An Executable Rewriting Logic Semantics for Concurrent Haskell

Patrick Bahr
s0404888@inf.tu-dresden.de

Technische Universität Dresden

January 11, 2008

Outline

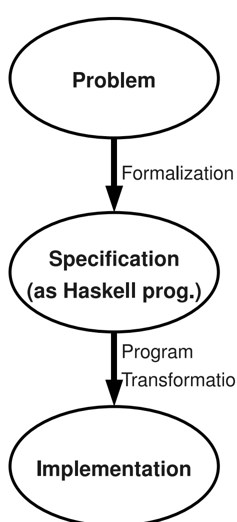
1. Introduction
   - Motivation
   - Preliminaries
2. Rewriting Logic Semantics of Haskell
   - Pure Haskell
   - Concurrent Haskell
   - Properties of the Semantic Theory
3. Conclusion
   - Summary
   - Future Work

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?

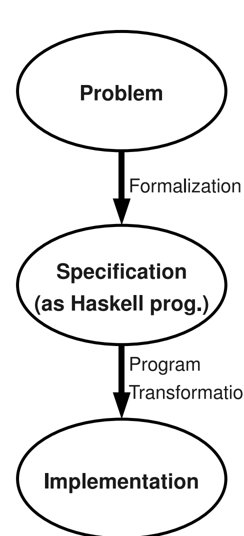
- declarative reading
- use as specification language
- program transformation to obtain efficient implementation



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

$\rightsquigarrow$ **Membership Equational Logic (MEL)**:

- many-kinded term language
- atomic formulae:
  - $t = t'$
  - $t : \mathbf{s}$ "t has sort $\mathbf{s}$"

Membership Equational Signature

Definition

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

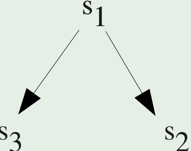
- S is a finite set of sorts,
- $<$ is a strict order on S,
- $\Sigma = \{\Sigma_{w,k}\}_{(w,k) \in K^* \times K}$ is a $K^* \times K$-indexed family of function symbols and
- $K := {}^{S}/_{\equiv_<}$ is the set of kinds induced by the equivalence closure of $<$.

$[\mathbf{s}]$ will denote the equivalence class of $\mathbf{s}$ w.r.t $\equiv_<$, i.e. its kind.

Sorts and Kinds — Example

Example

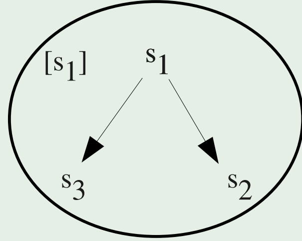
- $S = \{s_1, s_2, s_3, s_4, s_5\}$ — the set of sorts and
- $<$ is s.t.
  - $s_2 < s_1$,
  - $s_3 < s_1$,
  - $s_5 < s_4$.

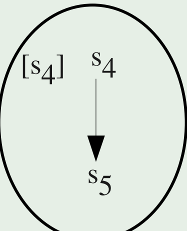


Sorts and Kinds — Example

Example

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





$\rightsquigarrow K = \{[s_1], [s_4]\}$, where $[s_1] = \{s_1, s_2, s_3\}$ and $[s_4] = \{s_4, s_5\}$.

MEL Sentences, MEL Theories

Definition

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

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.

MEL Sentences, MEL Theories

Definition

The following are membership equational sentences (or MEL sentences):

$$\frac{u_i = v_i \qquad w_j : s_j}{t = t'} \qquad \text{(Equation)}$$

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

Definition

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.

MEL Sentences, MEL Theories

Definition

The following are membership equational sentences (or MEL sentences):

$$\frac{u_i = v_i \qquad w_j : s_j}{t = t'} \qquad \text{(Equation)}$$

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

Definition

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.

