]\!] = ecs_1 \qquad \mathcal{F}[\![e_2]\!] = ecs_2}{\mathcal{F}[\![e_1\ e_2]\!] = ecs_1 \cup ecs_2}\ \textbf{app arg}\ \lightning$$

$$\frac{}{\mathcal{F}[\![rc_1]\!] = \mathcal{F}[\![\mathsf{unfold}(rc_1)]\!]}\ \textbf{closure} \qquad \frac{e_1 \Downarrow = \mathsf{true}}{\mathcal{F}[\![e_1]\!] = e_1}\ \textbf{whnf} \qquad \frac{\mathsf{otherwise}}{\mathcal{F}[\![e_1]\!] = \mathsf{typeExc}(e_1)}\ \textbf{type}\ \lightning$$

Fig. 3.8: Definitions describing the transformation of structural Haskell expressions in whnf.

$$\frac{\mathcal{F}\llbracket e_1 \rrbracket = bc_1}{\mathcal{F}\llbracket \texttt{not}\ e_1 \rrbracket = \mathsf{fromBool}(\neg\mathsf{toBool}(bc_1))}\ \textbf{not} \qquad \frac{\mathcal{F}\llbracket e_1 \rrbracket = ecs_1}{\mathcal{F}\llbracket \texttt{not}\ e_1 \rrbracket = ecs_1}\ \textbf{not}$$

$$\frac{\mathcal{F}\llbracket e_1 \rrbracket = ece_1}{\mathcal{F}\llbracket \texttt{raise}\ e_1 \rrbracket = \{\langle ece_1\rangle_{\lightning}\}}\ \textbf{raise} \qquad \frac{\mathcal{F}\llbracket e_1 \rrbracket = ecs_1}{\mathcal{F}\llbracket \texttt{raise}\ e_1 \rrbracket = ecs_1}\ \textbf{raise}\ \lightning$$

$$\frac{\mathcal{F}\llbracket e_1 \rrbracket : \mathbf{Exp}}{\mathcal{F}\llbracket \mathsf{seq}\ e_1 e_2 \rrbracket = \mathcal{F}\llbracket e_2 \rrbracket}\ \textbf{seq} \qquad \frac{\mathcal{F}\llbracket e_1 \rrbracket \cup \mathcal{F}\llbracket e_2 \rrbracket = ecs_1}{\mathcal{F}\llbracket \mathsf{seq}\ e_1 e_2 \rrbracket = ecs_1}\ \textbf{seq}\ \lightning$$

$$\frac{\mathcal{F}\llbracket e_1 \rrbracket = fc_1 \qquad \mathcal{F}\llbracket e_2 \rrbracket = fc_2}{\mathcal{F}\llbracket e_1 + e_2 \rrbracket = \mathsf{fromFloat}(\mathsf{toFloat}(fc_1) + \mathsf{toFloat}(fc_2))}\ \textbf{plus} \qquad \frac{\mathcal{F}\llbracket e_1 \rrbracket \cup \mathcal{F}\llbracket e_2 \rrbracket = ecs_1}{\mathcal{F}\llbracket e_1 + e_2 \rrbracket = ecs_1}\ \textbf{plus}\ \lightning$$

Fig. 3.9: Additional equational sentences describing the semantics of a selection of built-in primitives of Haskell.

$$\mathcal{M}\llbracket \cdot \overset{?}{:=} \cdot \rrbracket : [\mathbf{PatList}]\ [\mathbf{ExpList}] \rightarrow [\mathbf{SimpSubstUExc}]$$

$$\frac{\neg\mathsf{trivPat?}(pt_1) = \mathsf{true} \qquad \mathcal{F}\llbracket e_1 \rrbracket = ecs_1}{\mathcal{M}\llbracket pt_1 \overset{?}{:=} e_1 \rrbracket = ecs_1}\ \lightning$$

$$\frac{}{\mathcal{M}\llbracket pt_1\ ,\ pl_1 \overset{?}{:=} e_1\ ,\ nl_1 \rrbracket = \mathcal{M}\llbracket pl_1 \overset{?}{:=} nl_1 \rrbracket \uplus \mathcal{M}\llbracket pt_1 \overset{?}{:=} e_1 \rrbracket}\ \textbf{list1} \qquad \frac{}{\mathcal{M}\llbracket \mathsf{nil} \overset{?}{:=} \mathsf{nil} \rrbracket = \varepsilon}\ \textbf{list2}$$

$$\frac{}{\mathcal{M}\llbracket v_1 \overset{?}{:=} e_1 \rrbracket = \{v_1 \mapsto e_1\}}\ \textbf{var} \qquad \frac{}{\mathcal{M}\llbracket \_ \overset{?}{:=} e_1 \rrbracket = \varepsilon}\ \textbf{wildcard}$$

$$\frac{\mathsf{toList}(pt_1) = ct_1\ ,\ pl_1 \qquad \mathsf{toList}(\mathcal{F}\llbracket e_1 \rrbracket) = ct_1\ ,\ l_1}{\mathcal{M}\llbracket pt_1 \overset{?}{:=} e_1 \rrbracket = \mathcal{M}\llbracket pl_1 \overset{?}{:=} l_1 \rrbracket}\ \textbf{ctor}$$

$$\frac{\mathcal{F}\llbracket e_3 \rrbracket = e_1 : e_2}{\mathcal{M}\llbracket pt_1 : pt_2 \overset{?}{:=} e_3 \rrbracket = \mathcal{M}\llbracket pt_1 \overset{?}{:=} e_1 \rrbracket \uplus \mathcal{M}\llbracket pt_2 \overset{?}{:=} e_2 \rrbracket}\ \textbf{cons}$$

$$\frac{\mathcal{F}\llbracket e_1 \rrbracket = \texttt{(}\ l_1\ \texttt{)}}{\mathcal{M}\llbracket \texttt{(}\ pl_1\ \texttt{)} \overset{?}{:=} e_1 \rrbracket = \mathcal{M}\llbracket pl_1 \overset{?}{:=} l_1 \rrbracket}\ \textbf{tuple} \qquad \frac{\mathsf{otherwise}}{\mathcal{M}\llbracket pt_1 \overset{?}{:=} e_1 \rrbracket = \mathsf{noMatching}}\ \textbf{fail}$$

Fig. 3.10: Definitions describing the matching of expressions with patterns.

$$\mathcal{M}\,[\![\,\cdot\ :\stackrel{?}{=}\ \cdot\,]\!]\ :\ [\textbf{DefMatches}]\ \ [\textbf{ExpList}]\ \rightarrow\ [\textbf{DefMatches}]$$

$$\frac{}{\mathcal{M}\,[\![\mathsf{nil} :\stackrel{?}{=} e_1]\!] = \mathsf{nil}}\ \textbf{nil} \qquad \frac{}{\mathcal{M}\,[\![:\longrightarrow ecs_1; ms_1 :\stackrel{?}{=} e_1]\!] =:\longrightarrow ecs_1}\ \textbf{prop}\ \lightning$$

$$\frac{\mathcal{M}\,[\![pt_1 :\stackrel{?}{=} e_1]\!] = \mathsf{noMatching}}{\mathcal{M}\,[\![(pt_1\ ,\ pl_1 :\longrightarrow e_2); ms_1 :\stackrel{?}{=} e_1]\!] = \mathcal{M}\,[\![ms_1 :\stackrel{?}{=} e_1]\!]}\ \textbf{skip}$$

$$\frac{\mathcal{M}\,[\![pt_1 :\stackrel{?}{=} e_1]\!] = ss_1}{\mathcal{M}\,[\![(pt_1 :\longrightarrow e_2); ms_1 :\stackrel{?}{=} e_1]\!] = \mathsf{nil} :\longrightarrow e_2[ss_1]}\ \textbf{succ}$$

$$\frac{\mathcal{M}\,[\![pt_1 :\stackrel{?}{=} e_1]\!] = ss_1}{\mathcal{M}\,[\![(pt_1\ ,\ npl_1 :\longrightarrow e_2); ms_1 :\stackrel{?}{=} e_1]\!] = (npl_1 :\longrightarrow e_2[ss_1]); \mathcal{M}\,[\![ms_1 :\stackrel{?}{=} e1]\!]}\ \textbf{part}$$

$$\frac{}{\mathcal{M}\,[\![(\mathsf{nil} :\longrightarrow e_2); ms_1 :\stackrel{?}{=} e_1]\!] = (\mathsf{nil} :\longrightarrow e_2\ e_1); \mathcal{M}\,[\![ms_1 :\stackrel{?}{=} e1]\!]}\ \textbf{triv}$$

$$\frac{\mathcal{M}\,[\![pt_1 :\stackrel{?}{=} e_1]\!] = ecs_1}{\mathcal{M}\,[\![(pt_1\ ,\ pl_1 :\longrightarrow e_2); ms_1 :\stackrel{?}{=} e_1]\!] =:\longrightarrow ecs_1}\ \lightning$$

Fig. 3.11: Definitions describing the matching of expressions with defined matches.

$$\mathcal{M}\,[\![\,\cdot\ :\stackrel{?}{=}\ \cdot\,]\!]\ :\ [\textbf{Cases}]\ \ [\textbf{Exp}]\ \rightarrow\ [\textbf{ExpUExc}]$$

$$\frac{\mathcal{M}\,[\![ca_1 :\stackrel{?}{=} e_1]\!] = ee_1}{\mathcal{M}\,[\![ca_1\ ;\ cas_1 :\stackrel{?}{=} e_1]\!] = ee_1}\ \textbf{succ mult} \qquad \frac{\mathcal{M}\,[\![ca_1 :\stackrel{?}{=} e_1]\!] = \mathsf{noMatching}}{\mathcal{M}\,[\![ca_1\ ;\ cas_1 :\stackrel{?}{=} e_1]\!] = \mathcal{M}\,[\![cas_1 :\stackrel{?}{=} e_1]\!]}\ \textbf{skip}$$

$$\frac{\mathcal{M}\,[\![pt_1 :\stackrel{?}{=} e_1]\!] = sse_1}{\mathcal{M}\,[\![pt_1\ \texttt{->}\ e_2 :\stackrel{?}{=} e_1]\!] = e_2[sse_1]}\ \textbf{succ} \qquad \frac{\mathcal{M}\,[\![pt_1 :\stackrel{?}{=} e_1]\!] = \mathsf{noMatching}}{\mathcal{M}\,[\![pt_1\ \texttt{->}\ e_2 :\stackrel{?}{=} e_1]\!] = \mathsf{noMatching}}\ \textbf{fail}$$

Fig. 3.12: Definitions describing the matching of expressions with cases.

stating which type of normal form has to be considered:

$$\begin{aligned} \mathsf{whnf} : \mathbf{NfType}\ , \qquad & \mathsf{chnf} : \mathbf{NfType} \\ \mathcal{F}[\![\,\cdot\,]\!]_{\cdot} : [\mathbf{Exp}]\ [\mathbf{NfType}] & \rightarrow [\mathbf{ExpUExc}] \\ \cdot \Downarrow_{\cdot} : [\mathbf{Exp}]\ [\mathbf{NfType}] & \rightarrow [\mathbf{Bool}] \end{aligned}$$

Hence all defining equations for $\cdot \Downarrow$ have to be changed to use $\cdot \Downarrow_{\mathsf{whnf}}$ instead and appropriate equations for $\cdot \Downarrow_{\mathsf{chnf}}$ have to be given by translating the bnf definition given above[2]. Moreover the equational sentences defining $\mathcal{F}[\![\,\cdot\,]\!]$ can be changed to use $\mathcal{F}[\![\,\cdot\,]\!]_{nt_1}$ instead — and $\cdot \Downarrow_{nt_1}$ for equation **whnf** which should be called **nf** then. That means those equations do also hold for the chnf. Additionally the following equational sentences for the chnf are needed:

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{chnf}} = e_3 \qquad \mathsf{ctorApp?}(e_3) = \mathsf{true} \qquad \mathcal{F}[\![e_2]\!]_{\mathsf{chnf}} = e_4}{\mathcal{F}[\![e_1 e_2]\!]_{\mathsf{chnf}} = e_3 e_4}\ \textbf{ctor app}$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{chnf}} = e_3 \qquad \mathsf{ctorApp?}(e_3) = \mathsf{true} \qquad \mathcal{F}[\![e_2]\!]_{\mathsf{chnf}} = ecs_1}{\mathcal{F}[\![e_1 e_2]\!]_{\mathsf{chnf}} = ecs_1}\ \textbf{ctor app}\ \lightning$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{chnf}} = e_2 \qquad \mathcal{F}[\![\,(\,l_1\,)\,]\!]_{\mathsf{chnf}} = (\,l_2\,)}{\mathcal{F}[\![\,(\,e_1\,,\,l_1\,)\,]\!]_{\mathsf{chnf}} = (\,e_2\,,\,l_2\,)}\ \textbf{tuple} \qquad \frac{\mathcal{F}[\![e_1]\!] \cup \mathcal{F}[\![\,(\,l_1\,)\,]\!] = ecs_1}{\mathcal{F}[\![\,(\,e_1\,,\,l_1\,)\,]\!]_{\mathsf{chnf}} = ecs_1}\ \textbf{tuple}\ \lightning$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{chnf}} = e_3 \qquad \mathcal{F}[\![e_2]\!]_{\mathsf{chnf}} = e_4}{\mathcal{F}[\![e_1 : e_2]\!]_{\mathsf{chnf}} = e_3 : e_4}\ \textbf{list} \qquad \frac{\mathcal{F}[\![e_1]\!] \cup \mathcal{F}[\![e_2]\!] = ecs_1}{\mathcal{F}[\![e_1 : e_2]\!]_{\mathsf{chnf}} = ecs_1}\ \textbf{list}\ \lightning$$

All other previous occurrences of $\mathcal{F}[\![\,\cdot\,]\!]$ can be safely changed to $\mathcal{F}[\![\,\cdot\,]\!]_{\mathsf{whnf}}$, since pattern matching only needs — for the laziness — the whnf.

Giving the equational sentences defining the equality `==` for both normal forms is now trivial:

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{chnf}} = e_3 \qquad \mathcal{F}[\![e_2]\!]_{\mathsf{chnf}} = e_4}{\mathcal{F}[\![e_1 \mathtt{==} e_2]\!]_{nt_1} = \mathsf{fromBool}(e_1 = e_2)}\ \mathtt{==} \qquad \frac{\mathcal{F}[\![e_1]\!]_{\mathsf{chnf}} \cup \mathcal{F}[\![e_2]\!]_{\mathsf{chnf}} = ecs_1}{\mathcal{F}[\![e_1 \mathtt{==} e_2]\!]_{nt_1} = ecs_1}\ \mathtt{==}\ \lightning$$

Having this and the MEL definition of $\cdot \Downarrow_{\cdot}$ according to the bnf definition, the following is easy to show.

Lemma 3.1. *Let $\mathcal{E}_H$ be the MEL theory developed so far, e, e' terms of sort* **Exp** *and $n \in \{\mathsf{whnf}, \mathsf{chnf}\}$. Then the following holds:*

$$\textit{if} \quad \mathcal{E}_H \vdash \mathcal{F}[\![e]\!]_n = e' \quad \textit{then} \quad \mathcal{E}_H \vdash e' \Downarrow_n = \mathsf{true}$$

Proof. Straightforward induction on e. □

Now that the semantics of Haskell expressions is defined, defining the semantics of a whole Haskell program is easy. It is the fixed point of the substitution induced by the function bindings of the program. Additionally the semantics of an Haskell expression in the context of a Haskell program can be defined as the semantics of the program applied to the expression:

$$\begin{aligned} \mathsf{env}(\,\cdot\,) : \mathbf{Program} & \rightarrow \mathbf{SimpSubst} \\ \mathcal{H}[\![\,\cdot\,]\!] : \mathbf{Program} & \rightarrow \mathbf{Subst} \\ \mathcal{H}[\![\,\cdot\ \mathsf{in}\ \cdot\,]\!] : [\mathbf{Exp}]\ [\mathbf{Program}] & \rightarrow [\mathbf{Exp}] \end{aligned}$$

[2]Actually some equations can be shared by both normal form predicates by using $\cdot \Downarrow_{nt_1}$

$$\frac{\mathsf{toList}(\mathit{fl}_1) = v_1\ ,\ pl_1}{\mathsf{env}(\mathit{fl}_1 = e_1) = \{v_1 \mapsto \{pl_1 :\!\rightarrow e_1\}\}}\ \textbf{bind}$$

$$\frac{\mathsf{toList}(\mathit{fl}_1) = v_1\ ,\ pl_1 \qquad \mathsf{env}(p_1) = ss_1 \qquad ss_1(v_1) = \bot}{\mathsf{env}(\mathit{fl}_1 = e_1\ ;\ p_1) = ss_1 \overleftarrow{\cup} \{v_1 \mapsto \{pl_1 :\!\rightarrow e_1\}\}}\ \textbf{binds1}$$

$$\frac{\mathsf{toList}(\mathit{fl}_1) = v_1\ ,\ pl_1 \qquad \mathsf{env}(p_1) = ss_1 \qquad ss_1(v_1) = e_2}{\mathsf{env}(\mathit{fl}_1 = e_1\ ;\ p_1) = ss_1 \overleftarrow{\cup} \{v_1 \mapsto \mathsf{cons}(pl_1 :\!\rightarrow e_1, e_2)\}}\ \textbf{binds2}$$

$$\frac{}{\mathcal{H}[\![p_1]\!] = \mathsf{fix}(\mathsf{env}(p_1))}\ \textbf{prog} \qquad\qquad \frac{}{\mathcal{H}[\![e_1\ \mathsf{in}\ p_1]\!] = \mathcal{F}[\![e_1[\mathcal{H}[\![p_1]\!]]]\!]_{\mathsf{chnf}}}\ \textbf{progexp}$$

where $\mathsf{cons}(\,\cdot\,,\,\cdot\,)$ is defined in the following way:

$$\mathsf{cons}(\,\cdot\,,\,\cdot\,) : [\textbf{DefMatch}]\ \ [\textbf{Exp}]\ \rightarrow\ [\textbf{Exp}]$$

$$\frac{}{\mathsf{cons}(m_1, \{ms_1\}) = \{m_1; ms_1\}}$$

So $\mathcal{H}[\![e_1\ \mathsf{in}\ p_1]\!]$ denotes the functional semantics — that is, the whnf — of the Haskell expression e_1 in the context of the Haskell program p_1.

3.5 Executability

Well, until now nothing was said about the executability of the semantic MEL theory $\mathcal{E}_H$ given so far. That is preregularity, ground termination, ground confluence and ground sort-decreasingness of the theory respectively its induced rewriting system have to be established. Checking preregularity is tedious but trivial. It turns out that the given theory $\mathcal{E}_H$ provides a smallest codomain sort for every operator symbol. Also checking sort-decreasingness is easy. As a matter of fact sort-decreasingness is not only essential for executability but it is also desirable for readability as it informally just means that functions has be given the "right" sort declaration. For example the definition of $\mathsf{unfold}(\,\cdot\,) : \textbf{RecCl} \rightarrow \textbf{Exp}$. This declaration truly states that unfold "applied" to a recursive closure yields an expression. Yet if sort-decreasingness is not required this need not be true. For example the semantics of MEL allows unfold — only regarding its declaration — to "evaluate" a recursive closure to, say, a term of sort **Exceptions** as it is in the same connected component as **Exp**. But of course **Exceptions** is not smaller than **Exp** w.r.t. the sort order, hence such behaviour would destroy sort-decreasingness.

