An Executable Rewriting Logic Semantics for Concurrent Haskell

Patrick Bahr
s0404888@inf.tu-dresden.de

Technische Universität Dresden

January 11, 2008
Outline

1. Introduction
   - Motivation
   - Preliminaries

2. Rewriting Logic Semantics of Haskell
   - Pure Haskell
   - Concurrent Haskell
   - Properties of the Semantic Theory

3. Conclusion
   - Summary
   - Future Work
Outline

1 Introduction
   - Motivation
   - Preliminaries

2 Rewriting Logic Semantics of Haskell
   - Pure Haskell
   - Concurrent Haskell
   - Properties of the Semantic Theory

3 Conclusion
   - Summary
   - Future Work
Why Haskell?

- declarative reading
- use as specification language
- program transformation to obtain efficient implementation
Why Haskell?

- declarative reading
- use as specification language
- program transformation to obtain efficient implementation

- need for exact semantics to justify program transformations!
- yet: semantics of Haskell is well-studied
Why yet another semantics?

- collect different aspects of the semantics in one semantic framework
- view the semantics from a different perspective
- use the semantics as input for tools to analyse the language
Why Rewriting Logic?

- mature well-studied semantic framework
- there are rewrite semantics for many other languages
- implementation for rewriting logic is available
- remote goal: study relationship between Concurrent Haskell and Rewriting Logic
Rewriting Logic

- atomic formulae: $t \rightarrow t'$ where $t, t' \in T_\Sigma(X)$
- in our context:
  - $t$ and $t'$ encode program states
  - $t \rightarrow t'$ reads: “During the computation the program state changes from $t$ to $t'$.”
- terms are not taken as pure syntax: reasoning modulo an equational subtheory

\[ \leadsto \text{Membership Equational Logic (MEL):} \]

- many-kindred term language
- atomic formulae:
  - $t = t'$
  - $t : s$ “$t$ has sort $s$”
Definition

A membership equational signature (or MEL signature) is a triple \( \Omega = (\Sigma, S, \prec) \), where

- \( S \) is a finite set of sorts,
- \( \prec \) is a strict order on \( S \),
- \( \Sigma = \{ \Sigma_{w,k} \mid (w,k) \in K^* \times K \} \) is a \( K^* \times K \)-indexed family of function symbols and
- \( K := S/\equiv_{\prec} \) is the set of kinds induced by the equivalence closure of \( \prec \).

\([s]\) will denote the equivalence class of \( s \) w.r.t \( \equiv_{\prec} \), i.e. its kind.
Sorts and Kinds — Example

Example

- $S = \{s_1, s_2, s_3, s_4, s_5\}$ — the set of sorts and
- $< \text{ is s.t.}$
  - $s_2 < s_1$,
  - $s_3 < s_1$,
  - $s_5 < s_4$.
Sorts and Kinds — Example

**Example**

- $S = \{s_1, s_2, s_3, s_4, s_5\}$ — the set of sorts and
- $< $ is s.t.
  - $s_2 < s_1$,
  - $s_3 < s_1$,
  - $s_5 < s_4$.

$\Rightarrow K = \{[s_1], [s_4]\}$, where $[s_1] = \{s_1, s_2, s_3\}$ and $[s_4] = \{s_4, s_5\}$. 
The following are membership equational sentences (or MEL sentences):

\[(\forall X) \; t = t' \iff \bigwedge_{i \in I} u_i = v_i \land \bigwedge_{j \in J} w_j : s_j\]  
(Equation)

\[(\forall X) \; t'' : s \iff \bigwedge_{i \in I} u_i = v_i \land \bigwedge_{j \in J} w_j : s_j\]  
(Membership)

A membership equational theory (or MEL theory) \( \mathcal{E} \) is a pair \( (\Omega, E) \) where \( \Omega \) is a MEL signature and \( E \) a is set of MEL sentences.
The following are **membership equational sentences** (or MEL sentences):

\[
\begin{align*}
  u_i &= v_i \\
  w_j &:: s_j \\
  \implies t = t' \\
  (\forall X) \ t'' : s &\iff \bigwedge_{i \in I} u_i = v_i \land \bigwedge_{j \in J} w_j : s_j
\end{align*}
\]

(Equation) (Membership)

**Definition**

A **membership equational theory** (or MEL theory) \( \mathcal{E} \) is a pair \((\Omega, E)\) where \(\Omega\) is a MEL signature and \(E\) a is set of MEL sentences.
The following are membership equational sentences (or MEL sentences):

\[
\frac{u_i = v_i \quad w_j : S_j}{t = t'}
\]  
(Equation)

\[
\frac{u_i = v_i \quad w_j : S_j}{t'' : s}
\]  
(Membership)

A membership equational theory (or MEL theory) \( \mathcal{E} \) is a pair \((\Omega, E)\) where \(\Omega\) is a MEL signature and \(E\) a is set of MEL sentences.
MEL Semantics in a Nutshell

- if \( t = t' \) \( \iff \bigwedge_{i \in I} u_i = v_i \land \bigwedge_{j \in J} w_j : s_j \in E \) and \( E \vdash u_i = v_i \) and \( E \vdash w_j : s_j \) hold then also \( E \vdash t = t' \).

