Towards Operations on Operational Semantics

Mauro Jaskelioff
mjj@cs.nott.ac.uk

School of Computer Science & IT

22nd British Colloquium for Theoretical Computer Science
The Context

▶ We need semantics to *reason* about programs.

▶ Operational semantics is a popular way of giving semantics to languages.

▶ Languages evolve over time and need to be extended.

▶ We want to use what we already knew to reason about the extended language.

▶ However, operational semantics have poor modularity.
Modularity in SOS

- An arithmetics language

\[ a ::= \text{Con } n \mid \text{Add } a \ a \]

where \( n \) ranges over \( \mathbb{Z} \).

\[ \text{Con } x \downarrow x \]
\[ \text{Add } t \ u \downarrow (x + y) \]

\[ t \downarrow x \quad u \downarrow y \]
\[ \text{Just } x \quad \text{Just } x \]
\[ \text{Just } x \quad \text{Nothing } u \quad y \]
Modularity in SOS

- An arithmetics language

\[ a ::= \text{Con } n \mid \text{Add } a \]

where \( n \) ranges over \( \mathbb{Z} \).

- An exceptions language

\[ e ::= \text{Throw} \mid \text{Catch } e \; e \]

\[ \frac{t \downarrow x \quad u \downarrow y}{\text{Add } t \; u \downarrow (x + y)} \]

\[ \frac{t \downarrow \text{Just } x}{\text{Catch } t \; u \downarrow \text{Just } x} \]

\[ \frac{t \downarrow \text{Nothing} \quad u \downarrow y}{\text{Catch } t \; u \downarrow y} \]
A Combined Language

\[ t ::= Con \ n \mid Add \ t \ t \mid Throw \mid Catch \ t \ t \]
A Combined Language

\[ t ::= \text{Con } n \mid \text{Add } t \ t \mid \text{Throw} \mid \text{Catch } t \ t \]

\[ \begin{align*}
  & \text{Con } x \Downarrow \text{Just } x \\
  & \text{Add } t \ u \Downarrow \text{Just } (x + y) \\
  & \text{Throw} \Downarrow \text{Nothing} \\
  & \text{Catch } t \ u \Downarrow \text{Just } x \\
  & \text{Catch } t \ u \Downarrow \text{Just } x \\
  & \text{Catch } t \ u \Downarrow \text{Just } x \\
  & \text{Add } t \ u \Downarrow \text{Nothing} \\
  & \text{Add } t \ u \Downarrow \text{Nothing} \\
  & \text{Add } t \ u \Downarrow \text{Nothing}
\end{align*} \]
A Combined Language

\[ t ::= \text{Con } n \mid \text{Add } t \ t \mid \text{Throw} \mid \text{Catch } t \ t \]

\[ \text{Con } x \downarrow \text{Just } x \quad \text{Add } t \ u \downarrow \text{Just } (x + y) \]

\[ \text{Throw} \downarrow \text{Nothing} \quad \text{Catch } t \ u \downarrow \text{Just } x \quad \text{Catch } t \ u \downarrow y \]

\[ \text{Add } t \ u \downarrow \text{Nothing} \]

\[ \text{Add } t \ u \downarrow \text{Nothing} \]

\[ \text{Add } t \ u \downarrow \text{Nothing} \]

\[ \text{Add } t \ u \downarrow \text{Nothing} \]

What is the relation between this semantics and the previous ones?
A Combined Language

\[ t ::= \text{Con } n \mid \text{Add } t \ t \mid \text{Throw} \mid \text{Catch } t \ t \]

- \[ \text{Con } x \Downarrow \text{Just } x \]
- \[ \text{Add } t \ u \Downarrow \text{Just } (x + y) \]
- \[ \text{Throw} \Downarrow \text{Nothing} \]
- \[ \text{Catch } t \ u \Downarrow \text{Just } x \]
- \[ \text{Catch } t \ u \Downarrow y \]
- \[ \text{Add } t \ u \Downarrow \text{Nothing} \]

- What is the relation between this semantics and the previous ones?
- Can we obtain rules that just propagate Nothing for free?
Functorial Operational Semantics

- Abstract formulation of operational semantics using category theory.

- Rules of SOS are expressed in terms of
  - The signature $\Sigma$ (set of operations)
  - The observable behaviour $B$

That is,

$$R(\Sigma, B)$$

What if... we have some operations on rules \( \mathcal{R}(\Sigma, B) \) such that:

- We could join to languages with different signatures, but same behaviour. 
  
  \[
  \text{join}: (\mathcal{R}(\Sigma, B), \mathcal{R}(\Sigma', B)) \rightarrow \mathcal{R}(\Sigma + \Sigma', B)
  \]

- We could lift a rule to some effect \( F \).
  
  \[
  \text{lift}: \mathcal{R}(\Sigma, B) \rightarrow \mathcal{R}(\Sigma, F \cdot B)
  \]

- We could construct rules with behaviour \( F \cdot B \) that are well-defined for any \( B \).
  
  \[
  \rho \tau: \forall B. \mathcal{R}(\Sigma, F \cdot B)
  \]
What if...  

