CONVENTIONAL SEMANTIC APPROACHES

Grigore Rosu
CS522 – Programming Language Semantics
A language designer should understand the existing design approaches, techniques and tools, to know what is possible and how, or to come up with better ones. This part of the course will cover the major PL semantic approaches, such as:

- Big-step structural operational semantics (Big-step SOS)
- Small-step structural operational semantics (Small-step SOS)
- Denotational semantics
- Modular structural operational semantics (Modular SOS)
- Reduction semantics with evaluation contexts
- Abstract Machines
- The chemical abstract machine
- Axiomatic semantics
A simple imperative language
IMP – A Simple Imperative Language

We will exemplify the conventional semantic approaches by means of IMP, a very simple non-procedural imperative language, with

- Arithmetic expressions
- Boolean expressions
- Assignment statements
- Conditional statements
- While loop statements
- Blocks
**IMP Syntax**

- \( \text{Int} ::= \) the domain of (unbounded) integer numbers, with usual operations on them
- \( \text{Bool} ::= \) the domain of Booleans
- \( \text{Id} ::= \) standard identifiers
- \( \text{AExp} ::= \) \( \text{Int} \)
  - \( \text{Id} \)
  - \( \text{AExp} + \text{AExp} \)
  - \( \text{AExp} / \text{AExp} \)
- \( \text{BExp} ::= \) \( \text{Bool} \)
  - \( \text{AExp} <= \text{AExp} \)
  - \( \lnot \text{BExp} \)
  - \( \text{BExp} \&\& \text{BExp} \)
- \( \text{Block} ::= \) \{ \}
  - \{ Stmt \}
- \( \text{Stmt} ::= \) \( \text{Block} \)
  - \( \text{Id} = \text{AExp} ; \)
  - \( \text{Stmt} \text{ Stmt} \)
  - if ( \( \text{BExp} \) ) \( \text{Block} \) else \( \text{Block} \)
  - while ( \( \text{BExp} \) ) \( \text{Block} \)
- \( \text{Pgm} ::= \) int List\{Id\} ; Stmt

Suppose that, for demonstration purposes, we want “+” and “/” to be non-deterministically strict, “\(<=\)” to be sequentially strict, and “\&\&” to be short-circuited.
mod IMP-SYNTAX is including PL-INT + PL-BOOL + PL-ID .

--- AEExp
  sort AEExp . subsorts Int Id < AEExp .
  op _+_ : AEExp AEExp -> AEExp [prec 33 gather (E e format (d b o d))] .
  op _/-_ : AEExp AEExp -> AEExp [prec 31 gather (E e format (d b o d))] .

--- BExp
  sort BExp . subsort Bool < BExp .
  op _<=_ : AEExp AEExp -> BExp [prec 37 format (d b o d)] .
  op !_ : BExp -> BExp [prec 53 format (b o d)] .
  op _&&_ : BExp BExp -> BExp [prec 55 gather (E e format (d b o d))] .

--- Block and Stmt
  sorts Block Stmt . subsort Block < Stmt .
  op {} : -> Block [format (b b o)] .
  op {_} : Stmt -> Block [format (d n+i n--i d)] .
  op _=_; : Id AEExp -> Stmt [prec 40 format (d b o b o)] .
  op __ : Stmt Stmt -> Stmt [prec 60 gather (e E) format (d ni d)] .
  op if(_)_else_ : BExp Block Block -> Stmt [prec 59 format (b so d d s nib o d)] .
  op while(_) : BExp Block -> Stmt [prec 59 format (b so d d s d)] .

--- Pgm
  sort Pgm .
  op int_;_ : List{Id} Stmt -> Pgm [prec 70 format (nb o d ni d)] .
endm
Most semantics need some notion of state. A state holds all the semantic ingredients to fully define the meaning of a given program or fragment of program.

For IMP, a state is a partial finite-domain function from identifiers to integers (i.e., a function defined only on a finite subset of identifiers and undefined on the rest), written using a half-arrow:

$$\sigma : Id \rightarrow Int$$

We let State denote the set of such functions, and may write it

$$[Id \rightarrow Int]_{\text{finite}}$$

or

$$\text{Map} \{Id \mapsto Int\}$$
Lookup, Update and Initialization

- We may write states by enumerating each identifier binding. For example, the following state binds $x$ to 8 and $y$ to 0:
  \[ \sigma = x \mapsto 8, y \mapsto 0 \]

- Typical state operations are lookup, update and initialization.

  - **Lookup**
    \[ \_(_): State \times Id \to Int \]

  - **Update**
    \[ \_[/]_: State \times Int \times Id \to State \]

  - **Initialization**
    \[ \_ \mapsto \_ : List\{Id\} \times Int \to State \]
mod STATE is including PL-INT + PL-ID .
   sort State .

   op _|->_ : List{Id} Int -> State [prec 0] .
   op .State : -> State .
   op _&_ : State State -> State [assoc comm id: .State format(d s s d)] .

   op _(_) : State Id -> Int [prec 0] .     --- lookup
   op _[/_ ] : State Int Id -> State [prec 0] .  --- update

var Sigma : State . var I I’ : Int . var X X’ : Id . var Xl : List{Id} .

eq X |-> undefined = .State .                        --- "undefine" a state in a variable

eq (Sigma & X |-> I)(X) = I .

eq Sigma(X) = undefined [owise] .

eq (Sigma & X |-> I)[I’ / X ] = (Sigma & X |-> I’ ) .

eq Sigma[I / X] = (Sigma & X |-> I) [owise] .

eq (X,X’,Xl) |-> I = X |-> I & X’ |-> I & Xl |-> I .

eq .List{Id} |-> I = .State .

endm
BIG-STEP SOS

Big-step structural operational semantics
Big-Step Structural Operational Semantics (Big-Step SOS)

- Gilles Kahn (1987), under the name natural semantics. Also known as relational semantics, or evaluation semantics. We can regard a big-step SOS as a recursive interpreter, telling for a fragment of code and state what it evaluates to.

- **Configuration**: tuple containing code and semantic ingredients
  - E.g., \( \langle a_1, \sigma \rangle \quad \langle a_1 + a_2, \sigma \rangle \quad \langle i_1 \rangle \quad \langle i_1 + \text{Int} \ i_2 \rangle \quad \langle \sigma \rangle \)

- **Sequent**: Pair of configurations, to be derived or proved
  - E.g., \( \langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_1 + a_2, \sigma \rangle \Downarrow \langle i_1 + \text{Int} \ i_2 \rangle \)

- **Rule**: Tells how to derive a sequent from others
  - E.g.,

\[
\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i_1 + \text{Int} \ i_2 \rangle}
\]

Read “evaluates to”

May omit line when no premises
Big-Step SOS of IMP - Arithmetic

\[ \langle i, \sigma \rangle \downarrow \langle i \rangle \]

State lookup

\[ \langle x, \sigma \rangle \downarrow \langle \sigma(x) \rangle \quad \text{if} \quad \sigma(x) \neq \perp \]

\[ \langle a_1, \sigma \rangle \downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \downarrow \langle i_2 \rangle \]

\[ \langle a_1 + a_2, \sigma \rangle \downarrow \langle i_1 + \text{Int } i_2 \rangle \]

\[ \langle a_1, \sigma \rangle \downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \downarrow \langle i_2 \rangle \]

\[ \langle a_1 / a_2, \sigma \rangle \downarrow \langle i_1 / \text{Int } i_2 \rangle \]

Side condition ensures rule will never apply when \( a_2 \) evaluates to 0

Read: “provided that \( a_1 \) evaluates to \( i_1 \) in \( \sigma \) and \( a_2 \) evaluates to \( i_2 \) in \( \sigma \), then \( a_1 + a_2 \) evaluates to the integer sum of \( i_1 \) and \( i_2 \) in \( \sigma \)
Big-Step SOS of IMP - Boolean

\[
\begin{align*}
\langle t, \sigma \rangle & \Downarrow \langle t \rangle \\
\langle a_1, \sigma \rangle & \Downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle \\
\langle a_1 \leq a_2, \sigma \rangle & \Downarrow \langle i_1 \leq_{Int} i_2 \rangle \\
\langle b, \sigma \rangle & \Downarrow \langle \text{true} \rangle \\
\langle \neg b, \sigma \rangle & \Downarrow \langle \text{false} \rangle \\
\langle \neg b, \sigma \rangle & \Downarrow \langle \text{true} \rangle \\
\langle b_1, \sigma \rangle & \Downarrow \langle \text{false} \rangle \\
\langle b_1 \&\& b_2, \sigma \rangle & \Downarrow \langle \text{false} \rangle \\
\langle b_1, \sigma \rangle & \Downarrow \langle \text{true} \rangle \quad \langle b_2, \sigma \rangle \Downarrow \langle t \rangle \\
\langle b_1 \&\& b_2, \sigma \rangle & \Downarrow \langle t \rangle
\end{align*}
\]
Big-Step SOS of IMP - Statements

\[
\langle \{\}, \sigma \rangle \Downarrow \langle \sigma \rangle \quad \text{(BigStep-Empty-Block)}
\]

\[
\langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle \quad \langle \{ s \}, \sigma \rangle \Downarrow \langle \sigma' \rangle \quad \text{(BigStep-Block)}
\]

\[
\langle a, \sigma \rangle \Downarrow \langle i \rangle \quad \langle x = a; , \sigma \rangle \Downarrow \langle \sigma[i/x] \rangle \quad \text{if } \sigma(x) \neq \perp \quad \text{(BigStep-Asgn)}
\]

\[
\langle s_1, \sigma \rangle \Downarrow \langle \sigma_1 \rangle \quad \langle s_2, \sigma_1 \rangle \Downarrow \langle \sigma_2 \rangle \quad \langle s_1 \; s_2, \sigma \rangle \Downarrow \langle \sigma_2 \rangle \quad \text{(BigStep-Seq)}
\]

\[
\langle b, \sigma \rangle \Downarrow \langle \text{true} \rangle \quad \langle s_1, \sigma \rangle \Downarrow \langle \sigma_1 \rangle \quad \langle \text{if } (b) \; s_1 \text{ else } s_2, \sigma \rangle \Downarrow \langle \sigma_1 \rangle \quad \text{(BigStep-If-True)}
\]

\[
\langle b, \sigma \rangle \Downarrow \langle \text{false} \rangle \quad \langle s_2, \sigma \rangle \Downarrow \langle \sigma_2 \rangle \quad \langle \text{if } (b) \; s_1 \text{ else } s_2, \sigma \rangle \Downarrow \langle \sigma_2 \rangle \quad \text{(BigStep-If-False)}
\]

\[
\langle b, \sigma \rangle \Downarrow \langle \text{false} \rangle \quad \langle \text{while } (b) \; s, \sigma \rangle \Downarrow \langle \sigma \rangle \quad \text{(BigStep-While-False)}
\]

\[
\langle b, \sigma \rangle \Downarrow \langle \text{true} \rangle \quad \langle s \; \text{while } (b) \; s, \sigma \rangle \Downarrow \langle \sigma' \rangle \quad \langle \text{while } (b) \; s, \sigma \rangle \Downarrow \langle \sigma' \rangle \quad \text{(BigStep-While-True)}
\]
Big-Step SOS of IMP - Programs