More interesting is the property of ground termination of the induced rewriting system. This property is of course trivially given for the inductive definitions of functions on Haskell expressions (fv, substitution application, predicates, ...) as well as for simple shorthand definitions(fix, unfold, ...). On the other hand the transformation to whnf cannot be terminating for diverging Haskell expressions. Hence it should not be a far too big surprise that the given theory does not induce a ground terminating rewriting system. Too bad! Nevertheless it is terminating if we restrict the treatment to expressions that are not diverging, where an Haskell expression e should be called diverging if $\mathcal{E}_H \nvdash \mathcal{F}[\![e]\!]_{\mathsf{whnf}} : \textbf{ExpUExc}$, i.e. the induced rewriting system does not terminate for $\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}}$. But before the sarcastic reader bursts into applause because of this statement saying that the rewriting system terminates if it terminates, it should be pointed out that this notion of diverging expressions will be shown to coincide with the coinductively defined one of [MLJ99] in the upcoming section.

Showing confluence is quite simple as almost all equational sentences preclude every other sentences. There are exceptions like the overlapping of sentences **seq** and **seq** $\lightning$ for the case that $\mathcal{E}_H \vdash \mathcal{F}[\![e_2]\!]_{\mathsf{whnf}} : \textbf{Exceptions}$. But as in this case all of those overlappings are non-critical as they produce equal results or results that are joinable.

3.6 Relation to Existing Semantics

As already mentioned the semantics given here is inspired by the big step operational semantics given in [MLJ99]. And a question might be: Are these two semantics equal in some sense? Well, of course both semantics cannot be equal as the latter's syntax comprises only a small subset of the syntax considered here. Yet on the other side it introduces a primitive — the strict let — that is not part of the rewrite semantics given here. In the following $\mathcal{E}_H$ is used to denote the MEL theory of the Haskell semantics developed in this section.

But, as it will turn out, the strict let and the `seq` can be mutually transformed, such that the syntax in [MLJ99] can be seen as a subset of the syntax considered in this thesis. Thus for examining the relation of these both semantics it is appropriate to restrict the considered syntax to this fragment:

$$\begin{array}{rcl} M, N & ::= & \mid x \mid \lambda x.M \mid M\ N \mid \mathsf{let!}\ x = M\ \mathsf{in}\ N \mid \mathsf{seq}\ M\ N \mid \mathsf{raise}\ M \mid e \\ U, V & ::= & \mid x \mid \lambda x.M \mid e \end{array}$$

Where x ranges over variables and e over exceptions[3]. M and N describe expressions whereas U and V describe values, i.e. whnfs. We will use these meta variables, possibly indexed and/or primed, as ranging over the given domains. The same holds true for $\mathcal{S}$, which will range over finite sets of exceptions (that is, those referred to by e).

This is the full syntax introduced in [MLJ99] plus the additional seq. To be able to examine this primitive a defining equation has to be given:

$$\mathsf{let!}\ x = M\ \mathsf{in}\ N \ := \ \mathsf{seq}\ M\ N[{}^{M}/_{x}] \tag{slet}$$

Where $M[{}^{N}/_{x}]$ denotes the usual substitution of the free occurrences of the variable x by N in M. The notion of free variables $\mathsf{fv}(M)$ is the usual one of lambda terms where strict lets are treated like

$$\mathsf{fv}(\mathsf{let!}\ x = M\ \mathsf{in}\ N) = (\mathsf{fv}(N)\backslash\{x\}) \cup \mathsf{fv}(M).$$

The following fact can be easily established by induction:

Fact 3.2. *Let M,N be arbitrary and $x \notin \mathsf{fv}(M)$. Then the following holds:*

$$M[{}^{N}/_{x}] = M$$

Hence the following can be easily derived from equation (slet):

$$\mathsf{seq}\ M\ N \ = \ \mathsf{let!}\ x = M\ \mathsf{in}\ N \qquad x \notin \mathsf{fv}(N) \tag{slet'}$$

The semantics was given in a different way than it was done here so far. The relation $M \Downarrow V$ given again in figure 3.13 on the facing page was used to describe the convergence of the expression M to the value V. On the other hand an exceptional convergence relation $M \uparrow e$ was defined coinductively, hence including divergence. But as the executable rewrite semantics is limited to converging programs the focus here is rather on the relation $M \nearrow \mathcal{S}$ of exceptional convergence for the expression M and the set of exceptions $\mathcal{S}$ given here again in figure 3.14 on the next page.

As stated in theorem 3.1 and lemma 3.2 in [MLJ99] these two relations describe exactly the semantics of converging — that is, terminating — expressions. So they can be taken as the basis for a comparison. All other expressions not (exceptionally) converging are diverging — in symbol $M \Uparrow$ — and vice versa. Hence $M \not\Uparrow$ holds if M (exceptionally) converges.

Unfortunately this semantics additionally introduced a new symbol $\mathbb{0}$ to define the semantics of the strict let. If this additional symbol is in fact necessary the two semantics cannot be equal — at least in the definition for seq / let!. But as it will turn out the additional symbol $\mathbb{0}$ is dispensable.

The following trivial fact stating that the value of an expression does not introduces new free variables can easily be proven by induction.

Fact 3.3. *Let M and V be expressions such that $M \Downarrow V$ and $x \notin \mathsf{fv}(M)$, then $x \notin \mathsf{fv}(V)$.*

[3]In this treatment exceptions and exception expressions were confused, but in the context the meaning will be clear.

$$\frac{}{V \Downarrow V}\ (\text{Value}_{\Downarrow}) \qquad \frac{M \Downarrow \lambda x.M' \qquad M'[^{N}/_{x}] \Downarrow V}{M\ N \Downarrow V}\ (\text{App}_{\Downarrow})$$

$$\frac{M \Downarrow U \qquad N[^{U}/_{x}] \Downarrow V}{\textsf{let!}\ x = M\ \textsf{in}\ N \Downarrow V}\ (\text{Strict Let}_{\Downarrow})$$

Fig. 3.13: Rules defining the convergence relation $\Downarrow$. From [MLJ99].

$$\frac{}{\mathbb{0} \nearrow \emptyset}\ (\text{Stuck}_{\nearrow}) \qquad \frac{M \Downarrow e}{M\ N \nearrow \{\,\texttt{TypeError}\,\}}\ (\text{App}_{\nearrow_0}) \qquad \frac{M \nearrow \mathcal{S} \qquad N \Downarrow V}{M\ N \nearrow \mathcal{S}}\ (\text{App}_{\nearrow_1})$$

$$\frac{M \Downarrow \lambda x.M_0 \qquad M_0[^{N}/_{x}] \nearrow \mathcal{S}}{M\ N \nearrow \mathcal{S}}\ (\text{App}_{\nearrow_2}) \qquad \frac{M \nearrow \mathcal{S}_1 \qquad N \nearrow \mathcal{S}_2}{M\ N \nearrow \mathcal{S}_1 \cup \mathcal{S}_2}\ (\text{App}_{\nearrow_3})$$

$$\frac{M \Downarrow \lambda x.N}{\textsf{raise}\ M \nearrow \{\,\texttt{TypeError}\,\}}\ (\text{Raise}_{\nearrow_0}) \qquad \frac{M \Downarrow e}{\textsf{raise}\ M \nearrow \{e\}}\ (\text{Raise}_{\nearrow_1}) \qquad \frac{M \nearrow \mathcal{S}}{\textsf{raise}\ M \nearrow \mathcal{S}}\ (\text{Raise}_{\nearrow_2})$$

$$\frac{M \nearrow \mathcal{S} \qquad N[^{\mathbb{0}}/_{x}] \Downarrow V}{\textsf{let!}\ x = M\ \textsf{in}\ N \nearrow \mathcal{S}}\ (\text{Strict Let}_{\nearrow_1}) \qquad \frac{M \Downarrow V \qquad N[^{V}/_{x}] \nearrow \mathcal{S}}{\textsf{let!}\ x = M\ \textsf{in}\ N \nearrow \mathcal{S}}\ (\text{Strict Let}_{\nearrow_2})$$

$$\frac{M \nearrow \mathcal{S}_1 \qquad N[^{\mathbb{0}}/_{\mathcal{S}_2}] \nearrow}{\textsf{let!}\ x = M\ \textsf{in}\ N \nearrow \mathcal{S}_1 \cup \mathcal{S}_2}\ (\text{Strict Let}_{\nearrow_3})$$

Fig. 3.14: Rules defining the exceptional convergence relation $\nearrow$. From [MLJ99].

This can be used to get the following:

Corollary 3.4. *Let N be a closed expression and $M[{}^{N}\!/_{x}] \Downarrow \lambda x.M_0$, then*

$$M_0\left[{}^{N'[{}^{N''}\!/_{x}]}\!/_{y}\right] = M_0[{}^{N'}\!/_{y}][{}^{N''}\!/_{x}]$$

Proof. As $x \notin \mathsf{fv}(M[{}^{N}\!/_{x}])$ also $x \notin \mathsf{fv}(\lambda x.M_0)$ by fact 3.3. Hence $x \notin \mathsf{fv}(M_0)$. Now the stated equality can be easily proven by induction on M_0. $\square$

In the following lots of inductive proofs are necessary which need of course a well-founded order. The following proof tree orders will serve very well for this purpose:

Definition 3.5. $M >_{\Downarrow} N$ holds for two converging expressions M and N iff there are U and V such that there is a rule in figure 3.13 having $M \Downarrow U$ as its consequence and $N \Downarrow V$ in its premise. $>_{\nearrow}$ is defined analogously for the rules of $\nearrow$.

As proof trees for $\Downarrow$ and $\nearrow$ for converging expressions are finite, both $>_{\Downarrow}$ and $>_{\nearrow}$ do not induce infinite chains. Hence, assuming the axiom of choice, both are well-founded.

Now the following lemma states a kind of replacement invariance for expressions having the same value in the context of expressions in the (exceptional) convergence relation.

Lemma 3.6. *Let N and N' be expressions such that $N \Downarrow V'$ and $N' \Downarrow V'$ holds for some value V'. Then the following also holds true for all M, V, x and $\mathcal{S}$:*

$$\text{(i)}\ \ M[{}^{N}\!/_{x}] \Downarrow V \quad \text{iff} \quad M[{}^{N'}\!/_{x}] \Downarrow V$$

$$\text{(ii)}\ \ M[{}^{N}\!/_{x}] \nearrow \mathcal{S} \quad \text{iff} \quad M[{}^{N'}\!/_{x}] \nearrow \mathcal{S}$$

Proof. Straightforward inductions on M by $>_{\Downarrow}$ and $>_{\nearrow}$ respectively using corollary 3.4 for (i) and using (i) for (ii). $\square$

The next lemma simply states that convergence of expressions is independent of non-converging subexpressions. Note that $M \not\Downarrow$ means that $M \not\Downarrow V$ for all V, that is, M does not converge.

Lemma 3.7. *Let N be an expression such that $N \not\Downarrow$ and N' and x arbitrary. Then the following holds:*

$$\text{if} \quad M[{}^{N}\!/_{x}] \Downarrow V \quad \text{then} \quad M[{}^{N'}\!/_{x}] \Downarrow V$$

Proof. Straightforward induction on M by $>_{\Downarrow}$. $\square$

The following is then an immediate corollary of lemma 3.7:

Corollary 3.8. *Let M be an expression such that $M \nearrow \mathcal{S}$ for some set of exceptions $\mathcal{S}$ then the following holds:*

$$N[{}^{0}\!/_{x}] \Downarrow V \quad \text{iff} \quad N[{}^{M}\!/_{x}] \Downarrow V$$

A similar result can be proven for exceptional convergence:

Lemma 3.9. *Let M be an expression such that $M \nearrow \mathcal{S}_1$ for some set of exceptions $\mathcal{S}_1$, then there are some sets of exceptions $\mathcal{S}_2, \mathcal{S}_3$ satisfying $\mathcal{S}_2 \subseteq \mathcal{S}_3 \subseteq \mathcal{S}_1 \cup \mathcal{S}_2$ such that the following holds:*

$$N[{}^{0}\!/_{x}] \nearrow \mathcal{S}_2 \quad \text{iff} \quad N[{}^{M}\!/_{x}] \nearrow \mathcal{S}_3$$

Proof. Again this is an easy proof by induction on N using $>_{\nearrow}$. As an example the argument for the case $N = M'\ N'$ is given below:

$$\begin{aligned} & (M'\ N')[{}^{0}\!/_{x}] \nearrow \mathcal{S}_2 \\ \iff\ & (M'[{}^{0}\!/_{x}])\ (N'[{}^{0}\!/_{x}]) \nearrow \mathcal{S}_2 && \text{(subst.)} \end{aligned}$$

Here four different cases must be considered:

Case 1 (App$_{\nearrow_0}$) $M'[{}^{\mathbb{0}}\!/_{x}] \Downarrow e,\ \ \mathcal{S}_2 = \{\,\texttt{TypeError}\,\}$

$$
\begin{array}{llr}
\Longleftrightarrow & \underbrace{M'[{}^{M}\!/_{x}] \Downarrow e}_{(*)} & \text{(cor. 3.8)}\\
\Longleftrightarrow & (M'[{}^{M}\!/_{x}])\ (N'[{}^{M}\!/_{x}]) \nearrow \{\,\texttt{TypeError}\,\},\ \ (*) & (\text{App}_{\nearrow_0})\\
\Longleftrightarrow & (M'\ N')[{}^{M}\!/_{x}] \nearrow \{\,\texttt{TypeError}\,\},\ \ (*) & \text{(subst.)}\\
 & \text{Hence: } \{\,\texttt{TypeError}\,\} \subseteq \{\,\texttt{TypeError}\,\} \subseteq \{\,\texttt{TypeError}\,\} \cup \mathcal{S}_1 &
\end{array}
$$

Case 2 (App$_{\nearrow_1}$) $M'[{}^{\mathbb{0}}\!/_{x}] \nearrow \mathcal{S}_2,\ \ N'[{}^{\mathbb{0}}\!/_{x}] \Downarrow V$

$$
\begin{array}{llr}
\Longleftrightarrow & \underbrace{M'[{}^{M}\!/_{x}] \nearrow \mathcal{S}_3,\ \ N'[{}^{M}\!/_{x}] \Downarrow V}_{(*)} & \text{(I.H., cor. 3.8)}\\
\Longleftrightarrow & (M'[{}^{M}\!/_{x}])\ (N'[{}^{M}\!/_{x}]) \nearrow \mathcal{S}_3,\ \ (*) & (\text{App}_{\nearrow_1})\\
\Longleftrightarrow & (M'\ N')[{}^{M}\!/_{x}] \nearrow \mathcal{S}_3,\ \ (*) & \text{(subst.)}
\end{array}
$$

By the step using the I.H. we have: $\mathcal{S}_2 \subseteq \mathcal{S}_3 \subseteq \mathcal{S}_1 \cup \mathcal{S}_2$.

Case 3 (App$_{\nearrow_2}$) $M'[{}^{\mathbb{0}}\!/_{x}] \Downarrow \lambda y.M_0,\ \ M_0[{}^{N'[{}^{\mathbb{0}}\!/_{x}]}\!/_{y}] \nearrow \mathcal{S}_2$

$$
\begin{array}{llr}
\Longleftrightarrow & M'[{}^{M}\!/_{x}] \Downarrow \lambda y.M_0,\ \ M_0[{}^{N'}\!/_{y}][{}^{\mathbb{0}}\!/_{x}] \nearrow \mathcal{S}_2 & \text{(cor. 3.8, cor. 3.4)}\\
\Longleftrightarrow & M'[{}^{M}\!/_{x}] \Downarrow \lambda y.M_0,\ \ M_0[{}^{N'}\!/_{y}][{}^{M}\!/_{x}] \nearrow \mathcal{S}_3 & \text{(I.H.)}\\
\Longleftrightarrow & \underbrace{M'[{}^{M}\!/_{x}] \Downarrow \lambda y.M_0,\ \ M_0[{}^{N'[{}^{M}\!/_{x}]}\!/_{y}] \nearrow \mathcal{S}_3}_{(*)} & \text{(cor. 3.4)}\\
\Longleftrightarrow & (M'[{}^{M}\!/_{x}])\ (N'[{}^{M}\!/_{x}]) \nearrow \mathcal{S}_3,\ \ (*) & (\text{App}_{\nearrow_2})\\
\Longleftrightarrow & (M'\ N')[{}^{M}\!/_{x}] \nearrow \mathcal{S}_3,\ \ (*) & \text{(subst.)}
\end{array}
$$

By the step using the I.H. we have: $\mathcal{S}_2 \subseteq \mathcal{S}_3 \subseteq \mathcal{S}_1 \cup \mathcal{S}_2$.

Case 4 (App$_{\nearrow_3}$) $M'[{}^{\mathbb{0}}\!/_{x}] \nearrow \mathcal{S},\ \ N'[{}^{\mathbb{0}}\!/_{x}] \nearrow \mathcal{S}',\ \ \mathcal{S}_2 = \mathcal{S} \cup \mathcal{S}'$

$$
\begin{array}{llr}
\Longleftrightarrow & \underbrace{M'[{}^{M}\!/_{x}] \nearrow \mathcal{S}'',\ \ N'[{}^{M}\!/_{x}] \nearrow \mathcal{S}'''}_{(*)} & \text{(I.H.)}\\
\Longleftrightarrow & (M'[{}^{M}\!/_{x}])\ (N'[{}^{M}\!/_{x}]) \nearrow \mathcal{S}_3 = \mathcal{S}'' \cup \mathcal{S}''',\ \ (*) & (\text{App}_{\nearrow_3})\\
\Longleftrightarrow & (M'\ N')[{}^{M}\!/_{x}] \nearrow \mathcal{S}_3,\ \ (*) & \text{(subst.)}
\end{array}
$$

By the step using the I.H. we have both $\mathcal{S}' \subseteq \mathcal{S}''' \subseteq \mathcal{S}_1 \cup \mathcal{S}'$ and $\mathcal{S} \subseteq \mathcal{S}'' \subseteq \mathcal{S}_1 \cup \mathcal{S}$. By monotonicity we get $\underbrace{\mathcal{S} \cup \mathcal{S}'}_{=\mathcal{S}_2} \subseteq \underbrace{\mathcal{S}'' \cup \mathcal{S}'''}_{=\mathcal{S}_3} \subseteq \mathcal{S}_1 \cup \underbrace{\mathcal{S} \cup \mathcal{S}'}_{=\mathcal{S}_2}$.

The $(*)$ has to be carried through the respective argument chains since otherwise they would be only implications rather that equivalences. Checking the respective side conditions $(*)$ at the end of each argument yields that they cover all cases. Hence also the $\Leftarrow$ direction is proven. $\square$

The following lemma finally states that the additional element $\mathbb{0}$ can be omitted

Lemma 3.10. *The set of rules in figure 3.14 is equivalent to the set of rules that is gained by removing the rules (Stuck$_{\nearrow}$), (Strict Let$_{\nearrow_1}$) and (Strict Let$_{\nearrow_3}$) and adding the following rules:*

$$
\frac{M \nearrow \mathcal{S} \qquad N[{}^{M}\!/_{x}] \Downarrow V}{\textsf{let!}\ x = M\ \textsf{in}\ N \nearrow \mathcal{S}}\ \textit{(Strict Let}'_{\nearrow_1}\textit{)}
\qquad
\frac{M \nearrow \mathcal{S}_1 \qquad N[{}^{M}\!/_{x}] \nearrow \mathcal{S}_2'}{\textsf{let!}\ x = M\ \textsf{in}\ N \nearrow \mathcal{S}_1 \cup \mathcal{S}_2'}\ \textit{(Strict Let}'_{\nearrow_3}\textit{)}
$$

$$
\begin{aligned}
\overline{x} &= \mathtt{x} && (\overline{\text{var}})\\
\overline{\lambda x.M} &= \mathtt{\backslash}\,\overline{x}\,\mathtt{->}\,\overline{M} && (\overline{\text{lambda}})\\
\overline{M\ N} &= \overline{M}\ \overline{N} && (\overline{\text{app}})\\
\overline{\mathsf{let!}\ x = M\ \mathsf{in}\ N} &= \overline{\mathsf{seq}\ M\ N[{}^{M}/_{x}]} && (\overline{\text{seq}})\\
\overline{\mathsf{seq}\ M\ N} &= \mathtt{seq}\ \overline{M}\ \overline{N} && (\overline{\text{let!}})\\
\overline{\mathsf{raise}\ M} &= \mathtt{raise}\ \overline{M} && (\overline{\text{raise}})\\
\overline{\mathtt{TypeError}} &= \mathtt{TypeError} && (\overline{\text{exc}})\\
&= \langle\,\mathtt{TypeError}\,\rangle_{\lightning} && (\overline{\text{exc'}})\\
\overline{\{e_1,\dots,e_n\}} &= \{\overline{e_1},\dots,\overline{e_n}\} && (\overline{\text{excs}})
\end{aligned}
$$

Fig. 3.15: Defining equations for the translation into MEL terms.

