CONVENTIONAL SEMANTIC APPROACHES

Grigore Rosu

CS522 – Programming Language Semantics

Conventional Semantic Approaches

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

IMP

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

$$
\begin{array}{rcl}
Int & ::= & \text{the domain of (unbounded) integer numbers, with usual operations on them} \\
Bool & ::= & \text{the domain of Booleans} \\
Id & ::= & \text{standard identifiers} \\
AExp & ::= & Int \\
 & | & Id \\
 & | & AExp \;\texttt{+}\; AExp \\
 & | & AExp \;\texttt{/}\; AExp \\
BExp & ::= & Bool \\
 & | & AExp \;\texttt{<=}\; AExp \\
 & | & \texttt{!}\; BExp \\
 & | & BExp \;\texttt{\&\&}\; BExp \\
Block & ::= & \{\} \\
 & | & \{\; Stmt \;\} \\
Stmt & ::= & Block \\
 & | & Id \;\texttt{=}\; AExp \;\texttt{;} \\
 & | & Stmt\; Stmt \\
 & | & \texttt{if (}\; BExp \;\texttt{)}\; Block \;\texttt{else}\; Block \\
 & | & \texttt{while (}\; BExp \;\texttt{)}\; Block \\
Pgm & ::= & \texttt{int}\; \mathbf{List}\{Id\}\;\texttt{;}\; Stmt
\end{array}
$$

Suppose that, for demonstration purposes, we want "+" and "/" to be non-deterministically strict, "<=" to be sequentially strict, and "&&" to be short-circuited

Comma-separated list of identifiers

IMP Syntax in Maude


```
mod IMP-SYNTAX is including PL-INT + PL-BOOL + PL-ID .
--- AExp
  sort AExp .  subsorts Int Id < AExp .
  op _+_ : AExp AExp -> AExp [prec 33 gather (E e) format (d b o d)] .
  op _/_ : AExp AExp -> AExp [prec 31 gather (E e) format (d b o d)] .
--- BExp
  sort BExp .  subsort Bool < BExp .
  op _<=_ : AExp AExp -> 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 AExp -> 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
```

IMP State

- 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:

$$\sigma : Id \rightharpoonup Int$$

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

$$[Id \rightharpoonup Int]^{finite}$$

or

$$\mathbf{Map}\{Id \mapsto Int\}$$

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:

$$\sigma \quad = \quad x \mapsto 8, y \mapsto 0$$

- Typical state operations are lookup, update and initialization
- *Lookup*

$$\_(\_) : State \times Id \to Int$$

- *Update*

$$\_[\_/\_] : State \times Int \times Id \to State$$

- *Initialization*

$$\_ \mapsto \_ : \textbf{List}\{Id\} \times Int \to State$$

IMP State in Maude


```
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., $\langle a_1, \sigma \rangle$ $\langle a_1 + a_2, \sigma \rangle$ $\langle i_1 \rangle$ $\langle i_1 +_{Int} i_2 \rangle$ $\langle \sigma \rangle$
- *Sequent*: Pair of configurations, to be *derived* or *proved*
  - E.g., $\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle$ $\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i_1 +_{Int} i_2 \rangle$
- *Rule*: Tells how to derive a sequent from others
  - E.g.,

Read "evaluates to"

Premises

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \qquad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i_1 +_{Int} i_2 \rangle}$$

Conclusion

May omit line when no premises

Big-Step SOS of IMP - Arithmetic

$$\langle i, \sigma \rangle \Downarrow \langle i \rangle \qquad \text{(BigStep-Int)}$$

State lookup

$$\langle x, \sigma \rangle \Downarrow \langle \sigma(x) \rangle \quad \text{if } \sigma(x) \neq \bot \qquad \text{(BigStep-Lookup)}$$

Read: "provided that a_1 evaluates to i_1 in σ and a_2 evaluates to i_2 in σ, then $a_1 + a_2$ evaluates to the integer sum of i_1 and i_2 in σ

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \qquad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i_1 +_{Int} i_2 \rangle} \qquad \text{(BigStep-Add)}$$

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \qquad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 \,/\, a_2, \sigma \rangle \Downarrow \langle i_1 /_{Int} i_2 \rangle} \quad \text{if } i_2 \neq 0 \qquad \text{(BigStep-Div)}$$

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

Big-Step SOS of IMP - Boolean

$$\langle t, \sigma \rangle \Downarrow \langle t \rangle \qquad \text{(BigStep-Bool)}$$

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \qquad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 \texttt{ <= } a_2, \sigma \rangle \Downarrow \langle i_1 \leq_{Int} i_2 \rangle} \qquad \text{(BigStep-Leq)}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \langle \texttt{true} \rangle}{\langle \texttt{!}\, b, \sigma \rangle \Downarrow \langle \texttt{false} \rangle} \qquad \text{(BigStep-Not-True)}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \langle \texttt{false} \rangle}{\langle \texttt{!}\, b, \sigma \rangle \Downarrow \langle \texttt{true} \rangle} \qquad \text{(BigStep-Not-False)}$$

$$\frac{\langle b_1, \sigma \rangle \Downarrow \langle \texttt{false} \rangle}{\langle b_1 \texttt{ \&\& } b_2, \sigma \rangle \Downarrow \langle \texttt{false} \rangle} \qquad \text{(BigStep-And-False)}$$

$$\frac{\langle b_1, \sigma \rangle \Downarrow \langle \texttt{true} \rangle \qquad \langle b_2, \sigma \rangle \Downarrow \langle t \rangle}{\langle b_1 \texttt{ \&\& } b_2, \sigma \rangle \Downarrow \langle t \rangle} \qquad \text{(BigStep-And-True)}$$

Big-Step SOS of IMP - Statements

$$\langle \{\}, \sigma \rangle \Downarrow \langle \sigma \rangle \qquad \text{(BigStep-Empty-Block)}$$

$$\frac{\langle s, \sigma \rangle \Downarrow \langle \sigma' \rangle}{\langle \{\, s \,\}, \sigma \rangle \Downarrow \langle \sigma' \rangle} \qquad \text{(BigStep-Block)}$$

State update

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

$$\frac{\langle s_1, \sigma \rangle \Downarrow \langle \sigma_1 \rangle \qquad \langle s_2, \sigma_1 \rangle \Downarrow \langle \sigma_2 \rangle}{\langle s_1 \; s_2, \sigma \rangle \Downarrow \langle \sigma_2 \rangle} \qquad \text{(BigStep-Seq)}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \langle \texttt{true} \rangle \qquad \langle s_1, \sigma \rangle \Downarrow \langle \sigma_1 \rangle}{\langle \texttt{if } (b)\; s_1 \texttt{ else } s_2, \sigma \rangle \Downarrow \langle \sigma_1 \rangle} \qquad \text{(BigStep-If-True)}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \langle \texttt{false} \rangle \qquad \langle s_2, \sigma \rangle \Downarrow \langle \sigma_2 \rangle}{\langle \texttt{if } (b)\; s_1 \texttt{ else } s_2, \sigma \rangle \Downarrow \langle \sigma_2 \rangle} \qquad \text{(BigStep-If-False)}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \langle \texttt{false} \rangle}{\langle \texttt{while } (b)\; s, \sigma \rangle \Downarrow \langle \sigma \rangle} \qquad \text{(BigStep-While-False)}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \langle \texttt{true} \rangle \qquad \langle s \texttt{ while } (b)\; s, \sigma \rangle \Downarrow \langle \sigma' \rangle}{\langle \texttt{while } (b)\; s, \sigma \rangle \Downarrow \langle \sigma' \rangle} \qquad \text{(BigStep-While-True)}$$

Big-Step SOS of IMP - Programs

State initialization

$$\frac{\langle s, \boxed{xl \mapsto 0}\rangle \Downarrow \langle \sigma \rangle}{\langle \texttt{int } xl;\ s\rangle \Downarrow \langle \sigma \rangle} \qquad (\textsc{BigStep-Var})$$

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

$$\frac{\langle x, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 8\rangle \qquad \langle 2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 2\rangle}{\langle x/2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 4\rangle}$$

$$\frac{\langle x, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 8\rangle \qquad \langle 2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 4\rangle}{\langle x/2, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 2\rangle}$$

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):

$$\frac{\langle x, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 8\rangle \qquad \langle y, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle 0\rangle}{\langle x/y, (x \mapsto 8, y \mapsto 0)\rangle \Downarrow \langle ?\rangle}$$

Big-Step SOS Derivation

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 $\sigma(\mathbf{x})=8$ and $\sigma(\mathbf{y})=0$.

(BigStep-Lookup)

(BigStep-Int)

$$\dfrac{\dfrac{\cdot}{\langle \mathrm{x},\sigma\rangle \Downarrow \langle 8\rangle} \qquad \dfrac{\dfrac{\dfrac{\cdot}{\langle \mathrm{y},\sigma\rangle\Downarrow\langle 0\rangle}\qquad \dfrac{\cdot}{\langle \mathrm{x},\sigma\rangle\Downarrow\langle 8\rangle}}{\langle \mathrm{y} \,/\, \mathrm{x},\sigma\rangle \Downarrow \langle 0\rangle} \qquad \dfrac{\cdot}{\langle 2,\sigma\rangle\Downarrow\langle 2\rangle}}{\langle \mathrm{y} \,/\, \mathrm{x} + 2,\sigma\rangle\Downarrow\langle 2\rangle}}{\langle \mathrm{x} \,/\, (\mathrm{y} \,/\, \mathrm{x} + 2),\sigma\rangle \Downarrow\langle 4\rangle}$$

(BigStep-Add)

(BigStep-Div)

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* Γ, can be assigned a certain type τ. 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, Γ can be replaced by a list of untyped variables in our case. In general, however, a type environment Γ contains *typed variables*, that is, pairs "$x : \tau$".

Typing Arithmetic Expressions

$$xl \vdash i : int \qquad \text{(BigStepTypeSystem-Int)}$$

$$xl \vdash x : int \quad \text{if } x \in xl \qquad \text{(BigStepTypeSystem-Lookup)}$$

$$\frac{xl \vdash a_1 : int \qquad xl \vdash a_2 : int}{xl \vdash a_1 + a_2 : int} \qquad \text{(BigStepTypeSystem-Add)}$$

$$\frac{xl \vdash a_1 : int \qquad xl \vdash a_2 : int}{xl \vdash a_1 \,/\, a_2 : int} \qquad \text{(BigStepTypeSystem-Div)}$$

$$xl \vdash t : bool \quad \text{if } t \in \{\texttt{true}, \texttt{false}\} \qquad \text{(BigStepTypeSystem-Bool)}$$