State initialization

\[
\frac{\langle s, x \leftarrow 0 \rangle \Downarrow \langle \sigma \rangle}{\langle \text{int } x; s \rangle \Downarrow \langle \sigma \rangle}
\]

(BigStep-Var)
Big-Step Rule Instances

- Rules are schemas, allowing recursively enumerable many instances; side conditions filter out instances

  - E.g., these are correct instances of the rule for division

    \[
    \frac{\langle x, (x \mapsto 8, y \mapsto 0) \rangle \downarrow \langle 8 \rangle \quad \langle 2, (x \mapsto 8, y \mapsto 0) \rangle \downarrow \langle 2 \rangle}{\langle x/2, (x \mapsto 8, y \mapsto 0) \rangle \downarrow \langle 4 \rangle}
    \]

    \[
    \frac{\langle x, (x \mapsto 8, y \mapsto 0) \rangle \downarrow \langle 8 \rangle \quad \langle 2, (x \mapsto 8, y \mapsto 0) \rangle \downarrow \langle 4 \rangle}{\langle x/2, (x \mapsto 8, y \mapsto 0) \rangle \downarrow \langle 2 \rangle}
    \]

    The second may look suspicious, but it is not. Normally, one should never be able to apply it, because one cannot prove its hypotheses

  - However, the following is not a correct instance (no matter what \(?\) is):

    \[
    \frac{\langle x, (x \mapsto 8, y \mapsto 0) \rangle \downarrow \langle 8 \rangle \quad \langle y, (x \mapsto 8, y \mapsto 0) \rangle \downarrow \langle 0 \rangle}{\langle x/y, (x \mapsto 8, y \mapsto 0) \rangle \downarrow \langle ? \rangle}
    \]
The following is a valid proof derivation, or proof tree, using the big-step SOS proof system of IMP above.

Suppose that $x$ and $y$ are identifiers and $\sigma(x) = 8$ and $\sigma(y) = 0$.
Big-Step SOS for Type Systems

- Big-Step SOS is routinely used to define type systems for programming languages.
- The idea is that a fragment of code $c$, in a given type environment $\Gamma$, can be assigned a certain type $\tau$. We typically write

$$\Gamma \vdash c : \tau$$

instead of

$$\langle c, \Gamma \rangle \Downarrow \langle \tau \rangle$$

- Since all variables in IMP have integer type, $\Gamma$ can be replaced by a list of untyped variables in our case. In general, however, a type environment $\Gamma$ contains typed variables, that is, pairs “$x : \tau$”. 
Typing Arithmetic Expressions

\[
\begin{align*}
xl & \leftarrow i : \text{int} \\
\text{if } x \in xl & \\
\frac{xl \leftarrow a_1 : \text{int} \quad xl \leftarrow a_2 : \text{int}}{xl \leftarrow a_1 + a_2 : \text{int}} \\
\frac{xl \leftarrow a_1 : \text{int} \quad xl \leftarrow a_2 : \text{int}}{xl \leftarrow a_1 / a_2 : \text{int}} \\
xl & \leftarrow t : \text{bool} \quad \text{if } t \in \{\text{true, false}\}
\end{align*}
\]

(BigStepTypeSystem-Int)

(BigStepTypeSystem-Lookup)

(BigStepTypeSystem-Add)

(BigStepTypeSystem-Div)

(BigStepTypeSystem-Bool)
Typing Boolean Expressions

\[
\begin{align*}
xl &\leftarrow a_1 : int && xl &\leftarrow a_2 : int \\
\hline
xl &\leftarrow a_1 \leq a_2 : bool \\
\hline
xl &\leftarrow b : bool \\
\hline
xl &\leftarrow \neg b : bool \\
\hline
xl &\leftarrow b_1 : bool && xl &\leftarrow b_2 : bool \\
\hline
xl &\leftarrow b_1 \&\& b_2 : bool \\
\hline
xl &\leftarrow \{\} : block \\
\hline
& (\texttt{BigStepTypeSystem-Leq}) \\
& (\texttt{BigStepTypeSystem-Not}) \\
& (\texttt{BigStepTypeSystem-And}) \\
& (\texttt{BigStepTypeSystem-Empty-Block})
\end{align*}
\]
Typing Statements

The type of $s$ can be either block or stmt

\[
\frac{x_l \leftarrow s : \tau}{x_l \leftarrow \{ s \} : \text{block}} \quad \text{if } \tau \in \{\text{block, stmt}\}
\]

\[
\frac{x_l \leftarrow a : \text{int}}{x_l \leftarrow x = a ; : \text{stmt}} \quad \text{if } x \in x_l
\]

\[
\frac{x_l \leftarrow s_1 : \tau_1 \quad x_l \leftarrow s_2 : \tau_2}{x_l \leftarrow s_1 \ s_2 : \text{stmt}} \quad \text{if } \tau_1, \tau_2 \in \{\text{block, stmt}\}
\]

\[
\frac{x_l \leftarrow b : \text{bool} \quad x_l \leftarrow s_1 : \text{block} \quad x_l \leftarrow s_2 : \text{block}}{x_l \leftarrow \text{if } (b) \ s_1 \ \text{else } s_2 : \text{stmt}}
\]

\[
\frac{x_l \leftarrow b : \text{bool} \quad x_l \leftarrow s : \text{block}}{x_l \leftarrow \text{while } (b) \ s : \text{stmt}}
\]
Typing Programs

\[
\frac{x : \tau \vdash s}{\text{int } x : \tau; \ s : \text{pgm}} \quad \text{if } \tau \in \{\text{block}, \text{stmt}\}
\]
Big-Step SOS Type Derivation

Like the big-step rules for the concrete semantics of IMP, the ones for its type system are also rule schemas. We next show a proof derivation for the well-typed-ness of an IMP program that adds all the numbers from 1 to 100:

\[
\begin{align*}
\text{tree}_1 & \quad \text{tree}_2 \quad \text{tree}_3 \\
\text{n,s} \vdash (\text{while } (! (\text{n} \leq 0)) \{ \text{s} = \text{s} + \text{n}; \text{n} = \text{n} + -1; \}) : \text{stmt} \\
\text{n,s} \vdash (\text{n} = 100; \text{s} = 0; \text{while } (! (\text{n} \leq 0)) \{ \text{s} = \text{s} + \text{n}; \text{n} = \text{n} + -1; \}) : \text{stmt} \\
\vdash (\text{int n,s; n} = 100; \text{s} = 0; \text{while } (! (\text{n} \leq 0)) \{ \text{s} = \text{s} + \text{n}; \text{n} = \text{n} + -1; \}) : \text{pgm}
\end{align*}
\]

where
Big-Step SOS Type Derivation

\[
tree_1 = \left\{ \begin{array}{c}
\frac{n,s \vdash 100 : \text{int}}{n,s \vdash (n = 100;) : \text{stmt}} \quad \frac{n,s \vdash 0 : \text{int}}{n,s \vdash (s = 0;) : \text{stmt}} \\
n,s \vdash (n = 100; s = 0;) : \text{stmt}
\end{array} \right.
\]

\[
tree_2 = \left\{ \begin{array}{c}
\frac{n,s \vdash n : \text{int}}{n,s \vdash 0 : \text{int}} \\
n,s \vdash (n \leq 0) : \text{bool} \\
n,s \vdash (! (n \leq 0)) : \text{bool}
\end{array} \right.
\]
Big-Step SOS Type Derivation

\[ \text{tree}_3 = \{ \]

\[
\begin{array}{c}
\quad n,s \vdash s : \text{int} & \quad n,s \vdash n : \text{int} & \quad n,s \vdash \neg 1 : \text{int} \\
\hline
\quad n,s \vdash (s + n) : \text{int} & \quad n,s \vdash (n + -1) : \text{int} \\
\hline
\quad n,s \vdash (s = s + n; ) : \text{stmt} & \quad n,s \vdash (n = n + -1; ) : \text{stmt} \\
\hline
\quad n,s \vdash (s = s + n; n = n + -1; ) : \text{stmt} \\
\hline
\quad n,s \vdash \{ s = s + n; n = n + -1; \} : \text{block}
\end{array}
\]
Any big-step SOS can be associated a rewrite logic theory (or, equivalently, a Maude module)

The idea is to associate to each big-step SOS rule a rewrite rule

\[
\frac{C_1 \downarrow R_1 \quad C_2 \downarrow R_2 \quad \ldots \quad C_n \downarrow R_n}{C \downarrow R} \quad \text{[if condition]}
\]

a rewrite rule

\[\overline{C} \rightarrow \overline{R} \quad \text{if} \quad \overline{C_1} \rightarrow \overline{R_1} \wedge \overline{C_2} \rightarrow \overline{R_2} \wedge \ldots \wedge \overline{C_n} \rightarrow \overline{R_n} \quad \text{[\wedge condition]}\]

(over-lining means “algebraization”)

SMALL-STEP SOS

Small-step structural operational semantics
Small-Step Structural Operational Semantics (Small-Step SOS)

- Gordon Plotkin (1981)
- Also known as transitional semantics, or reduction semantics
- One can regard a small-step SOS as a device capable of executing a program step-by-step

**Configuration**: tuple containing code and semantic ingredients
- E.g., \( \langle a, \sigma \rangle \), \( \langle b, \sigma \rangle \), \( \langle s, \sigma \rangle \), \( \langle p \rangle \)