Proof. By corollary 3.8 the premises of the rules (Strict Let$_{\nearrow_1}$) and (Strict Let$'_{\nearrow_1}$) are equivalent. The same holds true for the rules (Strict Let$_{\nearrow_3}$) and (Strict Let$'_{\nearrow_3}$) by lemma 3.9. Yet, unlike for (Strict Let$_{\nearrow_1}$) and (Strict Let$'_{\nearrow_1}$), the consequences seem to be different, as $\mathcal{S}_2$ and $\mathcal{S}'_2$ are in general not equal. But also by lemma 3.9 the inclusions $\mathcal{S}_2 \subseteq \mathcal{S}'_2 \subseteq \mathcal{S}_1 \cup \mathcal{S}_2$ hold, which implies by idempotency and monotonicity of the set union $\mathcal{S}_1 \cup \mathcal{S}_2 \subseteq \mathcal{S}_1 \cup \mathcal{S}'_2 \subseteq \mathcal{S}_1 \cup \mathcal{S}_2$. Hence $\mathcal{S}_1 \cup \mathcal{S}_2 = \mathcal{S}_1 \cup \mathcal{S}'_2$, which makes the respective consequences of (Strict Let$_{\nearrow_3}$) and (Strict Let$'_{\nearrow_3}$) equivalent. Consequently the rules (Strict Let$'_{\nearrow_1}$) and (Strict Let$'_{\nearrow_3}$) compensate for (Strict Let$_{\nearrow_1}$) and (Strict Let$_{\nearrow_3}$), respectively. Since $\mathbb{0}$ is not referred by the remaining rules and as it is not part of the desired syntax, the rule (Stuck$_{\nearrow}$) is dispensable. □

Before comparing both semantics there is sill a problem to be solved. Until now there is no connection neither between the Haskell expressions referred to by the variables M and N and the MEL terms of sort **Exp** nor between sets of exceptions (i.e. those described by the variable e) and MEL terms of sort **Exceptions**. Therefore an overloaded meta symbol $\overline{\cdot}$ is introduced that describes the MEL term correspondent to a Haskell expression or a set of exceptions respectively. That is, $\mathcal{E}_H \vdash \overline{M} : \textbf{Exp}$ and $\mathcal{E}_H \vdash \overline{\mathcal{S}} : \textbf{Exceptions}$. The inductive definition of $\overline{\cdot}$ is given in figure 3.15.

Please note that strictly speaking $\overline{\cdot}$ is not a well-defined function, as there is no unique image for exceptions, which can be seen in the ambiguous definition of $\overline{\mathtt{TypeError}}$. The reason for this is that in this treatment there was made a distinction between exceptions as a data type in Haskell programs and exceptions as a denotation for Haskell expressions. This distinction was not made in the semantic treatment that is referred to here. So we take the liberty of using the implicit distinction that was assumed there and take the corresponding definition of $\overline{e}$ that fits into the considered context. For example the identity ($\overline{\text{excs}}$) in figure 3.15 refers to the latter, ($\overline{\text{exc'}}$).

The following lemma relates (the informally introduced) substitutions on Haskell expressions to the MEL defined application of simple substitutions and the union of sets to the union of sets of exceptions as defined in $\mathcal{E}_H$.

Lemma 3.11.

(i) $\mathcal{E}_H \vdash \overline{M[{}^{N}/_{x}]} = \overline{M}[\{\overline{x} \mapsto \overline{N}\}]$

(ii) $\mathcal{E}_H \vdash \overline{\mathcal{S} \cup \mathcal{S}'} = \overline{\mathcal{S}} \cup \overline{\mathcal{S}'}$

Proof. (i) Straightforward induction on the structure of M.

(ii) Easy equational argument; in abbreviated form:

$$
\begin{aligned}
\overline{\mathcal{S} \cup \mathcal{S}'} &\overset{\text{finite sets}}{=} \overline{\{e_1,\dots,e_n\} \cup \{e'_1,\dots,e'_m\}} \overset{\text{union}}{=} \overline{\{e_1,\dots,e_n,e'_1,\dots,e'_m\}}\\
&\overset{\overline{\text{excs}}}{=} \{\overline{e_1},\dots,\overline{e_n},\overline{e'_1},\dots,\overline{e'_m}\} \overset{\cdot \cup \cdot}{=} \{\overline{e_1},\dots,\overline{e_n}\} \cup \{\overline{e'_1},\dots,\overline{e'_m}\} \overset{\overline{\text{excs}}}{=} \overline{\mathcal{S}} \cup \overline{\mathcal{S}'}
\end{aligned}
$$

□

Before considering the semantics it has to be verified that both sides agree upon their notion of normal forms (or values).

Lemma 3.12.

$$\exists V.M \equiv V \quad \textit{iff} \quad \mathcal{E}_H \vdash \overline{M} \Downarrow_{\mathsf{whnf}} = \mathsf{true}$$

Proof. Easy induction on the structure of M. □

Note that $\exists V.M \equiv V$ truly means that M is a value.
Finally the goal theorem can be stated.

Theorem 3.13. *Let $\mathcal{E}_H$ be the semantic MEL theory for pure Haskell. Then the following equivalences hold true:*

(i) $M \Downarrow V \quad \textit{iff} \quad \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{M}\right]\!\right]_{\mathsf{whnf}} = \overline{V}$

(ii) $M \nearrow \mathcal{S} \quad \textit{iff} \quad \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{M}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}}$

Proof. Using lemma 3.12 the statement (i) can be easily established by induction on $>_{\Downarrow}$ on M. Using this result also (ii) can be proven by induction on $>_{\nearrow}$ on M. As an example the argument for the the case $M = \mathsf{let!}\ x = M'\ \mathsf{in}\ N'$ is given below:

$$\mathsf{let!}\ x = M'\ \mathsf{in}\ N' \nearrow \mathcal{S}$$

Here three different cases have to be considered:

Case 1 (Strict Let$'_{\nearrow_1}$) $M' \nearrow \mathcal{S},\ N'[{}^{M'}\!/_x] \Downarrow V$

$$\iff \underbrace{\mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{M'}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}},\ \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{N'[{}^{M'}\!/_x]}\right]\!\right]_{\mathsf{whnf}} = \overline{V}}_{(*)} \qquad \text{(I.H., (i))}$$

$$\iff \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{\mathsf{seq}\ M'\ N'[{}^{M'}\!/_x]}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}} \cup \overline{V},\ (*) \qquad (\textbf{seq}\ \lightning)$$

$$\iff \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{\mathsf{let!}\ x = M'\ \mathsf{in}\ N'}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}},\ (*) \qquad (\overline{\text{let!}}, \textbf{excs}\cup\textbf{exp})$$

Case 2 (Strict Let$_{\nearrow_2}$) $M' \Downarrow V,\ N'[{}^{V}\!/_x] \nearrow \mathcal{S}$

$$\iff M' \Downarrow V,\ N'[{}^{M'}\!/_x] \nearrow \mathcal{S} \qquad \text{(lem. 3.6)}$$

$$\iff \underbrace{\mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{M'}\right]\!\right]_{\mathsf{whnf}} = \overline{V},\ \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{N'[{}^{M'}\!/_x]}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}}}_{(*)} \qquad \text{((i), I.H.)}$$

$$\iff \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{\mathsf{seq}\ M'\ (N'[{}^{M'}\!/_x])}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}},\ (*) \qquad (\textbf{seq/seq}\ \lightning)$$

$$\iff \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{\mathsf{let!}\ x = M'\ \mathsf{in}\ N'}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}},\ (*) \qquad (\overline{\text{let!}})$$

Case 3 (Strict Let$_{\nearrow_3}$) $M' \nearrow \mathcal{S}_1,\; N'[{}^{M'}\!/x] \nearrow \mathcal{S}_2,\; \mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$

$$\iff \underbrace{\mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{M'}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}_1},\;\; \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{N'[{}^{M'}\!/x]}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}_2},\;\; \mathcal{E}_H \vdash \overline{\mathcal{S}} = \overline{\mathcal{S}_1} \cup \overline{\mathcal{S}_2}}_{(*)}$$

$$\text{(I.H., lemma 3.11)}$$

$$\iff \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{\mathsf{seq}\ M'\ (N'[{}^{M'}\!/x])}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}_1} \cup \overline{\mathcal{S}_2},\;\; (*) \qquad (\textbf{seq}\ \lightning)$$

$$\iff \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{\mathsf{let!}\ x = M'\ \mathsf{in}\ N'}\right]\!\right]_{\mathsf{whnf}} = \overline{\mathcal{S}},\;\; (*) \qquad (\overline{\text{let!}})$$

The $(*)$ has to be carried through the respective argument chains since otherwise they would be only implications rather that equivalences. Checking the respective side conditions $(*)$ at the end of each argument yields that all cases are covered. Hence also the $\Leftarrow$ direction is proven. □

The following is now an easy consequence of the previous theorem and the results in [MLJ99] and states how diverging expressions are characterised by our semantics.

Corollary 3.14.

$$M \Uparrow \quad \textit{iff} \quad \mathcal{E}_H \nvdash \mathcal{F}\left[\!\left[\overline{M}\right]\!\right]_{\mathsf{whnf}} : \mathbf{ExpUExc}$$

Proof.

$$\begin{aligned}
& M \Uparrow \\
\iff\ & \exists e. M \uparrow e,\;\; \neg\exists S. M \nearrow \mathcal{S} && \text{(lem. 3.2 in [MLJ99])}\\
\iff\ & \neg\exists V. M \Downarrow V,\;\; \neg\exists S. M \nearrow \mathcal{S} && \text{(lem. 3.1 in [MLJ99])}\\
\iff\ & \neg\exists V.\, \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{M}\right]\!\right]_{\mathsf{whnf}} = \overline{V},\;\; \neg\exists \mathcal{S}.\, \mathcal{E}_H \vdash \mathcal{F}\left[\!\left[\overline{M}\right]\!\right]_{\mathsf{whnf}} = \overline{S} && \text{(thm. 3.13)}\\
\iff\ & \mathcal{E}_H \nvdash \mathcal{F}\left[\!\left[\overline{M}\right]\!\right]_{\mathsf{whnf}} : \mathbf{Exp},\;\; \mathcal{E}_H \nvdash \mathcal{F}\left[\!\left[\overline{M}\right]\!\right]_{\mathsf{whnf}} : \mathbf{Exceptions} && \text{(lem. 3.1, 3.12)}\\
\iff\ & \mathcal{E}_H \nvdash \mathcal{F}\left[\!\left[\overline{M}\right]\!\right]_{\mathsf{whnf}} : \mathbf{ExpUExc}
\end{aligned}$$

□

Yet, as discussed in section 3.5 $\mathcal{E}_H$ is not executable for diverging Haskell expressions, that is, in particular for all $\overline{M}$ for which $M \Uparrow$ holds. Hence — what should not be too surprising — $\mathcal{E}_H$ does not enable to decide divergence of Haskell expressions.

4 Concurrency Semantics

Now that the pure functional part of the semantic rewrite theory — which is actually just a MEL theory $\mathcal{E}_H$ so far — is complete, the Concurrent Haskell extension can be defined. The concurrent extension discussed here is based on [MJMR01], where it is introduced in detail; so it will not be necessary to get into too much detail about it here. Instead the focus is set on how to translate the operational semantics given there into a rewrite theory. For this purpose we extend the MEL theory $\mathcal{E}_H$ of the previous section to the theory $\mathcal{E}_C$ which introduces additional auxiliary functions and also extends the functional semantics to include the small layer of functional semantics of the Concurrent Haskell extension. On top of this the GRT $\mathcal{R}_C$ will be presented that includes $\mathcal{E}_C$ and defines the concurrent part of the semantics of Concurrent Haskell. But firstly, as before, the considered syntax has to be described.

4.1 Syntax of the Concurrent Extension

The concurrent extension for Haskell introduces the `IO` monad. All concurrency is done inside this monad, that is, all primitives that are introduced have their result type in this monad. So the syntactic extension spans only the concurrent primitives plus the monad operations. To allow to present Concurrent Haskell programs more concisely also the do notation is included as syntactic sugar.

$$\begin{array}{rcl}
\textit{DoAtom} & ::= & \textit{Exp} \mid \textit{Pat}\,\texttt{<-}\,\textit{Exp} \\
\textit{DoBlock} & ::= & \textit{DoAtom} \mid \textit{DoBlock}\ \texttt{;}\ \textit{DoAtom} \\
\textit{ExPrim} & ::= & \texttt{putChar} \mid \texttt{getChar} \mid \texttt{putMVar} \mid \texttt{takeMVar} \mid \texttt{newEmptyMVar} \mid \texttt{sleep} \mid \texttt{return} \\
 & & \mid \texttt{throw} \mid \texttt{catch} \mid \texttt{throwTo} \mid \texttt{block} \mid \texttt{unblock} \mid \texttt{myThreadId} \mid \texttt{forkIO} \\
\textit{Primitive} & ::= & \ldots \mid \textit{ExPrim} \\
\textit{BinOp} & ::= & \ldots \mid \texttt{>>=} \mid \texttt{>>} \\
\textit{Exp} & ::= & \ldots \mid \texttt{do \{}\,\textit{DoBlock}\,\texttt{\}}
\end{array}$$

Formulating the syntax extension as an extension to the existing MEL theory is now quite simple. The sorts $\mathbf{DoAtom}\{da_i\}$, $\mathbf{DoBlock}\{db_i\}$ and $\mathbf{ExPrim}\{ep_i\}$ corresponding to the syntax variables mentioned above need to be introduced and defined as follows:

$$\mathbf{Exp} < \mathbf{DoAtom} < \mathbf{DoBlock}$$
$$\mathbf{ExPrim} < \mathbf{Primitive}$$

$$\begin{array}{rll}
\cdot\ \texttt{<-}\ \cdot : & \mathbf{Pat}\ \ \mathbf{Exp}\ \rightarrow\ \mathbf{DoAtom} & \\
\cdot\ \texttt{;}\ \cdot : & \mathbf{DoBlock}\ \ \mathbf{DoBlock}\ \rightarrow\ \mathbf{DoBlock} & \\
\texttt{do \{}\ \cdot\ \texttt{\}} : & \mathbf{DoBlock}\ \rightarrow\ \mathbf{Exp} & \\
p : & \mathbf{ExPrim} & (p \in \textit{ExPrim})
\end{array}$$

Please note, that, since by extending the bnf syntax definition as above in particular also the syntactic category of *BinOp* was extended by `>>=` and `>>`, the operator declarations and MEL sentences (meta-)quantified over the set *BinOp* automatically yield the corresponding operator declarations and MEL sentences (free variables, substitution application) for these two operators.

4.2 Semantics of the Concurrent Extension

The approach taken here to describe the concurrent semantics is nearly the same as the one introduced in [MJMR01]. So the plan is to develop a equationally defined sort of program states and to define transitions between them by rewrite sentences. Please observe the conceptual difference of this use of MEL to the one applied for defining pure Haskell's semantics. There, MEL was primarily used to define functions[1].

Now we will see how equality can be used to enable rewriting on a particular structure. Due to the rule (Eq) of the GRL semantics rewriting is done modulo the equational theory. That is, we can define structural properties by MEL and then exploit them when defining rewriting sentences. For example if we would consider a structure that should represent a finite set of some elements, say, natural numbers, we would define associativity, commutativity and idempotency sentences in MEL. Furthermore let us assume that we want to rewrite such a set by choosing an element of particular shape, say, even natural numbers to their successor. Now we can assume w.l.o.g. that such an element is at the first position of the set, because if it were not it could be move to the first position by using the equational sentence establishing commutativity. Hence the following rewrite sentence would have the desired semantics:

$$\frac{\mathsf{even}(n_1) = \mathsf{true}}{\{n_1, rest_1\} \;\longrightarrow\; \{\mathsf{s}(n_1), rest_1\}}$$

By using the MEL defined equality stating the commutativity, we could derive a rewrite that is done somewere in the middle of the set:

$$\{1, 2, 4, 9, 10\} \;\longrightarrow\; \{1, 2, 5, 9, 10\}$$

as $\{1, 2, 4, 9, 10\}$ is equal to $\{4, 1, 2, 9, 10\}$ which can be rewritten to $\{\mathsf{s}(4), 1, 2, 9, 10\}$ which is in turn equal to $\{1, 2, 5, 9, 10\}$. Please observe the different kinds of referring to the MEL subtheory in this example: Firstly, an implicit one by the (Eq) rule of GRL's semantics, which was mentioned above, and secondly, an explicit one by incorporating an equational condition in the rewriting sentence, in the example that was $\mathsf{even}(n_1) = \mathsf{true}$. But note that the implicit referring is not restricted to structural equalities; it is essential when equationally defined functions are used in the right-hand side term of the rewrite sentence like the use of $\mathsf{s}(n_1)$ in the example[2].

To describe program states a modular technique presented in [MB03] is used here. Particularly the extensible record structures introduced there will be used in the following.

For this purpose a rewrite theory $\mathcal{R}_{Rec}$ outside our semantic theory $\mathcal{R}_C$ defined so far will be given, describing this record structures so that we can include this theory later into $\mathcal{R}_C$ several times. The theory $\mathcal{R}_{Rec}$ is given in figure 4.1 on the next page; it also assumes the presence of the sort **Bool**.

So a record structure is a set containing fields each of which consists of an index describing the kind of data it represents and a component, the actual data. As the sort of fields can be freely extended by defining a constant of sort **Index** and adding a sort to its component by a simple membership sentence, the sort **Record** is highly extensible. Moreover sentences that only use a certain set of those fields keep their meaning — if properly defined — also for an extended set of fields, such that language features can be easily added without touching the sentences already given for the original language.

[1]Well, actually there were also some structural equational sentences, like those defining lists of expressions associative, or — implicitly assumed — associativity, commutativity and idempotency for sets, or something more differently the respective equalities of trivial lambda abstractions and trivial defined matches (i.e. those having an empty list of argument patterns). Nevertheless those structural equalities are just introduced to make the equational function definitions more concise. Hence there are on the same level, that is, MEL.