Typing Boolean Expressions

$$\frac{xl \vdash a_1 : int \qquad xl \vdash a_2 : int}{xl \vdash a_1 \mathtt{<=} a_2 : bool} \qquad (\textsc{BigStepTypeSystem-Leq})$$

$$\frac{xl \vdash b : bool}{xl \vdash \mathtt{!}\, b : bool} \qquad (\textsc{BigStepTypeSystem-Not})$$

$$\frac{xl \vdash b_1 : bool \qquad xl \vdash b_2 : bool}{xl \vdash b_1 \mathtt{\&\&}\, b_2 : bool} \qquad (\textsc{BigStepTypeSystem-And})$$

$$xl \vdash \{\} : block \qquad (\textsc{BigStepTypeSystem-Empty-Block})$$

Typing Statements

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

$$\frac{xl \vdash s : \boxed{\tau}}{xl \vdash \{\, s \,\} : block} \quad \text{if } \tau \in \{block, stmt\} \qquad \text{(BigStepTypeSystem-Block)}$$

$$\frac{xl \vdash a : int}{xl \vdash x = a; : stmt} \quad \text{if } x \in xl \qquad \text{(BigStepTypeSystem-Asgn)}$$

$$\frac{xl \vdash s_1 : \boxed{\tau_1} \qquad xl \vdash s_2 : \boxed{\tau_2}}{xl \vdash s_1\ s_2 : stmt} \quad \text{if } \tau_1, \tau_2 \in \{block, stmt\} \qquad \text{(BigStepTypeSystem-Seq)}$$

$$\frac{xl \vdash b : bool \qquad xl \vdash s_1 : block \qquad xl \vdash s_2 : block}{xl \vdash \texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2 : stmt} \qquad \text{(BigStepTypeSystem-If)}$$

$$\frac{xl \vdash b : bool \qquad xl \vdash s : block}{xl \vdash \texttt{while}\ (b)\ s : stmt} \qquad \text{(BigStepTypeSystem-While)}$$

Typing Programs

$$\frac{xl \vdash s : \boxed{\tau}}{\vdash \texttt{int}\ xl\,;\ s : pgm} \quad \text{if } \tau \in \{block, stmt\} \qquad (\textsc{BigStepTypeSystem-Pgm})$$

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:

$$\frac{tree_1 \qquad \dfrac{tree_2 \qquad tree_3}{\texttt{n,s} \vdash (\texttt{while (!(n <= 0)) \{s = s + n; n = n + -1;\}}) : \mathit{stmt}}}{\dfrac{\texttt{n,s} \vdash (\texttt{n = 100; s = 0; while (!(n <= 0)) \{s = s + n; n = n + -1;\}}) : \mathit{stmt}}{\vdash (\texttt{int n,s; n = 100; s = 0; while (!(n <= 0)) \{s = s + n; n = n + -1;\}}) : \mathit{pgm}}}$$

where

Big-Step SOS Type Derivation

$$tree_1 = \left\{ \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{100} : int}}{\texttt{n,s} \vdash (\texttt{n = 100;}) : stmt} \qquad \dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{0} : int}}{\texttt{n,s} \vdash (\texttt{s = 0;}) : stmt}}{\texttt{n,s} \vdash (\texttt{n = 100; s = 0;}) : stmt} \right.$$

$$tree_2 = \left\{ \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : int} \qquad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{0} : int}}{\texttt{n,s} \vdash (\texttt{n <= 0}) : bool}}{\texttt{n,s} \vdash (\texttt{!(n <= 0)}) : bool} \right.$$

Big-Step SOS Type Derivation

$$tree_3 \quad = \quad \left\{ \dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{s} : \mathit{int}} \quad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : \mathit{int}}}{\texttt{n,s} \vdash (\texttt{s + n}) : \mathit{int}}}{\texttt{n,s} \vdash (\texttt{s = s + n;}) : \mathit{stmt}} \quad \dfrac{\dfrac{\dfrac{\cdot}{\texttt{n,s} \vdash \texttt{n} : \mathit{int}} \quad \dfrac{\cdot}{\texttt{n,s} \vdash \texttt{-1} : \mathit{int}}}{\texttt{n,s} \vdash (\texttt{n + -1}) : \mathit{int}}}{\texttt{n,s} \vdash (\texttt{n = n + -1;}) : \mathit{stmt}}}{\dfrac{\texttt{n,s} \vdash (\texttt{s = s + n; n = n + -1;}) : \mathit{stmt}}{\texttt{n,s} \vdash \{\ \texttt{s = s + n; n = n + -1;}\ \} : \mathit{block}}} \right.$$

Big-Step SOS in Rewriting Logic

- 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

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

a rewrite rule

$$\overline{C} \to \overline{R} \ \ \textbf{if} \ \ \overline{C_1} \to \overline{R_1} \ \wedge \ \overline{C_2} \to \overline{R_2} \ \wedge \ \ldots \ \wedge \ \overline{C_n} \to \overline{R_n} \ [\wedge \ \overline{condition}]$$

(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., $\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle$ $\langle a_1 + a_2, \sigma \rangle \to \langle a_1' + a_2, \sigma \rangle$
- *Rule*: Tells how to derive a sequent from others
  - E.g.,

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \to \langle a_1' + a_2, \sigma \rangle}$$

Small-Step SOS of IMP - Arithmetic

State lookup

$$\langle x, \sigma \rangle \to \langle \sigma(x), \sigma \rangle \quad \text{if } \sigma(x) \neq \bot \qquad (\textsc{SmallStep-Lookup})$$

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \to \langle a_1' + a_2, \sigma \rangle} \qquad (\textsc{SmallStep-Add-Arg1})$$

$$\frac{\langle a_2, \sigma \rangle \to \langle a_2', \sigma \rangle}{\langle a_1 + a_2, \sigma \rangle \to \langle a_1 + a_2', \sigma \rangle} \qquad (\textsc{SmallStep-Add-Arg2})$$

$$\langle i_1 + i_2, \sigma \rangle \to \langle i_1 +_{Int} i_2, \sigma \rangle \qquad (\textsc{SmallStep-Add})$$

+ is non-deterministic (its arguments can evaluate in any order, and interleaved

Small-Step SOS of IMP - Arithmetic

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \mathbin{/} a_2, \sigma\rangle \to \langle a_1' \mathbin{/} a_2, \sigma\rangle} \qquad (\textsc{SmallStep-Div-Arg1})$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 \mathbin{/} a_2, \sigma\rangle \to \langle a_1 \mathbin{/} a_2', \sigma\rangle} \qquad (\textsc{SmallStep-Div-Arg2})$$

$$\langle i_1 \mathbin{/} i_2, \sigma\rangle \to \langle i_1 \mathbin{/_{Int}} i_2, \sigma\rangle \quad \text{if } i_2 \neq 0 \qquad (\textsc{SmallStep-Div})$$

Side condition ensures rule will never apply when denominator is 0

/ is also non-deterministic

Small-Step SOS of IMP - Boolean

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 \texttt{ <= } a_2, \sigma \rangle \to \langle a_1' \texttt{ <= } a_2, \sigma \rangle} \quad (\textsc{SmallStep-Leq-Arg1})$$

$$\frac{\langle a_2, \sigma \rangle \to \langle a_2', \sigma \rangle}{\langle i_1 \texttt{ <= } a_2, \sigma \rangle \to \langle i_1 \texttt{ <= } a_2', \sigma \rangle} \quad (\textsc{SmallStep-Leq-Arg2})$$

$$\langle i_1 \texttt{ <= } i_2, \sigma \rangle \to \langle i_1 \leq_{Int} i_2, \sigma \rangle \quad (\textsc{SmallStep-Leq})$$

Ensures sequential strictness

Small-Step SOS of IMP - Boolean

$$\frac{\langle b,\sigma\rangle \rightarrow \langle b',\sigma\rangle}{\langle !\, b,\sigma\rangle \rightarrow \langle !\, b',\sigma\rangle} \qquad (\textsc{SmallStep-Not-Arg})$$

$$\langle !\, \texttt{true},\sigma\rangle \rightarrow \langle \texttt{false},\sigma\rangle \qquad (\textsc{SmallStep-Not-True})$$

$$\langle !\, \texttt{false},\sigma\rangle \rightarrow \langle \texttt{true},\sigma\rangle \qquad (\textsc{SmallStep-Not-False})$$

$$\frac{\langle b_1,\sigma\rangle \rightarrow \langle b_1',\sigma\rangle}{\langle b_1 \;\&\&\; b_2,\sigma\rangle \rightarrow \langle b_1' \;\&\&\; b_2,\sigma\rangle} \qquad (\textsc{SmallStep-And-Arg1})$$

$$\langle \texttt{false} \;\&\&\; b_2,\sigma\rangle \rightarrow \langle \texttt{false},\sigma\rangle \qquad (\textsc{SmallStep-And-False})$$

$$\langle \texttt{true} \;\&\&\; b_2,\sigma\rangle \rightarrow \langle b_2,\sigma\rangle \qquad (\textsc{SmallStep-And-True})$$

Short-circuit semantics

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 $\sigma(x)=1$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{\langle x,\sigma\rangle \rightarrow \langle 1,\sigma\rangle}\ (\textsc{SmallStep-Lookup})}{\langle y \,/\, x,\sigma\rangle \rightarrow \langle y \,/\, 1,\sigma\rangle}\ (\textsc{SmallStep-Div-Arg2})}{\langle x + (y \,/\, x),\sigma\rangle \rightarrow \langle x + (y \,/\, 1),\sigma\rangle}\ (\textsc{SmallStep-Add-Arg2})}{\langle (x + (y \,/\, x)) \mathrel{<=} x,\sigma\rangle \rightarrow \langle (x + (y \,/\, 1)) \mathrel{<=} x,\sigma\rangle}\ (\textsc{SmallStep-Leq-Arg1})$$