MEL Semantics in a Nutshell

- if $t = t' \Leftarrow \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j \in E$ and $\mathcal{E} \vdash u_i = v_i$ and $\mathcal{E} \vdash w_j : s_j$ hold then also $\mathcal{E} \vdash t = t'$.
- if $t : s \Leftarrow \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j \in E$ and $\mathcal{E} \vdash u_i = v_i$ and $\mathcal{E} \vdash w_j : s_j$ hold then also $\mathcal{E} \vdash t : s$.
- $=$ is a congruence,
- sort membership is preserved by $=$,
  i.e if $\mathcal{E} \vdash t = t'$ and $\mathcal{E} \vdash t : s$ then also $\mathcal{E} \vdash t' : s$
- $<$ means “sort” inclusion,
  i.e if $s < s'$ and $\mathcal{E} \vdash t : s$ then also $\mathcal{E} \vdash t : s'$

GRL Sentences, Generalised Rewrite Theories

Definition

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

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 .

GRL Sentences, Generalised Rewrite Theories

Definition

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

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

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 Ω .

GRL Semantics in a Nutshell

- if $(\forall X)\ t \longrightarrow t' \Leftarrow \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \longrightarrow t'_l \in R$ and $\mathcal{E} \vdash u_i = v_i$, $\mathcal{E} \vdash w_j : s_j$, $\mathcal{R} \vdash t_l \longrightarrow t'_l$ then $\mathcal{R} \vdash t \longrightarrow t'$
  plus "nested replacement"!
- $\longrightarrow$ is reflexive, transitive and congruent,
- reasoning about $\longrightarrow$ is done modulo the MEL subtheory, i.e. if $\mathcal{E} \vdash t = u$ and $\mathcal{E} \vdash t' = u'$ then $\mathcal{R} \vdash u \longrightarrow u'$ implies $\mathcal{R} \vdash t \longrightarrow t'$.

Notation

- $s_0 < s_1 < \ldots < s_n$
- operator declaration:
  $f : k_1 \cdots k_n \to k$ or $c : k \quad \rightsquigarrow \quad f \in \Sigma_{k_1 \cdots k_n, k}$ or $c \in \Sigma_{\varepsilon, k}$
- operator declaration on sorts:
  $f : s_1 \cdots s_n \to s$ or $c : s \quad \rightsquigarrow$ as above plus membership sentence
- mixfix operators:
  e.g. $(\backslash \ \cdot$ `->` $\cdot) :$ **Var Term** $\to$ **LambdaAbstraction**
- implicit universal quantification of variables
- kind/sort of variables given when sort is introduced:
  e.g. **Exp**$\{e_i\} \in S$: e_i range over sort **Exp**; $[e]_i$ range over kind [**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

Coverage of the Semantics

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.

Semantics in Rewriting Logic

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 Ω 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

Semantics in Rewriting Logic

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

Approach taken for Pure Haskell

- define signature Ω 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

```
    BinOp  ::=  + | - | * | / | <= | >= | < | >
              | /= | == | !! | && | || | ** | ++ | $ | $!
BoolConst  ::=  True | False
   PDCtor  ::=  BoolConst | Float | Int | Char | String | () | []
 CustCtor  ::=  < data constructor >
     Ctor  ::=  CustCtor | PDCtor
Primitive  ::=  seq | not | raise | ( BinOp )
    AtExp  ::=  Ctor | Var | Primitive
    AtPat  ::=  Var | _ | Ctor

     Case  ::=  Pat -> Exp
    Cases  ::=  Case | Cases ; Case
```

Considered Syntax Fragment: Expressions

$$
\begin{aligned}
\mathit{Exp} \quad ::= \quad & \mathit{AtExp} \\
|\ & \mathit{Exp}\ \mathit{Exp} \\
|\ & \backslash\ \mathit{NePatList}\ \texttt{->}\ \mathit{Exp} \\
|\ & \texttt{case}\ \mathit{Exp}\ \texttt{of}\ \ \{\ \mathit{Cases}\ \} \\
|\ & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\
|\ & \mathit{Exp}\ \texttt{:}\ \mathit{Exp} \\
|\ & \mathit{Exp}\ \mathit{BinOp}\ \mathit{Exp} \\
|\ & [\ \mathit{ExpList}\ ] \\
|\ & (\ \mathit{ExpList}\ ) \\
\mathit{ExpList} \quad ::= \quad & \mathit{Exp} \mid \mathit{ExpList}\ ,\ \mathit{Exp}
\end{aligned}
$$