[2]In the general case this would also apply to equationally defined function symbols used in the left-hand side term. But note that this would immediately destroy the coherence of the theory if no additional rewrite sentence is added which covers the result of the function used on the left-hand side. But this would make the first rewrite sentence, that contains the function symbol, redundant. Hence for executability this scenario does not occur.

$$\text{Sorts:}\quad \mathbf{Index}\{in_i\}, \mathbf{Component}\{co_i\}, \mathbf{Field}, \mathbf{PreRecord}\{pr_i\}, \mathbf{Record}$$

$$\mathbf{Field} < \mathbf{PreRecord}$$

$$\begin{aligned}
\mathsf{null} &: \mathbf{PreRecord}\\
\cdot\ \cdot &: \mathbf{PreRecord}\ \mathbf{PreRecord} \rightarrow \mathbf{PreRecord}\\
\cdot : \cdot &: [\mathbf{Index}]\ [\mathbf{Component}] \rightarrow [\mathbf{Field}]\\
\{\cdot\} &: [\mathbf{PreRecord}] \rightarrow [\mathbf{Record}]\\
\mathsf{duplicated?} &: \mathbf{PreRecord} \rightarrow \mathbf{Bool}
\end{aligned}$$

$$\frac{}{pr_1\ (pr_2\ pr_3) = (pr_1\ pr_2)\ pr_3}\ \textbf{assoc} \qquad \frac{}{pr_1\ pr_2 = pr_2\ pr_1}\ \textbf{comm} \qquad \frac{}{pr_1\ \mathsf{null} = pr_1}\ \textbf{id}$$

$$\frac{}{\mathsf{duplicated?}((in_1 : co_1)\ (i_1 : co_2)\ pr_1) = \mathsf{true}}\ \textbf{dup1} \qquad \frac{\mathsf{otherwise}}{\mathsf{duplicated?}(pr_1) = \mathsf{false}}\ \textbf{dup2}$$

$$\frac{\neg\mathsf{duplicated?}(pr_1) = \mathsf{true}}{\{pr_1\} : \mathbf{Record}}\ \textbf{rec}$$

Fig. 4.1: Definition of $\mathcal{R}_{Rec}$, the theory of a record structure.

So let us switch back to our original theory $\mathcal{R}_C$. The program state can now be defined as a record structure by including the theory $\mathcal{R}_{Rec}$ with the following renamings:

$$\begin{aligned}
\mathbf{PreRecord} &\mapsto \mathbf{PreState}\{ps_i\},\\
\mathbf{Record} &\mapsto \mathbf{State}\{s_i\},\\
\mathbf{Field} &\mapsto \mathbf{SField}\{sf_i\},\\
\mathbf{Component} &\mapsto \mathbf{SComp}\{sco_i\},\\
\mathbf{Index} &\mapsto \mathbf{SIndex}\{si_i\}\\
\{\cdot\} &\mapsto \{\!\!\{ \cdot \}\!\!\}
\end{aligned}$$

Now the state record structure needs to be populated with appropriate fields necessary to define the semantics. At first two fields are needed for representing the output and input of a program. The necessary definitions are given in figures 4.2 and 4.3 on the following page.

Note that incorporating input and output into the program state departs our notion of states from the one used in [MJMR01] where this was considered as an external behaviour. So the states defined here somewhat extend the notion of a program state to a state of the hole computational environment. For both input and output there are functions hasNext respectively full to check whether the corresponding facility can be used, i.e. whether the input is nonempty respectively the output is not full. For the current design of the output facility this consideration is not necessary as an unbounded output buffer is described. Yet as we aim for a general abstract description of input and output this is included, such that for later language designs this can be used to define different behaviour or if for example the output of one program should be connected to the input of another program this connection could be made synchronised using an alternative output structure.

$$\text{Sorts:}\quad \mathbf{Inp}\{\mathit{inp}_i\}$$

$$\mathbf{Inp} < \mathbf{SComp}$$

$$\begin{aligned}
\mathsf{in} &: \mathbf{SIndex}\\
\langle\,\cdot\,\rangle_{\mathsf{I}} &: \mathbf{String} \;\rightarrow\; \mathbf{Inp}\\
\mathsf{hasNext}(\,\cdot\,) &: \mathbf{Inp} \;\rightarrow\; \mathbf{Bool}\\
\mathsf{getNext}(\,\cdot\,) &: [\mathbf{Inp}] \;\rightarrow\; [\mathbf{CharConst}]\\
\mathsf{skip}(\,\cdot\,) &: [\mathbf{Inp}] \;\rightarrow\; [\mathbf{Inp}]
\end{aligned}$$

$$\frac{}{(\mathsf{in} : \mathit{inp_1}) : \mathbf{SField}}\ \textbf{field} \qquad \frac{}{\mathsf{hasNext}(\langle \mathit{str}_1\rangle_{\mathsf{I}}) = \mathsf{length}(\mathit{str_1}) > 0}\ \textbf{next?}$$

$$\frac{\mathsf{length}(\mathit{str_1}) > 0}{\mathsf{getNext}(\langle varstr_1\rangle_{\mathsf{I}}) = \mathsf{fromChar}(\mathsf{substr}(\mathit{str_1}, 0, 1))}\ \textbf{next}$$

$$\frac{\mathsf{length}(\mathit{str_1}) > 0}{\mathsf{skip}(\langle \mathit{str_1}\rangle_{\mathsf{I}}) = \langle \mathsf{substr}(\mathit{str_1}, 0, \mathsf{length}(\mathit{str_1}) - 1)\rangle_{\mathsf{I}}}\ \textbf{skip}$$

Fig. 4.2: Definitions for the input field.

$$\text{Sorts:}\quad \mathbf{Out}\{\mathit{out}_i\}$$

$$\mathbf{Out} < \mathbf{SComp}$$

$$\begin{aligned}
\mathsf{out} &: \mathbf{SIndex}\\
\langle\,\cdot\,\rangle_{\mathsf{O}} &: \mathbf{String} \;\rightarrow\; \mathbf{Out}\\
\langle\rangle_{\mathsf{O}} &: \mathbf{Out}\\
\mathsf{full}(\,\cdot\,) &: \mathbf{Out} \;\rightarrow\; \mathbf{Bool}\\
\cdot \leftarrow \cdot &: [\mathbf{Out}]\ [\mathbf{CharExp}] \;\rightarrow\; [\mathbf{Out}]
\end{aligned}$$

$$\frac{}{(\mathsf{out} : \mathit{out_1}) : \mathbf{SField}}\ \textbf{field} \qquad \frac{}{\langle\rangle_{\mathsf{O}} = \langle\text{""}\rangle_{\mathsf{O}}}\ \textbf{empty}$$

$$\frac{}{\langle \mathit{str_1}\rangle_{\mathsf{O}} \leftarrow \mathit{cc_1} = \langle \mathit{str_1} + \mathsf{toChar}(\mathit{cc_1})\rangle_{\mathsf{O}}}\ \textbf{put} \qquad \frac{}{\mathsf{full}(\mathit{out_1}) = \mathsf{false}}\ \textbf{full}$$

Fig. 4.3: Definitions for the output field.

Threads, being the entities where the actual computation takes place, can be easily defined using the record structure. It is included in the semantic theory $\mathcal{R}_C$ under the following renamings:

$$\begin{aligned}
\mathbf{PreRecord} &\mapsto \mathbf{PreThread}\{\mathit{prt}_i\},\\
\mathbf{Record} &\mapsto \mathbf{Thread}\{t_i\},\\
\mathbf{Field} &\mapsto \mathbf{TField}\{\mathit{tf}_i\},\\
\mathbf{Component} &\mapsto \mathbf{TComp}\{\mathit{tc}_i\},\\
\mathbf{Index} &\mapsto \mathbf{TIndex},\\
\{\,\cdot\,\} &\mapsto [\![\,\cdot\,]\!]
\end{aligned}$$

A thread basically consist only of an expression that indicates the state of the computation. The termination of a thread should be indicated by an “empty” expression. To be able to express this a supersort **Content** is needed:

$$\begin{aligned}
\mathbf{Exp} &< \mathbf{Content}\\
\varepsilon &: \mathbf{Content}
\end{aligned}$$

Well, actually a thread is a bit more complex of course. Additionally an identifier for threads is needed. As this is also supposed to be used as an abstract data type for Haskell programs the sort **ThreadId**$\{\mathit{tid}_1\}$ of thread ids has to be made a subsort of **Exp**. But not directly; since the same has to be done for MVars later, a sort **ExCtor**$\{\mathit{ect}_1\}$ is introduced, making the necessary definitions shorter.

Yet aren't we confusing syntactic entities with their semantic counterparts? In this case, no. As thread ids are only used as abstract data types in Haskell, that is there are only accessor primitives (like `throwTo`) and creator primitives (like `myThreadId`) that enable interacting with them, they are merely metasyntactical entities like defined matches of the functional semantics. Hence there is no need for an explicit discrimination between syntactic thread ids and semantic thread ids as this was done for strings, integers etc.:

$$\begin{aligned}
\mathbf{ThreadId} &< \mathbf{ExCtor}\\
\langle\,\cdot\,\rangle_{\mathsf{Th}} &: \mathbf{Nat} \;\rightarrow\; \mathbf{ThreadId}
\end{aligned}$$

The needed additional equational sentences for substitution application and free variables are trivial:

$$\frac{}{\mathit{ect}_1[s_1] = \mathit{ect}_1} \qquad\qquad \frac{}{\mathsf{fv}(\mathit{ect}_1) = \emptyset}$$

Moreover it is necessary to have defined a sort **Flag**$\{\mathit{fl}_i\}$ to indicate whether a thread is stuck:

$$\mathsf{yes}, \mathsf{no} : \mathbf{Flag}$$

Now the still empty thread record structure is going to be filled introducing the sort **ThreadCont**$\{\mathit{tc}_i\}$ as a supersort of **Content**. Additionally a function symbol is introduced that constructs threads:

$$\begin{aligned}
\mathbf{Content} < \mathbf{ThreadCont} &< \mathbf{TComp}\\
\mathbf{ThreadId}, \mathbf{Flag} &< \mathbf{TComp}.
\end{aligned}$$

$$\begin{aligned}
\mathsf{tid}, \mathsf{val}, \mathsf{stuck} &: \mathbf{TIndex}\\
[\![\,\cdot\,]\!]_{\cdot} &: \mathbf{ThreadCont}\;\; \mathbf{ThreadId} \;\rightarrow\; \mathbf{Thread}
\end{aligned}$$

$$\frac{}{(\mathsf{tid} : \mathit{ti_1}) : \textbf{TField}}\ \textbf{id} \qquad \frac{}{(\mathsf{val} : \mathit{tc_1}) : \textbf{TField}}\ \textbf{val} \qquad \frac{}{(\mathsf{stuck} : \mathit{fl_1}) : \textbf{TField}}\ \textbf{stuck}$$

$$\frac{}{(\!|\mathit{tc_1}|\!)_{\mathit{ti_1}} = (\!|\mathsf{val} : \mathit{tc_1} \quad \mathsf{tid} : \mathsf{ti_1} \quad \mathsf{stuck} : \mathsf{no}|\!)}\ \textbf{new}$$

Note that $(\!|\cdot|\!)_.$ is more than just a shortcut. As the record structure can be extended to contain additional elements for later language revisions, using the explicit construction in the semantic rules would force us to change them if the thread record structure is extended to include all newly introduced elements. With this design only the definition of $(\!|\cdot|\!)_.$ has to be changed.

Of course also MVars need an identifier. Therefore the sort $\textbf{MVarId}\{\mathit{mi_i}\}$ is introduced. The discussion for thread ids regarding its status in the Haskell syntax also holds true for MVar ids. Hence:

$$\begin{aligned}\textbf{MVarId} &< \textbf{ExCtor}\\ \langle\cdot\rangle_{\mathsf{MV}} &: \textbf{Nat} \rightarrow \textbf{MVarId}\end{aligned}$$

For the representation of MVars pretty much the same is done as for threads. At first the record structure is included under the following renamings:

$$\begin{aligned}\textbf{PreRecord} &\mapsto \textbf{PreMVar}\{\mathit{pm_i}\},\\ \textbf{Record} &\mapsto \textbf{MVar}\{\mathit{m_i}\},\\ \textbf{Field} &\mapsto \textbf{MField},\\ \textbf{Component} &\mapsto \textbf{MComp},\\ \textbf{Index} &\mapsto \textbf{MIndex},\\ \{\cdot\} &\mapsto \langle\cdot\rangle\end{aligned}$$

Introducing the sort $\textbf{MVarCont}\{\mathit{mc_i}\}$ for the its content, analogously to **ThreadCont** for threads, the MVar record structure is populated:

$$\begin{aligned}&\textbf{Content} < \textbf{MVarCont} < \textbf{MComp}\\ &\textbf{MVarId} < \textbf{MComp}\end{aligned}$$

$$\begin{aligned}\mathsf{mid}, \mathsf{cont} &: \textbf{MIndex}\\ \langle\cdot\rangle_. &: \textbf{MVarCont}\ \textbf{MVarId} \rightarrow \textbf{MVar}\\ \langle\rangle_. &: \textbf{MVarId} \rightarrow \textbf{MVar}\end{aligned}$$

$$\frac{}{(\mathsf{mid} : \mathit{mi_1}) : \textbf{MField}}\ \textbf{id} \qquad \frac{}{(\mathsf{cont} : \mathit{mc_1}) : \textbf{MField}}\ \textbf{cont}$$

$$\frac{}{\langle \mathit{mc_1}\rangle_{\mathit{mi_1}} = \langle \mathsf{cont} : \mathit{mc_1} \quad \mathsf{mid} : \mathit{mi_1}\rangle}\ \textbf{new1} \qquad \frac{}{\langle\rangle_{\mathit{mi_1}} = \langle\varepsilon\rangle_{\mathit{mi_1}}}\ \textbf{new2}$$

The next ingredients are the asynchronous exceptions, which are syntactically just a pair consisting of an ordinary exception expression and the target thread's id. So here are the necessary definitions for the corresponding sort $\textbf{AException}\{\mathit{ae_i}\}$:

$$\begin{aligned}\langle\!|\cdot \lightning \cdot|\!\rangle &: \textbf{ThreadId}\ \textbf{ExcExp} \rightarrow \textbf{AException}\\ \mathsf{tgt}(\cdot) &: \textbf{AException} \rightarrow \textbf{ThreadId}\\ \mathsf{exc}(\cdot) &: \textbf{AException} \rightarrow \textbf{ExcExp}\end{aligned}$$

$$\frac{}{\mathsf{tgt}(\langle\!|\mathit{ti_1} \lightning \mathit{ee_1}|\!\rangle) = \mathit{ti_1}}\ \textbf{tgt} \qquad \frac{}{\mathsf{exc}(\langle\!|\mathit{ti_1} \lightning \mathit{ee_1}|\!\rangle) = \mathit{ee_1}}\ \textbf{exc}$$

Now, almost similar to the original treatment in [MJMR01], threads, MVars and asynchronous exceptions are taken as process like entities and are put into a pool of parallel processes. Therefore the sorts $\textbf{ProcPool}\{\mathit{pp_i}\}$ and $\textbf{Proc}\{\mathit{pc_i}\}$ are introduced:

$$\begin{aligned}
\langle\cdot\rangle_{\mathsf{ThGen}} &: \mathbf{Nat} \;\rightarrow\; \mathbf{ThreadIdGen}\\
\langle\cdot\rangle_{\mathsf{MVGen}} &: \mathbf{Nat} \;\rightarrow\; \mathbf{MVarIdGen}\\
\mathsf{newThreadIdGen} &: \mathbf{ThreadIdGen}\\
\mathsf{newMVarIdGen} &: \mathbf{MVarIdGen}\\
\mathsf{mainThreadId?}(\cdot) &: \mathbf{ThreadId} \;\rightarrow\; \mathbf{Bool}\\
\mathsf{mainThreadId} &: \mathbf{ThreadId}\\
\mathsf{currentId}\cdot &: \mathbf{MVarIdGen} \;\rightarrow\; \mathbf{MVarId}\\
\mathsf{nextGen}\cdot &: \mathbf{MVarIdGen} \;\rightarrow\; \mathbf{MVarIdGen}\\
\mathsf{currentId}\cdot &: \mathbf{ThreadIdGen} \;\rightarrow\; \mathbf{ThreadId}\\
\mathsf{nextGen}\cdot &: \mathbf{ThreadIdGen} \;\rightarrow\; \mathbf{ThreadIdGen}
\end{aligned}$$

$$\frac{}{\mathsf{newThreadIdGen} = \langle 1\rangle_{\mathsf{ThGen}}}\ \mathbf{newTh} \qquad \frac{}{\mathsf{newMVarIdGen} = \langle 1\rangle_{\mathsf{MVGen}}}\ \mathbf{newMV}$$

$$\frac{}{\mathsf{mainThreadId?}(\langle n_1\rangle_{\mathsf{Th}}) = (n_1 = 0)}\ \mathbf{main?} \qquad \frac{}{\mathsf{mainThreadId} = \langle 0\rangle_{\mathsf{Th}}}\ \mathbf{main}$$

$$\frac{}{\mathsf{currentId}\langle n_1\rangle_{\mathsf{ThGen}} = \langle n_1\rangle_{\mathsf{Th}}}\ \mathbf{thId} \qquad \frac{}{\mathsf{currentId}\langle n_1\rangle_{\mathsf{MVGen}} = \langle n_1\rangle_{\mathsf{MV}}}\ \mathbf{mvId}$$

$$\frac{}{\mathsf{currentId}\langle n_1\rangle_{\mathsf{ThGen}} = \langle n_1+1\rangle_{\mathsf{Th}}}\ \mathbf{thGen} \qquad \frac{}{\mathsf{currentId}\langle n_1\rangle_{\mathsf{MVGen}} = \langle n_1+1\rangle_{\mathsf{MV}}}\ \mathbf{mvGen}$$

Fig. 4.4: Equational definition of id generators.

$$\mathbf{Thread}, \mathbf{MVar}, \mathbf{AException} < \mathbf{Proc} < \mathbf{ProcPool}$$

$$\begin{aligned}
\mathsf{null} &: \mathbf{ProcPool}\\
\cdot \mid \cdot &: \mathbf{ProcPool}\;\; \mathbf{ProcPool} \;\rightarrow\; \mathbf{ProcPool}
\end{aligned}$$

$$\frac{}{pp_1 \mid (pp_2 \mid pp_3) = (pp_1 \mid pp_2) \mid pp_3}\ \mathbf{assoc} \qquad \frac{}{pp_1 \mid pp_2 = pp_2 \mid pp_1}\ \mathbf{comm} \qquad \frac{}{pp_1 \mid \mathsf{null} = pp_1}\ \mathbf{id}$$

This pool of processes can now be made part of the state record structure:

$$\mathbf{ProcPool} < \mathbf{SComp}$$

$$\mathsf{pool} : \mathbf{SIndex}$$

$$\frac{}{(\mathsf{pool} : pp_1) : \mathbf{SField}}\ \mathbf{pool}$$

Unfortunately the original approach in [MJMR01] of producing fresh thread and MVar ids by the use of a restriction operator can not be taken here. Though it is expressible in MEL it would yield a non-executable theory. Hence a different approach is taken involving generators (sorts $\mathbf{ThreadIdGen}\{tg_i\}$ and $\mathbf{MVarIdGen}\{mg_i\}$) that produce fresh ids. The necessary definitions are given in figure 4.4.