Small-Step SOS of IMP - Statements

$$\langle \{\, s \,\}, \sigma\rangle \to \langle s, \sigma\rangle \qquad \text{(SmallStep-Block)}$$

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma\rangle}{\langle x = a\,;, \sigma\rangle \to \langle x = a'\,;, \sigma\rangle} \qquad \text{(SmallStep-Asgn-Arg2)}$$

State update

$$\langle x = i\,;, \sigma\rangle \to \langle \{\}, \boxed{\sigma[i/x]}\rangle \quad \text{if } \sigma(x) \neq \bot \qquad \text{(SmallStep-Asgn)}$$

$$\frac{\langle s_1, \sigma\rangle \to \langle s_1', \sigma'\rangle}{\langle s_1\ s_2, \sigma\rangle \to \langle s_1'\ s_2, \sigma'\rangle} \qquad \text{(SmallStep-Seq-Arg1)}$$

$$\langle \{\}\ s_2, \sigma\rangle \to \langle s_2, \sigma\rangle \qquad \text{(SmallStep-Seq-Empty-Block)}$$

Small-Step SOS of IMP - Statements

$$\frac{\langle b, \sigma \rangle \to \langle b', \sigma \rangle}{\langle \texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2, \sigma \rangle \to \langle \texttt{if}\ (b')\ s_1\ \texttt{else}\ s_2, \sigma \rangle} \qquad (\textsc{SmallStep-If-Arg1})$$

$$\langle \texttt{if}\ (\texttt{true})\ s_1\ \texttt{else}\ s_2, \sigma \rangle \to \langle s_1, \sigma \rangle \qquad (\textsc{SmallStep-If-True})$$

$$\langle \texttt{if}\ (\texttt{false})\ s_1\ \texttt{else}\ s_2, \sigma \rangle \to \langle s_2, \sigma \rangle \qquad (\textsc{SmallStep-If-False})$$

$$\langle \texttt{while}\ (b)\ s, \sigma \rangle \to \langle \texttt{if}\ (b)\ \{\ s\ \texttt{while}\ (b)\ s\ \}\ \texttt{else}\ \{\}, \sigma \rangle \qquad (\textsc{SmallStep-While})$$

$$\langle \texttt{int}\ xl;\ s \rangle \to \langle s, xl \mapsto 0 \rangle \qquad (\textsc{SmallStep-Var})$$

State initialization

Small-Step SOS in Rewriting Logic

- Any small-step SOS can be associated a rewrite logic theory (or, equivalently, a Maude module)
- The idea is to associate to each small-step SOS rule

$$\frac{C_1 \to C'_1 \qquad C_2 \to C'_2 \qquad \ldots \qquad C_n \to C'_n}{C \to C'} \quad [\text{if } condition]$$

a rewrite rule

$$\circ\overline{C} \to \overline{C'} \quad \textbf{if} \quad \circ\,\overline{C_1} \to \overline{C'_1} \;\wedge\; \circ\overline{C_2} \to \overline{C'_2} \;\wedge\; \ldots \;\wedge\; \circ\overline{C_n} \to \overline{C'_n} \; [\wedge\; \overline{condition}]$$

(the circle means “ready for one step”)

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*

$$[\![\_]\!] : \ \mathtt{AExp} \rightarrow (State \rightharpoonup Int)$$

  - The denotation of a statement in IMP is a *partial function from states to states*

$$[\![\_]\!] : \ \mathtt{Stmt} \rightarrow (State \rightharpoonup State)$$

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*

$$[\![\_]\!] : \ \texttt{AExp} \to (State \rightharpoonup Int)$$

  - The denotation of a statement in IMP is a *partial function from states to states*

$$[\![\_]\!] : \ \texttt{Stmt} \to (State \rightharpoonup State)$$

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

Partial because some statements in some states may use undefined expressions, or may not terminate

Denotational Semantics - Terminology

$$[\![\_]\!] : \mathtt{AExp} \to (State \to Int)$$

Denotation (function)

Fragment of program

Mathematical domain

$$[\![\_]\!] : \mathtt{Stmt} \to (State \to State)$$

Denotational Semantics - Compositional

- Once the right mathematical domains are chosen, giving a denotational semantics to a language should be a straightforward and *compositional* process; e.g.

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

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

- 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* $(D, \sqsubseteq)$ is set D and binary rel. $\sqsubseteq$ which is
  - Reflexive: $x \sqsubseteq x$
  - Transitive: $x \sqsubseteq y \ \mathit{and} \ y \sqsubseteq z \ \mathit{imply} \ x \sqsubseteq z$
  - Anti-symmetric: $x \sqsubseteq y \ \mathit{and} \ y \sqsubseteq x \ \mathit{imply} \ x = y$
- Total order = partial order with $x \sqsubseteq y \ \mathit{or} \ y \sqsubseteq x$
- Important example: domains of *partial functions*

$$(A \rightharpoonup B, \leq)$$

$f \leq g$ iff g defined everywhere f is defined and $f(a) = g(a)$ whenever $f(a)$ is defined

(Least) Upper Bounds

- An *upper bound (u.b.)* of $X \subseteq D$ is any element $p \in D$ such that $x \sqsubseteq p$ for any $x \in X$
- The *least upper bound (l.u.b.)* of $X \subseteq D$, written $\sqcup X$, is an upper bound with $\sqcup X \sqsubseteq q$ for any u.b. q
  - When they exist, least upper bounds are *unique*
- The domains of partial functions, $(A \rightharpoonup B, \leq)$, 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

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \ldots$$

  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 $(A \rightharpoonup B, \leq, \bot)$ is a BCPO, where $\bot$ is the partial function undefined everywhere

Monotone and Continuous Functions

- $\mathcal{F} : (D, \sqsubseteq) \to (D', \sqsubseteq')$ *monotone* iff

  $x \sqsubseteq y$ implies $\mathcal{F}(x) \sqsubseteq' \mathcal{F}(y)$

- Monotone functions preserve chains:

  $\{d_n \mid n \in \mathbb{N}\}$ chain implies $\{\mathcal{F}(d_n) \mid n \in \mathbb{N}\}$ chain

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

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

- $Cont((D, \sqsubseteq, \perp), (D', \sqsubseteq', \perp'))$, the domain of continuous functions between two BCPOs, is itself a BCPO

Fixed-Point Theorem

- Let $(D, \sqsubseteq, \bot)$ be a BCPO and $\mathcal{F} : (D, \sqsubseteq, \bot) \to (D, \sqsubseteq, \bot)$ 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 $\mathit{fix}(\mathcal{F})$
- Proof sketch:

$$
\begin{aligned}
\mathcal{F}(\mathit{fix}(\mathcal{F})) &= \mathcal{F}(\textstyle\bigsqcup_{n\in\mathbb{N}} \mathcal{F}^n(\bot)) \\
&= \textstyle\bigsqcup_{n\in\mathbb{N}} \mathcal{F}^{n+1}(\bot) \\
&= \textstyle\bigsqcup_{n\in\mathbb{N}} \mathcal{F}^n(\bot) \\
&= \mathit{fix}(\mathcal{F}).
\end{aligned}
$$

Applications of Fixed-Point Theorem

- Consider the following "definition" of the factorial:

$$f(n) = \begin{cases} 1 & , \text{ if } n = 0 \\ n * f(n-1) & , \text{ if } n > 0 \end{cases}$$

- 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 $\mathbb{N} \to \mathbb{N}$ to itself

$$\mathcal{F}(g)(n) = \begin{cases} 1 & , \text{ if } n = 0 \\ n * g(n-1) & , \text{ if } n > 0 \text{ and } g(n-1) \text{ defined} \\ \text{undefined} & , \text{ if } n > 0 \text{ and } g(n-1) \text{ undefined} \end{cases}$$

Denotational Semantics of IMP
Arithmetic Expressions

$$[\![\_]\!] : AExp \rightarrow (State \rightarrow Int)$$

$$[\![i]\!]\sigma = i$$

$$[\![x]\!]\sigma = \sigma(x)$$

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

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

Denotational Semantics of IMP Boolean Expressions

$$[\![\_]\!] : BExp \rightarrow (State \rightharpoondown Bool)$$

$$[\![t]\!]\sigma = t$$

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

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

$$[\![b_1 \; \&\& \; b_2]\!]\sigma = \begin{cases} [\![b_2]\!]\sigma & \text{if} \quad [\![b_1]\!]\sigma = \texttt{true} \\ \texttt{false} & \text{if} \quad [\![b_1]\!]\sigma = \texttt{false} \\ \perp & \text{if} \quad [\![b_1]\!]\sigma = \perp \end{cases}$$

Denotational Semantics of IMP Statements (without loops)

$$[\![\_]\!] : \mathit{Stmt} \to (\mathit{State} \rightharpoonup \mathit{State})$$

$$[\![\{\}]\!]\sigma = \sigma$$