- if \( t : s \) \( \iff \bigwedge_{i \in I} u_i = v_i \land \bigwedge_{j \in J} w_j : s_j \in E \) and \( E \vdash u_i = v_i \) and \( E \vdash w_j : s_j \) hold then also \( E \vdash t : s \).

- = is a congruence,

- sort membership is preserved by =,
  i.e if \( E \vdash t = t' \) and \( E \vdash t : s \) then also \( E \vdash t' : s \)

- < means “sort” inclusion,
  i.e if \( s < s' \) and \( E \vdash t : s \) then also \( E \vdash t : s' \)
Definition

The following is a generalised rewrite sentence (or GRL sentence):

\[(\forall X) \; t \rightarrow t' \iff \bigwedge_{i \in I} u_i = v_i \land \bigwedge_{j \in J} w_j : s_j \land \bigwedge_{l \in L} t_l \rightarrow t'_l\] (Rewrite)

Definition

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 \( \Omega \).
GRL Sentences, Generalised Rewrite Theories

**Definition**

The following is a **generalised rewrite sentence** (or **GRL sentence**):

\[
\frac{u_i = v_i}{s_j : t_i \rightarrow t'_i} \quad (\text{Rewrite})
\]

**Definition**

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 \( \Omega \).
GRL Semantics in a Nutshell

- if $(\forall X) \ t \rightarrow t' \iff \bigwedge_{i \in I} u_i = v_i \land \bigwedge_{j \in J} w_j : s_j \land \bigwedge_{l \in L} t_l \rightarrow t'_l \in R$ and
  \[ E \vdash u_i = v_i, \ E \vdash w_j : s_j, \ R \vdash t_l \rightarrow t'_l \text{ then } R \vdash t \rightarrow t' \]
  plus “nested replacement”!

- $\rightarrow$ is reflexive, transitive and congruent,

- reasoning about $\rightarrow$ is done modulo the MEL subtheory,
  i.e. if $E \vdash t = u$ and $E \vdash t' = u'$ then $R \vdash u \rightarrow u'$ implies $R \vdash t \rightarrow t'$. 
Notation

- $s_0 < s_1 < \ldots < s_n$
- operator declaration:
  \[ f : k_1 \cdots k_n \rightarrow k \text{ or } c : k \quad \rightsquigarrow \quad f \in \Sigma_{k_1 \cdots k_n, k} \text{ or } c \in \Sigma_{\epsilon, k} \]
- operator declaration on sorts:
  \[ f : s_1 \cdots s_n \rightarrow s \text{ or } c : s \quad \rightsquigarrow \quad \text{as above plus membership sentence} \]
- mixfix operators:
  e.g. \((\mid \cdot \rightarrow \cdot) : \text{Var Term} \rightarrow \text{LambdaAbstraction}\)
- implicit universal quantification of variables
- kind/sort of variables given when sort is introduced:
  e.g. \(\text{Exp}\{e_i\} \in S : e_i \text{ range over sort } \text{Exp}; [e]_i \text{ range over kind } \text{[Exp]}\)
- variables ranging over sort \(\rightsquigarrow\) additional membership condition where variable is used
- “otherwise” condition
Outline

1 Introduction
   - Motivation
   - Preliminaries

2 Rewriting Logic Semantics of Haskell
   - Pure Haskell
   - Concurrent Haskell
   - Properties of the Semantic Theory

3 Conclusion
   - Summary
   - Future Work
The semantics includes

- laziness,
- pattern matching,
- mutual recursion (function binding),
- imprecise exceptions (synchronous and asynchronous),
- I/O and
- concurrency.
Coverage of the Semantics

The semantics includes

- laziness,
- pattern matching,
- mutual recursion (function binding),
- imprecise exceptions (synchronous and asynchronous),
- I/O and
- concurrency.

The semantics does not include

- recursive pattern bindings,
- full static semantics, i.e. context sensitive syntax and
- particularly the type system.
How can the semantics of a (programming) language be described in RL/MEL?
Semantics in Rewriting Logic

How can the semantics of a (programming) language be described in RL/MEL?

Approach taken for Pure Haskell

- define signature $\Omega$ s.t. a program of the considered language is a term of MEL (hence, the need for mixfix operators).
- define operators that transform programs (and fragments of them) into its denotation by giving equational sentences.
How can the semantics of a (programming) language be described in RL/MEL?

