

***Programming Language Concepts and Semantics
Part II***

***Lecture X
Modular Structural Operational Semantics***

In this lecture we will look at the modifications made to the Structural Operational Semantics introduced in Lecture V to make it modular

The lecture follows Chapters 2 and 3 of “Fundamental Concepts and Formal Semantics of Programming Languages, Lecture Notes by Peter Mosses. University of Aarhus”

Motivation for MSOS

The SOS description of programming languages are *not* modular.

Adding modularity is not as simple as Plotkin thought.

The way in which a construct is described depends on what other constructs are described in the language.

MSOS modifies SOS by allowing *reusable* descriptions of constructs.

Transition Systems: Basics

A **transition system** TS consists of a set $State$ of states S , and a binary relation $\rightarrow \subseteq (State \times State)$. We write $S \rightarrow S'$ to indicate there is a transition from S to S'

A **derivation** starting from state S_1 is a possibly empty, possibly infinite sequence of transitions $S_1 \rightarrow S_2 \rightarrow \dots$

A **computation** from S_1 is a maximal derivation starting from S_1 , i.e., a derivation that cannot be extended further.

We will only allow computations which :

- start from states that belong to a given set of **initial** states called *Init*
- end in states that belong to a given set of **final** states called *Final*

States that do not occur in any such computation are called **unreachable**

Non-final reachable states from which there are no further transitions are called **stuck**

Reachable states which are neither final nor stuck are **intermediate**

Labelled Transition Systems

A **labelled transition system** LTS consists of a set $State$ of states S , a set $Label$ of labels L , and a ternary relation $\rightarrow \subseteq (State \times Label \times State)$. We write $S \xrightarrow{L} S'$ to indicate there is a transition labelled L from S to S'

In traditional SOS, information such as the values of identifiers is represented by the states. Labels are used to represent concurrency information, such as transition of messages and synchronization signals.

To allow for modularity, MSOS uses labels to represent:

- **why** a transition occurs
- what has **changed** during such transition
- what has been **produced** by such transition

It also modifies the definition of computation to require labels of adjacent transitions to be **composable**.

An **info-labelled transition system** $ILTS$ is an LTS together with a binary relation $Composable \subseteq (Label \times Label)$ such that the labels of all adjacent transitions are composable.

Modelling Flow of Control

The abstract syntax tree of the entire program gives the initial control state for its computations (thus, *Init* will have a single element).

Nodes of the abstract syntax trees often get replaced by simpler trees during computation (ultimately to their resulting values).

But nodes can also get replaced by bigger trees (e.g., a call to a function gets replaced by the function definition).

Each intermediate state represents what remains to be computed plus the previous computed values.

The set *Final* contains all computed values.

Recall that *big-step* semantics have no intermediate values: computations have at most one transition from the initial to the final state.

Modelling Flow of Information

Information in the labels is classified according to how it flows through the derivations:

- information that remains *fixed*
- information that can be *changed*
- new information that is *produced*

Components of the processed information are referred to by *symbolic* indices. Thus, knowledge about position and number of other components is unnecessary.

Given a label L and an index i :

- $L.i$ refers to component i of the *fixed or changeable information at the start* of the transition.
- $L.i'$ refers to component i of the *changeable or produced information at the end* of the transition.

A component that is only referred to by $L.i$ is fixed information; if it is only referred to by $L.i'$ is produced information; and if it is referred to by both is changeable information.

Components of produced information are assumed to be sequences. This allows us to use concatenation to combine the labels in sequences of transitions. The empty sequence is $()$

Properties of Labels

Composability: $Composable(L_1, L_2)$ holds iff L_1, L_2 have the same set of indices and for each index i :

- if i indexes fixed information, then $L_1.i = L_2.i$
- if i indexes changeable information, then $L_1.i' = L_2.i$

Note that produced information does not affect labels

Unobservability: a label is unobservable if no information changes and no new information is produced.

This allows us to specify that a transition does not change or produce any information *without having to refer to its components*. This is crucial for modularity.

Operational Specifications

Specifying an operational model in MSOS involves defining:

- the sets of states (including *Init* and *Final*),
- the components of the labels (and their values), and
- the info-labelled transition relation.

This will be done using a *Modular SOS Definition Formalism (MSDF)* which provides appropriate notation to define such models.

The states are the abstract syntax tree of the program (constructs and sub-constructs) and the computed values.

The labels are defined by specifying sets of *extensible* records.