Both thread id and MVar id generator can be made fields of the state record structure now:

$$\mathbf{ThreadIdGen}, \mathbf{MVarIdGen} < \mathbf{SComp}$$

$$\mathsf{tgen}, \mathsf{mgen} : \mathbf{SIndex}$$

$$\frac{}{(\mathsf{tgen} : tg_1) : \mathbf{SField}}\ \textbf{tgen} \qquad \frac{}{(\mathsf{mgen} : mg_1) : \mathbf{SField}}\ \textbf{mgen}$$

Finally the initial state being dependent on the Haskell program has to be defined:

$$\mathcal{C}[\![\,\cdot\,]\!](\,\cdot\,) : \mathbf{Program}\ \ \mathbf{String}\ \to\ \mathbf{State}$$
$$\mathcal{C}[\![\,\cdot\ \mathsf{in}\ \cdot\,]\!](\,\cdot\,) : \mathbf{Exp}\ \ \mathbf{Program}\ \ \mathbf{String}\ \to\ \mathbf{State}$$

$$\frac{}{\begin{array}{c}\mathcal{C}[\![e_1\ \mathsf{in}\ pr_1]\!](str_1) = \{\!|\ \mathsf{in} : \langle str_1\rangle_{\mathsf{I}} \quad \mathsf{out} : \langle\rangle_{\mathsf{O}} \quad \mathsf{tgen} : \mathsf{newThreadIdGen} \\ \mathsf{mgen} : \mathsf{newMVarIdGen} \quad \mathsf{pool} : (\!|\mathcal{H}[\![\mathtt{unblock}\ \ e_1\ \mathsf{in}\ pr_1]\!]|\!)_{\mathsf{mainThreadId}}\ |\!\}\end{array}}$$

$$\frac{}{\mathcal{C}[\![pr_1]\!](str_1) = \mathcal{C}[\![\mathtt{main}\ \mathsf{in}\ pr_1]\!](str_1)}$$

Before translating the transitions in [MJMR01] into rewrite sentences the definition for the normal forms of Haskell expressions must be extended to include the new primitives. That is the small layer of functional semantics that is needed for the Concurrent Haskell extension. Therefore we have to distinguish between the theory $\mathcal{E}_H$ presented in section 3 and the extended theory $\mathcal{E}_C$ presented in this section.

Of course as for the functional primitives partially applied concurrent primitives are in normal form. Hence the syntactic category of *BinPrim* has to be extended:

$$\mathit{BinPrim} \quad ::= \quad \ldots \mid \mathtt{putMVar} \mid \mathtt{catch} \mid \mathtt{throwTo}$$

Furthermore, as they cannot be further evaluated, primitives applied to arguments of the “right type” are also considered to be in normal form:

$$\begin{array}{rcl}
\mathit{MVarId} & ::= & \ldots \qquad \text{(cf. MEL signature)} \\
\mathit{ThreadId} & ::= & \ldots \qquad \text{(cf. MEL signature)} \\
\mathit{ExcExp} & ::= & \ldots \qquad \text{(cf. MEL signature)} \\
\mathit{ExCtor} & ::= & \ldots \mid \mathit{MVarId} \mid \mathit{ThreadId} \\
\mathit{AtExp} & ::= & \ldots \mid \mathit{ExCtor} \\
\mathit{Whnf}, \mathit{Chnf} & ::= & \ldots \mid \mathtt{return}\ \mathit{Exp} \mid \mathtt{putChar}\ \mathit{Char} \mid \mathtt{sleep}\ \mathit{Int} \mid \mathtt{putMVar}\ \mathit{MVarId}\ \mathit{Exp} \\
& & \mid \mathtt{takeMVar}\ \mathit{MVarId} \mid \mathtt{forkIO}\ \mathit{Exp} \mid \mathtt{throw}\ \mathit{ExcExp} \mid \mathtt{catch}\ \mathit{Exp}\ \mathit{Exp} \\
& & \mid \mathtt{block}\ \mathit{Exp} \mid \mathtt{unblock}\ \mathit{Exp} \mid \mathit{Exp} >\!\!>\!= \mathit{Exp}
\end{array}$$

Consequently corresponding equational sentences have to be added for the normal form predicate $\cdot \Downarrow .$ [3]. But note that to keep the theory preregular one has to be carefull when trying to translate the syntactic variable *BinPrim* into a MEL sort. The best way to do this is to split it up into a sort **FuncBinPrim** (as a subsort of **FuncPrim**) and a sort **ExBinPrim** (as a subsort of **ExPrim**) and to make those two sorts subsorts of **BinPrim**.

All other additional expressions being not in normal form have to be considered by $\mathcal{F}[\![\,\cdot\,]\!]\,.$, hence the additional sentences given in figure 4.5 on the facing page are needed.

Yet the `do` notation is missing. But as already mentioned, this is considered syntactic sugar only. Hence for their definition it suffices to give the equivalent un-sugared expression for it:

$$\frac{}{\mathtt{do}\ \{\ e_1\,\}\ =\ e_1}\ \textbf{exp} \qquad \frac{}{\mathtt{do}\ \{\ e_1\ ;\ db_1\,\}\ =\ e_1 \mathtt{>>}\ \mathtt{do}\ \{\ db_1\,\}}\ \textbf{exp'}$$

$$\frac{}{\mathtt{do}\ \{\,pt_1\ \mathtt{<-}\ e_1\,\}\ =\ e_1\,\mathtt{>>=}\ (\ \backslash\, pt_1\ \mathtt{->}\ \mathtt{return}\ \ \mathtt{()}\ )}\ \textbf{assign}$$

$$\frac{}{\mathtt{do}\ \{\,pt_1\ \mathtt{<-}\ e_1\ ;\ db_1\,\}\ =\ e_1\,\mathtt{>>=}\ (\ \backslash\, pt_1\ \mathtt{->}\ \mathtt{do}\ \{\ db_1\,\}\ )}\ \textbf{assign'}$$

[3]Of course whnf and chnf share all these additional sentences.

$$\frac{}{\mathcal{F}[\![e_1 \texttt{ >> } e_2]\!]_{nt_1} = e_1 \texttt{ >>= \textbackslash\_ -> } e_2}\ \textbf{bind}$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = cc_1}{\mathcal{F}[\![\texttt{putChar}\ e_1]\!]_{nt_1} = \texttt{putChar}\ cc_1}\ \textbf{putChar} \qquad \frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = ecs_1}{\mathcal{F}[\![\texttt{putChar}\ e_1]\!]_{nt_1} = ecs_1}\ \textbf{putChar}\ \lightning$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = mi_1}{\mathcal{F}[\![\texttt{putMVar}\ e_1\ e_2]\!]_{nt_1} = \texttt{putMVar}\ mi_1\ e_2}\ \textbf{putMVar}$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = ecs_1}{\mathcal{F}[\![\texttt{putMVar}\ e_1\ e_2]\!]_{nt_1} = ecs_1}\ \textbf{putMVar}\ \lightning$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = mi_1}{\mathcal{F}[\![\texttt{takeMVar}\ e_1]\!]_{nt_1} = \texttt{takeMVar}\ mi_1}\ \textbf{takeMVar} \qquad \frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = ecs_1}{\mathcal{F}[\![\texttt{takeMVar}\ e_1]\!]_{nt_1} = ecs_1}\ \textbf{takeMVar}\ \lightning$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = ic_1}{\mathcal{F}[\![\texttt{sleep}\ e_1]\!]_{nt_1} = \texttt{sleep}\ ic_1}\ \textbf{sleep} \qquad \frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = ecs_1}{\mathcal{F}[\![\texttt{sleep}\ e_1]\!]_{nt_1} = ecs_1}\ \textbf{sleep}\ \lightning$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = ece_1}{\mathcal{F}[\![\texttt{throw}\ e_1]\!]_{nt_1} = \texttt{throw}\ ece_1}\ \textbf{throw} \qquad \frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = ecs_1}{\mathcal{F}[\![\texttt{throw}\ e_1]\!]_{nt_1} = ecs_1}\ \textbf{throw}\ \lightning$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} = ti_1 \qquad \mathcal{F}[\![e_2]\!]_{\mathsf{whnf}} = ece_1}{\mathcal{F}[\![\texttt{throwTo}\ e_1 e_2]\!]_{nt_1} = \texttt{throwTo}\ ti_1 ece_1}\ \textbf{throwTo} \qquad \frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} \cup \mathcal{F}[\![e_2]\!]_{\mathsf{whnf}} = ecs_1}{\mathcal{F}[\![\texttt{throwTo}\ e_1 e_2]\!]_{nt_1} = ecs_1}\ \textbf{throwTo}\ \lightning$$

Fig. 4.5: Definitions for functional semantics of concurrent primitives.

Moreover the definition of evaluation contexts both in its $\mathbb{E}$ and $\mathbb{F}$ flavour have to be translated into MEL sentences. The original definition was:

$$\begin{aligned} \mathbb{F} &::= [\cdot] \mid \mathbb{F} \texttt{ >>= } M \mid \texttt{catch}\ \mathbb{F}\ H \\ \mathbb{E} &::= \mathbb{F} \mid \mathbb{F}[\texttt{block}\ \mathbb{E}] \mid \mathbb{F}[\texttt{unblock}\ \mathbb{E}] \end{aligned}$$

Unfortunately this cannot be translated directly into an executable theory. Yet a more operational approach is definable. That is, functions getting subexpressions from such evaluation contexts and respective functions that replace expressions in that context. But still there is a problem, since the expressions considered in evaluation contexts itselves may contain `>>=`, `catch`, `block` or `unblock` such that there is a necessity for constants describing whether and which of such expression is considered. Therefore the sort **ContPat**$\{cp_i\}$ is introduced containing these constants, where the abbreviations `c`, `b`, `u`, `t` and `r` are used to denote `catch`, `block`, `unblock`, `throw` and `return`, respectively:

$$\mathsf{none},\ {}^{\texttt{r}}/\texttt{>>=},\ {}^{\texttt{t}}/\texttt{>>=},\ {}^{\texttt{c}}/\texttt{r},\ {}^{\texttt{c}}/\texttt{t},\ {}^{\texttt{b}}/\texttt{r},\ {}^{\texttt{b}}/\texttt{t},\ {}^{\texttt{u}}/\texttt{r},\ {}^{\texttt{u}}/\texttt{t}\ :\ \textbf{ContPat}$$

Of course it has to be defined which particular expression patterns[4] each single constant that is introduced above actually represents:

$$\cdot \in [\cdot] : \textbf{Exp}\ \ \textbf{ContPat}\ \rightarrow\ \textbf{Bool}$$

[4]These expression patterns have nothing to do with the patterns used for Haskell's pattern matching feature.

$$\frac{}{\texttt{return}\,e_1 \texttt{>>=} e_2 \in [\,^{\texttt{r}}/_{\texttt{>>=}}\,] = \mathsf{true}}\ ^{\texttt{r}}/_{\texttt{>>=}} \qquad \frac{}{\texttt{throw}\,e_1 \texttt{>>=} e_2 \in [\,^{\texttt{t}}/_{\texttt{>>=}}\,] = \mathsf{true}}\ ^{\texttt{t}}/_{\texttt{>>=}}$$

$$\frac{}{\texttt{catch}\,(\,\texttt{return}\,e_1)e_2 \in [\,^{\texttt{c}}/_{\texttt{r}}\,] = \mathsf{true}}\ ^{\texttt{c}}/_{\texttt{r}} \qquad \frac{}{\texttt{block}\,(\,\texttt{return}\,e_1) \in [\,^{\texttt{b}}/_{\texttt{r}}\,] = \mathsf{true}}\ ^{\texttt{b}}/_{\texttt{r}}$$

$$\frac{}{\texttt{block}\,(\,\texttt{throw}\,e_1) \in [\,^{\texttt{b}}/_{\texttt{t}}\,] = \mathsf{true}}\ ^{\texttt{b}}/_{\texttt{t}} \qquad \frac{}{\texttt{unblock}\,(\,\texttt{return}\,e_1) \in [\,^{\texttt{u}}/_{\texttt{r}}\,] = \mathsf{true}}\ ^{\texttt{u}}/_{\texttt{r}}$$

$$\frac{}{\texttt{unblock}\,(\,\texttt{throw}\,e_1) \in [\,^{\texttt{u}}/_{\texttt{t}}\,] = \mathsf{true}}\ ^{\texttt{u}}/_{\texttt{t}} \qquad \frac{\mathsf{otherwise}}{e_1 \in [cp_1] = \mathsf{false}}\ \textbf{fail}$$

Where $e_1 \in [cp_1]$ is supposed to be understood as "e_1 is of the form cp_1".

Now we are able to describe the $\mathbb{E}$ context by the two symbols $\mathbb{E}[\cdot](\cdot)$, for retrieving, and $\mathbb{E}[\cdot](\cdot \leftarrow \cdot)$, for replacing expressions in $\mathbb{E}$ contexts. The necessary operator declarations and equational sentences are given in figure 4.6.

$$\mathbb{E}[\cdot](\cdot) : \textbf{ContPat}\ \textbf{Exp} \rightarrow \textbf{Exp}$$
$$\mathbb{E}[\cdot](\cdot \leftarrow \cdot) : \textbf{ContPat}\ \textbf{Exp}\ \textbf{Exp} \rightarrow \textbf{Exp}$$

$$\frac{\mathsf{otherwise}}{\mathbb{E}[cp_1](e_1) = e_1}\ \mathbb{E}\ \textbf{match}$$

$$\frac{e_1 = \texttt{block}\ e_2 \quad e_1 \in [cp_1] = \mathsf{false}}{\mathbb{E}[cp_1](e_1) = \mathbb{E}[cp_1](e_2)}\ \mathbb{E}\ \textbf{block} \qquad \frac{e_1 = \texttt{unblock}\ e_2 \quad e_1 \in [cp_1] = \mathsf{false}}{\mathbb{E}[cp_1](e_1) = \mathbb{E}[cp_1](e_2)}\ \mathbb{E}\ \textbf{unblock}$$

$$\frac{e_1 = e_2 \texttt{>>=} e_3 \quad e_1 \in [cp_1] = \mathsf{false}}{\mathbb{E}[cp_1](e_1) = \mathbb{E}[cp_1](e_2)}\ \mathbb{E}\ \textbf{bind} \qquad \frac{e_1 = \texttt{catch}\ e_2\ e_3 \quad e_1 \in [cp_1] = \mathsf{false}}{\mathbb{E}[cp_1](e_1) = \mathbb{E}[cp_1](e_2)}\ \mathbb{E}\ \textbf{catch}$$

$$\frac{\mathsf{otherwise}}{\mathbb{E}[cp_1](e_1 \leftarrow e_2) = e_2}\ \mathbb{E}_{\leftarrow}\textbf{match}$$

$$\frac{e_1 = \texttt{block}\ e_3 \qquad e_1 \in [cp_1] = \mathsf{false}}{\mathbb{E}[cp_1](e_1 \leftarrow e_2) = \texttt{block}\ \mathbb{E}[cp_1](e_3 \leftarrow e_2)}\ \mathbb{E}_{\leftarrow}\textbf{block}$$

$$\frac{e_1 = \texttt{unblock}\ e_3 \qquad e_1 \in [cp_1] = \mathsf{false}}{\mathbb{E}[cp_1](e_1 \leftarrow e_2) = \texttt{unblock}\ \mathbb{E}[cp_1](e_3 \leftarrow e_2)}\ \mathbb{E}_{\leftarrow}\textbf{unblock}$$

$$\frac{e_1 = e_3 \texttt{>>=} e_4 \qquad e_1 \in [cp_1] = \mathsf{false}}{\mathbb{E}[cp_1](e_1 \leftarrow e_2) = \mathbb{E}[cp_1](e_3 \leftarrow e_2) \texttt{>>=} e_4}\ \mathbb{E}_{\leftarrow}\textbf{bind}$$

$$\frac{e_1 = \texttt{catch}\ e_3\ e_4 \qquad e_1 \in [cp_1] = \mathsf{false}}{\mathbb{E}[cp_1](e_1 \leftarrow e_2) = \texttt{catch}\ \mathbb{E}[cp_1](e_3 \leftarrow e_2)\ e_4}\ \mathbb{E}_{\leftarrow}\textbf{catch}$$

Fig. 4.6: Operational definition of $\mathbb{E}$ contexts.

The following definitions merely introduce a shortcut for using the none pattern constant.

$$\mathbb{E}(\cdot) : \textbf{Exp} \rightarrow \textbf{Exp}$$
$$\mathbb{E}(\cdot \leftarrow \cdot) : \textbf{Exp}\ \textbf{Exp} \rightarrow \textbf{Exp}$$

$$\frac{}{\mathbb{E}(e_1) = \mathbb{E}[\mathsf{none}](e_1)}\ \mathbb{E} \qquad \frac{}{\mathbb{E}(e_1 \leftarrow e_2) = \mathbb{E}[\mathsf{none}](e_1 \leftarrow e_2)}\ \mathbb{E}_{\leftarrow}$$

$$\mathbb{F}(\,\cdot\,) : \textbf{Exp} \;\rightarrow\; \textbf{Exp}$$
$$\mathbb{F}(\,\cdot \leftarrow \cdot\,) : \textbf{Exp}\;\textbf{Exp} \;\rightarrow\; \textbf{Exp}$$

$$\frac{\textsf{otherwise}}{\mathbb{F}(e_1) = e_1}\;\mathbb{F}\;\textbf{def} \qquad \frac{}{\mathbb{F}(e_1\,\texttt{>>=}\,e_2) = \mathbb{F}(e_1)}\;\mathbb{F}\;\textbf{bind} \qquad \frac{}{\mathbb{F}(\,\texttt{catch}\;e_1\;e_2) = \mathbb{F}(e_1)}\;\mathbb{F}\;\textbf{catch}$$

$$\frac{\textsf{otherwise}}{\mathbb{F}(e_1 \leftarrow e_2) = e_2}\;\mathbb{F}_{\leftarrow}\textbf{def} \qquad \frac{}{\mathbb{F}(e_1\,\texttt{>>=}\,e_2 \leftarrow e_3) = \mathbb{F}(e_1 \leftarrow e_3)\,\texttt{>>=}\,e_2}\;\mathbb{F}_{\leftarrow}\textbf{bind}$$

$$\frac{}{\mathbb{F}(\,\texttt{catch}\;e_1\;e_2 \leftarrow e_3) = \;\texttt{catch}\;\mathbb{F}(e_1 \leftarrow e_3)\;e_2}\;\mathbb{F}_{\leftarrow}\textbf{catch}$$

Fig. 4.7: Operational definition of $\mathbb{F}$ contexts.

Similarly but in detail differently $\mathbb{F}$ contexts are defined in figure 4.7. In this particular case it can be forbeared from considering special expression that include `>>=`, `catch` as such expressions will not be used in these contexts by the transition system.

The definition of $\mathbb{F}$ contexts is merely for defining $\mathbb{E}\,[\textsf{unblock}\;\mathbb{F}]$ contexts that will be used by the transition system directly. Hence a MEL definition of these contexts is necessary, too. This is also done operationally by defining an operational $\mathbb{U}$ context in figure 4.8 on the next page.

Apart from evaluation contexts there is a auxiliary predicate needed to determine whether an expression has the `block` primitive as its head element:

$$\textsf{blockHead?} : \textbf{Exp} \;\rightarrow\; \textbf{Bool}$$

$$\frac{}{\textsf{blockHead?}(\,\texttt{block}\,e_1) = \textsf{true}} \qquad \frac{\textsf{otherwise}}{\textsf{blockHead?}(e_1) = \textsf{false}}$$

Finally all preparations are made to give the rewrite sentences that describe the transition system of the original operational semantics. They are given in figures 4.9 and 4.10 on pages 45 ff. These are exactly those rewrite sentences R_C that together with the MEL theory $\mathcal{E}_C = (\Omega_C, E_C)$ constitute the desired GRT $\mathcal{E}_C = (\Omega_C, E_C, R_C)$ of the semantics of Concurrent Haskell.

There are several differences compared to the original operational semantics. The first and most obvious difference is that some rewrites only consider a subterm of the **State** term. Yet by (Cong) of the rewriting logic semantics they still apply to a full **State** term. But still there is a subtlety of this subterm rewrites that is typical for rewriting logic. The (Nested Repl) rule enables parallel rewriting provided the individual rewrites are independent. This behaviour — coined *true concurrency* — is possible for example for the sentence (**Bind**) in conjunction with any other sentences including itself.

Another more obvious difference is the fact that the original operational semantics is a *small step* semantics. The rewrite semantics on the other hand is by the (Trans) rule of the semantics of rewriting logic a *big step* semantics. This could be avoided by using the idea of [ŞRM07, MB03] of introducing operators that are wrapped around the state terms to make the rewrite sentences asymmetric and hence prevent transitivity. But since this would destroy true concurrency and is more importantly not necessary as the transition system is completely shallow[5], that is transitions are not used as a condition for another transition, this approach is not taken.