$$[\![\{\ s\ \}]\!]\sigma = [\![s]\!]\sigma$$

$$[\![x = a\,;]\!]\sigma = \begin{cases} \sigma[[\![a]\!]\sigma/x] & \text{if} \quad \sigma(x) \neq \bot \\ \bot & \text{if} \quad \sigma(x) = \bot \end{cases}$$

$$[\![s_1\ \ s_2]\!]\sigma = [\![s_2]\!][\![s_1]\!]\sigma$$

$$[\![\texttt{if}\,(b)\ s_1\ \texttt{else}\ s_2]\!]\sigma = \begin{cases} [\![s_1]\!]\sigma & \text{if} \quad [\![b]\!]\sigma = \texttt{true} \\ [\![s_2]\!]\sigma & \text{if} \quad [\![b]\!]\sigma = \texttt{false} \\ \bot & \text{if} \quad [\![b]\!]\sigma = \bot \end{cases}$$

Denotational Semantics of IMP While

- We first define a continuous function as follows

$$\mathcal{F} : (State \rightharpoonup State) \rightarrow (State \rightharpoonup State)$$

$$\mathcal{F}(\alpha)(\sigma) = \begin{cases} \alpha(\llbracket s \rrbracket \sigma) & \text{if} \quad \llbracket b \rrbracket \sigma = \texttt{true} \\ \sigma & \text{if} \quad \llbracket b \rrbracket \sigma = \texttt{false} \\ \perp & \text{if} \quad \llbracket b \rrbracket \sigma = \perp \end{cases}$$

- Then we define the denotational semantics of while

$$\llbracket \texttt{while}\ (b)\ s \rrbracket = \mathit{fix}(\mathcal{F})$$

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!

Denotational Semantics in Rewriting Logic

- Rewriting/equational logics do not have builtin functions and fixed-points, so they need 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:

$$Var_{CPO},\ CPO$$

subsorts:

$$Var_{CPO},\ Bool \quad < \quad CPO$$

operations:

$$\begin{aligned}
\mathtt{fun}_{CPO}\ _\texttt{->}\ _ &\ :\ \ Var_{CPO} \times CPO \to CPO \\
\mathtt{app}_{CPO}(_,_) &\ :\ \ CPO \times CPO \to CPO \\
\mathtt{fix}_{CPO}\ _ &\ :\ \ CPO \to CPO \\
\mathtt{if}_{CPO}(_,_,_) &\ :\ \ CPO \times CPO \times CPO \to CPO
\end{aligned}$$

equations:

$$\begin{aligned}
\mathtt{app}_{CPO}(\mathtt{fun}_{CPO}\ V\texttt{->}C, C') &\ =\ \ C[C'/V] \\
\mathtt{fix}_{CPO}\ \mathtt{fun}_{CPO}\ V\texttt{->}C &\ =\ \ C[(\mathtt{fix}_{CPO}\ \mathtt{fun}_{CPO}\ V\texttt{->}C)/V] \\
\mathtt{if}_{CPO}(\mathtt{true}, C, C') &\ =\ \ C \\
\mathtt{if}_{CPO}(\mathtt{false}, C, C') &\ =\ \ C'
\end{aligned}$$

Denotational Semantics of IMP in RL

Signature

sort:

Syntax // generic sort for syntax

subsorts:

$\mathit{AExp}, \mathit{BExp}, \mathit{Stmt}, \mathit{Pgm} < \mathit{Syntax}$ // syntactic categories fall under Syntax

$\mathit{Int}, \mathit{Bool}, \mathit{State} < \mathit{CPO}$ // basic domains are regarded as CPOs

operation:

$[\![\_]\!] : \mathit{Syntax} \to \mathit{CPO}$ // denotation of syntax

Denotational Semantics of IMP in RL
Arithmetic Expressions

$[\![I]\!] = \texttt{fun}_{CPO}\ \sigma \texttt{->}\ I$

$[\![X]\!] = \texttt{fun}_{CPO}\ \sigma \texttt{->}\ \sigma(X)$

$[\![A_1 + A_2]\!] = \texttt{fun}_{CPO}\ \sigma \texttt{->}\ \texttt{app}_{CPO}([\![A_1]\!], \sigma) +_{Int} \texttt{app}_{CPO}([\![A_2]\!], \sigma)$

$[\![A_1\ /\ A_2]\!] = \texttt{fun}_{CPO}\ \sigma \texttt{->}\ \texttt{if}_{CPO}(\texttt{app}_{CPO}([\![A_2]\!], \sigma) \neq_{Int} 0, \texttt{app}_{CPO}([\![A_1]\!], \sigma)\ /_{Int}\ \texttt{app}_{CPO}([\![A_2]\!], \sigma), \bot)$

Denotational Semantics of IMP in RL
Boolean Expressions

$[\![T]\!] = \texttt{fun}_{CPO}\ \sigma \texttt{->}\ T$

$[\![A_1 \texttt{<=} A_2]\!] = \texttt{fun}_{CPO}\ \sigma \texttt{->}\ \texttt{app}_{CPO}([\![A_1]\!], \sigma) \leq_{Int} \texttt{app}_{CPO}([\![A_2]\!], \sigma)$

$[\![\,!\ B]\!] = \texttt{fun}_{CPO}\ \sigma \texttt{->}\ \neg_{Bool}\texttt{app}_{CPO}([\![B]\!], \sigma)$

$[\![B_1\ \texttt{\&\&}\ B_2]\!] = \texttt{fun}_{CPO}\ \sigma \texttt{->}\ \texttt{if}_{CPO}(\texttt{app}_{CPO}([\![B_1]\!], \sigma), \texttt{app}_{CPO}([\![B_2]\!], \sigma), \texttt{false})$

Denotational Semantics of IMP in RL Statements and Programs

$[\![\{\}]\!] = \mathtt{fun}_{CPO}\ \sigma\texttt{->}\ \sigma$

$[\![\{\ S\ \}]\!] = \mathtt{fun}_{CPO}\ \sigma\texttt{->}\ \mathtt{app}_{CPO}([\![S]\!], \sigma)$

$[\![X = A\,;]\!] = \mathtt{fun}_{CPO}\ \sigma\texttt{->}\ \mathtt{app}_{CPO}(\mathtt{fun}_{CPO}\ arg\texttt{->}\ \mathtt{if}_{CPO}(\sigma(X) \neq_{Int} \bot, \sigma[arg/X], \bot), \mathtt{app}_{CPO}([\![A]\!], \sigma))$

$[\![S_1\ S_2]\!] = \mathtt{fun}_{CPO}\ \sigma\texttt{->}\ \mathtt{app}_{CPO}([\![S_2]\!], \mathtt{app}_{CPO}([\![S_1]\!], \sigma))$

$[\![\mathtt{if}\ (B)\ S_1\ \mathtt{else}\ S_2]\!] = \mathtt{fun}_{CPO}\ \sigma\texttt{->}\ \mathtt{if}_{CPO}(\mathtt{app}_{CPO}([\![B]\!], \sigma), \mathtt{app}_{CPO}([\![S_1]\!], \sigma), \mathtt{app}_{CPO}([\![S_2]\!], \sigma))$

$[\![\mathtt{while}\ (B)\ S]\!] = \mathtt{fix}_{CPO}\ \mathtt{fun}_{CPO}\ \alpha\texttt{->}\ \mathtt{fun}_{CPO}\ \sigma\texttt{->}\ \mathtt{if}_{CPO}(\mathtt{app}_{CPO}([\![B]\!], \sigma), \mathtt{app}_{CPO}(\alpha, \mathtt{app}_{CPO}([\![S]\!], \sigma)), \sigma)$

$[\![\mathtt{int}\ Xl;\ S]\!] = \mathtt{app}_{CPO}([\![S]\!], (Xl \mapsto 0))$

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

Philosophy of MSOS

- *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

MSOS Transitions

- An MSOS transition has the form

$$P \xrightarrow{\Delta} P'$$

  - *P* and *P'* are programs or tragments of program
  - Δ is a label describing the changes in the configuration components, defined as a record; primed fields stay for "after" the transition
- Example: $x := i \xrightarrow{\{\text{state}=\sigma,\ \text{state}'=\sigma[i/x],\ \ldots\}} \texttt{skip} \quad \text{if } \sigma(x) \neq \bot$
  - This rule can be automatically "desugared" into the SOS rule

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

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

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

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

$$x := i \xrightarrow{\{\mathsf{state}=\sigma,\ \mathsf{state'}=\sigma[i/x],\ \ldots\}} \mathtt{skip} \qquad \text{if } \sigma(x) \neq \bot$$

  - Write-only = fields which appear only primed

$$\mathtt{print}\ i \xrightarrow{\{\mathsf{output'}=i,\ \ldots\}} \mathtt{skip}$$

  - Read –only = fields which appear only unprimed

$$\frac{e_2 \xrightarrow{\{\mathsf{env}=\rho[v_1/x],\ldots\}} e_2'}{\mathtt{let}\ x = v_1\ \mathtt{in}\ e_2 \xrightarrow{\{\mathsf{env}=\rho,\ldots\}} \mathtt{let}\ x = v_1\ \mathtt{in}\ e_2'}$$

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:

$$\frac{s_1 \xrightarrow{\Delta} s_1'}{s_1\,;\,s_2 \xrightarrow{\Delta} s_1'\,;\,s_2}$$

  - Same Δ 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:

$$\frac{s_1 \rightarrow s_1'}{s_1\,;\,s_2 \rightarrow s_1'\,;\,s_2}$$

MSOS of IMP - Arithmetic

State lookup

$$x \xrightarrow{\{\text{state}=\sigma,\ldots\}} \boxed{\sigma(x)} \quad \text{if } \sigma(x) \neq \bot \qquad (\text{MSOS-Lookup})$$

$$\frac{a_1 \to a_1'}{a_1 + a_2 \to a_1' + a_2} \qquad (\text{MSOS-Add-Arg1})$$

$$\frac{a_2 \to a_2'}{a_1 + a_2 \to a_1 + a_2'} \qquad (\text{MSOS-Add-Arg2})$$

$$i_1 + i_2 \to i_1 +_{Int} i_2 \qquad (\text{MSOS-Add})$$

MSOS of IMP - Arithmetic

$$\frac{a_1 \to a_1'}{a_1 \,/\, a_2 \to a_1' \,/\, a_2} \qquad (\text{MSOS-}\textsc{Div-Arg1})$$

$$\frac{a_2 \to a_2'}{a_1 \,/\, a_2 \to a_1 \,/\, a_2'} \qquad (\text{MSOS-}\textsc{Div-Arg2})$$

$$i_1 \,/\, i_2 \to i_1 \,/_{Int}\, i_2 \quad \text{if } i_2 \neq 0 \qquad (\text{MSOS-}\textsc{Div})$$

MSOS of IMP - Boolean

$$\frac{a_1 \rightarrow a_1'}{a_1 \texttt{<=} a_2 \rightarrow a_1' \texttt{<=} a_2} \qquad (\text{MSOS-Leq-Arg1})$$

$$\frac{a_2 \rightarrow a_2'}{i_1 \texttt{<=} a_2 \rightarrow i_1 \texttt{<=} a_2'} \qquad (\text{MSOS-Leq-Arg2})$$

$$i_1 \texttt{<=} i_2 \rightarrow i_1 \leq_{Int} i_2 \qquad (\text{MSOS-Leq})$$