The transition relation for

- primitive constructs is specified directly (using big-step semantics).
- compound constructs, usually depends on transitions for one or more of their components, and it is done using conditional inference rules.

The entire set of transition rules is called an *inductive definition* of the transition relation.

Specification of Sets and Constructors

Identifiers with a *fixed notation* as sets and set constructors.

Boolean set consisting of the usual boolean truth-values: *true*, *false*. Unary: *not*. Binary: $\setminus /$, $/ \setminus$, $=$, $\setminus =$ (equality and inequality are defined on all sets).

Integer set of all integer values. Unary: $-$. Binary: $+$, $-$, $*$, *div*, *mod*, *rem*, $<$, $=<$, $>$, $>=$.

set+ set of non-empty, finite, non-nested sequences of elements in *set*.

set? set consisting of the empty sequence and the elements of *set* (i.e., sequences of length 0 or 1).

set* set of possibly-empty, finite, non-nested sequences of elements in *set*. It includes both *set+* and *set?* as subsets. Values: $(term_1, \dots, term_n)$. Binary: \wedge (concatenate).

(set)List set of possibly-empty, finite lists of elements in *set*. Values: $[term_1, \dots, term_n]$. Unary: *first*, *rest*, *front*, *last*, *length*. Binary: $+$ (concatenation), *nth*, *take*, *drop* (list, integer).

(set)Set set of possibly-empty, finite sets of elements in *set*. Values: $\{term_1, \dots, term_n\}$ ($n \geq 1$), *empty*. Unary: *size*. Binary: $+$ (ordinary union), $-$ (ordinary difference), $*$ (intersection), *in* (element, set), $<$, $=<$, $>$, $>=$.

(set1, set2)Map set of finite mappings from *set1* to *set2*. Values: $term_1 / -> term_2$, *void*. Unary: *dom* (domain), *ran* (range). Binary: $+$ (disjoint domains), $-$ (domain restriction), $/$ (first map overrides second), *lookup* (element, map).

Specification of Sets and Constructors (Cont.)

Reserved set identifiers for commonly-needed sets.

Id is an open set of identifiers.

Bindable is an open set of values that can be bound to identifiers.

Env is the set $(Id, Bindable)Map$.

Cell is an open set of storage cells (also known as 'locations' or 'addresses').

Storable is an open set of values that can be stored in (single) cells..

Store is the set $(Cell, Storable)Map$.

State, *Final*, *Label*, and *Unobs* are reserved as identifiers for the corresponding sets that define an info-labelled transition system.

Specification of Sets and Constructors (Cont.)

Unreserved set identifiers can be declared. Let *setid* range over all set identifiers.

setid declares a set identifier which starts with a capital letter, and may contain hyphens, letters and digits.

Example: `Cmd`.

setid ::= *opid* declares *opid* as a constructor constant of *setid* which starts with a lowercase letter, and may contain hyphens and letters.

Example: `Cmd ::= skip`.

setid ::= *opid*(*set1*, ..., *setn*) declares *opid* as an n-ary constructor function for *setid* values with arguments in *set1*, ..., *setn*.

Example: `Cmd ::= cond-nz(Exp, Cmd)`.

setid ::= *set* declares *setid* to include *set* as a subset.

Example: `Exp ::= Boolean`.

setid = *set* declares *setid* as an abbreviation for *set*.

Example: `Env = (Id, Bindable)Map`.

Label = {*opid1*:*set1*, ..., *opidn*:*setn*, ...} declares each *opidi* (may be followed by a prime ') as a label index for a component with values in *seti*. No need primed and unprimed sets.

Example: `Label = {env:Env, store, store':S, out':Value*, ...}`.

Variable identifiers

Variable identifiers (used in transition rules) have to be declared **globally** as ranging over particular sets. Let *varid* range over variable identifiers.

varid:*set* declares that *varid*, (with *pr* without digits and/or primes), are restricted to range over *set*. It is formed from uppercase letters only. The variable declarations `X:Label` and `U:Unobs` are implicit in MSDF.

Example: `E:Exp` declares the variables `E`, `E'`, `E23`, etc.

The declarations of a set identifier, its constructors, and its subsets can be specified **together**. The declaration of a *varid* can be combined with declarations for the set over which it ranges.

Example: `E: Exp ::= null | Boolean | app(Op, Arg)`.

The combined declarations in a complete MSDF specification define all the elements of all the declared sets.