[5]Actually there are structural rules in the original SOS but there are compensated by equational rules and RL's semantics, that is the rules (Cong, Eq).

$$\mathbb{U}(\,\cdot\,) : [\mathbf{Exp}] \;\rightarrow\; [\mathbf{Exp}]$$
$$\mathbb{U}(\,\cdot\, \leftarrow \,\cdot\,) : [\mathbf{Exp}]\;\; [\mathbf{Exp}] \;\rightarrow\; [\mathbf{Exp}]$$

$$\frac{}{\mathbb{U}(e_1 \texttt{>>=}\, e_2) = \mathbb{U}(e_1)}\;\textbf{U bind} \qquad \frac{}{\mathbb{U}(\,\texttt{catch}\;\; e_1\; e_2) = \mathbb{U}(e_1)}\;\textbf{U catch}$$

$$\frac{}{\mathbb{U}(\,\texttt{block}\;\; e_1) = \mathbb{U}(e_1)}\;\textbf{U block} \qquad \frac{\mathbb{U}(e_1) = e_2}{\mathbb{U}(\,\texttt{unblock}\;\; e_1) = e_2}\;\textbf{U unblock}$$

$$\frac{\textsf{otherwise}}{\mathbb{U}(\,\texttt{unblock}\;\; e_1) = \mathbb{F}(e_1)}\;\textbf{U match}$$

$$\frac{}{\mathbb{U}(e_1 \texttt{>>=}\, e_2 \leftarrow e_3) = \mathbb{U}(e_1 \leftarrow e_3) \texttt{>>=}\, e_2}\;\mathbf{U_{\leftarrow}bind}$$

$$\frac{}{\mathbb{U}(\,\texttt{catch}\;\; e_1\; e_2 \leftarrow e_3) = \;\texttt{catch}\;\; \mathbb{U}(e_1 \leftarrow e_3)\; e_2}\;\mathbf{U_{\leftarrow}catch}$$

$$\frac{}{\mathbb{U}(\,\texttt{block}\;\; e_1 \leftarrow e_3) = \;\texttt{block}\, \mathbb{U}(e_1 \leftarrow e_3)}\;\mathbf{U_{\leftarrow}block}$$

$$\frac{\mathbb{U}(e_1 \leftarrow e_2) = e_3}{\mathbb{U}(\,\texttt{unblock}\;\; e_1 \leftarrow e_2) = \;\texttt{unblock}\;\; e_3}\;\mathbf{U_{\leftarrow}unblock}$$

$$\frac{\textsf{otherwise}}{\mathbb{U}(\,\texttt{unblock}\;\; e_1 \leftarrow e_2) = \;\texttt{unblock}\;\; \mathbb{F}(e_1 \leftarrow e_2)}\;\mathbf{U_{\leftarrow}match}$$

Fig. 4.8: Operational definition of $\mathbb{E}\,[\textsf{unblock}\;\mathbb{F}]$ contexts.

$$\frac{\mathbb{E}(e_1) = \mathtt{putChar}\ cc_1 \qquad \mathbb{E}(e_1 \leftarrow \mathtt{return}\ \ (\,)\,) = e_2 \qquad \neg\mathsf{full}(out_1) = \mathsf{true}}{\begin{array}{c}\mathsf{pool} : \llparenthesis \mathsf{val} : e_1 \quad \mathsf{stuck} : fl_1 \quad prt_1 \rrparenthesis \mid pp_1 \quad \mathsf{out} : out_1 \\ \longrightarrow \\ \mathsf{pool} : \llparenthesis \mathsf{val} : e_2 \quad \mathsf{stuck} : \mathsf{no} \quad prt_1 \rrparenthesis \mid pp_1 \quad \mathsf{out} : out_1 \leftarrow cc_1\end{array}} \quad \textbf{PutChar}$$

$$\frac{\begin{array}{c}\mathbb{E}(e_1) = \mathtt{getChar} \qquad \mathsf{hasNext}(inp_1) = true \\ \mathbb{E}(e_1 \leftarrow \mathtt{return}\ \mathsf{fromChar}(\mathsf{getNext}(inp_1))) = e_2\end{array}}{\begin{array}{c}\mathsf{pool} : \llparenthesis \mathsf{val} : e_1 \quad \mathsf{stuck} : fl_1 \quad prt_1 \rrparenthesis \mid pp_1 \quad \mathsf{in} : inp_1 \\ \longrightarrow \\ \mathsf{pool} : \llparenthesis \mathsf{val} : e_2 \quad \mathsf{stuck} : \mathsf{no} \quad prt_1 \rrparenthesis \mid pp_1 \quad \mathsf{in} : \mathsf{skip}(inp_1)\end{array}} \quad \textbf{GetChar}$$

$$\frac{\mathbb{E}(e_1) = \mathtt{sleep}\ ic_1 \qquad \mathbb{E}(e_1 \leftarrow \mathtt{return}\ \ (\,)\,) = e_2}{\mathsf{val} : e_1 \quad \mathsf{stuck} : fl_1 \ \longrightarrow\ \mathsf{val} : e_2 \quad \mathsf{stuck} : \mathsf{no}} \quad \textbf{Sleep}$$

$$\frac{\mathbb{E}(e_1) = \mathtt{putMVar}\ mi_1\ e_3 \qquad \mathbb{E}(e_1 \leftarrow \mathtt{return}\ \ (\,)\,) = e_2}{\begin{array}{c}\llparenthesis \mathsf{val} : e_1 \quad \mathsf{stuck} : fl_1 \quad prt_1 \rrparenthesis \mid \langle \mathsf{cont} : \varepsilon \quad \mathsf{mid} : mi_1 \quad pm_1 \rangle \\ \longrightarrow \\ \llparenthesis \mathsf{val} : e_2 \quad \mathsf{stuck} : \mathsf{no} \quad prt_1 \rrparenthesis \mid \langle \mathsf{cont} : e_3 \quad \mathsf{mid} : mi_1 \quad pm_1 \rangle\end{array}} \quad \textbf{PutMVar}$$

$$\frac{\mathbb{E}(e_1) = \mathtt{takeMVar}\ mi_1 \qquad \mathbb{E}(e_1 \leftarrow \mathtt{return}\ e_3) = e_2}{\begin{array}{c}\llparenthesis \mathsf{val} : e_1 \quad \mathsf{stuck} : fl_1 \quad prt_1 \rrparenthesis \mid \langle \mathsf{cont} : e_3 \quad \mathsf{mid} : mi_1 \quad pm_1 \rangle \\ \longrightarrow \\ \llparenthesis \mathsf{val} : e_2 \quad \mathsf{stuck} : \mathsf{no} \quad prt_1 \rrparenthesis \mid \langle \mathsf{cont} : \varepsilon \quad \mathsf{mid} : mi_1 \quad pm_1 \rangle\end{array}} \quad \textbf{TakeMVar}$$

$$\frac{\begin{array}{c}\mathbb{E}(e_1) = \mathtt{newEmptyMVar} \qquad \mathsf{currentId}(mg_1) = mi_1 \\ \mathbb{E}(e_1 \leftarrow \mathtt{return}\ mi_1) = e_2\end{array}}{\begin{array}{c}\mathsf{pool} : \llparenthesis \mathsf{val} : e_1 \quad prt_1 \rrparenthesis \mid pp_1 \quad \mathsf{mgen} : mg_1 \\ \longrightarrow \\ \mathsf{pool} : \llparenthesis \mathsf{val} : e_2 \quad prt_1 \rrparenthesis \mid \langle\rangle_{mi_1} \mid pp_1 \quad \mathsf{mgen} : \mathsf{nextGen}(mg_1)\end{array}} \quad \textbf{NewMVar}$$

$$\frac{\mathbb{E}(e_1) = \mathtt{forkIO}\ e_3 \qquad \mathsf{currentId}(tg_1) = ti_1 \qquad \mathbb{E}(e_1 \leftarrow \mathtt{return}\ ti_1) = e_2}{\begin{array}{c}\mathsf{pool} : \llparenthesis \mathsf{val} : e_1 \quad prt_1 \rrparenthesis \mid pp_1 \quad \mathsf{tgen} : tg_1 \\ \longrightarrow \\ \mathsf{pool} : \llparenthesis \mathsf{val} : e_2 \quad prt_1 \rrparenthesis \mid \llparenthesis \mathtt{unblock}\ e_3 \rrparenthesis_{ti_1} \mid pp_1 \quad \mathsf{tgen} : \mathsf{nextGen}(tg_1)\end{array}} \quad \textbf{Fork}$$

$$\frac{\mathbb{E}(e_1) = \mathtt{myThreadId} \qquad \mathbb{E}(e_1 \leftarrow \mathtt{return}\ ti_1) = e_2}{\mathsf{val} : e_1 \quad \mathsf{tid} : ti_1 \ \longrightarrow\ \mathsf{val} : e_2 \quad \mathsf{tid} : ti_1} \quad \textbf{ThreadId}$$

$$\frac{\mathbb{E}[\mathtt{t}/\mathtt{>>=}](e_1) = (\mathtt{throw}\, ece_1)\, \mathtt{>>=}\, e_3 \qquad \mathbb{E}[\mathtt{t}/\mathtt{>>=}](e_1 \leftarrow \mathtt{throw}\, ece_1) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2} \quad \textbf{Propagate}$$

$$\frac{\mathbb{E}[\mathtt{c}/\mathtt{r}](e_1) = \mathtt{catch}\,(\mathtt{return}\, e_3) e_4 \qquad \mathbb{E}[\mathtt{c}/\mathtt{r}](e_1 \leftarrow \mathtt{return}\ e_3) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2} \quad \textbf{Catch}$$

$$\frac{\mathbb{E}[\mathtt{c}/\mathtt{t}](e_1) = \mathtt{catch}\,(\mathtt{throw}\, ecs_1) e_3 \qquad \mathbb{E}[\mathtt{c}/\mathtt{t}](e_1 \leftarrow e_3\ ecs_1) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2} \quad \textbf{Handle}$$

$$\frac{}{\mathsf{val} : \mathtt{return}\ e_1 \ \longrightarrow\ \mathsf{val} : \varepsilon} \quad \textbf{Return GC}$$

$$\frac{}{\mathsf{val} : \mathtt{throw}\ ece_1 \ \longrightarrow\ \mathsf{val} : \varepsilon} \quad \textbf{Throw GC}$$

$$\frac{\mathsf{mainThreadId?}(ti_1) = \mathsf{true}}{\llparenthesis \mathsf{val} : \varepsilon \quad \mathsf{tid} : ti_1 \quad prt_1 \rrparenthesis \mid pc_1 \ \longrightarrow\ \llparenthesis \mathsf{val} : \varepsilon \quad \mathsf{tid} : ti_1 \quad prt_1 \rrparenthesis} \quad \textbf{Proc GC}$$

Fig. 4.9: Rewrite sentences describing the concurrency semantics (1).

$$\frac{\mathbb{E}[{}^{\mathtt{r}}/\mathtt{>>=}](e_1) = (\,\mathtt{return}\ \ e_3) \mathbin{\mathtt{>>=}} e_4 \qquad \mathbb{E}[{}^{\mathtt{r}}/\mathtt{>>=}](e_1 \leftarrow e_4\ e_3) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2}$$

Bind

$$\frac{\mathbb{E}(e_1) = e_3 \qquad \neg e_3 \Downarrow_{\mathsf{whnf}} = \mathsf{true} \qquad \mathcal{F}[\![e_3]\!]_{\mathsf{whnf}} = e_4 \qquad \mathbb{E}(e_1 \leftarrow e_4) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2}$$

Eval

$$\frac{\mathbb{E}(e_1) = e_3 \qquad \mathcal{F}[\![e_3]\!]_{\mathsf{whnf}} = \{ec_1; pecs_1\} \qquad \mathbb{E}(e_1 \leftarrow \mathtt{throw}\ \mathsf{exp}(ec_1)) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2}$$

Raise

$$\frac{\mathbb{E}[{}^{\mathtt{b}}/\mathtt{r}](e_1) = \mathtt{block}\,(\,\mathtt{return}\,e_3) \qquad \mathbb{E}[{}^{\mathtt{b}}/\mathtt{r}](e_1 \leftarrow \mathtt{return}\,e_3) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2}$$

Block Return

$$\frac{\mathbb{E}[{}^{\mathtt{u}}/\mathtt{r}](e_1) = \mathtt{unblock}\,(\,\mathtt{return}\,e_3) \qquad \mathbb{E}[{}^{\mathtt{u}}/\mathtt{r}](e_1 \leftarrow \mathtt{return}\,e_3) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2}$$

Unblock Return

$$\frac{\mathbb{E}[{}^{\mathtt{b}}/\mathtt{t}](e_1) = \mathtt{block}\,(\,\mathtt{throw}\,ece_1) \qquad \mathbb{E}[{}^{\mathtt{b}}/\mathtt{r}](e_1 \leftarrow \mathtt{throw}\,ece_1) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2}$$

Block Throw

$$\frac{\mathbb{E}[{}^{\mathtt{u}}/\mathtt{t}](e_1) = \mathtt{unblock}\,(\,\mathtt{throw}\,ece_1) \qquad \mathbb{E}[{}^{\mathtt{u}}/\mathtt{r}](e_1 \leftarrow \mathtt{throw}\,ece_1) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2}$$

Unblock Throw

$$\frac{\mathbb{E}(e_1) = \mathtt{throwTo}\ \ ti_1\ ece_1 \qquad \mathbb{E}(e_1 \leftarrow \mathtt{return}\ \ (\,)\,) = e_2}{(\!|\,\mathsf{val} : e_1 \quad prt_1\,|\!) \ \longrightarrow\ (\!|\,\mathsf{val} : e_2 \quad prt_1\,|\!) \mid \{\!|\, ti_1 \lightning\, ece_1 \,|\!\}}$$

ThrowTo

$$\frac{\begin{array}{c}\mathbb{U}(e_1) = e_3 \qquad \neg\mathsf{blockHead?}(e_3) = \mathsf{true} \\ \mathsf{tgt}(ae_1) = ti_1 \qquad \mathbb{U}(e_1 \leftarrow \mathtt{throw}\ \mathsf{exc}(ae_1)) = e_2\end{array}}{(\!|\,\mathsf{val} : e_1 \quad \mathsf{tid} : ti_1 \quad prt_1\,|\!) \mid ae_1 \ \longrightarrow\ (\!|\,\mathsf{val} : e_2 \quad \mathsf{tid} : ti_1 \quad prt_1\,|\!)}$$

Receive

$$\frac{\mathbb{E}(e_1) = e_3 \qquad \mathsf{tgt}(ae_1) = ti_1 \qquad \mathbb{E}(e_1 \leftarrow \mathtt{throw}\ \mathsf{exc}(ae_1)) = e_2}{\begin{array}{c}(\!|\,\mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{yes} \quad \mathsf{tid} : ti_1 \quad prt_1\,|\!) \mid \{\!|\, ti_1 \lightning\, ece_1 \,|\!\} \\ \xrightarrow{\hspace{20em}} \\ (\!|\,\mathsf{val} : e_2 \quad \mathsf{stuck} : \mathsf{no} \quad \mathsf{tid} : ti_1 \quad prt_1\,|\!)\end{array}}$$

Interrupt

$$\frac{\mathbb{E}(e_1) = \mathtt{putChar}\ \ cc_1}{\mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{no} \ \longrightarrow\ \mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{yes}}$$

Stuck PutChar

$$\frac{\mathbb{E}(e_1) = \mathtt{getChar}}{\mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{no} \ \longrightarrow\ \mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{yes}}$$

Stuck GetChar

$$\frac{\mathbb{E}(e_1) = \mathtt{sleep}\ \ ic_1}{\mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{no} \ \longrightarrow\ \mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{yes}}$$

Stuck Sleep

$$\frac{\mathbb{E}(e_1) = \mathtt{putMVar}\ \ mi_1\ e_3}{\begin{array}{c}(\!|\,\mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{no} \quad prt_1\,|\!) \mid \langle \mathsf{cont} : e_2 \quad \mathsf{mid} : mi_1 \quad pm_1 \rangle \\ \xrightarrow{\hspace{20em}} \\ (\!|\,\mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{yes} \quad prt_1\,|\!) \mid \langle \mathsf{cont} : e_2 \quad \mathsf{mid} : mi_1 \quad pm_1 \rangle\end{array}}$$

Stuck PutMVar

$$\frac{\mathbb{E}(e_1) = \mathtt{takeMVar}\ \ mi_1}{\begin{array}{c}(\!|\,\mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{no} \quad prt_1\,|\!) \mid \langle \mathsf{cont} : \varepsilon \quad \mathsf{mid} : mi_1 \quad pm_1 \rangle \\ \xrightarrow{\hspace{20em}} \\ (\!|\,\mathsf{val} : e_1 \quad \mathsf{stuck} : \mathsf{yes} \quad prt_1\,|\!) \mid \langle \mathsf{cont} : \varepsilon \quad \mathsf{mid} : mi_1 \quad pm_1 \rangle\end{array}}$$

Stuck TakeMVar

Fig. 4.10: Rewrite sentences describing the concurrency semantics (2).

4.3 Relation to Existing Semantics

As the semantics given in the previous section is almost only a simple transformation of the operational semantics in [MJMR01] it should be easy to show their equivalence. But as a matter of fact as already mentioned the the previous section they are — in a strict sense — *not* equivalent. Yet in a slightly weaker sense they are.

Now we have to consider an extended syntax including the primitives of Concurrent Haskell:

$$\begin{array}{rcl}
M, N & ::= & \mid \dots \quad (cf.previousdefinition) \\
& & \mid ch \mid d \mid m \mid t \\
& & \mid M >\!\!>\!= N \mid \mathsf{return} \mid \mathsf{throw} \mid \mathsf{putChar} \mid \dots \\
U, V & ::= & \mid \dots \quad (cf.previousdefinition) \\
& & \mid ch \mid d \mid m \mid t \\
& & \mid \mathsf{return}\ M \mid M >\!\!>\!= N \mid \mathsf{throw}\ e \mid \mathsf{putChar}\ ch \mid \mathsf{getChar} \mid \mathsf{putMVar}\ m\ M \mid \mathsf{takeMVar}\ m \\
& & \mid \mathsf{newEmptyMVar} \mid \mathsf{sleep}\ d \mid \mathsf{catch}\ M\ N \mid \mathsf{throwTo}\ t\ e \mid \mathsf{block}\ M \mid \mathsf{unblock} \ \mid \mathsf{forkIO}\ M \\
& & \mid \mathsf{myThreadId}
\end{array}$$

where ch ranges over the set of characters, d over $\mathbb{Z}$, m over the set of MVar ids $MVar$ and t over the set of thread ids $Thread$. We assume that both sets $MVar$ and $Thread$ are countably infinite.

To be able to relate both semantics the translation symbol $\overline{\cdot}$ has to be extended to cover the syntax extension as well:

$$\begin{array}{rcll}
\overline{M >\!\!>\!= N} & = & \overline{M}\ \texttt{>>=}\ \overline{M} & (\overline{\text{bind}}) \\
\overline{\mathsf{catch}\ M\ N} & = & \texttt{catch}\ \overline{M}\ \overline{M} & (\overline{\text{catch}}) \\
\overline{\mathsf{block}\ M\ N} & = & \texttt{block}\ \overline{M}\ \overline{M} & (\overline{\text{block}}) \\
\overline{\mathsf{unblock}\ M\ N} & = & \texttt{unblock}\ \overline{M}\ \overline{M} & (\overline{\text{unblock}}) \\
& \dots & &
\end{array}$$

For the special constants the choice for $\overline{\cdot}$ is quite obvious: Let $\mathsf{intConst} : \mathbb{N} \to T_{\Sigma[\mathbf{IntConst}]}$ be the mapping that maps every integer to the corresponding term of sort **IntConst**. Moreover let both $\mathsf{mInd} : MVar \to \mathbb{N}$ and $\mathsf{tInd} : Thread \to \mathbb{N}$ be arbitrary but fixed bijections (which exist as $MVar$ and $Thread$ are countably infinite). Then define $\overline{d} = \mathsf{intConst}(d)$; $\overline{m} = \langle \mathsf{intConst}(\mathsf{mInd}(m)) \rangle_{\mathsf{MV}}$ and $\overline{t} = \langle \mathsf{intConst}(\mathsf{tInd}(t)) \rangle_{\mathsf{Th}}$. Furthermore let $\overline{ch}$ be defined as the corresponding term of sort **CharConst**.