Considered Syntax Fragment: Patterns, Programs

$$
\begin{aligned}
Pat \quad &::= \quad AtPat \\
&\quad\ \mid Pat\ Pat \\
&\quad\ \mid Pat : Pat \\
&\quad\ \mid [\ PatList\ ] \\
&\quad\ \mid (\ PatList\ ) \\
PatList \quad &::= \quad Pat \mid PatList\ ,\ Pat
\end{aligned}
$$

$$
\begin{aligned}
FuncBindLhs \quad &::= \quad Var \mid FuncBindLhs\ Pat \\
FuncBind \quad &::= \quad FuncBindLhs = Exp \\
Program \quad &::= \quad FuncBind \mid Program\ ;\ FuncBind
\end{aligned}
$$

Formulate Haskell Syntax as a MEL Theory: Examples

BNF definition

$$
\begin{aligned}
Pat \quad &::= \quad AtPat \\
&\quad\;\; \mid Pat\ Pat \\
&\quad\;\; \mid Pat : Pat \\
&\quad\;\; \mid [\ PatList\ ] \\
&\quad\;\; \mid (\ PatList\ ) \\
\\
Primitive \quad &::= \quad \texttt{seq} \mid \texttt{not} \\
&\quad\;\; \mid \texttt{raise} \mid \ldots
\end{aligned}
$$

Formulate Haskell Syntax as a MEL Theory: Examples

BNF definition

$$
\begin{aligned}
\mathit{Pat} \quad &::= \quad \mathit{AtPat} \\
&\quad\ \mid \mathit{Pat}\ \mathit{Pat} \\
&\quad\ \mid \mathit{Pat} : \mathit{Pat} \\
&\quad\ \mid [\,\mathit{PatList}\,] \\
&\quad\ \mid (\,\mathit{PatList}\,)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Primitive} \quad &::= \quad \texttt{seq} \mid \texttt{not} \\
&\quad\ \mid \texttt{raise} \mid \ldots
\end{aligned}
$$

MEL definition

$$
\begin{aligned}
&\textbf{AtPat} < \textbf{Pat} \\
\cdot\ \cdot &: \textbf{Pat}\ \ \textbf{Pat} \to \textbf{Pat} \\
\cdot : \cdot &: \textbf{Pat}\ \ \textbf{Pat} \to \textbf{Pat} \\
[\ \cdot\ ] &: \textbf{PatList} \to \textbf{Pat} \\
(\ \cdot\ ) &: \textbf{PatList} \to \textbf{Pat}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{seq} &: \textbf{Primitive} \\
\texttt{not} &: \textbf{Primitive} \\
\texttt{raise} &: \textbf{Primitive} \\
&\vdots
\end{aligned}
$$

Haskell Semantics

What is the semantics of a Haskell expression?

Haskell Semantics

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.

Exceptions

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) * (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) + (raise SomeException)` either raises `ArithException DivideByZero` or `SomeException` depending on the order of evaluation

Exceptions

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) * (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) + (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:
  whnf : **NfType** , chnf : **NfType**
- predicate to characterise normal form:
  $\cdot \Downarrow . \; : \; \mathbf{Exp} \;\; \mathbf{NfType} \; \rightarrow \; \mathbf{Bool}$
- expressions can have multiple (imprecise) exceptional behaviour
- result of transformation is either the normal form or a set of exceptions
- sort **ExpUExc** as "union" of **Exp** and **Exceptions**
- transformation function $\mathcal{F}[\![\cdot]\!]. \; : \; [\mathbf{Exp}] \;\; [\mathbf{NfType}] \; \rightarrow \; [\mathbf{ExpUExc}]$

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.