Approach taken for Pure Haskell
- define signature $\Omega$ s.t. a program of the considered language is a term of MEL (hence, the need for mixfix operators).
- define operators that transform programs (and fragments of them) into its denotation by giving equational sentences.

Approach taken for Concurrent Haskell
- programs are also terms
- include function symbols that construct program states
- give rewrite sentences that describe the execution of a program by rewriting a program state to some successor state.
Considered Syntax Fragment: Basic Syntax

\[
\begin{align*}
BinOp & ::= + \mid - \mid * \mid / \mid <= \mid >= \mid < \mid > \\
& \mid /= \mid == \mid !! \mid && \mid || \mid ** \mid ++ \mid $ \mid $!
\end{align*}
\]

\[
\begin{align*}
BoolConst & ::= \text{True} \mid \text{False}
\end{align*}
\]

\[
\begin{align*}
PDCtor & ::= \text{BoolConst} \mid \text{Float} \mid \text{Int} \mid \text{Char} \mid \text{String} \mid () \mid []
\end{align*}
\]

\[
\begin{align*}
CustCtor & ::= < \text{data constructor} >
\end{align*}
\]

\[
\begin{align*}
Ctor & ::= CustCtor \mid PDCtor
\end{align*}
\]

\[
\begin{align*}
Primitive & ::= \text{seq} \mid \text{not} \mid \text{raise} \mid (BinOp)
\end{align*}
\]

\[
\begin{align*}
AtExp & ::= Ctor \mid Var \mid Primitive
\end{align*}
\]

\[
\begin{align*}
AtPat & ::= Var \mid _ \mid Ctor
\end{align*}
\]

\[
\begin{align*}
Case & ::= Pat \rightarrow Exp
\end{align*}
\]

\[
\begin{align*}
Cases & ::= Case \mid Cases \mid Case
\end{align*}
\]
Considered Syntax Fragment: Expressions

\[
Exp ::= \text{AtExp} \\
| \text{Exp Exp} \\
| \text{\ NePatList \rightarrow Exp} \\
| \text{case Exp of } \{ \text{Cases} \} \\
| \text{if Exp then Exp else Exp} \\
| \text{Exp : Exp} \\
| \text{Exp BinOp Exp} \\
| \left[ \text{ExpList} \right] \\
| \left( \text{ExpList} \right)
\]

\[
\text{ExpList ::= Exp } | \text{ExpList } , \text{Exp}
\]
Considered Syntax Fragment: Patterns, Programs

\[
Pat \quad ::= \quad \text{AtPat} \\
| \quad \text{Pat Pat} \\
| \quad \text{Pat : Pat} \\
| \quad [\text{PatList}] \\
| \quad (\text{PatList}) \\
\]

\[
\text{PatList} \quad ::= \quad \text{Pat} | \text{PatList} , \text{Pat} \\
\]

\[
\text{FuncBindLhs} \quad ::= \quad \text{Var} | \text{FuncBindLhs Pat} \\
\]

\[
\text{FuncBind} \quad ::= \quad \text{FuncBindLhs} = \text{Exp} \\
\]

\[
\text{Program} \quad ::= \quad \text{FuncBind} | \text{Program} ; \text{FuncBind} \\
\]
Formulate Haskell Syntax as a MEL Theory: Examples

BNF definition

\[
Pat \ ::= \ AtPat \\
| Pat \ Pat \\
| Pat : Pat \\
| [ PatList ] \\
| ( PatList )
\]

Primitive \ ::= \ seq | not \\
| raise | ...

Patrick Bahr
An Executable Rewriting Logic Semantics for Concurrent Haskell
Formulate Haskell Syntax as a MEL Theory: Examples

\[
Pat ::= \begin{align*}
   \text{AtPat} & \mid Pat \ Pat \\
   \text{Pat} : \text{Pat} & \mid [ \text{PatList} ] \\
   ( \text{PatList} ) &
\end{align*}
\]

\[
\text{Primitive ::= seq} \mid \text{not} \mid \text{raise} \mid \ldots
\]

\[
\text{AtPat} < \text{Pat}
\]

\[
\begin{align*}
   \cdots : \text{Pat} \text{ Pat} & \rightarrow \text{Pat} \\
   \cdots : \cdots : \text{Pat} \text{ Pat} & \rightarrow \text{Pat} \\
   [ \cdots ] : \text{PatList} & \rightarrow \text{Pat} \\
   ( \cdots ) : \text{PatList} & \rightarrow \text{Pat}
\end{align*}
\]

\[
\begin{align*}
   \text{seq} : \text{Primitive} \\
   \text{not} : \text{Primitive} \\
   \text{raise} : \text{Primitive}
\end{align*}
\]
Haskell Semantics

What is the semantics of a Haskell expression?
What is the semantics of a Haskell expression? — Its normal form.
Haskell Semantics

What is the semantics of a Haskell expression? — Its **normal form**. Since Haskell is lazy (two) different other normal forms are considered:

**Weak head normal form (WHNF) — for pattern matching semantics**

- Constructor applied to several expressions
  - e.g.: (1, 2+3, digitToInt ’3’) or Node (3*5) (1+2)
- function expression (lambda expression, built-in function, function representation, …) applied to too few arguments
  - e.g.: (+) 2; (\x -> x + 1).