...we have some operations on rules $\mathcal{R}(\Sigma, B)$ such that:

- We could join to languages with different signatures, but same behaviour.

  $$join: (\mathcal{R}(\Sigma, B), (\mathcal{R}(\Sigma', B))) \rightarrow \mathcal{R}(\Sigma + \Sigma', B)$$

- We could lift a rule to some effect $F$.

  $$lift: \mathcal{R}(\Sigma, B) \rightarrow \mathcal{R}(\Sigma, F \cdot B)$$

- We could construct rules with behaviour $F \cdot B$ that are well-defined for any $B$. 

  $$\rho \tau: \forall B. \mathcal{R}(\Sigma, F \cdot B)$$
What if . . .

. . . we have some operations on rules $\mathcal{R}(\Sigma, B)$ such that:

- We could join to languages with different signatures, but same behaviour.

$$\text{join}: (\mathcal{R}(\Sigma, B), (\mathcal{R}(\Sigma', B))) \rightarrow \mathcal{R}(\Sigma + \Sigma', B)$$

- We could lift a rule to some effect $F$.

$$\text{lift}: \mathcal{R}(\Sigma, B) \rightarrow \mathcal{R}(\Sigma, F \cdot B)$$
What if . . .

. . . we have some operations on rules $\mathcal{R}(\Sigma, B)$ such that:

- We could join to languages with different signatures, but same behaviour.

\[
\text{join: } (\mathcal{R}(\Sigma, B), (\mathcal{R}(\Sigma', B))) \rightarrow \mathcal{R}(\Sigma + \Sigma', B)
\]

- We could lift a rule to some effect $F$.

\[
lift: \mathcal{R}(\Sigma, B) \rightarrow \mathcal{R}(\Sigma, F \cdot B)
\]

- We could construct rules with behaviour $F \cdot B$ that are well-defined for any $B$.

\[
\rho_T: \forall B. \mathcal{R}(\Sigma, F \cdot B)
\]
Then we could... 

...answer the previous questions.
Semantics of arithmetics:

\[ \rho_A : \mathcal{R}(\Sigma_A, K_Z) \]

Semantics of exceptions:

\[ \rho_T : \forall B. \mathcal{R}(\Sigma_E, \text{Maybe} \cdot B) \]

with \text{Maybe } X = 1 + X
Then we could... 

... answer the previous questions.

Semantics of arithmetics:

\[ \rho_A : \mathcal{R}(\Sigma_A, K_{\mathbb{Z}}) \]

Semantics of exceptions:

\[ \rho_T : \forall B. \mathcal{R}(\Sigma_E, \text{Maybe} \cdot B) \]

with \( \text{Maybe} X = 1 + X \)

\[ \rho_A : \mathcal{R}(\Sigma_A, K_{\mathbb{Z}}) \]
\[ \rho_T : \forall B. \mathcal{R}(\Sigma_E, \text{Maybe} \cdot B) \]

\[ \text{lift}(\rho_A) : \mathcal{R}(\Sigma_A, \text{Maybe} \cdot K_{\mathbb{Z}}) \]
\[ \rho_T : \forall B. \mathcal{R}(\Sigma_E, \text{Maybe} \cdot B) \]

\[ \rho_{\tau_K} : \mathcal{R}(\Sigma_E, \text{Maybe} \cdot K_{\mathbb{Z}}) \]
\[ \text{join}(\rho_A, \rho_{\tau_K}) : \mathcal{R}(\Sigma_A + \Sigma_E, \text{Maybe} \cdot K_{\mathbb{Z}}) \]
Abstract Operational Rules

Our rules $\mathcal{R}(\Sigma, B)$, are actually abstract operational rules, natural transformations

$$\rho: \Sigma \cdot (\text{Id} \times B) \rightarrow B \cdot T\Sigma$$

where

- $T\Sigma$ is the free monad on the signature $\Sigma$. ($T\Sigma X$ is the set of terms with variables from $X$.)

$$\rho_X: \Sigma \cdot (X \times BX) \rightarrow (B \cdot T\Sigma)X$$
Abstract Operational Rules

Our rules \( \mathcal{R}(\Sigma, B) \), are actually \textit{abstract operational rules}, natural transformations

\[
\rho: \Sigma \cdot (Id \times B) \rightarrow B \cdot T\Sigma
\]

where

- \( T\Sigma \) is the free monad on the signature \( \Sigma \). (\( T\Sigma X \) is the set of terms with variables from \( X \).)

\[
\rho_X: \Sigma \cdot (X \times BX) \rightarrow (B \cdot T\Sigma)X
\]

Example: One rule for a binary sequence operator

\( (\cdot): X \times X \rightarrow X \)

\[
\frac{t \xrightarrow{a} t'}{t; u \xrightarrow{a} t'; u} \quad \Rightarrow \quad ((X \times BX \times X \times BX)) \rightarrow (B \cdot T\Sigma)X
\]

\[
(\cdot)(( t , \langle a, t' \rangle ) \times ( u , _ )) \rightarrow \langle a, t'; u \rangle
\]
Joining Rules

`join` puts together two languages with different signatures, but same behaviour.