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

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

```
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

Haskell function $\rightsquigarrow$ MEL representation

```
foo 1 y = y;
foo 2 y = 4 * y
```

$$\texttt{foo} \mapsto \{ \texttt{1 , y :}\!\longrightarrow \texttt{y;}\quad \texttt{2 , y :}\!\longrightarrow \texttt{4 * y} \}$$

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

Haskell function

```
foo 1  y = y;
foo 2  y = 4  *  y
```

$\rightsquigarrow$

MEL representation

$$\texttt{foo} \mapsto \{ \texttt{1 , y} :\rightarrow \texttt{y;} \quad \texttt{2 , y} :\rightarrow \texttt{4 * y} \}$$

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

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

Sentences

$$\frac{}{\mathcal{H}[\![p_1]\!] = \mathrm{fix}(\mathrm{env}(p_1))} \quad \textbf{program}$$

$$\frac{s_1 = \mathcal{H}[\![p_1]\!]}{\mathcal{H}[\![e_1 \text{ in } p_1]\!] = \mathcal{F}\left[\!\left[e_1[s_1]\right]\!\right]_{\mathrm{chnf}}} \quad \textbf{expression}$$

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

Program States

P, Q, R	$::=$	$(\!\lvert M \rvert\!)_t$	thread of computation named t
	$\mid$	$\mathbb{0}_t$	finished thread named t
	$\mid$	$\langle\rangle_m$	empty `MVar` named m
	$\mid$	$\langle M \rangle_m$	full `MVar` named m, holding M
	$\mid$	$(\!\lvert t \lightning e \rvert\!)$	pending asynchronous exception e for thread t
	$\mid$	$\nu x.P$	restriction
	$\mid$	$P \mid Q$	parallel composition

Structural Congruence

Defines which program states are considered equal.

Commutativity and associativity

$$P \mid Q \equiv Q \mid P \quad (\text{Comm}) \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad (\text{Assoc})$$

$\rightsquigarrow$ Straightforward translation into equational sentences.

Restrictions

$$\begin{aligned} \nu x.\nu y.P &\equiv \nu y.\nu x.P & & (\text{Swap}) \\ (\nu x.P) \mid Q &\equiv \nu x.(P \mid Q) & x \notin fn(Q) & \quad (\text{Extrude}) \\ \nu x.P &\equiv \nu y.P[{}^{y}/_{x}] & y \notin fn(P) & \quad (\text{Alpha}) \end{aligned}$$

$\rightsquigarrow$ Cannot be translated into an executable MEL theory.

Skip details on MEL-representation

Program States as MEL Terms — Modularity

- 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: $[\![\cdot , \cdot , \cdot]\!]$: **ThreadId ThreadCont Flag** $\rightarrow$ **Thread**
  example term: $[\![\langle 1\rangle_{\mathsf{Th}}, \varepsilon, \mathsf{no}]\!]$.
- Record structure approach:
  definition: a bit more complicated
  example term: $[\![\mathsf{tid} : \langle 1\rangle_{\mathsf{Th}} \quad \mathsf{val} : \varepsilon \quad \mathsf{stuck} : \mathsf{no}]\!]$

Program States

The record structure of a program state contains

- input
- output
- thread id generator
- MVar id generator
- process pool (contains threads, MVars, asynchronous exceptions)

Process Pool

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

$$\mathrm{null} : \mathbf{ProcPool}$$
$$\cdot \mid \cdot : \mathbf{ProcPool}\ \mathbf{ProcPool} \rightarrow \mathbf{ProcPool}$$

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

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

$$\frac{}{pp_1 \mid pp_2 = pp_2 \mid pp_1}\ \textbf{comm} \qquad \frac{}{pp_1 \mid \mathrm{null} = pp_1}\ \textbf{id}$$

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:

$$\mathbb{0}_t^\circ \quad \leadsto \quad (\!|\mathsf{tid} : \langle 1 \rangle_{\mathsf{Th}} \quad \mathsf{val} : \varepsilon \quad \mathsf{stuck} : \mathsf{no}|\!)$$

stuck thread:

$$(\!|\,\mathtt{MVar}\ m\ 0|\!)_t^\bullet \quad \leadsto \quad (\!|\mathsf{tid} : \langle 0 \rangle_{\mathsf{Th}} \quad \mathsf{val} : \mathtt{putMVar}\ \langle 1 \rangle_{\mathsf{MV}}\ 0 \quad \mathsf{stuck} : \mathsf{yes}|\!)$$

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

$$\langle\!| \cdot \lightning \cdot |\!\rangle : \textbf{ThreadId}\ \ \textbf{ExcExp} \longrightarrow \textbf{AException}$$

$$\mathrm{tgt}(\cdot) : \textbf{AException} \longrightarrow \textbf{ThreadId}$$

$$\mathrm{exc}(\cdot) : \textbf{AException} \longrightarrow \textbf{ExcExp}$$

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

Initial Program States

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

$$\mathcal{C}[\![\, \cdot \text{ in } \cdot \,]\!](\,\cdot\,) : \textbf{Exp}\ \ \textbf{Program}\ \ \textbf{String}\ \rightarrow\ \textbf{State} \quad (1)$$

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

Initial state of a program

$$\mathcal{C}[\![\,\cdot\,]\!](\,\cdot\,) : \textbf{Program}\ \ \textbf{String}\ \rightarrow\ \textbf{State}$$

$$\overline{\mathcal{C}[\![pr_1]\!](str_1) = \mathcal{C}[\![\texttt{main} \text{ in } pr_1]\!](str_1)}$$

Rules of Peyton Jones' Semantics

Structural rules

$$\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \; (\mathrm{Par}) \qquad \frac{P \xrightarrow{\alpha} Q}{\nu x.P \xrightarrow{\alpha} \nu x.Q} \; (\mathrm{Nu})$$

$$\frac{P \equiv P' \qquad P' \xrightarrow{\alpha} Q' \qquad Q \equiv Q'}{P \xrightarrow{\alpha} Q} \; (\mathrm{Equiv})$$

- 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 `throwTo`

$$(\!|\mathbb{E}\,[\,\texttt{throwTo}\ \ t\ e]\,|\!)_u \quad\longrightarrow\quad (\!|\mathbb{E}\,[\,\texttt{return}\ \ ()\,]\,|\!)_u \mid \langle\!| t \lightning e |\!\rangle$$

Axioms of the Concurrency Semantics — An Example

Original axiom for throwTo

$$(\!|\mathbb{E}[\,\texttt{throwTo}\ \ t\ e\,]|\!)_u \quad \rightarrow \quad (\!|\mathbb{E}[\,\texttt{return}\ \ ()\,]|\!)_u \mid \langle\!| t \lightning e |\!\rangle$$

Rewrite sentence formulation

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

Axioms of the Concurrency Semantics — Another Example

Original axiom for forkIO

$$(\!|\mathbb{E}\,[\,\texttt{forkIO}\;\; M]\,|\!)_t \quad\rightarrow\quad \nu u.((\!|\mathbb{E}\,[\,\texttt{return}\;\; u]\,|\!)_t \,|\, (\!|\,\texttt{unblock}\;\; M|\!)_u)$$

$$\text{where} \quad u \notin fn(\mathbb{E}, M)$$

Axioms of the Concurrency Semantics — Another Example

Original axiom for forkIO

$$(\!|\mathbb{E}[\texttt{forkIO}\ M]|\!)_t \quad \rightarrow \quad \nu u.((\!|\mathbb{E}[\texttt{return}\ u]|\!)_t \mid (\!|\texttt{unblock}\ M|\!)_u)$$
$$\text{where} \quad u \notin fn(\mathbb{E}, M)$$

Rewrite sentence formulation

$$\frac{\begin{array}{c}\mathbb{E}(e_1) = \texttt{forkIO}\ e_3 \\ \mathsf{currentId}(tg_1) = ti_1 \qquad \mathbb{E}(e_1 \leftarrow \texttt{return}\ ti_1) = e_2\end{array}}{\mathsf{pool} : (\!|\mathsf{val} : e_1 \quad prt_1|\!) \mid pp_1 \quad \mathsf{tgen} : tg_1 \xrightarrow{\qquad} \mathsf{pool} : (\!|\mathsf{val} : e_2 \quad prt_1|\!) \mid (\!|\texttt{unblock}\ e_3|\!)_{ti_1} \mid pp_1 \quad \mathsf{tgen} : \mathsf{nextGen}(tg_1)}$$

Executability

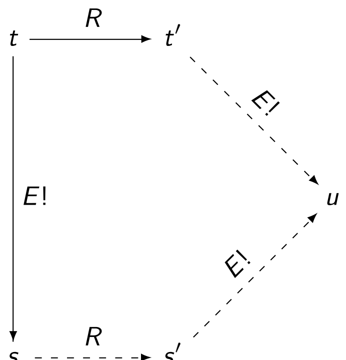
Executability subsumes the following properties