**Constructor head normal form (CHNF) — for equality semantics**

- same as weak head normal form but constructors must be applied to CHNFs e.g. Node (3*5) (1+2) is not in CHNF; Node 15 3 is the “equivalent” CHNF

The overall semantics of a Haskell expression is then its **CHNF**.
Hence, expressions must be transformed to their CHNF/WHNF.

**Problem — order of evaluation**

- the order of evaluation of Haskell expressions will not be defined (thanks to referential transparency this is possible):
  e.g. the “result” of \((1+2) \times (3+4)\) does not depend on whether \(1+2\) or \(3+4\) is evaluated first.

- the order of evaluation is significant if exceptions are raised!
  e.g. the expression \((2 \div 0) + (\text{raise SomeException})\) either raises \text{ArithException DivideByZero} or \text{SomeException} depending on the order of evaluation.
Hence, expressions must be transformed to their CHNF/WHNF.

**Problem — order of evaluation**

- the order of evaluation of Haskell expressions will not be defined (thanks to referential transparency this is possible):
  e.g. the “result” of \((1+2) \times (3+4)\) does not depend on whether \(1+2\) or \(3+4\) is evaluated first.

- the order of evaluation is significant if exceptions are raised!
  e.g. the expression \((2 / 0) + (\text{raise SomeException})\) either raises ArithException DivideByZero or SomeException depending on the order of evaluation.

**Solution**

- the semantics of a Haskell expression is either its CHNF or a set of exceptions.
- the set of exceptions contains all exceptions that are raised with some order of evaluation.
Transformation to Normal Form

- constants to identify normal form type:
  \( \text{whnf} : \text{NfType}, \text{chnf} : \text{NfType} \)

- predicate to characterise normal form:
  \( \cdot \downarrow : \text{Exp} \text{ NfType} \rightarrow \text{Bool} \)

- expressions can have multiple (imprecise) exceptional behaviour

- result of transformation is either the normal form or a set of exceptions

- sort \( \text{ExpUEExc} \) as “union” of \( \text{Exp} \) and \( \text{Exceptions} \)

- transformation function \( \mathcal{F}[] : [\text{Exp}] [\text{NfType}] \rightarrow [\text{ExpUEExc}] \)

Note: As both normal forms share some properties and the resp. operators are parametric w.r.t. the normal form type, some sentences can be shared by both normal forms.
Programs induce substitutions

- each Haskell program induces a substitution
- every function defines by which expression the function name can be replaced
Semantics of Haskell Programs

Programs induce substitutions

- each Haskell program induces a substitution
- every function defines by which expression the function name can be replaced

Example

Haskell function

```haskell
foo 1 y = y;
foo 2 y = 4 * y
```
Semantics of Haskell Programs

Programs induce substitutions

- each Haskell program induces a substitution
- every function defines by which expression the function name can be replaced

Example

<table>
<thead>
<tr>
<th>Haskell function</th>
<th>MEL representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo 1 y = y ;</td>
<td>foo \mapsto { 1 \to y ; }</td>
</tr>
<tr>
<td>foo 2 y = 4 * y</td>
<td>foo \mapsto { 2 \to 4 \cdot y }</td>
</tr>
</tbody>
</table>
Semantics of Haskell Programs

Programs induce substitutions

- each Haskell program induces a substitution
- every function defines by which expression the function name can be replaced

Example

Haskell function

\[
\begin{align*}
\text{foo 1 } \ y &= y \\
\text{foo 2 } \ y &= 4 \times y
\end{align*}
\]

MEL representation

\[
\begin{array}{l}
\text{foo} \mapsto \{ \\
1 \ , \ y \mapsto y \\
2 \ , \ y \mapsto 4 \times y
\}
\end{array}
\]

Semantics of a program

- The semantics of a Haskell program is the “least fixed point” of the substitution induced by the program.
- Taking the least fixed point enables mutual recursion!
Program Semantics

Symbols

\[ \text{env}(\cdot) : \text{Program} \rightarrow \text{SimpSubst} \]
\[ \mathcal{H}[\cdot] : \text{Program} \rightarrow \text{Subst} \]
\[ \mathcal{H}[\cdot \text{ in } \cdot] : [\text{Exp}] [\text{Program}] \rightarrow [\text{ExpUExc}] \]