\[
\begin{align*}
\rho : \Sigma \cdot (Id \times B) & \to B \cdot T_\Sigma \\
\rho' : \Sigma' \cdot (Id \times B) & \to B \cdot T_{\Sigma'} \\
join (\rho, \rho') & : (\Sigma + \Sigma') \cdot (Id \times B) \\
& = \{ \text{ Coproduct of Functors } \} \\
& \Sigma \cdot (Id \times B) + \Sigma' \cdot (Id \times B) \\
& \to \{ \rho + \rho' \} \\
& B \cdot T_\Sigma + B \cdot T_{\Sigma'} \\
& \to \{ [Binl + Binr] \} \\
& B \cdot (T_\Sigma + T_{\Sigma'}) \\
& \to \{ B[\text{fold (inl, inr.inl)}, \text{fold (inl, inr.inr)}] \} \\
& B \cdot (T_{\Sigma + \Sigma'})
\end{align*}
\]
Lifting Rules

*lift* lifts a rule with behaviour $B$ to a behaviour $F \cdot B$.

- For $F$ strong and a distributivity law $\Sigma \cdot F \rightarrow F \cdot \Sigma$

  
  \[
  \rho : \Sigma \cdot (Id \times B) \rightarrow B \cdot T_{\Sigma}
  \]

  \[
  \text{lift}_F \rho : \Sigma \cdot (Id \times F \cdot B)
  \]

  \[
  \rightarrow \{ \text{strength of } F \} \Sigma \cdot F \cdot (Id \times B)
  \]

  \[
  \rightarrow \{ \text{distributivity law } \} F \cdot \Sigma \cdot (Id \times B)
  \]

  \[
  \rightarrow \{ F\rho \} F \cdot B \cdot T_{\Sigma}
  \]

  If $F$ is applicative and $\Sigma$ traversable, we obtain the strength and distributivity law for free.

For simple signatures and all monadic effects, we get "propagation rules" for free.
Lifting Rules

\( \text{lift} \) lifts a rule with behaviour \( B \) to a behaviour \( F \cdot B \).

- For \( F \) strong and a distributivity law \( \Sigma \cdot F \rightarrow F \cdot \Sigma \)

\[
\rho : \Sigma \cdot (\text{Id} \times B) \rightarrow B \cdot T_{\Sigma}
\]

\[
\text{lift}_F \rho : \Sigma \cdot (\text{Id} \times F \cdot B)
\rightarrow \{ \text{ strength of } F \} \\
\Sigma \cdot F \cdot (\text{Id} \times B)
\rightarrow \{ \text{ distributivity law } \} \\
F \cdot \Sigma \cdot (\text{Id} \times B)
\rightarrow \{ F \rho \} \\
F \cdot B \cdot T_{\Sigma}
\]

- if \( F \) is \textit{applicative} and \( \Sigma \) \textit{traversable}, we obtain the strength and distributivity law for free.

- For simple signatures and all monadic effects, we get “propagation rules” for free.
Rule Transformers

- A rule transformer is a mapping from a behaviour $B$ to a rule $\rho_\tau : \Sigma \cdot (Id \times F \cdot B) \rightarrow F \cdot B \cdot T_\Sigma$. 
Rule Transformers

- A rule transformer is a mapping from a behaviour $B$ to a rule $\rho_\tau : \Sigma \cdot (Id \times F \cdot B) \rightarrow F \cdot B \cdot T_\Sigma$.
- They can be generated from a transformer germ: a natural transformation $\tau : \Sigma \cdot F \rightarrow F$.

\[
\tau : \Sigma \cdot F \rightarrow F
\]

\[
\rho_\tau : \Sigma \cdot (Id \times F \cdot B) \\
\rightarrow \{ \Sigma \pi_2 \} \\
\Sigma \cdot F \cdot B \\
\rightarrow \{ \tau_B \} \\
F \cdot B \\
\rightarrow \{ (F \cdot B)\eta \} \\
F \cdot B \cdot T_\Sigma
\]
Lifting $T$ to $D$-coalgebras

Functorial operational semantics are a distributivity law

$$\lambda: T \cdot D \rightarrow D \cdot T$$

between

- a monad $T$ (corresponding to syntax)
- a comonad $D$ (corresponding to behaviours)

Equivalently, a lifting $\tilde{T}$ of $T$ to the $D$-coalgebras:

For all $k: X \rightarrow DX$,

$$\tilde{T}(k): TX \rightarrow D(TX)$$

To execute a program (a closed term $T\emptyset$) we unfold

$$\tilde{T}(e): T\emptyset \rightarrow D(T\emptyset),$$

where $e: \emptyset \rightarrow D\emptyset$. 
We can easily reason about operational semantics by working in the abstract (category-theoretical) setting of functorial operational semantics.

We can build complex semantics out of simpler building blocks, using operations on abstract operational rules (but with some limitations.)

Future Work
- Broaden the class of languages that we can represent (variable binding).
- Construct more powerful operations to combine two languages (instead of transforming one.)
Thanks for listening

Haskell code will be available for downloading at
http://www.cs.nott.ac.uk/~mjj/