**Sequent (transition)**: Pair of configurations, to be derived (proved)
- E.g., \( \langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle \), \( \langle a_1 + a_2, \sigma \rangle \rightarrow \langle a'_1 + a_2, \sigma \rangle \)

**Rule**: Tells how to derive a sequent from others
- E.g.,

\[
\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a'_1 + a_2, \sigma \rangle}
\]
Small-Step SOS of IMP - Arithmetic

\[ \langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle \quad \text{if } \sigma(x) \neq \bot \]

\[ \langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle \]

\[ \frac{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a_1' + a_2, \sigma \rangle \quad (\text{SMALLSTEP-ADD-ARG1})}{\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma \rangle} \]

\[ \frac{\langle a_1 + a_2, \sigma \rangle \rightarrow \langle a_1 + a'_2, \sigma \rangle \quad (\text{SMALLSTEP-ADD-ARG2})}{\langle i_1 + i_2, \sigma \rangle \rightarrow \langle i_1 + \text{Int} i_2, \sigma \rangle \quad (\text{SMALLSTEP-ADD})} \]

\[ + \text{ is non-deterministic (its arguments can evaluate in any order, and interleaved) } \]
Small-Step SOS of IMP - Arithmetic

\[
\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle \\
\langle a_1 / a_2, \sigma \rangle \rightarrow \langle a'_1 / a_2, \sigma \rangle \\
\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma \rangle \\
\langle a_1 / a_2, \sigma \rangle \rightarrow \langle a_1 / a'_2, \sigma \rangle \\
\langle i_1 / i_2, \sigma \rangle \rightarrow \langle i_1 / \text{Int} \ i_2, \sigma \rangle \quad \text{if } i_2 \neq 0
\]

(SMALLSTEP-DIV-ARG1)

(SMALLSTEP-DIV-ARG2)

(SMALLSTEP-DIV)

Side condition ensures rule will never apply when denominator is 0

/ is also non-deterministic
Small-Step SOS of IMP - Boolean

\[
\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle \\
\frac{\langle a_1 \leq a_2, \sigma \rangle \rightarrow \langle a'_1 \leq a_2, \sigma \rangle}{\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma \rangle} \\
\frac{\langle i_1 \leq a_2, \sigma \rangle \rightarrow \langle i'_1 \leq a_2, \sigma \rangle}{\langle i_1 \leq i_2, \sigma \rangle \rightarrow \langle i'_1 \leq \text{Int} i_2, \sigma \rangle}
\]

(SMALLSTEP-LEQ-ARG1)

(SMALLSTEP-LEQ-ARG2)

(SMALLSTEP-LEQ)

Ensures sequential strictness
Small-Step SOS of IMP - Boolean

\[
\begin{align*}
\langle b, \sigma \rangle & \rightarrow \langle b', \sigma \rangle \\
\langle \neg b, \sigma \rangle & \rightarrow \langle \neg b', \sigma \rangle & (\text{SMALLSTEP-NOT-ARG}) \\
\langle \neg \text{true}, \sigma \rangle & \rightarrow \langle \text{false}, \sigma \rangle & (\text{SMALLSTEP-NOT-TRUE}) \\
\langle \neg \text{false}, \sigma \rangle & \rightarrow \langle \text{true}, \sigma \rangle & (\text{SMALLSTEP-NOT-FALSE}) \\
\langle b_1, \sigma \rangle & \rightarrow \langle b'_1, \sigma \rangle \\
\langle b_1 \&\& b_2, \sigma \rangle & \rightarrow \langle b'_1 \&\& b_2, \sigma \rangle & (\text{SMALLSTEP-AND-ARG1}) \\
\langle \text{false} \&\& b_2, \sigma \rangle & \rightarrow \langle \text{false}, \sigma \rangle & (\text{SMALLSTEP-AND-FALSE}) \\
\langle \text{true} \&\& b_2, \sigma \rangle & \rightarrow \langle b_2, \sigma \rangle & (\text{SMALLSTEP-AND-TRUE})
\end{align*}
\]
Small-Step SOS Derivation

The following is a valid proof derivation, or proof tree, using the small-step SOS proof system for expressions of IMP above. Suppose that $x$ and $y$ are identifiers and $\sigma(x) = 1$.
Small-Step SOS of IMP - Statements

\[ \langle \{ s \}, \sigma \rangle \rightarrow \langle s, \sigma \rangle \]

\[ \langle a, \sigma \rangle \rightarrow \langle a', \sigma \rangle \]

\[ \langle x = a; , \sigma \rangle \rightarrow \langle x = a'; , \sigma \rangle \]

\[ \langle x = i; , \sigma \rangle \rightarrow \langle \{ \}, \sigma[i/x] \rangle \] if \( \sigma(x) \neq \bot \)

\[ \langle s_1, \sigma \rangle \rightarrow \langle s'_1, \sigma' \rangle \]

\[ \langle s_1 \ s_2, \sigma \rangle \rightarrow \langle s'_1 \ s_2, \sigma' \rangle \]

\[ \langle \{ \} \ s_2, \sigma \rangle \rightarrow \langle s_2, \sigma \rangle \]

State update

(SMALLSTEP-BLOCK)

(SMALLSTEP-ASGN-ARG2)

(SMALLSTEP-ASGN)

(SMALLSTEP-SEQ-ARG1)

(SMALLSTEP-SEQ-EMPTY-BLOCK)
Small-Step SOS of IMP - Statements

\[ \langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle \]
\[ \langle \text{if} \ (b) \ s_1 \ \text{else} \ s_2, \sigma \rangle \rightarrow \langle \text{if} \ (b') \ s_1 \ \text{else} \ s_2, \sigma \rangle \]
\[ \langle \text{if} \ (\text{true}) \ s_1 \ \text{else} \ s_2, \sigma \rangle \rightarrow \langle s_1, \sigma \rangle \]
\[ \langle \text{if} \ (\text{false}) \ s_1 \ \text{else} \ s_2, \sigma \rangle \rightarrow \langle s_2, \sigma \rangle \]
\[ \langle \text{while} \ (b) \ s, \sigma \rangle \rightarrow \langle \text{if} \ (b) \ \{ \ s \ \text{while} \ (b) \ s \ \} \ \text{else} \ \{ \}, \sigma \rangle \]
\[ \langle \text{int} \ xl; \ s \rangle \rightarrow \langle s, xl \leftarrow 0 \rangle \]

(SMALLSTEP-IF-ARG1)
(SMALLSTEP-IF-TRUE)
(SMALLSTEP-IF-FALSE)
(SMALLSTEP-WHILE)
(SMALLSTEP-VAR)
Any small-step SOS can be associated with a rewrite logic theory (or, equivalently, a Maude module).

The idea is to associate to each small-step SOS rule

\[
C_1 \rightarrow C'_1 \quad C_2 \rightarrow C'_2 \quad \ldots \quad C_n \rightarrow C'_n
\]

a rewrite rule

\[
\overline{\circ C} \rightarrow \overline{C'} \quad \text{if} \quad \overline{\circ C_1} \rightarrow \overline{C'_1} \land \overline{\circ C_2} \rightarrow \overline{C'_2} \land \ldots \land \overline{\circ C_n} \rightarrow \overline{C'_n} \land \text{condition}
\]

(the circle means “ready for one step”)

\[ C \rightarrow C' \quad \text{[if condition]} \]
DENOTATIONAL

Denotational or fixed-point semantics
Denotational Semantics

Christopher Strachey and Dana Scott (1970)

- Associate *denotation*, or meaning, to (fragments of) programs into *mathematical domains*; for example,
  - The denotation of an arithmetic expression in IMP is a *partial function from states to integer numbers*:
    \[
    [-] : \text{AExp} \rightarrow (\text{State} \rightarrow \text{Int})
    \]
  - The denotation of a statement in IMP is a *partial function from states to states*:
    \[
    [-] : \text{Stmt} \rightarrow (\text{State} \rightarrow \text{State})
    \]
Denotational Semantics

- **Partial because some expressions may be undefined in some states (e.g., division by zero)**

  - The denotation of an arithmetic expression in IMP is a *partial function from states to integer numbers*
    \[
    [-] : \text{AExp} \rightarrow (\text{State} \rightarrow \text{Int})
    \]

  - The denotation of a statement in IMP is a *partial function from states to states*
    \[
    [-] : \text{Stmt} \rightarrow (\text{State} \rightarrow \text{State})
    \]

- **Partial because some statements in some states may use undefined expressions, or may not terminate**
Denotational Semantics - Terminology

Denotation (function)  \[ [-] : \text{AExp} \rightarrow (\text{State} \rightarrow \text{Int}) \]

Fragment of program

Mathematical domain

\[ [-] : \text{Stmt} \rightarrow (\text{State} \rightarrow \text{State}) \]
Once the right mathematical domains are chosen, giving a denotational semantics to a language should be a straightforward and compositional process; e.g.

\[
[a_1 + a_2]_\sigma = [a_1]_\sigma +_{\text{Int}} [a_2]_\sigma
\]

\[
[a_1 / a_2]_\sigma = \begin{cases}
[a_1]_\sigma /_{\text{Int}} [a_2]_\sigma & \text{if } [a_2]_\sigma \neq 0 \\
\bot & \text{if } [a_2]_\sigma = 0
\end{cases}
\]

The hardest part is to give semantics to recursion. This is done using fixed-points.
Mathematical Domains

- Mathematical domains can be anything; it is common though that they are organized as complete partial orders with bottom element.

- The partial order structure captures the intuition of informativeness: \( a \leq b \) means \( a \) is less informative than \( b \). E.g., as a loop is executed, we get more and more information about its semantics.

- Completeness means that chains of more and more informative elements have a limit.

- The bottom element, written \( \perp \), stands for undefined, or no information at all.
Partial Orders

- Partial order \((D, \subseteq)\) is a set \(D\) and binary relation \(\subseteq\) which is
  - Reflexive: \(x \subseteq x\)
  - Transitive: \(x \subseteq y \text{ and } y \subseteq z \implies x \subseteq z\)
  - Anti-symmetric: \(x \subseteq y \text{ and } y \subseteq x \implies x = y\)
- Total order = partial order with \(x \subseteq y \text{ or } y \subseteq x\)
- Important example: domains of partial functions

\[(A \rightarrow B, \leq)\]

\[f \leq g \text{ iff } g \text{ defined everywhere } f \text{ is defined and } f(a) = g(a) \text{ whenever } f(a) \text{ is defined}\]
(Least) Upper Bounds

- An upper bound (u.b.) of $X \subseteq D$ is any element $p \in D$ such that $x \subseteq p$ for any $x \in X$.