- preregularity: terms have a smallest sort if any
- equations are applied from left to right
  $\rightsquigarrow$ order-sorted term rewriting system $\to_{\mathcal{E}}$
- $\to_{\mathcal{E}}$ must be:
  - ground terminating
  - ground confluent
  - ground sort decreasing
- but: associativity and commutativity sentences can be disregarded for this properties
- the TRS $\to_{\mathcal{R}}$ induced by the rewrite sentences must be coherent w.r.t. $\to_{\mathcal{E}}$

Skip executability

Coherence

That is, the following diagramm has to commute:



Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

Preregularity
Confluence
Sort-decreasingness
Coherence
Termination

Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

Preregularity
Confluence
Sort-decreasingness
Coherence
Termination

Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

Preregularity
Confluence
Sort-decreasingness
Coherence
Termination

Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

Preregularity	✔
Confluence	✔
Sort-decreasingness	✔
Coherence	
Termination	

Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

Preregularity	✔
Confluence	✔
Sort-decreasingness	✔
Coherence	✔
Termination	

Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

Property	
Preregularity	✔
Confluence	✔
Sort-decreasingness	✔
Coherence	✔
Termination	✘

Properties Providing Executability

Properties of the semantic theory of Concurrent Haskell

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

Termination issue

- Problem: $\mathcal{F}[\![t]\!]_{\mathsf{chnf}}$ 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
$\mathcal{E}_C \vdash \mathcal{F}[\![t]\!]_{\mathsf{chnf}} : \textbf{ExpUExc}$.

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

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

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

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

Corollary

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

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:

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

Relation to Peyton Jones' SOS of Concurrent Haskell

Non-executable extension of $\mathcal{R}_C$ to $\mathcal{R}'_C$

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

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

$$\frac{\mathbb{E}(e_1) = e_3 \qquad e_3 \Uparrow = \mathsf{true} \qquad \mathbb{E}(e_1 \leftarrow \texttt{throw}\ ece_1) = e_2}{\mathsf{val} : e_1 \longrightarrow \mathsf{val} : e_2}\ \textbf{Raise div}$$

Relation to Peyton Jones' SOS of Concurrent Haskell

Non-executable extension of $\mathcal{R}_C$ to $\mathcal{R}'_C$

$$\cdot \Uparrow : \textbf{Exp} \;\rightarrow\; \textbf{Bool}$$

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

$$\frac{\mathbb{E}(e_1) = e_3 \qquad e_3 \Uparrow \,=\, \text{true} \qquad \mathbb{E}(e_1 \leftarrow \texttt{throw}\ \ ece_1) = e_2}{\text{val} : e_1 \;\longrightarrow\; \text{val} : e_2}\ \textbf{Raise div}$$

Theorem

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

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

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

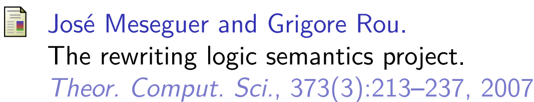
Summary

- 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.

For Further Reading



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

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.

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.