Of course also the operational MEL definitions of evaluation contexts, that is in particular $\mathbb{E}$ and $\mathbb{U}$, have to match their original inductive definition $\mathbb{E}$ and $\mathbb{E}\,[\mathsf{unblock}\ \mathbb{F}]$. To prove this, some lemmas have to be established.

The following lemma, being prototypical for the upcoming lemmas, states that $\mathbb{E}$ skips $\mathbb{F}\,[\mathsf{block}\ [\,\cdot\,]]$ and $\mathbb{F}\,[\mathsf{unblock}\ [\,\cdot\,]]$ contexts:

Lemma 4.1. *Let M and N be arbitrary Haskell expressions and $B \in \{\mathsf{block}, \mathsf{block}\}$. Then the following holds:*

(i) $\mathcal{E}_C \vdash \mathbb{E}(\overline{\mathbb{F}\,[B\ M]}) = \mathbb{E}(\overline{M})$

(ii) $\mathcal{E}_C \vdash \mathbb{E}(\overline{\mathbb{F}\,[B\ M]} \leftarrow \overline{N}) = \overline{\mathbb{F}\,[B\ M']}$ *where* $\overline{M'} = \mathbb{E}(\overline{M} \leftarrow \overline{N})$

Proof. The proof of both statements is an easy induction on the structure of $\mathbb{F}$. The proofs for $B = \mathsf{block}$ and $B = \mathsf{unblock}$ are virtually identical. In the following the first case is treated. Pleas note that in this as well as in the upcoming proofs (meta-)syntactical equalities for example induced by $\overline{\cdot}$ and MEL equalities are mixed in the argument chain for the sake of brevity of the presentation. Nevertheless, as the equalities are justified by giving the equation that was used both qualities of equalities are kept distinguishable.

(i) **Case 1** $\mathbb{F} = [\cdot]$

$$\mathbb{E}(\overline{\mathsf{block}\ M}) \overset{\overline{\text{block}}}{=} \mathbb{E}(\,\texttt{block}\ \overline{M}) \overset{\mathbb{E}\ \textbf{block}}{=} \mathbb{E}(\overline{M})$$

Case 2 $\mathbb{F} = \mathbb{F}' >>= M'$

$$\mathbb{E}(\overline{\mathbb{F}'[\mathsf{block}\ M] >>= M'}) \overset{\overline{\text{bind}}}{=} \mathbb{E}(\overline{\mathbb{F}'[\mathsf{block}\ M]}\,\texttt{>>=}\,\overline{M'}) \overset{\mathbb{E}\ \textbf{bind}}{=} \mathbb{E}(\overline{\mathbb{F}'[\mathsf{block}\ M]}) \overset{\text{I.H.}}{=} \mathbb{E}(\overline{M})$$

Case 3 $\mathbb{F} = \mathsf{catch}\ \mathbb{F}'\ M'$ Analogously to case 2.

(ii) Analogously to (i).

□

Using this the the following essential property of the operational $\mathbb{E}$ contexts can be established. Please note the restriction on the argument expression.

Lemma 4.2. *Let M be an expression not having $>>=$, catch, block or $\mathsf{unblock}$ as the head symbol (i.e. M is not of the form $M' >>= N'$, $\mathsf{catch}\ M'\ N'$, $\mathsf{block}\ M'$ or $\mathsf{unblock}\ M'$ for some expressions M', N') and N an arbitrary Haskell expression. Then the following holds:*

(i) $\mathcal{E}_C \vdash \mathbb{E}(\overline{\mathbb{E}\,[M]}) = \overline{M}$

(ii) $\mathcal{E}_C \vdash \mathbb{E}(\overline{\mathbb{E}\,[M]} \leftarrow \overline{N}) = \overline{\mathbb{E}\,[N]}$

Proof. Both statements can be easily proven by induction on the structure of $\mathbb{E}$ and $\mathbb{F}$:

(i) **Case 1** $\mathbb{E} = \mathbb{F}$ induction on $\mathbb{F}$

Case 1.1 $\mathbb{F} = [\cdot]$ Here the restriction on the form of M is used.

$$\mathbb{E}(\overline{M}) \overset{\mathbb{E}\ \textbf{match}}{=} \overline{M}$$

Case 1.2 $\mathbb{F} = \mathbb{F}' >>= N$

$$\mathbb{E}(\overline{\mathbb{F}'[M] >>= N}) \overset{\overline{\text{bind}}}{=} \mathbb{E}(\overline{\mathbb{F}'[M]}\,\texttt{>>=}\,\overline{N}) \overset{\mathbb{E}\ \textbf{bind}}{=} \mathbb{E}(\overline{\mathbb{F}'[M]}) \overset{\text{I.H.}}{=} \overline{M}$$

Case 1.3 $\mathbb{F} = \mathsf{catch}\ \mathbb{F}'\ N$ Analogously to case 1.2.

Case 2 $\mathbb{E} = \mathbb{F}\,[B\ \mathbb{E}']$ for $B \in \{\mathsf{block}, \mathsf{unblock}\}$

$$\mathbb{E}(\overline{\mathbb{F}\,[B\ \mathbb{E}'[M]]}) \overset{\text{lemma 4.1}}{=} \mathbb{E}(\overline{\mathbb{E}'[M]}) \overset{\text{I.H.}}{=} \overline{M}$$

(ii) Analogously to (i).

□

To include $>>=$, catch, block or $\mathsf{unblock}$ in the argument expression a more specific version using **ContPat** is needed. Hence there is a similar lemma for each constant of **ContPat**, e.g. for `r/>>=`:

Lemma 4.3. *Let M, M' and N be arbitrary Haskell expressions. Then the following holds*

(i) $\mathcal{E}_C \vdash \mathbb{E}[\texttt{r/>>=}](\overline{\mathbb{E}\,[\mathsf{return}\ M >>= M']}) = \overline{\mathsf{return}\ M >>= M'}$

(ii) $\mathcal{E}_C \vdash \mathbb{E}[\texttt{r/>>=}](\overline{\mathbb{E}\,[\mathsf{return}\ M >>= M']} \leftarrow \overline{N}) = \overline{\mathbb{E}\,[N]}$

Proof. Similar to the proof of lemma 4.2. □

An analogous property also holds true for $\mathbb{F}$ contexts, which are in fact only a subset of $\mathbb{E}$ contexts.

Lemma 4.4. *Let M be a Haskell expression not having* catch *or* >>= *as the head symbol (i.e. M is not of the form M' >>= N' or* catch M' N' *for some expressions M', N') and N an arbitrary Haskell expression. Then the following holds:*

(i) $\mathcal{E}_C \vdash \mathbb{F}(\overline{\mathbb{F}\,[M]}) = \overline{M}$

(ii) $\mathcal{E}_C \vdash \mathbb{F}(\overline{\mathbb{F}\,[M]} \leftarrow \overline{N}) = \overline{\mathbb{F}\,[N]}$

Proof. Analogous to the proof lemma 4.2 □

Now we want to show that $\mathbb{E}\,[\mathsf{unblock}\ \mathbb{F}]$ context are represented by their operational correspondent $\mathbb{U}$. Therefore a few lemmas have to be established as well. But beforehand, we need an definition for an modified version of the $\mathbb{E}$ context which does not contain an `unblock`:

$$\mathbb{G} \quad ::= \quad \mathbb{F} \,|\, \mathbb{F}\,[\texttt{block}\ \mathbb{G}]$$

Now the first lemma on $\mathbb{U}$ contexts states that $\mathbb{U}$ contexts skip $\mathbb{G}$ contexts:

Lemma 4.5. *Let M and N be arbitrary Haskell expressions. Then the following holds:*

(i) $\mathcal{E}_C \vdash \mathbb{U}(\overline{\mathbb{G}\,[M]}) = \mathbb{U}(\overline{M})$

(ii) $\mathcal{E}_C \vdash \mathbb{U}(\overline{\mathbb{G}\,[M]} \leftarrow \overline{N}) = \overline{\mathbb{G}\,[M']}$ *where* $\overline{M'} = \mathbb{U}(\overline{M} \leftarrow \overline{N})$

Proof. Straightforward induction on the structure of $\mathbb{G}$ analogously to lemma 4.1. □

The next lemma states that $\mathbb{U}$ contexts cannot be evaluated for argument terms that do not contain `unblock` at some "reachable" position, where "reachable" means inside a $\mathbb{G}$ context.

Lemma 4.6. *Let M be a Haskell expression such that there is no Haskell expression M' and no context $\mathbb{G}'$ for which $M \equiv \mathbb{G}'[\mathsf{unblock}\ M']$. Then the following holds:*

(i) $\mathcal{E}_C \nvdash \mathbb{U}(\overline{\mathbb{G}\,[M]}) : \mathbf{Exp}$

(ii) $\mathcal{E}_C \nvdash \mathbb{U}(\overline{\mathbb{G}\,[M]} \leftarrow \overline{N}) : \mathbf{Exp}$

Proof. (i) Proof by contradiction:

Assume $\mathcal{E}_C \vdash \mathbb{U}(\overline{\mathbb{G}\,[M]}) : \mathbf{Exp}$. Because of the declaration of $\mathbb{U}(\,\cdot\,)$ this can only be due to some deducible equality $\mathcal{E}_C \vdash \mathbb{U}(\overline{\mathbb{G}\,[M]}) = \mathbb{U}(\texttt{unblock}\ \overline{M'})$ since then $\mathcal{E}_C \vdash \mathbb{U}(\texttt{unblock}\ \overline{M'}) : Exp$ by either **$\mathbb{U}$ unblock** or **$\mathbb{U}$ match**. $\mathcal{E}_C \vdash \mathbb{U}(\overline{\mathbb{G}\,[M]}) = \mathbb{U}(\texttt{unblock}\ \overline{M'})$ is only derivable by repeated use of the sentences **$\mathbb{U}$ bind**, **$\mathbb{U}$ catch** and **$\mathbb{U}$ block**. Thus there is some context $\mathbb{G}'$ s.t. $\mathbb{G}\,[M] \equiv \mathbb{G}\,[\mathbb{G}'[\mathsf{unblock}\ M']]$ and consequently $M \equiv \mathbb{G}'[\mathsf{unblock}\ M']]$ which contradicts the assumption on the form of M. Hence the initial assumption was wrong. Therefore $\mathcal{E}_C \nvdash \mathbb{U}(\overline{\mathbb{G}\,[M]}) : \mathbf{Exp}$.

(ii) Analogously.

□

The next lemma finally states the aspired qualified equivalence of $\mathbb{E}\,[\mathsf{unblock}\ \mathbb{F}]$ and $\mathbb{U}$ in the style of lemma 4.2.

Lemma 4.7. *Let M be an expression not having* >>=, catch, block *or* unblock *as the head symbol. Then the following holds*

(i) $\mathcal{E}_C \vdash \mathbb{U}(\overline{\mathbb{E}\,[\mathsf{unblock}\ \mathbb{F}\,[M]]}) = \overline{M}$

(ii) $\mathcal{E}_C \vdash \mathbb{U}(\overline{\mathbb{E}\,[\mathsf{unblock}\ \mathbb{F}\,[M]]} \leftarrow \overline{N}) = \overline{\mathbb{E}\,[\mathsf{unblock}\ \mathbb{F}\,[N]]}$

Proof. Both statements can be proven by induction on the structure of $\mathbb{E}$ and $\mathbb{F}$:

(i) **Case 1** $\mathbb{E} = \mathbb{F}'$

Case 1.1 $\mathbb{F}' = [\,\cdot\,]$

$$\mathbb{U}(\overline{\mathsf{unblock}\ \mathbb{F}\,[M]}) \overset{\overline{\text{unblock}}}{=} \mathbb{U}(\,\texttt{unblock}\ \overline{\mathbb{F}\,[M]}) \overset{\substack{\text{lem. 4.6}\\ \mathbb{U}\ \textbf{match}}}{=} \mathbb{F}(\overline{\mathbb{F}\,[M]}) \overset{\text{lem. 4.4}}{=} \overline{M}$$

Note that lemma 4.6 is applicable to $\mathbb{U}(\overline{\mathbb{F}\,[M]})$ since $\mathbb{F}$ is also a $\mathbb{G}$ context and because of the restriction of the shape of M there is also no expression M' and no context $\mathbb{G}'$ s.t. $M \equiv \mathbb{G}'[\mathsf{unblock}\ M']$ (Suppose there are such M' and $\mathbb{G}$. Consider the case $\mathbb{G}' = [\,\cdot\,]$. Then $M \equiv \mathsf{unblock}\ M'$ would have $\mathsf{unblock}$ as head symbol. The other case $\mathbb{G}' \neq [\,\cdot\,]$ clearly implies that M has $>>=$, catch or block as head symbol).

Case 1.2 $\mathbb{F}' = \mathbb{F}'' >>= N$

$$\mathbb{U}(\overline{\mathbb{F}''[\mathsf{unblock}\ \mathbb{F}\,[M]] >>= N}) \overset{\overline{\text{bind}}}{=} \mathbb{U}(\overline{\mathbb{F}''[\mathsf{unblock}\ \mathbb{F}\,[M]]}\,\texttt{>>=}\,\overline{N}) \overset{\mathbb{U}\ \textbf{bind}}{=} \mathbb{U}(\overline{\mathbb{F}''[\mathsf{unblock}\ \mathbb{F}\,[M]]}) \overset{\text{I.H.}}{=} \overline{M}$$

Case 1.3 $\mathbb{F}' = \mathsf{catch}\ \mathbb{F}''\ N$ Analogously to case 1.2

Case 2 $\mathbb{E} = \mathbb{F}'[\mathsf{block}\ \mathbb{E}']$

Case 2.1 $\mathbb{F}' = [\,\cdot\,]$

$$\mathbb{U}(\overline{\mathsf{block}\ \mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]}) \overset{\overline{\text{block}}}{=} \mathbb{U}(\,\texttt{block}\ \overline{\mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]}) \overset{\mathbb{U}\ \textbf{block}}{=} \mathbb{U}(\overline{\mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]}) \overset{\text{I.H.}}{=} \overline{M}$$

Case 2.2 $\mathbb{F}' = \mathbb{F}'' >>= N$ Observe that $\mathbb{F}''[\mathsf{block}\ [\,\cdot\,]] >>= N$ is a $\mathbb{G}$ context.

$$\mathbb{U}(\overline{\mathbb{F}''[\mathsf{block}\ \mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]] >>= N}) \overset{\text{lem. 4.5}}{=} \mathbb{U}(\overline{\mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]}) \overset{\text{I.H.}}{=} \overline{M}$$

Case 2.3 $\mathbb{F}' = \mathsf{catch}\ \mathbb{F}''\ N$ Analogously to case 2.2

Case 3 $\mathbb{E} = \mathbb{F}'[\mathsf{unblock}\ \mathbb{E}']$

Case 3.1 $\mathbb{F}' = [\,\cdot\,]$

$$\mathbb{U}(\overline{\mathsf{unblock}\ \mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]}) \overset{\substack{\overline{\text{unblock}}\\ \mathbb{U}\ \textbf{unblock}}}{=} \mathbb{U}(\overline{\mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]}) \overset{\text{I.H.}}{=} \overline{M}$$

Case 3.2 $\mathbb{F}' = \mathbb{F}'' >>= N$ Observe that $\mathbb{F}'' >>= N$ is a $\mathbb{G}$ context.

$$\mathbb{U}(\overline{\mathbb{F}''[\mathsf{unblock}\ \mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]] >>= N}) \overset{\text{lem. 4.5}}{=} \mathbb{U}(\overline{\mathsf{unblock}\ \mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]})$$

$$\overset{\substack{\overline{\text{unblock}}\\ \mathbb{U}\ \textbf{unblock}}}{=} \mathbb{U}(\overline{\mathbb{E}'[\mathsf{unblock}\ \mathbb{F}\,[M]]}) \overset{\text{I.H.}}{=} \overline{M}$$

Case 3.3 $\mathbb{F}' = \mathsf{catch}\ \mathbb{F}''\ N$ Analogously to case 3.2

(ii) Analogously to (i).

□

Now program states can be considered. Therefore the definition of the symbol $\overline{\cdot}$ has to be extended — at first to cover threads, MVars and asynchronous exceptions as well.

$$\begin{aligned}
\overline{\circ} &= \mathsf{no}; \quad \overline{\bullet} = \mathsf{yes} && (\overline{\text{stuck}})\\
\overline{(\!|M|\!)^b_t} &= (\!|\mathsf{val} : \overline{M} \quad \mathsf{tid} : \overline{t} \quad \mathsf{stuck} : \overline{b}|\!) && (\overline{\text{thread}})\\
\overline{\mathbb{O}_t} &= (\!|\mathsf{val} : \varepsilon \quad \mathsf{tid} : \overline{t} \quad \mathsf{stuck} : \mathsf{no}|\!) && (\overline{\text{empty thread}})\\
\overline{\langle M\rangle_m} &= \langle \overline{M}\rangle_{\overline{m}} && (\overline{\text{MVar}})\\
\overline{\langle\rangle_m} &= \langle\rangle_{\overline{m}} && (\overline{\text{empty MVar}})\\
\overline{\langle\!|t \lightning e|\!\rangle} &= \langle\!|\overline{t} \lightning \overline{e}|\!\rangle && (\overline{\text{asynch exc}})\\
\overline{P\,|\,Q} &= \overline{P}\,|\,\overline{Q} \qquad (P, Q \text{ contain no } \nu) && (\overline{\text{parallel}})
\end{aligned}$$

Unfortunately the corresponding definition for program states is a bit cumbersome. Instead of a term of sort **State** the correspondent of a program state is a set of those terms. To be able to describe this more concisely a relation $\approx$ between program states and terms of sort **State** is introduced.

$$\frac{\begin{array}{c} P \text{ contains no } \nu \\ \mathcal{E}_C \vdash s_l : \textbf{StringConst}, l = 1, 2 \\ d > \max\{\mathsf{tInd}(t_i) | 1 \le i \le n\} \qquad d' > \max\{\mathsf{mInd}(m_j) | 1 \le j \le k\} \end{array}}{\begin{array}{c} \nu t_1 . \ldots \nu t_n . \nu m_1 . \ldots \nu m_k . P \\ \approx \{\!|\mathsf{pool} : \overline{P} \quad \mathsf{in} : \langle s_1 \rangle_{\mathsf{I}} \quad \mathsf{out} : \langle s_2 \rangle_{\mathsf{O}} \quad \mathsf{tgen} : \langle \overline{d} \rangle_{\mathsf{ThGen}} \quad \mathsf{mgen} : \langle \overline{d'} \rangle_{\mathsf{MVGen}} |\!\} \end{array}} \text{(approx)}$$

For the sake of convenience $\approx$ will be used symmetrically in the following. So $s \approx P$ means $P \approx s$.