- The least upper bound (l.u.b.) of $X \subseteq D$, written $\uparrow X$, is an upper bound with $\uparrow X \subseteq q$ for any u.b. $q$.
  
  - When they exist, least upper bounds are unique.

- The domains of partial functions, $(A \rightarrow B, \leq)$, admit upper bounds and least upper bounds if and only if all the partial functions in the considered set are compatible: any two agree on any element in which both are defined.
Complete Partial Orders (CPO)

- A chain in \((D, \sqsubseteq)\) is an infinite sequence
  \[d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots \sqsubseteq d_n \sqsubseteq \ldots\]
  also written
  \[\{d_n \mid n \in \mathbb{N}\}\]

- Partial order \((D, \sqsubseteq)\) is a complete partial order (CPO) iff any of its chains admits a least upper bound

- \((D, \sqsubseteq, \bot)\) is a bottomed CPO (BCPO) iff \(\bot\) is a minimal element of \(D\), also called its bottom

- The domain of partial functions \((A \rightarrow B, \leq, \bot)\) is a BCPO, where \(\bot\) is the partial function undefined everywhere
Monotone and Continuous Functions

- \( \mathcal{F} : (D, \sqsubseteq) \to (D', \sqsubseteq') \) is monotone iff
  \[ x \sqsubseteq y \implies \mathcal{F}(x) \sqsubseteq' \mathcal{F}(y) \]

- Monotone functions preserve chains:
  \[ \{d_n \mid n \in \mathbb{N}\} \text{ chain implies } \{\mathcal{F}(d_n) \mid n \in \mathbb{N}\} \text{ chain} \]

  However, they do not always preserve l.u.b. of chains.

- \( \mathcal{F} : (D, \sqsubseteq) \to (D', \sqsubseteq') \) is continuous iff monotone and preserves l.u.b. of chains:
  \[ \sqcup \mathcal{F}(d_n) = \mathcal{F}(\sqcup d_n) \]

- \( \text{Cont}((D, \sqsubseteq, \bot), (D', \sqsubseteq', \bot')) \), the domain of continuous functions between two BCPOs, is itself a BCPO
Fixed-Point Theorem

Let \((D, \sqsubseteq, \bot)\) be a BCPO and \(\mathcal{F}: (D, \sqsubseteq, \bot) \rightarrow (D, \sqsubseteq, \bot)\) be a continuous function. Then the l.u.b. of the chain \(\{\mathcal{F}^n(\bot) \mid n \in \mathbb{N}\}\) is the least fixed-point of \(\mathcal{F}\).

Typically written \(\text{fix}(\mathcal{F})\).

Proof sketch:

\[
\mathcal{F}(\text{fix}(\mathcal{F})) = \mathcal{F}(\bigsqcup_{n \in \mathbb{N}} \mathcal{F}^n(\bot)) \\
= \bigsqcup_{n \in \mathbb{N}} \mathcal{F}^{n+1}(\bot) \\
= \bigsqcup_{n \in \mathbb{N}} \mathcal{F}^n(\bot) \\
= \text{fix}(\mathcal{F}).
\]
Consider the following “definition” of the factorial:

\[ f(n) = \begin{cases} 
1 & , \text{if } n = 0 \\
 n \times f(n - 1) & , \text{if } n > 0 
\end{cases} \]

This is a recursive definition

- Is it well-defined? Why?
- Yes. Because it is the least fixed-point of the following continuous (prove it!) function from \( \mathbb{N} \rightarrow \mathbb{N} \) to itself

\[ \mathcal{F}(g)(n) = \begin{cases} 
1 & , \text{if } n = 0 \\
 n \times g(n - 1) & , \text{if } n > 0 \text{ and } g(n - 1) \text{ defined} \\
\text{undefined} & , \text{if } n > 0 \text{ and } g(n - 1) \text{ undefined} 
\end{cases} \]
\([\_]: AExp \rightarrow (State \rightarrow Int)\)

\([i]_\sigma = i\)

\([x]_\sigma = \sigma(x)\)

\([a_1 + a_2]_\sigma = [a_1]_\sigma +_{Int} [a_2]_\sigma\)

\([a_1 / a_2]_\sigma = \begin{cases} 
[a_1]_\sigma /_{Int} [a_2]_\sigma & \text{if } [a_2]_\sigma \neq 0 \\
\bot & \text{if } [a_2]_\sigma = 0
\end{cases}\)
Denotational Semantics of IMP

Boolean Expressions

\[ [\_] : BExp \rightarrow (State \rightarrow Bool) \]

\[ [t] \sigma = t \]

\[ [a_1 <= a_2] \sigma = [a_1] \sigma \leq_{Int} [a_2] \sigma \]

\[ ![b] \sigma = \neg_{Bool}([b] \sigma) \]

\[ [b_1 \&\& b_2] \sigma = \begin{cases} [b_2] \sigma & \text{if } [b_1] \sigma = \text{true} \\ \text{false} & \text{if } [b_1] \sigma = \text{false} \\ \bot & \text{if } [b_1] \sigma = \bot \end{cases} \]
Denotational Semantics of IMP Statements (without loops)

\[
\llbracket \_ \rrbracket : Stmt \rightarrow (State \rightarrow State)
\]

\[
\llbracket \{ \} \rrbracket \sigma = \sigma
\]

\[
\llbracket \{ s \} \rrbracket \sigma = \llbracket s \rrbracket \sigma
\]

\[
\llbracket x = a ; \rrbracket \sigma = \begin{cases} 
\sigma[\llbracket a \rrbracket \sigma / x] & \text{if } \sigma(x) \neq \bot \\
\bot & \text{if } \sigma(x) = \bot
\end{cases}
\]

\[
\llbracket s_1 \ s_2 \rrbracket \sigma = \llbracket s_2 \rrbracket \llbracket s_1 \rrbracket \sigma
\]

\[
\llbracket \text{if} \ (b) \ s_1 \ \text{else} \ s_2 \rrbracket \sigma = \begin{cases} 
\llbracket s_1 \rrbracket \sigma & \text{if } \llbracket b \rrbracket \sigma = \text{true} \\
\llbracket s_2 \rrbracket \sigma & \text{if } \llbracket b \rrbracket \sigma = \text{false} \\
\bot & \text{if } \llbracket b \rrbracket \sigma = \bot
\end{cases}
\]
Denotational Semantics of IMP

While

- We first define a continuous function as follows

\[ \mathcal{F} : (\text{State} \rightarrow \text{State}) \rightarrow (\text{State} \rightarrow \text{State}) \]

\[ \mathcal{F}(\alpha)(\sigma) = \begin{cases} 
\alpha([s]\sigma) & \text{if } [b]\sigma = \text{true} \\
\sigma & \text{if } [b]\sigma = \text{false} \\
\bot & \text{if } [b]\sigma = \bot 
\end{cases} \]

- Then we define the denotational semantics of while

\[ [[\text{while } (b) \ s]] = \text{fix}(\mathcal{F}) \]
Formalizing Denotational Semantics

- A denotational semantics is like a “compiler” of the defined programming language into mathematics.
- Formalizing denotational semantics in RL reduces to formalizing the needed mathematical domains:
  - Integers, Booleans, etc.
  - Functions (with cases) and function applications
  - Fixed-points
- Such mathematics is already available in functional programming languages, which makes them excellent candidates for denotational semantics!
Rewriting/equational logics do not have built-in functions and fixed-points, so they need to be defined. They are, however, easy to define using rewriting.

In fact, we do not need rewrite rules, all we need is equations to define these simple domains.

See next slide.
Functional CPO Domain

sorts:

$\text{Var}_{\text{CPO}}, \text{CPO}$

subsorts:

$\text{Var}_{\text{CPO}}, \text{Bool} \subseteq \text{CPO}$

operations:

$\text{fun}_{\text{CPO}} -> _ : \text{Var}_{\text{CPO}} \times \text{CPO} \rightarrow \text{CPO}$

$\text{app}_{\text{CPO}}(_,-) : \text{CPO} \times \text{CPO} \rightarrow \text{CPO}$

$\text{fix}_{\text{CPO}} - : \text{CPO} \rightarrow \text{CPO}$

$\text{if}_{\text{CPO}}(_,-,-) : \text{CPO} \times \text{CPO} \times \text{CPO} \rightarrow \text{CPO}$

equations:

$\text{app}_{\text{CPO}}(\text{fun}_{\text{CPO}} V-> C, C') = \text{C}[C'/V]$  

$\text{fix}_{\text{CPO}} \text{fun}_{\text{CPO}} V-> C = \text{C}[(\text{fix}_{\text{CPO}} \text{fun}_{\text{CPO}} V-> C)/V]$  

$\text{if}_{\text{CPO}}(\text{true}, C, C') = C$  

$\text{if}_{\text{CPO}}(\text{false}, C, C') = C'$
sort:  
  Syntax  // generic sort for syntax

subsorts:  
  AExp, BExp, Stmt, Pgm < Syntax  // syntactic categories fall under Syntax  
  Int, Bool, State < CPO  // basic domains are regarded as CPOs

operation:  
  \([\_\_\_] : Syntax \rightarrow CPO\)  // denotation of syntax
\[[I]\] = \text{fun}_{CPO} \sigma \rightarrow I
\[[X]\] = \text{fun}_{CPO} \sigma \rightarrow \sigma(X)
\[[A_1 + A_2]\] = \text{fun}_{CPO} \sigma \rightarrow \text{app}_{CPO}(\[[A_1]\], \sigma) +_{\text{Int}} \text{app}_{CPO}(\[[A_2]\], \sigma)
\[[A_1 / A_2]\] = \text{fun}_{CPO} \sigma \rightarrow \text{if}_{CPO}(\text{app}_{CPO}(\[[A_2]\], \sigma) \neq_{\text{Int}} 0, \text{app}_{CPO}(\[[A_1]\], \sigma) \mid_{\text{Int}} \text{app}_{CPO}(\[[A_2]\], \sigma), \bot)
Denotational Semantics of IMP in RL