MSOS of IMP - Boolean

$$\frac{b \rightarrow b'}{\texttt{not}\, b \rightarrow \texttt{not}\, b'} \qquad (\text{MSOS-Not-Arg})$$

$$\texttt{not true} \rightarrow \texttt{false} \qquad (\text{MSOS-Not-True})$$

$$\texttt{not false} \rightarrow \texttt{true} \qquad (\text{MSOS-Not-False})$$

$$\frac{b_1 \rightarrow b_1'}{b_1 \,\texttt{and}\, b_2 \rightarrow b_1' \,\texttt{and}\, b_2} \qquad (\text{MSOS-And-Arg1})$$

$$\texttt{false and}\, b_2 \rightarrow \texttt{false} \qquad (\text{MSOS-And-False})$$

$$\texttt{true and}\, b_2 \rightarrow b_2 \qquad (\text{MSOS-And-True})$$

MSOS of IMP - Statements

$$\frac{a \to a'}{x := a \to x := a'} \qquad \text{(MSOS-ASGN-ARG2)}$$

$$x := i \xrightarrow{\{\text{state}=\sigma,\ \text{state}'=\sigma[i/x],\ \ldots\}} \texttt{skip} \quad \text{if } \sigma(x) \neq \bot \qquad \text{(MSOS-ASGN)}$$

$$\frac{s_1 \to s_1'}{s_1\ ;\ s_2 \to s_1'\ ;\ s_2} \qquad \text{(MSOS-SEQ-ARG1)}$$

$$\texttt{skip}\ ;\ s_2 \to s_2 \qquad \text{(MSOS-SEQ-SKIP)}$$

MSOS of IMP - Statements

$$\frac{b \to b'}{\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \to \texttt{if}\ b'\ \texttt{then}\ s_1\ \texttt{else}\ s_2} \quad (\text{MSOS-If-Arg1})$$

$$\texttt{if true then}\ s_1\ \texttt{else}\ s_2 \to s_1 \quad (\text{MSOS-If-True})$$

$$\texttt{if false then}\ s_1\ \texttt{else}\ s_2 \to s_2 \quad (\text{MSOS-If-False})$$

$$\texttt{while}\ b\ \texttt{do}\ s \to \texttt{if}\ b\ \texttt{then}\ (s\ ;\ \texttt{while}\ b\ \texttt{do}\ s)\ \texttt{else skip} \quad (\text{MSOS-While})$$

$$\texttt{var}\ xl\ ;\ s \xrightarrow{\{\text{state}' = xl \mapsto 0, \ldots\}} s \quad (\text{MSOS-Var})$$

MSOS in Rewriting Logic

- Any MSOS can be associated 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`

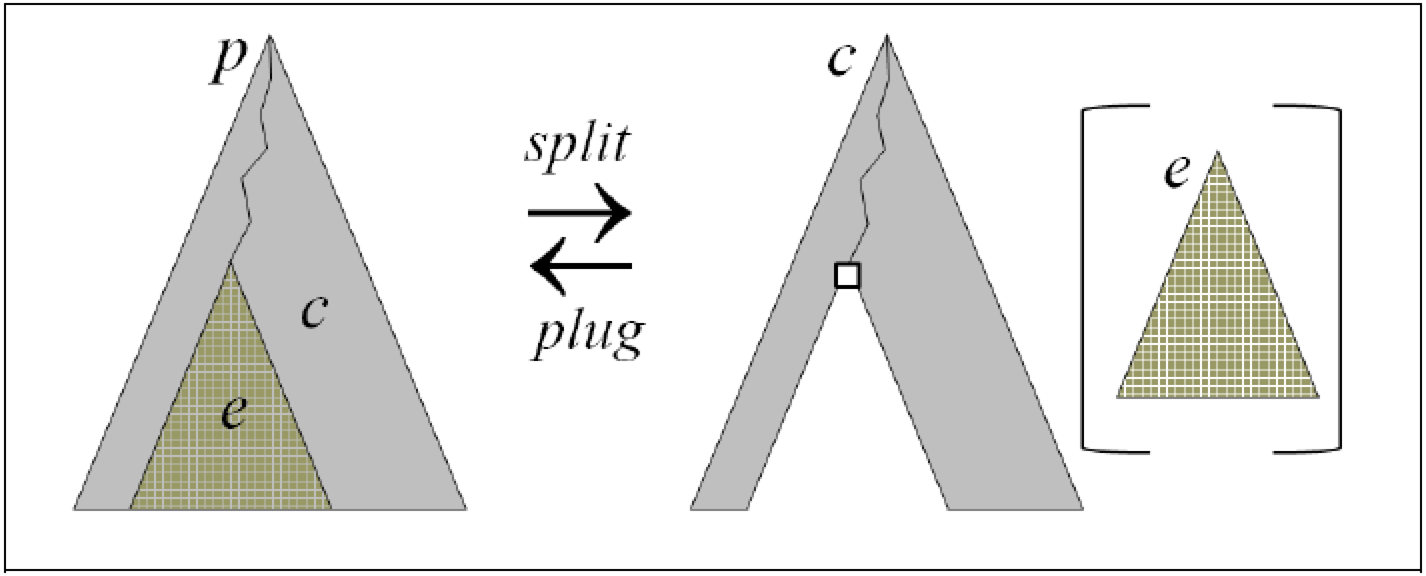
RSEC

Reduction Semantics with Evaluation Contexts (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:

$$
\begin{array}{lcl}
\mathit{Context} & ::= & \square \\
& | & \mathit{Context} \texttt{ <= } \mathit{AExp} \\
& | & \mathit{Int} \texttt{ <= } \mathit{Context} \\
& | & \mathit{Id} \texttt{ := } \mathit{Context} \\
& | & \mathit{Context} \texttt{ ; } \mathit{Stmt} \\
& | & \texttt{if } \mathit{Context} \texttt{ then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} \\
& | & \ldots
\end{array}
$$

Correct Evaluation Contexts

$\square$

3 <= $\square$

$\square$ <= 3

$\square$; x := 5

if $\square$ then s_1 else s_2

Wrong Evaluation Contexts

$\square$ `<=` $\square$

x `<=` 3

x `<=` $\square$

x `:=` 5 `;` $\square$

$\square$ `:=` 5

`if` x `<=` 7 `then` $\square$ `else` x `:=` 5

Splitting/Plugging of Syntax

$$7 \;=\; (\Box)[7]$$

$$\begin{aligned} 3 \texttt{<=} x \;&=\; (3 \texttt{<=} \Box)[x] \;=\; (\Box \texttt{<=} x)[3] \\ &=\; (\Box)[3 \texttt{<=} x] \end{aligned}$$

$$\begin{aligned} 3 \texttt{<=} (2 + x) + 7 \;&=\; (3 \texttt{<=} \Box + 7)[2 + x] \\ &=\; (\Box \texttt{<=} (2 + x) + 7)[3] \\ &=\; \ldots \end{aligned}$$

Characteristic Rule of RSEC

$$\frac{e \to e'}{c[e] \to c[e']}$$

- 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

IMP evaluation contexts syntax	IMP language syntax
$\begin{array}{rcl} \mathit{Context} & ::= & \square \\ & \mid & \mathit{Context} \texttt{ + } \mathit{AExp} \mid \mathit{AExp} \texttt{ + } \mathit{Context} \\ & \mid & \mathit{Context} \texttt{ / } \mathit{AExp} \mid \mathit{AExp} \texttt{ / } \mathit{Context} \\ & & \\ & \mid & \mathit{Context} \texttt{ <= } \mathit{AExp} \mid \mathit{Int} \texttt{ <= } \mathit{Cxt} \\ & \mid & \texttt{not } \mathit{Context} \\ & \mid & \mathit{Context} \texttt{ and } \mathit{BExp} \\ & \mid & \mathit{Id} \texttt{ := } \mathit{Context} \\ & \mid & \mathit{Context} \texttt{ ; } \mathit{Stmt} \\ & \mid & \texttt{if } \mathit{Context} \texttt{ then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} \end{array}$	$\begin{array}{rcl} \mathit{AExp} & ::= & \mathit{Int} \mid \mathit{Id} \mid \\ & \mid & \mathit{AExp} \texttt{ + } \mathit{AExp} \\ & \mid & \mathit{AExp} \texttt{ / } \mathit{AExp} \\ \mathit{BExp} & ::= & \mathit{Bool} \\ & \mid & \mathit{AExp} \texttt{ <= } \mathit{AExp} \\ & \mid & \texttt{not } \mathit{BExp} \\ & \mid & \mathit{BExp} \texttt{ and } \mathit{BExp} \\ \mathit{Stmt} & ::= & \mathit{Id} \texttt{ := } \mathit{AExp} \\ & \mid & \mathit{Stmt} \texttt{ ; } \mathit{Stmt} \\ & \mid & \texttt{if } \mathit{BExp} \texttt{ then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} \\ & \mid & \texttt{while } \mathit{BExp} \texttt{ do } \mathit{Stmt} \\ \mathit{Pgm} & ::= & \texttt{var } \mathrm{List}\{\mathit{Id}\} \texttt{ ; } \mathit{Stmt} \end{array}$

RSEC of IMP – Rules

$$Context ::= \ldots \mid \langle Context, State \rangle$$

$$\frac{e \to e'}{c[e] \to c[e']}$$

$$\begin{array}{rcll}
\langle c, \sigma\rangle[x] & \to & \langle c, \sigma\rangle[\sigma(x)] & \text{if } \sigma(x) \neq \bot \\
i_1 \texttt{+} i_2 & \to & i_1 +_{Int} i_2 & \\
i_1 \texttt{/} i_2 & \to & i_1 /_{Int} i_2 & \text{if } i_2 \neq 0 \\
i_1 \texttt{<=} i_2 & \to & i_1 \leq_{Int} i_2 & \\
\texttt{not true} & \to & \texttt{false} & \\
\texttt{not false} & \to & \texttt{true} & \\
\texttt{true and}\, b_2 & \to & b_2 & \\
\texttt{false and}\, b_2 & \to & \texttt{false} & \\
\langle c, \sigma\rangle[x \texttt{:=} i] & \to & \langle c, \sigma[i/x]\rangle[\texttt{skip}] & \text{if } \sigma(x) \neq \bot \\
\texttt{skip ;}\, s_2 & \to & s_2 & \\
\texttt{if true then}\, s_1\, \texttt{else}\, s_2 & \to & s_1 & \\
\texttt{if false then}\, s_1\, \texttt{else}\, s_2 & \to & s_2 & \\
\texttt{while}\, b\, \texttt{do}\, s & \to & \texttt{if}\, b\, \texttt{then}\, (s\, \texttt{;}\, \texttt{while}\, b\, \texttt{do}\, s)\, \texttt{else skip} & \\
\langle\, \texttt{var}\, xl\, \texttt{;}\, s \rangle & \to & \langle s, (xl \mapsto 0)\rangle &
\end{array}$$

RSEC Derivation

$$\langle \boxed{x := 1} \,;\, y := 2 \,;\, \texttt{if } x \texttt{<=} y \texttt{ then } x := 0 \texttt{ else } y := 0 \;,\; (x \mapsto 0, y \mapsto 0) \rangle$$

$$\rightarrow \langle \boxed{\texttt{skip} \,;\, y := 2} \,;\, \texttt{if } x \texttt{<=} y \texttt{ then } x := 0 \texttt{ else } y := 0 \;,\; (x \mapsto 1, y \mapsto 0) \rangle$$

$$\rightarrow \langle \boxed{y := 2} \,;\, \texttt{if } x \texttt{<=} y \texttt{ then } x := 0 \texttt{ else } y := 0 \;,\; (x \mapsto 1, y \mapsto 0) \rangle$$

$$\rightarrow \langle \texttt{skip} \,;\, \texttt{if } x \texttt{<=} y \texttt{ then } x := 0 \texttt{ else } y := 0 \;,\; (x \mapsto 1, y \mapsto 2) \rangle$$

$$\rightarrow \langle \texttt{if } \boxed{x} \texttt{<=} y \texttt{ then } x := 0 \texttt{ else } y := 0 \;,\; (x \mapsto 1, y \mapsto 2) \rangle$$

$$\rightarrow \langle \texttt{if } 1 \texttt{<=} \boxed{y} \texttt{ then } x := 0 \texttt{ else } y := 0 \;,\; (x \mapsto 1, y \mapsto 2) \rangle$$

$$\rightarrow \langle \texttt{if } \boxed{1 \texttt{<=} 2} \texttt{ then } x := 0 \texttt{ else } y := 0 \;,\; (x \mapsto 1, y \mapsto 2) \rangle$$

$$\rightarrow \langle \texttt{if true then } x := 0 \texttt{ else } y := 0 \;,\; (x \mapsto 1, y \mapsto 2) \rangle$$

$$\rightarrow \langle x := 0 \;,\; (x \mapsto 1, y \mapsto 2) \rangle$$

$$\rightarrow \langle \texttt{skip} \;,\; (x \mapsto 0, y \mapsto 2) \rangle$$

RSEC in Rewriting Logic

- 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

$$Context \quad ::= \quad \pi(N_1, \ldots, N_n, Context)$$

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

$$split(\pi(T_1, \ldots, T_n, T)) \to \pi(T_1, \ldots, T_n, C)[Syn] \ \textbf{if} \ split(T) \to C[Syn]$$

$$plug(\pi(T_1, \ldots, T_n, C)[Syn]) = \pi(T_1, \ldots, T_n, plug(C[Syn]))$$

