Towards an executable denotational semantics for causal block diagrams

Ben Denckla – Denckla Consulting
Pieter J. Mosterman – The MathWorks
Hans Vangheluwe – McGill University
What is the meaning of $\times$?

We’d like it to be $\lambda(x, y, z) \rightarrow (x \times y, x \times y \times z)$.

E.g., $(3, 4, 5) \rightarrow (12, 60)$.

How can we formalize these semantics?
We formalize by cascading two translational semantics and a traditional denotational semantics.

block diagram language
↓ (translation)
BdAppLang
↓ (translation)
AppLang
↓ (denotation)
Haskell
block diagram language to BdAppLang translation

\[
\begin{array}{ccc}
1 & \times \rightarrow 1  \\
2 & \times \rightarrow 1  \\
3 & \times \rightarrow 2
\end{array}
\]

\[
\downarrow
\]

**diagram blocks**

<table>
<thead>
<tr>
<th>name</th>
<th>function</th>
<th>inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>mula</td>
<td>\times</td>
<td>1 2</td>
</tr>
<tr>
<td>mulb</td>
<td>\times</td>
<td>mula 3</td>
</tr>
</tbody>
</table>

**diagram outputs**

<table>
<thead>
<tr>
<th>mula</th>
</tr>
</thead>
<tbody>
<tr>
<td>mulb</td>
</tr>
</tbody>
</table>
BdAppLang to AppLang translation

<table>
<thead>
<tr>
<th>name</th>
<th>function</th>
<th>inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>mula</td>
<td>×</td>
<td>1 2</td>
</tr>
<tr>
<td>mulb</td>
<td>×</td>
<td>mula 3</td>
</tr>
</tbody>
</table>

**diagram outputs**

| mula | mulb |

\[
\lambda \text{toplevel} \rightarrow
\]

letrec

\[
mula = \times (\text{toplevel}_1, \text{toplevel}_2)
\]

\[
mulb = \times (mula, \text{toplevel}_3)
\]

in (mula, mulb)
AppLang to Haskell denotation

\[ \lambda \text{toplevel} \rightarrow \]
\[ \text{letrec} \]
\[ \quad mula = \times (\text{toplevel}_1, \text{toplevel}_2) \]
\[ \quad mulb = \times (mula, \text{toplevel}_3) \]
\[ \quad \text{in} (mula, mulb) \]

\[ \downarrow \text{(abbreviated)} \]

\[ m [\lambda i \rightarrow x] e = \lambda v \rightarrow m x (ue i v e) \]
\[ m [f a] e = (m f e) (m a e) \]
\[ m [\text{letrec} \ ds \ \text{in} \ x] e = m x (\text{fix } \lambda e' \rightarrow ue (md \ ds \ e') \ e) \]
\[ m [(a, b)] e = (m a e, m b e) \]
\[ m [i] e = e i \]
\[ m [\text{int } n] e = n \]
I've implemented this all in Haskell.

\[
\text{Dia blockList outputList}
\]

where

\[
\text{blockList} = [
\text{Bld "mula" tupleMul [Din 0, Din 1]},
\text{Bld "mulb" tupleMul [Blo "mula" 0, Din 2]}
\]

\[
\text{outputList} = [\text{Blo "mula" 0}, \text{Blo "mulb" 0}]
\]

If \( bd \) is the BdAppLang block diagram above, \( trd \) translates from BdAppLang to AppLang, and \( mPro \) gives the meaning of a program in AppLang,

\[
mPro (\text{App (trd bd) (tixn [3, 4, 5])}) = (12, 60)
\]
But this is all kind of boring!

Instead of just arithmetic, we want to do signal processing, controls, and continuous-time simulation via numerical approximation of ODEs!

We can!

E.g., feedback already supported via fixed-point semantics.

With lazy lists (streams) and implicit state added, lots of applications are supported.

But it is all based on the small, clean semantics presented here with a boring arithmetic example.