Boolean Expressions

\[
\begin{align*}
\llbracket T \rrbracket &= \text{fun}_{\text{CPO}} \sigma \rightarrow T \\
\llbracket A_1 \leq A_2 \rrbracket &= \text{fun}_{\text{CPO}} \sigma \rightarrow \text{app}_{\text{CPO}}(\llbracket A_1 \rrbracket, \sigma) \leq_{\text{int}} \text{app}_{\text{CPO}}(\llbracket A_2 \rrbracket, \sigma) \\
\llbracket \neg B \rrbracket &= \text{fun}_{\text{CPO}} \sigma \rightarrow \neg_{\text{Bool}}\text{app}_{\text{CPO}}(\llbracket B \rrbracket, \sigma) \\
\llbracket B_1 \land B_2 \rrbracket &= \text{fun}_{\text{CPO}} \sigma \rightarrow \text{if}_{\text{CPO}}(\text{app}_{\text{CPO}}(\llbracket B_1 \rrbracket, \sigma), \text{app}_{\text{CPO}}(\llbracket B_2 \rrbracket, \sigma), \text{false})
\end{align*}
\]
Denotational Semantics of IMP in RL

Statements and Programs

\[
[\{\}]] = \text{fun}_{\text{CPO}} \sigma \rightarrow \sigma
\]

\[
[\{ S \}]] = \text{fun}_{\text{CPO}} \sigma \rightarrow \text{app}_{\text{CPO}}([[S]], \sigma)
\]

\[
[X = A;] = \text{fun}_{\text{CPO}} \sigma \rightarrow \text{app}_{\text{CPO}}(\text{fun}_{\text{CPO}} \text{arg} \rightarrow \text{if}_{\text{CPO}}(\sigma(X) \neq \bot, \sigma[\text{arg}/X], \bot), \text{app}_{\text{CPO}}([[A]], \sigma))
\]

\[
[S_1 S_2] = \text{fun}_{\text{CPO}} \sigma \rightarrow \text{app}_{\text{CPO}}([[S_2]], \text{app}_{\text{CPO}}([[S_1]], \sigma))
\]

\[
[\text{if} (B) S_1 \text{ else } S_2] = \text{fun}_{\text{CPO}} \sigma \rightarrow \text{if}_{\text{CPO}}(\text{app}_{\text{CPO}}([[B]], \sigma), \text{app}_{\text{CPO}}([[S_1]], \sigma), \text{app}_{\text{CPO}}([[S_2]], \sigma))
\]

\[
[\text{while} (B) S] = \text{fix}_{\text{CPO}} \text{fun}_{\text{CPO}} \alpha \rightarrow \text{fun}_{\text{CPO}} \sigma \rightarrow \text{if}_{\text{CPO}}(\text{app}_{\text{CPO}}([[B]], \sigma), \text{app}_{\text{CPO}}(\alpha, \text{app}_{\text{CPO}}([[S]], \sigma)), \sigma)
\]

\[
[\text{int } X = 0; S] = \text{app}_{\text{CPO}}([[S]], (X \mapsto 0))
\]
MSOS

Modular structural operational semantics
Modular Structural Operational Semantics (Modular SOS, or MSOS)

- Peter Mosses (1999)
- Addresses the non-modularity aspects of SOS
  - A definitional framework is non-modular when, in order to add a new feature to an existing language, one needs to revisit and change some of the already defined, unrelated language features
  - The non-modularity of SOS becomes clear when we define IMP++
- Why modularity is important
  - Modifying existing rules when new rules are added is error prone
  - When experimenting with language design, one needs to make changes quickly; having to do unrelated changes slows us down
  - Rapid language development, e.g., domain-specific languages
Separate the syntax from configurations and treat it differently.

Transitions go from syntax to syntax, hiding the other configuration components into transition labels.

Labels encode all the non-syntactic configuration changes.

Specialized notation in transition labels, to:

- Say that certain configuration components stay unchanged.
- Say that certain configuration changes are propagated from the premise to the conclusion of a rule.
An MSOS transition has the form

\[ P \xrightarrow{\Delta} P' \]

- \( P \) and \( P' \) are programs or fragments of program
- \( \Delta \) is a label describing the changes in the configuration components, defined as a record; primed fields stay for “after” the transition

Example:

\[
\begin{align*}
  x &:= i \quad \{ \text{state} = \sigma, \text{state} = \sigma[i/x], \ldots \} \\
  \quad \Rightarrow \quad \text{skip} \quad \text{if } \sigma(x) \neq \bot
\end{align*}
\]

This rule can be automatically “desugared” into the SOS rule

\[
\langle x := i, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[i/x] \rangle \quad \text{if } \sigma(x) \neq \bot
\]

But also into (if the configuration contains more components, like in IMP++)

\[
\langle x := i, \sigma, \omega \rangle \rightarrow \langle \text{skip}, \sigma[i/x], \omega \rangle \quad \text{if } \sigma(x) \neq \bot
\]
MSOS Labels

- Labels are field assignments, or records, and can use “…” for “and so on”, called record comprehension.

- Fields can be primed or not.
  - Unprimed = configuration component before the transition is applied
  - Primed = configuration component after the transition is applied

- Some fields appear both unprimed and primed (called read-write), while others appear only primed (called write-only) or only unprimed (called read-only).
MSOS Labels

- **Field types**
  - **Read/write** = fields which appear both unprimed and unprimed
    \[x := i \quad \text{state} = \sigma, \text{state}' = \sigma[i/x], \ldots \rightarrow \text{skip} \quad \text{if } \sigma(x) \neq \perp\]
  - **Write-only** = fields which appear only primed
    \[\text{print } i \quad \text{output}' = i, \ldots \rightarrow \text{skip}\]
  - **Read-only** = fields which appear only unprimed
    \[\begin{array}{c}
    e_2 \quad \text{env} = \rho[v_1/x], \ldots \rightarrow e'_2 \\
    \text{let } x = v_1 \text{ in } e_2 \quad \text{env} = \rho, \ldots \rightarrow \text{let } x = v_1 \text{ in } e'_2
    \end{array}\]
MSOS Rules

- Like in SOS, but using MSOS transitions as sequents
- Same labels or parts of them can be used multiple times in a rule
- Example:

\[
\frac{s_1 \rightarrow s_1'}{s_1 \triangleright s_1'}
\]

\[
\frac{s_1; s_2 \rightarrow s_1'; s_2}{s_1 \triangleright s_1'}
\]

- Same $\triangleright$ means that changes propagate from premise to conclusion
- The author of MSOS now promotes a simplifying notation
  - If the premise and the conclusion repeat the same label or part of it, simply drop that label or part of it. For example:

\[
\frac{s_1 \rightarrow s_1'}{s_1 \rightarrow s_1'}
\]

\[
\frac{s_1; s_2 \rightarrow s_1'; s_2}{s_1; s_2 \rightarrow s_1'; s_2}
\]
MSOS of IMP - Arithmetic

\[ x \xrightarrow{\text{state}=\sigma,...} \sigma(x) \quad \text{if } \sigma(x) \neq \bot \]

\[
\begin{align*}
  a_1 &\rightarrow a'_1 \\
  a_1 + a_2 &\rightarrow a'_1 + a_2 \\
  a_2 &\rightarrow a'_2 \\
  a_1 + a_2 &\rightarrow a_1 + a'_2 \\
i_1 + i_2 &\rightarrow i_1 + \text{Int} \ i_2
\end{align*}
\]

(\text{MSOS-Lookup})

(\text{MSOS-Add-Arg1})

(\text{MSOS-Add-Arg2})

(\text{MSOS-Add})
MSOS of IMP - Arithmetic

\[
\frac{a_1 \rightarrow a'_1}{a_1 / a_2 \rightarrow a'_1 / a_2} \\
\frac{a_2 \rightarrow a'_2}{a_1 / a_2 \rightarrow a_1 / a'_2}
\]

\[(\text{MSOS-Div-Arg1})\]

\[(\text{MSOS-Div-Arg2})\]

\[i_1 / i_2 \rightarrow i_1 / \text{Int} i_2 \quad \text{if } i_2 \neq 0\]

\[(\text{MSOS-Div})\]
MSOS of IMP - Boolean

\[ \frac{a_1 \rightarrow a'_1}{a_1 \leq a_2 \rightarrow a'_1 \leq a_2} \quad (\text{MSOS-Leq-Arg1}) \]

\[ \frac{a_2 \rightarrow a'_2}{i_1 \leq a_2 \rightarrow i_1 \leq a'_2} \quad (\text{MSOS-Leq-Arg2}) \]

\[ i_1 \leq i_2 \rightarrow i_1 \leq_{\text{Int}} i_2 \quad (\text{MSOS-Leq}) \]
MSOS of IMP - Boolean

\[ b \rightarrow b' \]
\[ \text{not } b \rightarrow \text{not } b' \]  
(MSOS-Not-Arg)

not true $\rightarrow$ false  
(MSOS-Not-True)

not false $\rightarrow$ true  
(MSOS-Not-False)

\[ b_1 \rightarrow b_1' \]
\[ b_1 \text{ and } b_2 \rightarrow b_1' \text{ and } b_2 \]  
(MSOS-And-Arg1)

false and $b_2$ $\rightarrow$ false  
(MSOS-And-False)

true and $b_2$ $\rightarrow$ $b_2$  
(MSOS-And-True)
### MSOS of IMP - Statements

\[
\begin{align*}
\frac{a \rightarrow a'}{x := a \rightarrow x := a'}
\end{align*}
\]

\[
\begin{align*}
x := i & \quad \{\text{state} = \sigma, \text{state}' = \sigma[i/x], \ldots\} \\
& \quad \frac{\text{skip}}{\text{if } \sigma(x) \neq 1}
\end{align*}
\]

\[
\begin{align*}
\frac{s_1 \rightarrow s_1'}{s_1 \;; s_2 \rightarrow s_1' \;; s_2}
\end{align*}
\]

\[
\begin{align*}
\text{skip} \;; s_2 \rightarrow s_2
\end{align*}
\]
\[ b \rightarrow b' \]

\[
\text{if } b \text{ then } s_1 \text{ else } s_2 \rightarrow \text{if } b' \text{ then } s_1 \text{ else } s_2
\]

\[
\text{if true then } s_1 \text{ else } s_2 \rightarrow s_1
\]

\[
\text{if false then } s_1 \text{ else } s_2 \rightarrow s_2
\]

\[
\text{while } b \text{ do } s \rightarrow \text{if } b \text{ then } (s ; \text{while } b \text{ do } s) \text{ else } \text{skip}
\]

\[
\text{var } x_l ; s \xrightarrow{\{\text{state } = x_l \rightarrow 0, \ldots\}} s
\]

(MSOS-IF-ARG1)  
(MSOS-IF-TRUE)  
(MSOS-IF-FALSE)  
(MSOS-WHILE)  
(MSOS-VAR)
Any MSOS can be associated with a rewrite logic theory (or, equivalently, a Maude module).