Sentences

\[
\begin{align*}
\mathcal{H}[p_1] &= \text{fix}(\text{env}(p_1)) \\
\ s_1 &= \mathcal{H}[p_1] \\
\mathcal{H}[e_1 \text{ in } p_1] &= \mathcal{F}[e_1[s_1]]_{\text{chnf}}
\end{align*}
\]

program

expression
What is Concurrent Haskell

New concepts
- Thread
- MVar: shared memory, synchronisation

New primitives in the IO monad
- `forkIO :: IO a -> IO ThreadId` — spawns a thread
- `newEmptyMVar :: IO (MVar a)` — creates new empty MVar
- `putMVar :: MVar a -> a -> IO ()` — stores value into an empty MVar
- `takeMVar :: MVar a -> IO a` — reads MVar’s content
- `throw :: Exception -> IO ()` — raises an exception
What is Concurrent Haskell

New concepts
- Thread
- MVar: shared memory, synchronisation

New primitives in the IO monad (cont.)
- `catch :: IO a -> (Exception -> IO a) -> IO a` — handles an exception
- `throwTo :: ThreadId -> Exception -> IO ()` — forces a thread to raise an (asynchronous) exception
- `block/unblock :: IO a -> IO a` — disallows / allows asynchronous exceptions
- `sleep, myThreadId`
Peyton Jones’ Semantics — Program States

<table>
<thead>
<tr>
<th>Program States</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P, Q, R$ ::=</td>
<td>thread of computation named $t$</td>
</tr>
<tr>
<td></td>
<td>finished thread named $t$</td>
</tr>
<tr>
<td></td>
<td>empty MVar named $m$</td>
</tr>
<tr>
<td></td>
<td>full MVar named $m$, holding $M$</td>
</tr>
<tr>
<td></td>
<td>pending asynchronous exception $e$ for thread $t$</td>
</tr>
<tr>
<td></td>
<td>restriction</td>
</tr>
<tr>
<td></td>
<td>parallel composition</td>
</tr>
</tbody>
</table>
Structural Congruence

Defines which program states are considered equal.

Commutativity and associativity

\[ P \mid Q \equiv Q \mid P \quad \text{(Comm)} \]
\[ P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad \text{(Assoc)} \]

\[ \Rightarrow \text{Straightforward translation into equational sentences.} \]

Restrictions

\[ \nu x. \nu y. P \equiv \nu y. \nu x. P \quad \text{(Swap)} \]
\[ (\nu x. P) \mid Q \equiv \nu x. (P \mid Q) \quad x \not\in \text{fn}(Q) \quad \text{(Extrude)} \]
\[ \nu x. P \equiv \nu y. P[y/x] \quad y \not\in \text{fn}(P) \quad \text{(Alpha)} \]

\[ \Rightarrow \text{Cannot be translated into an executable MEL theory.} \]
Extensibility: Record structure technique!

Instead of using only a single function symbol as a constructor (where each argument position defines some component)

Use a record structure, whose components are indexed by a name.

Example

Function symbol approach:
- **definition**: \((·, ·, ·)\) : ThreadId ThreadCont Flag → Thread
- **example term**: \((⟨1⟩_\text{Th}, ε, \text{no})\).

Record structure approach:
- **definition**: a bit more complicated
- **example term**: \((\text{tid} : ⟨1⟩_\text{Th} \ → \ \text{val} : ε \ → \ \text{stuck} : \text{no})\)
The record structure of a program state contains

- input
- output
- thread id generator
- MVar id generator
- process pool (contains threads, MVars, asynchronous exceptions)
Thread, MVar, AException $\prec$ Proc $\prec$ ProcPool

null : ProcPool
$\cdot \mid \cdot : \text{ProcPool} \times \text{ProcPool} \to \text{ProcPool}$

Plus some sentences making $\mid$ associative and commutative and make null its identity.

$\frac{pp_1 \mid (pp_2 \mid pp_3) = (pp_1 \mid pp_2) \mid pp_3}{\text{assoc}}$

$\frac{pp_1 \mid pp_2 = pp_2 \mid pp_1}{\text{comm}}$

$\frac{pp_1 \mid \text{null} = pp_1}{\text{id}}$
Thread as a Record Structure

Components
- tid: the thread’s id
- val: the thread’s value
- stuck: flag indicating whether the thread is stuck

Example

finished thread:
\[ 0_t \xrightarrow{\_} \{ \text{tid: } \langle 1 \rangle_{Th} \text{, } \text{val: } \varepsilon \text{, stuck: no} \} \]

stuck thread:
\[ \{ \text{MVar } m 0 \}_t \xrightarrow{\_} \{ \text{tid: } \langle 0 \rangle_{Th} \text{, } \text{val: } \text{putMVar } \langle 1 \rangle_{MV} 0 \text{, stuck: yes} \} \]
MVar as a Record Structure