Based on this the definition of the corresponding set of **State** terms for a program state can be defined like this:

$$\overline{P} = \{s \,|\, \exists Q, s'. P \equiv Q \wedge Q \approx s' \wedge \mathcal{E}_C \vdash s' = s\} \qquad (\overline{\text{state}})$$

It is easy to see that every program state P is $\equiv$-equal to a program state of the form $\nu t_1 . \ldots \nu t_n . \nu m_1 . \ldots \nu m_k . P'$ where P' does not contain any ν; it is a simple matter of repeatedly applying (Alpha) and (Extrude) of the definition of $\equiv$ (cf. [MJMR01]). Hence $\overline{P}$ is a nonempty set for every P. So such a set contains all terms that only differ in thread and MVar ids and id generator respectively and the input and the output field.

The equivalence of the original operational semantics and the rewrite semantics is now the following:

Theorem 4.8. *Let P and Q be program states and $\mathcal{R}_C$ the rewrite theory of the Haskell semantics given so far. Then, excluding functional nontermination, the following holds:*

$$P \to^* Q \quad \textit{iff} \quad \exists s \in \overline{P}, s' \in \overline{Q}. \quad \mathcal{R}_C \vdash s \longrightarrow s'$$

Proof. (i) The "only if" direction can be proven by induction on the transition length. The case $P \to^0 Q$ is trivial (by (Refl) of GRL's semantics). Let us assume a transition of length $n+1$, hence:

$$P \to^n R \to Q \qquad \text{for some } R$$

By induction we have $\mathcal{R}_C \vdash s_1 \longrightarrow s_2$ with $s_1 \in \overline{P}$ and $s_2 \in \overline{R}$. It now suffices to show that $\mathcal{R}_C \vdash s_2 \longrightarrow s_3$ for some $s_3 \in \overline{Q}$ since by rule (Trans) of GRL's semantics we get the aspired $\mathcal{R}_C \vdash s_1 \longrightarrow s_3$. Therefore the transition $R \to Q$ has to be analysed by case distinction, examining every transition rule (excluding structural rules). It may suffice for this purpose to restrict the case analysis to the rule (Bind). The argument for the remaining rules is analogous to this example.

As (Bind) is assumed the following holds by the structural congruence $\equiv$ and structural transition rules of $\to$:

$$\begin{aligned} R &\equiv \overbrace{\nu t_1 . \ldots \nu m_k . P' \,|\, (\!|\mathbb{E}\,[\mathsf{return}\ N >\!\!>\!= M]\,|\!)^b_t}^{R'} \\ Q &\equiv \underbrace{\nu t_1 . \ldots \nu m_k . P' \,|\, (\!|\mathbb{E}\,[M\ N]\,|\!)^b_t}_{Q'} \end{aligned}$$

Now it has to be shown that there is some s_3 s.t. $Q' \approx s_3$ and $\mathcal{R}_C \vdash s_2 \longrightarrow s_3$. To do this s_2 has to be examined:

$$\begin{aligned}
R' \approx s_2 &\overset{\text{(approx)}}{=} \{\!|\mathsf{pool} : \overline{P'} \mid \overline{(\!|\mathbb{E}\,[\mathsf{return}\ N >\!>\!= M]\,|\!)^b_t} \quad \mathsf{in} : \ldots \quad \mathsf{out} : \ldots \quad \mathsf{tgen} : \ldots \quad \mathsf{mgen} : \ldots |\!\} \\
&\overset{\overline{\text{(thread)}}}{=} \{\!|\mathsf{pool} : \overline{P'} \mid (\!|\mathsf{val} : \overline{\mathbb{E}\,[\mathsf{return}\ N >\!>\!= M]} \quad \mathsf{stuck} : \overline{b} \quad \mathsf{tid} : \langle \overline{t} \rangle_{\mathsf{Th}}|\!) \quad \ldots |\!\} =: s_2' \\
&\text{and} \quad \mathbb{E}[\mathtt{r}/\mathtt{>>=}](\overline{\mathbb{E}\,[\mathsf{return}\ N >\!>\!= M]}) \overset{\text{lem.4.3}}{=} \overline{\mathtt{return}\ N >\!>\!= M} \\
&\overset{\overline{\text{(bind)}}}{\overset{\overline{\text{(return)}}}{=}} \mathtt{return}\ \overline{N} >\!>\!= \overline{M}
\end{aligned}$$

Hence by the rewrite sentence **Bind** we get $\mathcal{R}_C \vdash s_2' \longrightarrow s_3'$ where

$$\begin{aligned}
s_3' &= \{\!|\mathsf{pool} : \overline{P'} \mid (\!|\mathsf{val} : \mathbb{E}[\mathtt{r}/\mathtt{>>=}](\overline{\mathbb{E}\,[\mathsf{return}\ N >\!>\!= M]} \leftarrow \overline{M}\ \overline{N}) \quad \mathsf{stuck} : \overline{b} \quad \mathsf{tid} : \langle \overline{t} \rangle_{\mathsf{Th}}|\!) \quad \ldots |\!\} \\
&\overset{\text{lem.4.3}}{\overset{\overline{\text{app}}}{=}} \{\!|\mathsf{pool} : \overline{P'} \mid (\!|\mathsf{val} : \overline{\mathbb{E}\,[M\ N]} \quad \mathsf{stuck} : \overline{b} \quad \mathsf{tid} : \langle \overline{t} \rangle_{\mathsf{Th}}|\!) \quad \ldots |\!\} =: s_3
\end{aligned}$$

By $\mathcal{E}_C \vdash s_2 = s_2', s_3 = s_3'$ and (Eq) of GRL's semantics we truly have $\mathcal{R}_C \vdash s_2 \longrightarrow s_3$. Furthermore $Q' \approx s_3$ holds. As $R' \approx s_2$ we have $s_2 \in \overline{R}$ and $s_3 \in \overline{Q}$.

(ii) The "if" direction is a bit more involved. It can be proven by induction on the size of the proof tree of $\mathcal{R}_C \vdash s \longrightarrow s'$ assuming that we have fixed a particular one for each such statement. To make things a bit easier we assume a certain shape of the proof trees. They are assumed to not use true concurrency, that is the deduction rule (Nested Repl) is only used to instantiate rewrite sentences. True concurrency can then be mimicked by interleaving, i.e., using the rule (Trans). Also (Cong) is assumed to be used linearly, that is all argument rewrites except one are assumed to be reflexive. Every other usage of (Cong) can be mimicked by interleaving as well. Thus these assumptions can be made without loss of generality. Then we have the following cases:

Case 1 (Refl) $\mathcal{R}_C \vdash s \longrightarrow s$

Let P be s.t. $s \in \overline{P}$. Then $P \rightarrow^0 P$.

Case 2 (Trans)

$$\frac{\mathcal{R}_C \vdash s_1 \longrightarrow s_2 \qquad \mathcal{R}_C \vdash s_2 \longrightarrow s_3}{\mathcal{R}_C \vdash s_1 \longrightarrow s_3}\ \text{(Trans)}$$

By induction hypothesis $P_1 \rightarrow^* P_2$ and $P_2 \rightarrow^* P_3$ for $\overline{P_i} \ni s_i,\ i = 1, 2, 3$ holds. By transitivity we get $P_1 \rightarrow^* P_3$.

Case 3 (Eq)

$$\frac{\mathcal{E}_C \vdash s_1 = s_1' \qquad \mathcal{R}_C \vdash s_1' \longrightarrow s_2' \qquad \mathcal{E}_C \vdash s_2' = s_2}{\mathcal{R}_C \vdash s_1 \longrightarrow s_2}\ \text{(Eq)}$$

By induction hypothesis $P_1' \rightarrow^* P_2'$ for $\overline{P_i'} \ni s_i',\ i = 1, 2$ holds. But since $\mathcal{E}_C \vdash s_i = s_i'$ also $s_i \in \overline{P_i'}$ holds.

Case 4 (Cong, Nested Repl) Due to the assumption that is made on the shape of the proof tree, only the following constellation is possible:

$$
\dfrac{(\text{Refl})\ \dfrac{\dots}{\dots}\quad \dots \qquad \begin{matrix} \vdots \\ \dfrac{(\text{Refl})\ \dfrac{\dots}{\dots}\quad \dots \qquad \dfrac{\begin{matrix} r : t_0 \longrightarrow t_0' \Leftarrow \bigwedge_{i\in I} u_i = v_i \in R_H \wedge \bigwedge_{j \in J} w_j : s_j \\ \overline{\theta}(t_0) = t_1, \overline{\theta}(t_0') = t_1' \qquad \mathcal{E}_C \vdash \overline{\theta}(w_j) : s_j \quad j \in J \\ \mathcal{E}_C \vdash \overline{\theta}(u_i) = \overline{\theta}(v_i) \quad i \in I \end{matrix}}{\mathcal{R}_C \vdash t_1 \longrightarrow t_1'}\ (\text{Nested Repl})}{\vdots}\ (\text{Cong}) \\ \mathcal{R}_C \vdash t_n \longrightarrow t_n' \end{matrix}}{\mathcal{R}_C \vdash s \longrightarrow s'}\ (\text{Cong})
$$

where $n \geq 0$ and $s = t_{(n+1)}$ and $s' = t'_{(n+1)}$. By writing

$$(\text{Refl})\ \dfrac{\dots}{\dots} \quad \dots$$

is meant the justification for all other direct subterms of t_{i+1} different from t_i. Due to our assumption on the shape of proof trees only one of the direct subterms of t_{i+1} — namely t_i — is non-reflexively rewritten. That is, assuming $t_{i+1} = f(t_{i+1,1}, \dots, t_{i+1,j-1}, t_i, t_{i+1,j+1}, \dots, t_{i+1,k})$ and consequently $t'_{i+1} = f(t_{i+1,1}, \dots, t_{i+1,j-1}, t_i', t_{i+1,j+1}, \dots, t_{i+1,k})$ the above statement is a shorthand for the $k-1$ statements

$$(\text{Refl})\ \dfrac{t_{i+1,l} \in T_\Sigma(X)}{\mathcal{R}_C \vdash t_{i+1,l} \longrightarrow t_{i+1,l}} \qquad \text{for every } l \in \{1,\dots,k\} \backslash \{j\}$$

Note that the restriction to ground substitutions in the application of the rule (Nested Repl) above is w.l.o.g. as the considered terms s and s' are ground. Therefore all respective subterms t_i and t_i' for $n \leq i \leq 1$ are ground too.

Now a case distinction has to be made for the rewrite sentence that is chosen. As an example the case for the sentence **Bind** is analysed. All other cases can be treated similarly[6].

By (Cong) and (Nested Repl) using sentence **Bind** we get:

$$
\begin{aligned}
\mathcal{E}_C \vdash s &= \overbrace{\{\!|\mathsf{pool} : \overline{P} \,|\, (\!|\mathsf{val} : \overline{M} \quad \mathsf{stuck} : \bar{b} \quad \mathsf{tid} : \langle \bar{t} \rangle_{\mathsf{Th}} |\!) \quad \dots |\!\}}^{s_0} \\
\mathcal{E}_C \vdash s' &= \underbrace{\{\!|\mathsf{pool} : \overline{P} \,|\, (\!|\mathsf{val} : \overline{N} \quad \mathsf{stuck} : \bar{b} \quad \mathsf{tid} : \langle \bar{t} \rangle_{\mathsf{Th}} |\!) \quad \dots |\!\}}_{s_0'}
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathcal{E}_C \vdash \mathbb{E}[\mathtt{r}/\mathtt{>>=}](\overline{M}) = \mathtt{return}\ \overline{N'}\, \mathtt{>>=} \,\overline{M'} \\
&\mathcal{E}_C \vdash \mathbb{E}[\mathtt{r}/\mathtt{>>=}](\overline{M} \leftarrow \overline{M'}\ \overline{N'}) = \overline{N}
\end{aligned}
$$

By $(\overline{\text{bind}})$,$(\overline{\text{return}})$, $(\overline{\text{app}})$ and lemma 4.3 we get:

$$
\begin{aligned}
M &= \mathbb{E}\,[\mathsf{return}\ N' \mathrel{>>=} M'] \\
N &= \mathbb{E}\,[M'\ N']
\end{aligned}
$$

[6]Note that for the sentences **Eval** and **Raise** the theorems proven in section 3.6 have to be used but in the more general context of the extended MEL theory $\mathcal{E}_C$ and on the other hand the extended definitions of expressions (i.e. M, N) and values (i.e. U, V). As this small extension to the syntax and the functional semantics is of course not considered in [MLJ99] it would have to be added in an obvious way. Just look at the corresponding definitions of values U, V and define the transformation of expressions M, N to these values, which is only an transformation of those arguments of primitives that the respective primitive is strict in to whnf. Consequently exceptional behaviour of those arguments must be propagated appropriately. Establishing the theorems for $\mathcal{E}_C$ is then trivial as this is exactly the way how $\mathcal{E}_C$ was conceived.

Hence:

$$s_0 \approx Q = \nu t_1. \dots \nu m_k.P \mid (\!|\mathbb{E}\,[\mathsf{return}\ N' >\!\!>\!= M']\,|\!)^b_t$$
$$s'_0 \approx R = \nu t_1. \dots \nu m_k.P \mid (\!|\mathbb{E}\,[M'\ N']\,|\!)^b_t$$

By (Bind) and structural rules we have $Q \to R$. Moreover $s \in \overline{Q}$ and $s' \in \overline{P}$.

□

4.4 Executability and beyond

Now it has to be checked whether the developed GRT $\mathcal{R}_C$ of the Concurrent Haskell semantics is in fact executable. The necessary properties for the MEL theory $\mathcal{E}_H$ were already discussed in section 3.5. The argument for the respective properties of the extended theory $\mathcal{E}_C$ is analogous.

What is still missing is a discussion concerning the required property of coherence for the rewrite sentences w.r.t. the MEL subtheory. But showing this also turns out to be rather trivial, as there are no equational sentences on **State** terms except associativity, commutativity and identity axioms which do not destroy coherence. Well, due to (Cong) of MEL also equational sentences on **Exp** have to be considered. But as the rewrite sentences of $\mathcal{R}_C$ only consider **Exp** terms being in nf w.r.t. rewriting system induced by the MEL theory, this does not affect coherence either.

Still there is a shortcoming of the semantics. It does not cover functional nontermination, that is expressions e for which we have $\mathcal{E}_C \nvdash \mathcal{F}[\![e]\!]_{\mathsf{whnf}} : \textbf{ExpUExc}$. But the problem can be solved by adding the following equational respectively rewrite sentences that use a new symbol $\cdot\Uparrow$ to denote divergence:

$$\cdot\Uparrow : \textbf{Exp} \ \to\ \textbf{Bool}$$

$$\frac{\mathcal{F}[\![e_1]\!]_{\mathsf{whnf}} : \textbf{ExpUExc}}{e_1\Uparrow = \mathsf{false}}\ \textbf{div1} \qquad\qquad \frac{\mathsf{otherwise}}{e_1\Uparrow = \mathsf{true}}\ \textbf{div2}$$

$$\frac{\mathbb{E}(e_1) = e_3 \qquad e_3\Uparrow = \mathsf{true} \qquad \mathbb{E}(e_1 \leftarrow \mathtt{throw}\,\mathsf{exp}(ec_1)) = e_2}{\mathsf{val} : e_1 \ \longrightarrow\ \mathsf{val} : e_2}\ \textbf{Raise div}$$

This extension to $\mathcal{R}_C$ will be denoted $\mathcal{R}'_C$. But of course reintroducing diverging expressions destroys executability. So these sentences are only mentioned to show that diverging is definable in GRL. With this extension the following can then be easily proven:

Theorem 4.9. *Let P and Q be program states Then the following holds:*

$$P \to^* Q \quad \textit{iff} \quad \exists s \in \overline{P}, s' \in \overline{Q}.\ \ \mathcal{R}'_C \vdash s \ \longrightarrow\ s'$$

This is the same statement as in theorem 4.8 but without the restriction to non-diverging expressions.

5 Conclusion

The rewrite theory of the semantics of Haskell and its concurrency extension was thoroughly presented and legitimated by showing equalities to existing semantics. Due to its properties this theory can be translated into an equivalent Maude specification very easily. Thereby a hole set of powerful tools for analysing the object language is produced. A survey of the possibilities a Maude specification offers is given in [MR07]. On account of its modular style it is well suited to analyse modifications of the semantics of the language or possible extensions by new features. Moreover the existence of rewrite semantics for a large set of other languages (cf. [MR07]) may serve as an inspiration on how to include new language features.

The presented semantics covers a large number of aspects of the Haskell language including laziness, pattern matching, mutual recursion, imprecise (a)synchronous exceptions, I/O and concurrency. Thereby rewriting logic has proved to be a well suited framework for an amalgamation of these different computational aspects. The deterministic parts were formulated in the equational sublogic whereas the nondeterministic parts were mostly defined using rewrite sentences. But it could be observed that this distinction is not always that rigid: For example the imprecise definition of exceptional behaviour is truly a kind of nondeterminism. But nevertheless it was chosen to define this equationally and to resolve this nondeterminism by a set construction. However the inherent nondeterminism is only deferred. It re-emerges in the Concurrent Haskell semantics where the set of possible exceptions induces a different rewrite for every exception in the set. Hence the rewrite sentences of the semantics only constitute a small layer on top of the functional semantics that aside from introducing the concurrent part of the semantics of the concurrency primitives also surfaces the deferred concurrency of the functional semantics.

Nevertheless a large part of the semantics of Concurrent Haskell was not treated — the static semantics. Hence future work on this subject is ought to be anxious for particularly formulating the type calculus of Haskell as a rewrite theory. This could also help to formulate the dynamic rewrite semantics of Haskell far more gracefully. Furthermore an amalgamation of both dynamic and static semantics inside the very same framework is certainly of notably interest as both are exceptionally interwoven e.g. in the case of ad-hoc polymorphism.

Bibliography

[Ben04] Andrew Douglas Bennett. Haskell-RL - an equational specification of haskell in maude. Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[Bir84] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 6(4):487–504, 1984.

[BM03] R. Bruni and J. Meseguer. Generalized rewrite theories, 2003.

[BN99] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

[BR00] Adam Bakewell and Colin Runciman. A space semantics for core haskell. *Electr. Notes Theor. Comput. Sci.*, 41(1), 2000.

[BT80] Jan A. Bergstra and J. V. Tucker. A characterisation of computable data types by means of a finite equational specification method. In *ICALP*, pages 76–90, 1980.

[CDE$^+$] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.3).*

[HH92] Kevin Hammond and Cordelia Hall. A dynamic semantics for haskell (draft), 1992.

[HM07] Joe Hendrix and José Meseguer. On the completeness of context-sensitive order-sorted specifications. In *RTA*, pages 229–245, 2007.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

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

[JW92] Simon L. Peyton Jones and Philip Wadler. A static semantics for haskell. Technical report, 1992.

[JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.

[MB03] José Meseguer and Christiano O. Braga. Modular rewriting semantics of programming languages. Submitted for publication, 2003.

[Mes97] José Meseguer. Membership algebra as a logical framework for equational specification. In *WADT '97: Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 18–61, London, UK, 1997. Springer-Verlag.

[MJMR01] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy. Asynchronous exceptions in haskell. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 274–285, 2001.

[MJT04] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Extending the haskell foreign function interface with concurrency. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 22–32, New York, NY, USA, 2004. ACM Press.

[MLJ99] Andrew Moran, Søren B. Lassen, and Simon Peyton Jones. Imprecise exceptions, co-inductively. In *HOOTS '99, Higher Order Operational Techniques in Semantics, ENTCS 26*, pages 122–141, 1999.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MR07] José Meseguer and Grigore Rou. The rewriting logic semantics project. *Theor. Comput. Sci.*, 373(3):213–237, 2007.

[ŞRM07] Traian Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics – extended abstract. In *SOS'07*, number to appear in ENTCS, 2007.

[Wec92] Wolfgang Wechler. *Universal Algebra for Computer Scientists (EATCS Monographs in Theoretical Computer Science).* Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 1992.