Idea:
- Desugar MSOS into SOS
- Apply the SOS-to-rewriting-logic representation, but
- Hold the non-syntactic configuration components in an ACI-data-structure, so that we can use ACI matching to retrieve only the fields of interest (which need to be read or written to).
MSOS of IMP in Maude

- See file
  - `imp-semantics-msos.maude`
RSEC

Reduction semantics with evaluation contexts
Reduction Semantics with Evaluation Contexts (RSEC)

- Matthias Felleisen and collaborators (1992)
- Previous operational approaches encoded the program execution context as a proof context, by means of rule conditions or premises
  - This has a series of advantages, but makes it hard to define control intensive features, such as abrupt termination, exceptions, call/cc, etc.
- We would like to have the execution context explicit, so that we can easily save it, change it, or even delete it
- Reduction semantics with evaluation contexts does precisely that
  - It allows to formally define evaluation contexts
  - Rules become mostly unconditional
  - Reductions can only happen “in context”
Splitting and Plugging

- RSEC relies on reversible implicit mechanisms to
  - Split syntax into an evaluation context and a redex
  - Plug a redex into an evaluation contexts and obtain syntax again

\[ p = c[e] \]
Evaluation Contexts

- Evaluation contexts are typically defined by the same means that we use to define the language syntax, that is, grammars.
- The hole □ represents the place where redex is to be plugged.
- Example:

\[
\text{Context} ::= \\
| \Box \\
| \text{Context} \leftarrow \text{AExp} \\
| \text{Int} \leftarrow \text{Context} \\
| \text{Id} ::= \text{Context} \\
| \text{Context} ; \text{Stmt} \\
| \text{if Context then Stmt else Stmt} \\
| \ldots \\
\]
Correct Evaluation Contexts

□

3 <= □

□ <= 3

□; x := 5

if □ then $s_1$ else $s_2$
Wrong Evaluation Contexts

□ <= □

x <= 3

x <= □

x := 5 ; □

□ := 5

if x <= 7 then □ else x := 5
Splitting/Plugging of Syntax

\[ 7 = (\square)[7] \]

\[ 3 \leq x = (3 \leq \square)[x] = (\square \leq x)[3] = (\square)[3 \leq x] \]

\[ 3 \leq (2 + x) + 7 = (3 \leq \square + 7)[2 + x] = (\square \leq (2 + x) + 7)[3] = \ldots \]
The characteristic rule of RSEC allows us to only define semantic rules stating how redexes are reduced:

- This significantly reduces the number of rules.
- The semantic rules are mostly unconditional (no premises).
- The overall result is a semantics which is compact and easy to read and understand.
# RSEC of IMP — Evaluation Contexts

<table>
<thead>
<tr>
<th>IMP evaluation contexts syntax</th>
<th>IMP language syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context</strong> ::= □</td>
<td><strong>AExp</strong> ::= <em>Int</em></td>
</tr>
<tr>
<td></td>
<td><strong>AExp + AExp</strong></td>
</tr>
<tr>
<td><strong>Context + AExp</strong></td>
<td><strong>BExp</strong> ::= <em>Bool</em></td>
</tr>
<tr>
<td><strong>AExp + Context</strong></td>
<td><strong>AExp &lt;= AExp</strong></td>
</tr>
<tr>
<td><strong>Context / AExp</strong></td>
<td><strong>BExp &lt;= BExp</strong></td>
</tr>
<tr>
<td><strong>AExp / Context</strong></td>
<td><strong>Id := AExp</strong></td>
</tr>
<tr>
<td><strong>Context &lt;= AExp</strong></td>
<td><strong>Id := Context</strong></td>
</tr>
<tr>
<td><strong>Int &lt;= Cxt</strong></td>
<td><strong>Context and BExp</strong></td>
</tr>
<tr>
<td><strong>not Context</strong></td>
<td><strong>Id := AExp</strong></td>
</tr>
<tr>
<td><strong>Context and BExp</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Id := Context</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Context ; Stmt</strong></td>
<td></td>
</tr>
<tr>
<td><strong>if Context then Stmt else Stmt</strong></td>
<td><strong>Pgm</strong> ::= <strong>var List{Id} ; Stmt</strong></td>
</tr>
</tbody>
</table>
RSEC of IMP – Rules

\[
\text{Context ::= } \ldots \mid \langle \text{Context, State} \rangle
\]

\[
e \rightarrow e' \\
\frac{c[e] \rightarrow c[e']}{c[e]} \rightarrow \langle c[e'] \rangle
\]

\[
\langle c, \sigma \rangle[x] \rightarrow \langle c, \sigma \rangle[\sigma(x)] \quad \text{if } \sigma(x) \neq \bot
\]

\[
i_1 + i_2 \rightarrow i_1 +_{\text{Int}} i_2
\]

\[
i_1 / i_2 \rightarrow i_1 /_{\text{Int}} i_2 \quad \text{if } i_2 \neq 0
\]

\[
i_1 \leq_{\text{Int}} i_2 \rightarrow i_1 \leq_{\text{Int}} i_2
\]

\[
\text{not true } \rightarrow \text{ false}
\]

\[
\text{not false } \rightarrow \text{ true}
\]

\[
\text{true and } b_2 \rightarrow b_2
\]

\[
\text{false and } b_2 \rightarrow \text{ false}
\]

\[
\langle c, \sigma \rangle[x := i] \rightarrow \langle c, \sigma[i/x] \rangle[\text{skip}] \quad \text{if } \sigma(x) \neq \bot
\]

\[
\text{skip}; s_2 \rightarrow s_2
\]

\[
f i l e \text{ true then } s_1 \text{ else } s_2 \rightarrow s_1
\]

\[
f i l e \text{ false then } s_1 \text{ else } s_2 \rightarrow s_2
\]

\[
\text{while } b \text{ do } s \rightarrow \text{ if } b \text{ then } (s; \text{ while } b \text{ do } s) \text{ else } \text{skip}
\]

\[
\langle \text{var } x l ; s \rangle \rightarrow \langle s, (x l \mapsto 0) \rangle
\]
RSEC Derivation

\[
\{ x := 1 ; y := 2 ; \text{if } x \leq y \text{ then } x := 0 \text{ else } y := 0 , (x \mapsto 0, y \mapsto 0) \}\]

\rightarrow \{ \text{skip ; } y := 2 ; \text{if } x \leq y \text{ then } x := 0 \text{ else } y := 0 , (x \mapsto 1, y \mapsto 0) \}\]

\rightarrow \{ y := 2 ; \text{if } x \leq y \text{ then } x := 0 \text{ else } y := 0 , (x \mapsto 1, y \mapsto 0) \}\]

\rightarrow \{ \text{skip ; if } x \leq y \text{ then } x := 0 \text{ else } y := 0 , (x \mapsto 1, y \mapsto 2) \}\]

\rightarrow \{ \text{if } x \leq y \text{ then } x := 0 \text{ else } y := 0 , (x \mapsto 1, y \mapsto 2) \}\]

\rightarrow \{ \text{if } 1 \leq y \text{ then } x := 0 \text{ else } y := 0 , (x \mapsto 1, y \mapsto 2) \}\]

\rightarrow \{ \text{if } 1 \leq 2 \text{ then } x := 0 \text{ else } y := 0 , (x \mapsto 1, y \mapsto 2) \}\]

\rightarrow \{ \text{if true then } x := 0 \text{ else } y := 0 , (x \mapsto 1, y \mapsto 2) \}\]

\rightarrow \{ x := 0 , (x \mapsto 1, y \mapsto 2) \}\]

\rightarrow \{ \text{skip , } (x \mapsto 0, y \mapsto 2) \}\]
Like with the other styles, RSEC can also be faithfully represented in rewriting logic and, implicitly, in Maude.

However, RSEC is context sensitive, while rewriting logic is not (rewriting logic allows rewriting strategies, but one can still not match and modify the context, as we can do in RSEC).

We therefore need:
- A mechanism to achieve context sensitivity (the splitting/plugging) in rewriting logic.
- Use that mechanism to represent the characteristic rule of RSEC.
Evaluation Contexts in Rewriting Logic

- An evaluation context CFG production in RSEC has the form

\[
\text{Context} \ ::= \ \pi(N_1, \ldots, N_n, \text{Context})
\]

- Associate to each such production one rule and one equation:

\[
\text{split}(\pi(T_1, \ldots, T_n, T)) \rightarrow \pi(T_1, \ldots, T_n, C)[\text{Syn}] \quad \text{if} \quad \text{split}(T) \rightarrow C[\text{Syn}]
\]

\[
\text{plug}(\pi(T_1, \ldots, T_n, C)[\text{Syn}]) = \pi(T_1, \ldots, T_n, \text{plug}(C[\text{Syn}]))
\]

- Plus, we add one generic rule and one generic equation:

\[
\text{split}(\text{Syn}) \rightarrow \square[\text{Syn}]
\]

\[
\text{plug}(\square[\text{Syn}]) = \text{Syn}
\]
IMP Examples:

- For productions
  
  \[
  \text{Context} ::= \begin{cases} 
  \text{Context} <= AExp \\
  \text{Int} <= \text{Context} \\
  \text{Id} : = \text{Context} 
  \end{cases}
  \]

  we add the following rewrite logic rules and equations:

  \[
  \text{split}(A_1 <= A_2) \rightarrow (C <= A_2)[\text{Syn}] \text{ if } \text{split}(A_1) \rightarrow C[\text{Syn}]
  \]

  \[
  \text{plug}((C <= A_2)[\text{Syn}]) = \text{plug}(C[\text{Syn}]) <= A_2
  \]

  \[
  \text{split}(I_1 <= A_2) \rightarrow (I_1 <= C)[\text{Syn}] \text{ if } \text{split}(A_2) \rightarrow C[\text{Syn}]
  \]

  \[
  \text{plug}((I_1 <= C)[\text{Syn}]) = I_1 <= \text{plug}(C[\text{Syn}])
  \]

  \[
  \text{split}(X : = A) \rightarrow (X : = C)[\text{Syn}] \text{ if } \text{split}(A) \rightarrow C[\text{Syn}]
  \]

  \[
  \text{plug}((X : = C)[\text{Syn}]) = X : = \text{plug}(C[\text{Syn}])
  \]
RSEC Reduction Rules in Rewriting Logic

// for each term $l$ that appears as the left-hand side of a reduction rule
// "$l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow \ldots\$", add the following conditional
// rewrite rule (there could be one $l$ for many reduction rules):

$\mathcal{O}(T_1, \ldots, T_n) \rightarrow T \textbf{ if } \text{plug}(\mathcal{O}(\text{split}(T_1), \ldots, \text{split}(T_n))) \rightarrow T$

// for each non-identity term $r$ appearing as right-hand side in a reduction rule
// "$\ldots \rightarrow r(c_1[r_1], \ldots, c_n[r_n])\$", add the following equation
//(there could be one $r$ for many reduction rules):

$\text{plug}(\mathcal{O}(\text{Syn}_1, \ldots, \text{Syn}_n)) = \mathcal{O}(\text{plug}(\text{Syn}_1), \ldots, \text{plug}(\text{Syn}_n))$

// for each reduction semantics rule "$l(c_1[l_1], \ldots, c_n[l_n]) \rightarrow r(c'_1[r_1], \ldots, c'_n[r_n'])\$"
// add the following "semantic" rewrite rule:

$\mathcal{O}(\overline{c_1[l_1]}, \ldots, \overline{c_n[l_n]}) \rightarrow \mathcal{O}(\overline{c'_1[r_1]}, \ldots, \overline{c'_n[r_n']})$
IMP Examples

- One rule of first kind
  \[ C_{fg} \rightarrow C_{fg}' \text{ if } \text{plug}(\circ \text{split}(C_{fg})) \rightarrow C_{fg}' \]

- No need for equations of second kind

- Characteristic rule of RSEC:
  \[ C[Syn] \rightarrow C[Syn'] \text{ if } C \neq \Box \wedge \circ Syn \rightarrow Syn' \]

- The remaining rules are as natural as can be:
  - \[ I_1 \leq I_2 \rightarrow I_1 \leq_{\text{Int}} I_2 \]
  - \[ \text{skip} ; S_2 \rightarrow S_2 \]
  - \[ \text{if true then } S_1 \text{ else } S_2 \rightarrow S_1 \]
RSEC of IMP in Maude

- See file
  - imp-split-plug.maude
- See files
  - imp-semantics-evaluation-contexts-1.maude
  - imp-semantics-evaluation-contexts-2.maude
  - imp-semantics-evaluation-contexts-3.maude
CHAM

The chemical abstract machine
The Chemical Abstract Machine (CHAM)

- Berry and Boudol (1992)
- Both a model of concurrency and a specific semantic style
- Chemical metaphor
  - States regarded as chemical solutions containing floating molecules
  - Molecules can interact with each other by means of reactions
  - Reactions take place concurrently, unrestricted by context
  - Solutions are encapsulated within new molecules, using membranes
    - The following is a solution containing \( k \) molecules:
      \[
      \{m_1 \ m_2 \ \ldots \ m_k\}
      \]
CHAM Syntax and Rules

\[
\begin{align*}
Molecule & ::= \text{Solution} \\
      & | \ Molecule \triangleleft \text{Solution} \\
Solution & ::= \{\text{Bag}\{Molecule\}\}
\end{align*}
\]
CHAM Rules

- Ordinary rewrite rules between solution terms:

\[ m_1 \ m_2 \ldots \ m_k \rightarrow m'_1 \ m'_2 \ldots m'_l \]

- Rewriting takes place only within solutions

- Three (metaphoric) kinds of rules
  - \textit{Heating} rules using \( \rightarrow \) : structurally rearrange solution
  - \textit{Cooling} rules using \( \rightarrow \) : clean up solution after reactions
  - \textit{Reaction} rules using \( \rightarrow \) : change solution irreversibly
CHAM Airlock

- Allows to extract molecules from encapsulated solutions
- Governed by two rules coming in a heating/cooling pair:

$$\{m_1 \ m_2 \ \ldots \ m_k\} \Leftrightarrow \{m_1 \triangleleft \{m_2 \ \ldots \ m_k\}\}$$
A top-level solution containing two subsolution molecules

- One for holding the syntax
- Another for holding the state

Example:

\[
\begin{align*}
&\{\{x := (3 / (x + 2))\}\} \quad \{x \to 1, y \to 0\}\}
\end{align*}
\]
Airlock can be Problematic

- Airlock cannot be used to encode evaluation strategies; heating/cooling rules of the form

\[
x := a \quad \Leftarrow \quad a \triangleleft \{x := \Box\}
\]

\[
a_1 + a_2 \quad \Leftarrow \quad a_1 \triangleleft \{\Box + a_2\}
\]

\[
a_1 + a_2 \quad \Leftarrow \quad a_2 \triangleleft \{a_1 + \Box\}
\]

are problematic, because they yield ambiguity, e.g.,

\[
\{x := (3 / (x + 2))\} \quad \Rightarrow \quad x := ((3 / x) + 2)
\]
Correct Representation of Syntax

- Other attempts fail, too (see the lecture notes)
- We need some mechanism which is not based on airlocks
- We borrow the representation approach of K
  - Term \( x := (3 / (x + 2)) \) represented as
    \[
    x \sim (\square + 2) \sim (3 / \square) \sim (x := \square) \sim \square
    \]
- Can be achieved using heating/cooling rules of the form
  
  \[
  (x := a) \sim c \quad \Rightarrow \quad a \sim (x := \square) \sim c
  
  (a_1 / a_2) \sim c \quad \Rightarrow \quad a_2 \sim (a_1 / \square) \sim c
  
  (a_1 + a_2) \sim c \quad \Rightarrow \quad a_1 \sim (\square + a_2) \sim c
  
  s \quad \Rightarrow \quad s \sim \square
  \]
CHAM Heating-Cooling Rules for IMP

\[ a_1 + a_2 \sim c \implies a_1 \sim \square + a_2 \sim c \]
\[ a_1 + a_2 \sim c \implies a_2 \sim a_1 + \square \sim c \]
\[ a_1 / a_2 \sim c \implies a_1 \sim \square / a_2 \sim c \]
\[ a_1 / a_2 \sim c \implies a_2 \sim a_1 / \square \sim c \]
\[ a_1 \leq a_2 \sim c \implies a_1 \sim \square \leq a_2 \sim c \]
\[ i_1 \leq a_2 \sim c \implies a_2 \sim i_1 \leq \square \sim c \]
\[ \text{not } b \sim c \implies b \sim \text{not } \square \sim c \]
\[ b_1 \text{ and } b_2 \sim c \implies b_1 \sim \square \text{ and } b_2 \sim c \]
\[ x := a \sim c \implies a \sim x := \square \sim c \]
\[ s_1 ; s_2 \sim c \implies s_1 \sim \square ; s_2 \sim c \]
\[ s \implies s \sim \square \]
\[ \text{if } b \text{ then } s_1 \text{ else } s_2 \sim c \implies b \sim \text{if } \square \text{ then } s_1 \text{ else } s_2 \sim c \]
Examples of Syntax Heating/Cooling

- The following is correct heating/cooling of syntax:

\[
\begin{align*}
\{x := 1 \ ; \ x := (3 / (x + 2))\} & \iff^* \\
\{x := 1 \leadsto (\square ; x := (3 / (x + 2))) \leadsto \square\} & \end{align*}
\]

- The following is incorrect heating/cooling of syntax:

\[
\begin{align*}
\{x := 1 \ ; \ x := (3 / (x + 2))\} & \iff^* \\
\{x := 1 \ ; \ (x \leadsto (\square + 2) \leadsto (3 / \square) \leadsto (x := \square) \leadsto \square)\} & \end{align*}
\]
CHAM Reaction Rules for IMP

\[
\begin{align*}
\{x \sim c\} \ {\{x \mapsto i \triangleright \sigma}\} &\rightarrow \{i \sim c\} \ {\{x \mapsto i \triangleright \sigma\}} \\
i_1 + i_2 \sim c &\rightarrow i_1 +_{\text{Int}} i_2 \sim c \\
i_1 / i_2 \sim c &\rightarrow i_1 /_{\text{Int}} i_2 \sim c \text{ when } i_2 \neq 0 \\
i_1 \leq i_2 \sim c &\rightarrow i_1 \leq_{\text{Int}} i_2 \sim c \\
\text{not true} \sim c &\rightarrow \text{false} \sim c \\
\text{not false} \sim c &\rightarrow \text{true} \sim c \\
\text{true and } b_2 \sim c &\rightarrow b_2 \sim c \\
\text{false and } b_2 \sim c &\rightarrow \text{false} \sim c \\
\{x := i \sim c\} \ {\{x \mapsto j \triangleright \sigma\}} &\rightarrow \{\text{skip} \sim c\} \ {\{x \mapsto i \triangleright \sigma\}} \\
\text{skip ; } s_2 \sim c &\rightarrow s_2 \sim c \\
\text{if true then } s_1 \text{ else } s_2 \sim c &\rightarrow s_1 \sim c \\
\text{if false then } s_1 \text{ else } s_2 \sim c &\rightarrow s_2 \sim c \\
\text{while } b \text{ do } s \sim c &\rightarrow \text{if } b \text{ then } (s ; \text{while } b \text{ do } s) \text{ else skip} \sim c \\
\text{var } x_l ; s &\rightarrow \{s\} \ {\{x_l \mapsto 0\}} \\
(x, x_l) \mapsto i &\rightarrow x \mapsto i \triangleright \{x_l \mapsto i\}
\end{align*}
\]
Sample CHAM Rewriting