Components
- mid: the MVar's id
- cont: the MVar's content

Example
- empty MVar:
  \[ \langle \rangle_m \rightsquigarrow \langle \text{mid} : \langle 1 \rangle_{\text{MV}} \text{, cont} : \varepsilon \rangle \]
- full MVar holding 1:
  \[ \langle 1 \rangle_m \rightsquigarrow \langle \text{mid} : \langle 0 \rangle_{\text{MV}} \text{, cont} : 1 \rangle \]
Asynchronous Exception as a Function Symbol

\[
\begin{align*}
\langle \cdot \; \# \; \cdot \rangle & : \text{ThreadId} \; \text{ExcExp} \rightarrow A\text{Exception} \\
tgt(\cdot) & : A\text{Exception} \rightarrow \text{ThreadId} \\
exc(\cdot) & : A\text{Exception} \rightarrow \text{ExcExp}
\end{align*}
\]

\[
\begin{align*}
tgt(\langle ti_1 \# ee_1 \rangle) &= ti_1 \\
exc(\langle ti_1 \# ee_1 \rangle) &= ee_1
\end{align*}
\]
Initial Program States

Initial state of a program w.r.t an expression

\[ C[\cdot \text{ in } \cdot](\cdot) : \text{Exp Program String } \rightarrow \text{State} \quad (1) \]

\[ C[e_1 \text{ in } pr_1](str_1) = \{ \text{ in : } \langle str_1 \rangle_1, \text{ out : } \langle \rangle_0, \text{ tgen : newThreadIdGen, mgen : newMVarIdGen, pool : (H[unblock e_1 in pr_1])}_{\text{mainThreadId}} \} \]

Initial state of a program

\[ C[\cdot](\cdot) : \text{Program String } \rightarrow \text{State} \]

\[ C[pr_1](str_1) = C[\text{main in } pr_1](str_1) \]
Rules of Peyton Jones’ Semantics

Structural rules

\[
P \xrightarrow{\alpha} Q \\
P \parallel R \xrightarrow{\alpha} Q \parallel R
\]

(Par)

\[
P \xrightarrow{\alpha} Q \\
\nu x. P \xrightarrow{\alpha} \nu x. Q
\]

(Nu)

\[
P \equiv P' \\
P' \xrightarrow{\alpha} Q' \\
Q \equiv Q'
\]

(Equiv)

- These rules are covered by the semantics of rewriting logic, i.e. deduction rules (Cong) and (Eq).
- For each primitive there is at least one axiom.
Axioms of the Concurrency Semantics — An Example

Original axiom for \( \text{throwTo} \)

\[
(\mathbb{E} [\text{throwTo } t \ e])_u \rightarrow (\mathbb{E} [\text{return }()] )_u \mid (\not\exists e)
\]
Axioms of the Concurrency Semantics — An Example

Original axiom for `throwTo`

\[
(E[\text{throwTo } t e])_u \rightarrow (E[\text{return } ()])_u \mid \langle t \not \leq e \rangle
\]

Rewrite sentence formulation

\[
\begin{align*}
\mathcal{E}(e_1) &= \text{throwTo } t_{i_1} e_{ce_1} \\
\mathcal{E}(e_1 \leftarrow \text{return } ()) &= e_2 \\
\{\text{val} : e_1 \quad \text{prt}_1\} &\rightarrow \{\text{val} : e_2 \quad \text{prt}_1\} \mid \langle t_{i_1} \not \leq e_{ce_1} \rangle
\end{align*}
\]
Axioms of the Concurrency Semantics — Another Example

Original axiom for `forkIO`

\[(\mathcal{E}[\text{forkIO } M])_t \rightarrow \nu u.((\mathcal{E}[\text{return } u])_t | (\text{unblock } M)_u)\]

where \( u \not\in \text{fn}(\mathcal{E}, M) \)
Axioms of the Concurrency Semantics — Another Example

Original axiom for \texttt{forkIO}

\[
(E[forkIO\ M])_t \rightarrow \nu u.((E[return\ u])_t | (unblock\ M)_u)
\]

where \( u \notin fn(E, M) \)

Rewrite sentence formulation

\[
E(e_1) = forkIO\ e_3
\]
\[
currentId(tg_1) = ti_1
\]
\[
E(e_1 \leftarrow return\ ti_1) = e_2
\]

\[
\begin{array}{c}
\text{pool : } [\text{val : } e_1\ prt_1] | pp_1 \quad \text{tgen : } tg_1 \\
\text{pool : } [\text{val : } e_2\ prt_1] \ | [\text{unblock } e_3]_t_i_1 | pp_1 \quad \text{tgen : } \text{nextGen}(tg_1)
\end{array}
\]
Executability