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

$$split(Syn) \to \square[Syn]$$
$$plug(\square[Syn]) = Syn$$

IMP Examples:

- For productions

$$
\begin{array}{lcl}
Context & ::= & Context \texttt{ <= } AExp \\
& | & Int \texttt{ <= } Context \\
& | & Id \texttt{ := } Context
\end{array}
$$

we add the following rewrite logic rules and equations:

$$split(A_1 \texttt{ <= } A_2) \to (C \texttt{ <= } A_2)[Syn] \ \textbf{if} \ split(A_1) \to C[Syn]$$
$$plug((C \texttt{ <= } A_2)[Syn]) = plug(C[Syn]) \texttt{ <= } A_2$$

$$split(I_1 \texttt{ <= } A_2) \to (I_1 \texttt{ <= } C)[Syn] \ \textbf{if} \ split(A_2) \to C[Syn]$$
$$plug((I_1 \texttt{ <= } C)[Syn]) = I_1 \texttt{ <= } plug(C[Syn])$$

$$split(X \texttt{ := } A) \to (X \texttt{ := } C)[Syn] \ \textbf{if} \ split(A) \to C[Syn]$$
$$plug((X \texttt{ := } C)[Syn]) = X \texttt{ := } plug(C[Syn])$$

RSEC Reduction Rules in Rewriting Logic

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

$$\circ\bar{l}(T_1, \ldots, T_n) \to T \;\; \textbf{if} \;\; plug(\circ\bar{l}(split(T_1), \ldots, split(T_n))) \to T$$

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

$$plug(\bar{r}(Syn_1, \ldots, Syn_n)) = \bar{r}(plug(Syn_1), \ldots, plug(Syn_n))$$

// for each reduction semantics rule "$l(c_1[l_1], \ldots, c_n[l_n]) \to r(c'_1[r_1], \ldots, c'_{n'}[r_{n'}])$"
// add the following "semantic" rewrite rule:

$$\circ\bar{l}(\overline{c_1}[\overline{l_1}], \ldots, \overline{c_n}[\overline{l_n}]) \to \bar{r}(\overline{c'_1}[\overline{r_1}], \ldots, \overline{c'_n}[\overline{r_{n'}}])$$

IMP Examples

- One rule of first kind

$$\circ\, Cfg \to Cfg' \;\; \textbf{if} \;\; plug(\circ\, split(Cfg)) \to Cfg'$$

- No need for equations of second kind
- Characteristic rule of RSEC:

$$\circ\, C[Syn] \to C[Syn'] \;\; \textbf{if} \;\; C \neq \square \wedge \circ\, Syn \to Syn'$$

- The remaining rules are as natural as can be:

$$\circ\, I_1 \texttt{<=} I_2 \to I_1 \leq_{Int} I_2$$
$$\circ\, \texttt{skip ;}\; S_2 \to S_2$$
$$\circ\, \texttt{if true then}\, S_1\, \texttt{else}\, S_2 \to S_1$$

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

Airlock

$$
\begin{array}{rcl}
Molecule & ::= & Solution \\
 & | & Molecule \triangleleft Solution
\end{array}
$$

$$
Solution \quad ::= \quad \{\!|\, \mathbf{Bag}\,\{Molecule\}\,|\!\}
$$

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
  - *Heating* rules using $\rightharpoonup$: structurally rearrange solution
  - *Cooling* rules using $\rightharpoondown$: clean up solution after reactions
  - *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:

$$\{\!|m_1\ m_2\ \ldots\ m_k|\!\} \rightleftharpoons \{\!|m_1 \triangleleft \{\!|m_2\ \ldots\ m_k|\!\}|\!\}$$

CHAM Molecule Configuration for IMP

- A top-level solution containing two subsolution molecules
  - One for holding the syntax
  - Another for holding the state

$$\{\!|\ \{\!|\,\text{Syntax}\,|\!\}\quad \{\!|\,\text{State}\,|\!\}\ |\!\}$$

- Example:

$$\{\!|\{\!|x := (3\,/\,(x+2))|\!\}\ \{\!|x \mapsto 1\quad y \mapsto 0|\!\}|\!\}$$

Airlock can be Problematic

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

$$\begin{array}{rcl} x := a & \rightleftharpoons & a \lhd \{\!| x := \square |\!\} \\ a_1 + a_2 & \rightleftharpoons & a_1 \lhd \{\!| \square + a_2 |\!\} \\ a_1 + a_2 & \rightleftharpoons & a_2 \lhd \{\!| a_1 + \square |\!\} \end{array}$$

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

$$\{\!| x := (3 / (x + 2)) |\!\} \rightleftharpoons x := ((3 / x) + 2)$$

$$\begin{array}{rcl} \{\!| x := (3 / (x + 2)) |\!\} & \rightleftharpoons & \{\!| (3 / (x + 2)) \lhd \{\!| x := \square |\!\} |\!\} \\ & \rightleftharpoons & \{\!| (3 / (x + 2)) \;\; (x := \square) |\!\} \\ & \rightleftharpoons & \{\!| (x + 2) \;\; (3 / \square) \;\; (x := \square) |\!\} \\ & \rightleftharpoons & \{\!| x \;\; (\square + 2) \;\; (3 / \square) \;\; (x := \square) |\!\} \end{array}$$

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

$$x \curvearrowright (\square + 2) \curvearrowright (3 \,/\, \square) \curvearrowright (x := \square) \curvearrowright \square$$

- Can be achieved using heating/cooling rules of the form

$$\begin{array}{rcl}
(x := a) \curvearrowright c & \rightleftharpoons & a \curvearrowright (x := \square) \curvearrowright c \\
(a_1 \,/\, a_2) \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright (a_1 \,/\, \square) \curvearrowright c \\
(a_1 + a_2) \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright (\square + a_2) \curvearrowright c \\
s & \rightleftharpoons & s \curvearrowright \square
\end{array}$$

CHAM Heating-Cooling Rules for IMP

$$
\begin{array}{rcl}
a_1 \,\texttt{+}\, a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \,\texttt{+}\, a_2 \curvearrowright c \\
a_1 \,\texttt{+}\, a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright a_1 \,\texttt{+}\, \square \curvearrowright c \\
a_1 \,\texttt{/}\, a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \,\texttt{/}\, a_2 \curvearrowright c \\
a_1 \,\texttt{/}\, a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright a_1 \,\texttt{/}\, \square \curvearrowright c \\
a_1 \,\texttt{<=}\, a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \,\texttt{<=}\, a_2 \curvearrowright c \\
i_1 \,\texttt{<=}\, a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright i_1 \,\texttt{<=}\, \square \curvearrowright c \\
\texttt{not}\, b \curvearrowright c & \rightleftharpoons & b \curvearrowright \texttt{not}\, \square \curvearrowright c \\
b_1 \,\texttt{and}\, b_2 \curvearrowright c & \rightleftharpoons & b_1 \curvearrowright \square \,\texttt{and}\, b_2 \curvearrowright c \\
x \,\texttt{:=}\, a \curvearrowright c & \rightleftharpoons & a \curvearrowright x \,\texttt{:=}\, \square \curvearrowright c \\
s_1 \,\texttt{;}\, s_2 \curvearrowright c & \rightleftharpoons & s_1 \curvearrowright \square \,\texttt{;}\, s_2 \curvearrowright c \\
s & \rightleftharpoons & s \curvearrowright \square \\
\texttt{if}\, b \,\texttt{then}\, s_1 \,\texttt{else}\, s_2 \curvearrowright c & \rightleftharpoons & b \curvearrowright \texttt{if}\, \square \,\texttt{then}\, s_1 \,\texttt{else}\, s_2 \curvearrowright c
\end{array}
$$

Examples of Syntax Heating/Cooling

- The following is *correct* heating/cooling of syntax:

$$\{\!|x := 1 \; ; \; x := (3 \,/\, (x + 2))|\!\} \rightleftharpoons^{*}$$
$$\{\!|x := 1 \curvearrowright (\square \; ; \; x := (3 \,/\, (x + 2))) \curvearrowright \square|\!\}$$

- The following is *incorrect* heating/cooling of syntax:

$$\{\!|x := 1 \; ; \; x := (3 \,/\, (x + 2))|\!\} \rightleftharpoons^{*}$$
$$\{\!|x := 1 \; ; \; (x \curvearrowright (\square + 2) \curvearrowright (3 \,/\, \square) \curvearrowright (x := \square) \curvearrowright \square)|\!\}$$

CHAM Reaction Rules for IMP

$$
\begin{array}{rcl}
\{\!|x \curvearrowright c|\!\}\ \{\!|x \mapsto i \rhd \sigma|\!\} & \to & \{\!|i \curvearrowright c|\!\}\ \{\!|x \mapsto i \rhd \sigma|\!\} \\
i_1 \,\texttt{+}\, i_2 \curvearrowright c & \to & i_1 +_{Int} i_2 \curvearrowright c \\
i_1 \,\texttt{/}\, i_2 \curvearrowright c & \to & i_1 /_{Int} i_2 \curvearrowright c \quad \text{when } i_2 \neq 0 \\
i_1 \,\texttt{<=}\, i_2 \curvearrowright c & \to & i_1 \leq_{Int} i_2 \curvearrowright c \\
\texttt{not true} \curvearrowright c & \to & \texttt{false} \curvearrowright c \\
\texttt{not false} \curvearrowright c & \to & \texttt{true} \curvearrowright c \\
\texttt{true and}\, b_2 \curvearrowright c & \to & b_2 \curvearrowright c \\
\texttt{false and}\, b_2 \curvearrowright c & \to & \texttt{false} \curvearrowright c \\