\[
\begin{array}{l}
\{ \{ \mathsf{var} \ x, y ; \ x := 1 ; \ x := (3 / (x+2)) \} \} \rightarrow \\
\{ \{ x := 1 ; \ x := (3 / (x+2)) \} \} \quad \{ x, y \mapsto 0 \} \\
\{ \{ x := 1 ; \ x := (3 / (x+2)) \} \} \sim \square \\
\{ \{ x := 0 ; \ x := (3 / (x+2)) \} \} \quad \{ x \mapsto 0 \quad y \mapsto 0 \} \\
\{ \{ x := 1 \sim \square ; \ x := (3 / (x+2)) \sim \square \} \} \quad \{ x \mapsto 0 \quad y \mapsto 0 \} \\
\{ \{ \mathsf{skip} \sim \square ; \ x := (3 / (x+2)) \sim \square \} \} \\
\{ \{ \mathsf{skip} ; \ x := (3 / (x+2)) \sim \square \} \} \\
\{ \{ x := (3 / (x+2)) \sim \square \} \} \\
\{ \{ x \sim \square + 2 \sim 3 / \square \sim x := \square \sim \square \} \} \\
\{ \{ 1 \sim \square + 2 \sim 3 / \square \sim x := \square \sim \square \} \} \\
\{ \{ 1 + 2 \sim 3 / \square \sim x := \square \sim \square \} \} \\
\{ \{ 3 \sim 3 / \square \sim x := \square \sim \square \} \} \\
\{ \{ x := 1 \sim \square \} \} \\
\{ \{ \mathsf{skip} \sim \square \} \} \\
\{ \{ \mathsf{skip} \} \} \\
\{ \{ \} \} \\
\end{array}
\]
CHAM in Rewriting Logic

- CHAM rules cannot be used unchanged as rewrite rules
  - They need to only apply in solutions, not anywhere they match
- We represent each CHAM rule
  \[ m_1 \ m_2 \ \ldots \ \ m_k \rightarrow m'_1 \ m'_2 \ \ldots \ m'_l \]

into a rewrite logic rule

\[ \{\overline{m_1 \ m_2 \ \ldots \ m_k} \ Ms\} \rightarrow \{\overline{m'_1 \ m'_2 \ \ldots \ m'_l} \ Ms\} \]

where Ms is a fresh bag-of-molecule variable and the overlined molecules are the algebraic variants of the original ones, replacing in particular their meta-variables by variables.
CHAM of IMP in Maude

- See file
  - `imp-heating-cooling.maude`
- See file
  - `imp-semantics-cham.maude`
COMPARING CONVENTIONAL EXECUTABLE SEMANTICS

How good are the various semantic approaches?
We next discuss the conventional executable semantics approaches in depth, aiming at understanding their pros and cons.

Our approach is to extend each semantics of IMP with various common features (we call the resulting language IMP++):

- **Variable increment** – this will add side effects to expressions
- **Input/Output** – this will require changes in the configuration
- **Abrupt termination** – this requires explicit handling of control
- **Dynamic threads** – this requires handling concurrency and sharing
- **Local variables** – this requires handling environments

We will first treat each extension of IMP independently, i.e., we do not pro-actively take semantic decisions when defining a feature that will help the definition of other features later on. Then, we will put all features together into our IMP++ final language.
**IMP++ Variable Increment**

- **Syntax:**

\[ AExp ::= \ldots | ++ \text{Id} \]

- Variable increment is very common (C, C++, Java, etc.)
  - We only consider pre-increment (first increment, then return value)

- The problem with increment in some semantic approaches is that it adds side effects to expressions. Therefore, if one did not pro-actively account for that then one needs to change many existing and unrelated semantics rules, if not all.
IMP++ Variable Increment

Big-Step SOS

- Previous big-step SOS rules had the form:

\[
\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \quad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 / a_2, \sigma \rangle \Downarrow \langle i_1 / \text{Int} \ i_2 \rangle}, \text{ where } i_2 \neq 0
\]

- Big-step SOS is the most affected by side effects
  - Needs to change its sequents from \( \langle a, \sigma \rangle \Downarrow \langle i \rangle \) to \( \langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle \)
  - And all the existing rules accordingly, e.g.:

\[
\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1, \sigma_1 \rangle, \quad \langle a_2, \sigma_1 \rangle \Downarrow \langle i_2, \sigma_2 \rangle}{\langle a_1 / a_2, \sigma \rangle \Downarrow \langle i_1 / \text{Int}i_2, \sigma_2 \rangle}, \text{ where } i_2 \neq 0
\]
Recall IMP operators like / were non-deterministically strict. Here is an attempt to achieve that with big-step SOS

\[
\begin{align*}
\langle a_1, \sigma \rangle & \Downarrow \langle i_1, \sigma_1 \rangle, \langle a_2, \sigma_1 \rangle \Downarrow \langle i_2, \sigma_2 \rangle \\
\langle a_1 / a_2, \sigma \rangle & \Downarrow \langle i_1 / \text{Int} i_2, \sigma_2 \rangle
\end{align*}
\]

, where \( i_2 \neq 0 \)

\[
\begin{align*}
\langle a_1, \sigma_2 \rangle & \Downarrow \langle i_1, \sigma_1 \rangle, \langle a_2, \sigma \rangle \Downarrow \langle i_2, \sigma_2 \rangle \\
\langle a_1 / a_2, \sigma \rangle & \Downarrow \langle i_1 / \text{Int} i_2, \sigma_1 \rangle
\end{align*}
\]

, where \( i_2 \neq 0 \)

All we got is “non-deterministic choice” strictness: choose an order, then evaluate the arguments in that order.

- Some behaviors are thus lost, but this is relatively acceptable in practice since programmers should not rely on those behaviors in their programs anyway (the loss of behaviors when we add threads is going to be much worse)
We are now ready to add the big-step SOS for variable increment (this is easy now, the hard part was to get here):

\[
\langle \texttt{++ \(x\), } \sigma \rangle \downarrow \langle \sigma(\(x\)) + \texttt{Int 1, } \sigma[\left(\sigma(\(x\)) + \texttt{Int 1}\right)/\(x\)]\rangle
\]

Example:

- How many values can the following expression possibly evaluate to under the big-step SOS of IMP++ above (assume \(x\) is initially 1)?

\[
\texttt{++ \(x\) / ( ++ \(x\) / \(x\) )}
\]

- Can it evaluate to 0 or even be undefined under a fully non-deterministic evaluation strategy?
IMP++ Variable Increment
Small-Step SOS

- Previous small-step SOS rules had the form:

\[
\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle
\]

\[
\langle a_1 / a_2, \sigma \rangle \rightarrow \langle a'_1 / a_2, \sigma \rangle
\]

- Small-step SOS less affected than big-step SOS, but still requires many rule changes to account for the side effects:

\[
\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma_1 \rangle
\]

\[
\langle a_1 / a_2, \sigma \rangle \rightarrow \langle a'_1 / a_2, \sigma_1 \rangle
\]
IMP++ Variable Increment
Small-Step SOS

- Since small-step SOS “gets back to the top” at each step, it actually does not lose any non-deterministic behaviors
  - We get fully non-deterministic evaluation strategies for all the IMP constructs instead of “non-deterministic choice” ones
- The semantics of variable increment almost the same as in big-step SOS (indeed, variable increment is an atomic operation):

\[
\langle \text{++ } x, \sigma \rangle \rightarrow \langle \sigma(x) +_{\text{Int}} 1, \sigma[(\sigma(x) +_{\text{Int}} 1)/x] \rangle
\]
IMP++ Variable Increment

MSOS

- Previous MSOS rules had the form:

\[
\frac{a_1 \to a'_1}{a_1 / a_2 \to a'_1 / a_2}
\]

- All semantic changes are hidden within labels, which are implicitly propagated through the general MSOS mechanism.

- Consequently, the MSOS of IMP only needs the following rule to accommodate variable updates; nothing else changes!

\[
\text{++ } x \quad \begin{cases} \text{state}=\sigma, \text{state'}=\sigma[(\sigma(x)+\text{Int}1)/x], \ldots \end{cases} \quad \sigma(x) + \text{Int} 1
\]
IMP++ Variable Increment
Reduction Semantics with Eval. Contexts

- Previous RSEC evaluation contexts and rules had the form:

\[
\text{Context} \ := \ \Box \mid \text{Context} / AExp \mid AExp / \text{Context}
\]

\[
i_1 / i_2 \rightarrow i_1/_{\text{Int}}i_2, \quad \text{when } i_2 \neq 0
\]

\[
\langle c, \sigma \rangle [x] \rightarrow \langle c, \sigma \rangle [\sigma(x)]
\]

- Evaluation contexts, together with the characteristic rule of RSEC, allows for compact unconditional rules, mentioning only what is needed from the entire configuration

- Consequently, the RSED of IMP only needs the following rule to accommodate variable updates; nothing else changes!

\[
\langle c, \sigma \rangle [\mathbf{++} x] \rightarrow \langle c, \sigma[(\sigma(x) +_{\text{Int}} 1)/x] \rangle [\sigma(x) +_{\text{Int}} 1]
\]
IMP++ Variable Increment

CHAM

- Previous CHAM heating/cooling/reaction rules had the form:

  \[
  a_1 / a_2 \sim c \quad \Rightarrow \quad a_1 \sim \Box / a_2 \sim c \\
  a_1 / a_2 \sim c \quad \Rightarrow \quad a_2 \sim a_1 / \Box \sim c \\
  i_1 / i_2 \sim c \quad \Rightarrow \quad i_1/_{Int}i_2 \sim c \quad \text{when } i_2 \neq 0
  \]

  \[
  \{x \sim c\} \quad \{x \leftrightarrow i \triangleright \sigma\} \quad \Rightarrow \quad \{i \sim c\} \quad \{x \leftrightarrow i \triangleright \sigma\}
  \]

- Since the heating/cooling rules achieve the role of the evaluation contexts and since one can only mention the necessary molecules in each rule, one does not need to change anything either!

- All one needs to do is to add the following rule:

  \[
  \{++x \sim c\} \quad \{x \leftrightarrow i \triangleright \sigma\} \quad \Rightarrow \quad \{i +_{Int} 1 \sim c\} \quad \{x \leftrightarrow i +_{Int} 1 \triangleright \sigma\}
  \]
Where is the rest?

- We discussed the remaining features in class, using the whiteboard and colors.
- The lecture notes contain the complete information, even more than we discussed in class.