- preregularity: terms have a smallest sort if any
- equations are applied from left to right
  \[ \Rightarrow \text{ order-sorted term rewriting system } \rightarrow E \]
- \( \rightarrow E \) must be:
  - ground terminating
  - ground confluent
  - ground sort decreasing
- but: associativity and commutativity sentences can be disregarded for this properties
- the TRS \( \rightarrow R \) induced by the rewrite sentences must be coherent w.r.t. \( \rightarrow E \)
That is, the following diagram has to commute:
Properties Providing Executability

<table>
<thead>
<tr>
<th>Properties of the semantic theory of Concurrent Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preregularity</td>
</tr>
<tr>
<td>Confluence</td>
</tr>
<tr>
<td>Sort-decreasingness</td>
</tr>
<tr>
<td>Coherence</td>
</tr>
<tr>
<td>Termination</td>
</tr>
</tbody>
</table>
## Properties Providing Executability

### Properties of the semantic theory of Concurrent Haskell

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preregularity</td>
</tr>
<tr>
<td>Confluence</td>
</tr>
<tr>
<td>Sort-decreasingness</td>
</tr>
<tr>
<td>Coherence</td>
</tr>
<tr>
<td>Termination</td>
</tr>
</tbody>
</table>

- Preregularity: ✔️
- Confluence
- Sort-decreasingness
- Coherence
- Termination

Problem: \( F \) does not have a normal form if the Haskell expression represented by \( t \) diverges. Hence: The semantic theory is only executable for converging expressions. Nevertheless: Divergence is exactly described by the theory: The Haskell expression represented by \( t \) converges iff \( E \vdash F \).
### Properties of the semantic theory of Concurrent Haskell

<table>
<thead>
<tr>
<th>Property</th>
<th>Executability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preregularity</td>
<td>✓</td>
</tr>
<tr>
<td>Confluence</td>
<td>✓</td>
</tr>
<tr>
<td>Sort-decreasingness</td>
<td></td>
</tr>
<tr>
<td>Coherence</td>
<td></td>
</tr>
<tr>
<td>Termination</td>
<td></td>
</tr>
</tbody>
</table>
Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

<table>
<thead>
<tr>
<th>Property</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preregularity</td>
<td>✔</td>
</tr>
<tr>
<td>Confluence</td>
<td>✔</td>
</tr>
<tr>
<td>Sort-decreasingness</td>
<td>✔</td>
</tr>
<tr>
<td>Coherence</td>
<td></td>
</tr>
<tr>
<td>Termination</td>
<td></td>
</tr>
</tbody>
</table>
### Properties of the semantic theory of Concurrent Haskell

<table>
<thead>
<tr>
<th>Property</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preregularity</td>
<td>✔</td>
</tr>
<tr>
<td>Confluence</td>
<td>✔</td>
</tr>
<tr>
<td>Sort-decreasingness</td>
<td>✔</td>
</tr>
<tr>
<td>Coherence</td>
<td>✔</td>
</tr>
<tr>
<td>Termination</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Problems:**

- **FJ**
- **Kchnf**

Does not have a normal form if the Haskell expression represented by \( t \) diverges. Hence: The semantic theory is only executable for converging expressions. Nevertheless: Divergence is exactly described by the theory: The Haskell expression represented by \( t \) converges iff \( E \subseteq F^{JF} \): ExpUExc.
Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

<table>
<thead>
<tr>
<th>Property</th>
<th>✔</th>
<th>✗</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preregularity</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Confluence</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Sort-decreasingness</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Coherence</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Termination</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>

Problem: \( t \text{chnf} \) does not have a normal form if the Haskell expression represented by \( t \) diverges. Hence: The semantic theory is only executable for converging expressions. Nevertheless: Divergence is exactly described by the theory: The Haskell expression represented by \( t \) converges iff \( \text{ExpUExc} \).
Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

<table>
<thead>
<tr>
<th>Property</th>
<th>✔️</th>
<th>✗</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preregularity</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Confluence</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Sort-decreasingness</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Coherence</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>Termination</td>
<td></td>
<td>✗</td>
</tr>
</tbody>
</table>

Termination issue

- Problem: \( F[t]_{chnf} \) does not have a normal form if the Haskell expression represented by \( t \) diverges.
- Hence: The semantic theory is only executable for converging expressions.
- Nevertheless: Divergence is exactly described by the theory:

The Haskell expression represented by \( t \) converges iff

\[ E_C \vdash F[t]_{chnf} : \text{ExpUExc}. \]
Relation to Moran’s Coinductive Semantics of Imprecise Exceptions