\{\!|x \,\texttt{:=}\, i \curvearrowright c|\!\}\ \{\!|x \mapsto j \rhd \sigma|\!\} & \to & \{\!|\,\texttt{skip} \curvearrowright c|\!\}\ \{\!|x \mapsto i \rhd \sigma|\!\} \\
\texttt{skip ;}\, s_2 \curvearrowright c & \to & s_2 \curvearrowright c \\
\texttt{if true then}\, s_1\, \texttt{else}\, s_2 \curvearrowright c & \to & s_1 \curvearrowright c \\
\texttt{if false then}\, s_1\, \texttt{else}\, s_2 \curvearrowright c & \to & s_2 \curvearrowright c \\
\texttt{while}\, b\, \texttt{do}\, s \curvearrowright c & \to & \texttt{if}\, b\, \texttt{then}\, (s\ \texttt{;}\ \texttt{while}\, b\, \texttt{do}\, s)\, \texttt{else skip} \curvearrowright c \\
\texttt{var}\, xl\ \texttt{;}\ s & \to & \{\!|s|\!\}\ \{\!|xl \mapsto 0|\!\} \\
\\
(x, xl) \mapsto i & \rightharpoonup & x \mapsto i \rhd \{\!|xl \mapsto i|\!\}
\end{array}
$$

Sample CHAM Rewriting

$$
\begin{array}{rl}
\{\!|\, \mathtt{var}\, x, y \,;\, x := 1 \,;\, x := (3 \,/\, (x + 2)) \,|\!\} & \rightarrow \\
\{\!|\, \{\!|\, x := 1 \,;\, x := (3 \,/\, (x + 2)) \,|\!\} \quad \{\!|\, x, y \mapsto 0 \,|\!\} \,|\!\} & \rightharpoonup \\
\{\!|\, \{\!|\, x := 1 \,;\, x := (3 \,/\, (x + 2)) \curvearrowright \square \,|\!\} \quad \{\!|\, x, y \mapsto 0 \,|\!\} \,|\!\} & \rightharpoonup^{*} \\
\{\!|\, \{\!|\, x := 1 \,;\, x := (3 \,/\, (x + 2)) \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 0 \quad y \mapsto 0 \,|\!\} \,|\!\} & \rightharpoonup \\
\{\!|\, \{\!|\, x := 1 \curvearrowright \square \,;\, x := (3 \,/\, (x + 2)) \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 0 \quad y \mapsto 0 \,|\!\} \,|\!\} & \rightharpoonup \\
\{\!|\, \{\!|\, x := 1 \curvearrowright \square \,;\, x := (3 \,/\, (x + 2)) \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 0 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightarrow \\
\{\!|\, \{\!|\, \mathtt{skip} \curvearrowright \square \,;\, x := (3 \,/\, (x + 2)) \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightharpoondown \\
\{\!|\, \{\!|\, \mathtt{skip} \,;\, x := (3 \,/\, (x + 2)) \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightharpoondown \\
\{\!|\, \{\!|\, x := (3 \,/\, (x + 2)) \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightharpoonup^{*} \\
\{\!|\, \{\!|\, x \curvearrowright \square + 2 \curvearrowright 3 \,/\, \square \curvearrowright x := \square \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightarrow \\
\{\!|\, \{\!|\, 1 \curvearrowright \square + 2 \curvearrowright 3 \,/\, \square \curvearrowright x := \square \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightharpoondown \\
\{\!|\, \{\!|\, 1 + 2 \curvearrowright 3 \,/\, \square \curvearrowright x := \square \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightarrow \\
\{\!|\, \{\!|\, 3 \curvearrowright 3 \,/\, \square \curvearrowright x := \square \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightarrow^{*} \\
\{\!|\, \{\!|\, x := 1 \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightarrow \\
\{\!|\, \{\!|\, \mathtt{skip} \curvearrowright \square \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} & \rightarrow \\
\{\!|\, \{\!|\, \mathtt{skip} \,|\!\} \quad \{\!|\, x \mapsto 1 \triangleright \{\!|\, y \mapsto 0 \,|\!\} \,|\!\} \,|\!\} &
\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

$$\{\!|\overline{m_1} \ \overline{m_2} \ \ldots \ \overline{m_k} \ Ms|\!\} \rightarrow \{\!|\overline{m'_1} \ \overline{m'_2} \ \ldots \ \overline{m'_l} \ Ms|\!\}$$

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?

IMP++: A Language Design Experiment

- 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:

$$AExp \ ::= \ \ldots \mid \ {+}{+}\, Id$$

- 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:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \Downarrow \langle i_1 \ /_{Int} \ i_2\rangle}\ , \text{ where } i_2 \neq 0$$

- 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.:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle,\ \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_2\rangle}\ , \quad \text{where } i_2 \neq 0$$

IMP++ Variable Increment Big-Step SOS

- Recall IMP operators like / were non-deterministically strict. Here is an attempt to achieve that with big-step SOS

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle,\ \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1\ /\ a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_2\rangle}\ , \quad \text{where } i_2 \neq 0$$

$$\frac{\langle a_1, \sigma_2\rangle \Downarrow \langle i_1, \sigma_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1\ /\ a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_1\rangle}\ , \quad \text{where } i_2 \neq 0$$

  - 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)

IMP++ Variable Increment Big-Step SOS

- We are now ready to add the big-step SOS for variable increment (this is easy now, the hard part was to get here):

$$\langle \texttt{++}\, x, \sigma \rangle \Downarrow \langle \sigma(x) +_{Int} 1, \sigma[(\sigma(x) +_{Int} 1)/x] \rangle$$

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

$$\texttt{++}\, x \,/\, (\, \texttt{++}\, x \,/\, x\,)$$

  - 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:

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a_1', \sigma \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \rightarrow \langle a_1' \ / \ a_2, \sigma \rangle}$$

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

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a_1', \sigma_1 \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \rightarrow \langle a_1' \ / \ a_2, \sigma_1 \rangle}$$

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):

$$\langle +\!+\, x, \sigma\rangle \rightarrow \langle \sigma(x) +_{Int} 1, \sigma[(\sigma(x) +_{Int} 1)/x]\rangle$$

IMP++ Variable Increment MSOS

- Previous MSOS rules had the form:

$$\frac{a_1 \rightarrow a_1'}{a_1 \,/\, a_2 \rightarrow a_1' \,/\, a_2}$$

- 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!

$$\texttt{++}\, x \xrightarrow{\{\text{state}=\sigma,\text{state}'=\sigma[(\sigma(x)+_{Int}1)/x],\ldots\}} \sigma(x) +_{Int} 1$$

IMP++ Variable Increment Reduction Semantics with Eval. Contexts

- Previous RSEC evaluation contexts and rules had the form:

$$Context \ ::= \ \square \ \mid \ Context \ / \ AExp \ \mid \ AExp \ / \ Context$$

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

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

- 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!

$$\langle c, \sigma \rangle [\texttt{++}\, x] \rightarrow \langle c, \sigma[(\sigma(x) +_{Int} 1)/x] \rangle [\sigma(x) +_{Int} 1]$$

IMP++ Variable Increment CHAM

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

$$
\begin{array}{rcl}
a_1 \,/\, a_2 \curvearrowright c & \rightleftharpoons & a_1 \curvearrowright \square \,/\, a_2 \curvearrowright c \\
a_1 \,/\, a_2 \curvearrowright c & \rightleftharpoons & a_2 \curvearrowright a_1 \,/\, \square \curvearrowright c \\
i_1 \,/\, i_2 \curvearrowright c & \rightarrow & i_1 /_{Int} i_2 \curvearrowright c \quad \text{when } i_2 \neq 0 \\
\{\!|x \curvearrowright c|\!\} \; \{\!|x \mapsto i \rhd \sigma|\!\} & \rightarrow & \{\!|i \curvearrowright c|\!\} \; \{\!|x \mapsto i \rhd \sigma|\!\}
\end{array}
$$

- 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:

$$
\{\!|\texttt{++}x \curvearrowright c|\!\} \; \{\!|x \mapsto i \rhd \sigma|\!\} \rightarrow \{\!|i +_{Int} 1 \curvearrowright c|\!\} \; \{\!|x \mapsto i +_{Int} 1 \rhd \sigma|\!\}
$$

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.