**Theorem**

Let $E_H$ be the semantic MEL theory for pure Haskell. Then the following equivalences hold true:

(i) $M \Downarrow V$ iff $E_H \vdash \mathcal{F}[\overline{M}]_{\text{whnf}} = \overline{V}$

(ii) $M \nrightarrow S$ iff $E_H \vdash \mathcal{F}[\overline{M}]_{\text{whnf}} = \overline{S}$
Relation to Moran’s Coinductive Semantics of Imprecise Exceptions

Theorem

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

(i) $M \downarrow V$ iff $\mathcal{E}_H \vdash \mathcal{F}[\overline{M}]_{\text{whnf}} = \overline{V}$

(ii) $M \not\uparrow S$ iff $\mathcal{E}_H \vdash \mathcal{F}[\overline{M}]_{\text{whnf}} = \overline{S}$

Corollary

$M \uparrow$ iff $\mathcal{E}_H \not\vdash \mathcal{F}[\overline{M}]_{\text{whnf}} : \text{ExpUExc}$
Relation to Peyton Jones’ SOS of Concurrent Haskell

Theorem

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

$$ P \rightarrow^* Q \iff \exists s \in \overline{P}, s' \in \overline{Q}. \quad \mathcal{R}_C \vdash s \xrightarrow{} s' $$
Non-executable extension of $\mathcal{R}_C$ to $\mathcal{R}'_C$

\[ \cdot \upharpoonright: \text{Exp} \rightarrow \text{Bool} \]

\[
\begin{align*}
\mathcal{F}[e_1]_{\text{whnf}} : \text{ExpUExc} & \quad \text{div1} \\
\quad e_1 \uparrow = \text{false} & \\
\text{div2} & \quad \text{otherwise} \\
\quad e_1 \uparrow = \text{true} & \\
\end{align*}
\]

\[
\begin{align*}
\mathcal{E}(e_1) = e_3 & \quad e_3 \uparrow = \text{true} \\
\text{Raise div} & \quad \mathcal{E}(e_1 \leftarrow \text{throw } ece_1) = e_2 \\
\text{val : e}_1 \rightarrow & \quad \text{val : e}_2
\end{align*}
\]
Relation to Peyton Jones’ SOS of Concurrent Haskell

Non-executable extension of \( R_C \) to \( R'_C \)

\[
\begin{align*}
\cdot \uparrow & : \text{Exp} \rightarrow \text{Bool} \\
\mathcal{F}[e_1]_{\text{whnf}} & : \text{Exp}\cup\text{Exc} \\
\quad e_1 \uparrow & = \text{false} \quad \text{div1} \\
\quad \text{otherwise} & \quad \text{div2} \\
E(e_1) & = e_3 \quad e_3 \uparrow = \text{true} \quad E(e_1 \leftarrow \text{throw } ece_1) = e_2 \\
\text{val} : e_1 & \rightarrow \text{val} : e_2 \\
\text{Raise div}
\end{align*}
\]

Theorem

Let \( P \) and \( Q \) be program states Then the following holds:

\[
P \rightarrow^* Q \quad \text{iff} \quad \exists s \in \overline{P}, s' \in \overline{Q}. \quad R'_C \vdash s \rightarrow s'
\]
Outline

1 Introduction
   - Motivation
   - Preliminaries

2 Rewriting Logic Semantics of Haskell
   - Pure Haskell
   - Concurrent Haskell
   - Properties of the Semantic Theory

3 Conclusion
   - Summary
   - Future Work
Dynamic semantics for Concurrent Haskell including most of the language features:
- laziness
- pattern matching
- mutual recursion
- imprecise (a)synchronous exceptions
- I/O
- concurrency

Proofs for equivalence to different existing semantics.

The given theory is executable in the Maude system, i.e.:
- interpreter
- semi-automated inductive proofs
- model checking
  
Modularity ensures extendibility of the semantics to include further features as well as flexibility to change the semantics.
Future Work

- Minor features of the dynamic semantics are missing (in particular recursive pattern bindings).
- How can rewrite sentences be used to describe imprecise exceptions more naturally?
- A formulation of the static semantics in rewriting logic is still missing!
- Particularly the type calculus of Haskell is interesting.
- Amalgamation of the dynamic and static semantics in one framework is desirable, as both are interwoven (ad-hoc polymorphism)!
- Use of free theorems derived from the type information in semi-automated proofs.
José Meseguer and Grigore Rou.  
The rewriting logic semantics project.  

Andrew Moran, Søren B. Lassen, and Simon Peyton Jones.  
Imprecise exceptions, co-inductively.  
In *HOOTS ’99, Higher Order Operational Techniques in Semantics,* 

Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy.  
Asynchronous exceptions in haskell.  