The *elements* of a declared set are all values constructed with its declared constructors (and relevant fixed interpretations) and values of all included sets. Sets that have no constructors or subsets are thus defined to be empty.

The *order* of declarations in MSDF is insignificant, and repetitions of declarations are ignored.

Specification of Transitions

Specified by *simple and conditional rules* with the label in the middle of the arrow.

Let *term* range over terms denoting individual values, and *form* over formulae.

$term1 \dashrightarrow term2 \dashrightarrow term3$ is an unconditional rule for transitions from state *term1* with label *term2* to state *term3*.

Example: `print(V):Cmd --{out'=V,---}-> skip.`

$form1, \dots, formn \dashdashdash term1 \dashrightarrow term2 \dashrightarrow term3$ is a transition rule with conditions *form1*, ..., *formn*. Usually the '-----' separating the condition(s) from the conclusion is written on a line by itself:

```
      E --{...}-> E'
-----
print(E):Cmd --{...}-> print(E')
```

Definition .1 A rule is structural when every term on the left of a transition in a condition is a subterm of the term on the left of the transition in the conclusion.

Rules in MSOS are mostly structural, and the terms on the left of transitions in the conditions are mostly variables.

Conditions of rules can be transitions or side-conditions.

Formulae

Formulae *form* are used as conditions and interpreted as follows:

$term1 \dashrightarrow term2 \dashrightarrow term3$ holds when the indicated transition can be derived.

Example: `E --{...}-> E'.`

$term1 \dashdashdash term3$ abbreviates $term1 \dashrightarrow U \dashrightarrow term3$, i.e., the label is unobservable, and identified by the variable *U*.

Example: `E --> VT.`

$term1 = term2$ holds when the values of *term1* and *term2* are defined and equal.

Example: `lookup(CL,S) = V.`

def *term* holds when the value of *term* is defined.

Example: `def lookup(CL,S).`

not *form* holds when *form* does not hold, and vice versa.

Example: `not def lookup(CL,S).`

term abbreviates $term = true$ (when *term* is boolean-valued).

Example: `N \= 0` (where '`\=`' is defined as a boolean-valued operation for all sets, and '0' is the zero integer).

Formulae other than transitions are known as *side-conditions*.

Terms

`var` gives the value assigned to `var`.

Example: E1'.

`opid` gives the value of a constructor or a fixed constant.

Example: skip.

`opid(term1, ..., termn)` gives the value (if defined) of applying `opid` to the values of `term1, ..., termn`. If the value of any `term` is undefined, so is that of the application.

Example: cond-nz(E,C).

`opid` may be also a variable ranging over a specified set of operations. **Example:** O(V1,V2).

`{opid1=term1, ..., opidn=termn, term}` gives the label where the components indexed by the `opids` have the values of the corresponding `terms`. All other components are determined by the label given by the value of the last `term`.

Example: {store=S, store'=S', U}.

The last `term` may also be `'...'` (equivalent to X), or `'---'` (equivalent to U); in these cases, `n` may be 0.

Example: {...}.

Example: {store=S, store'=S', ---}.

`term.opid` gives the `opid` component of the label given by the `term`.

Example: X.store'.

`term:set` restricts the values of `term` to be elements of `set`.

Example: N:Exp.

Modular Prototyping

The aim of the following slides is to show how to write in MSDF so that it can be *automatically translated* to Prolog.

The Prolog translation helps *validate* our language descriptions: to check the MSOS is complete and specifies the *intended semantics*.

Prolog is a *natural choice* for implementing MSOS because:

- Rules can be easily transcribed as Prolog rules
- Mapping abstract to concrete syntax is easy through DCGs
- Prolog computations closely correspond to derivations in MSOS
- Prolog modules can be combined without explicit references
- Names of vars and constructors in Prolog are not declared

MSOS modules are organised to maximise their reusability. All relevant files are available at

<http://www.bricks.dk/řdm/MSOS>

Individual Constructs: Abstract Syntax

Each construct is divided into three parts describing its *abstract syntax*, *static semantics*, *dynamic semantics*.

The abstract syntax of each construct is stored under directory ABS. Example, that of `Cmd` is stored at `Cons/Cmd/ABS`.

The *number of sets of constructs is kept low* by distinguishing by avoiding inessential distinctions. Example: we consider booleans and numerical expressions together in a more general set of expressions (no problem with values which might be boolean or numeric depending on context).

Basic constructs and variants are *named systematically*: by adding a suffix to a fixed root that has no suffix. Variants with alternative number of arguments keep the original name. Example `cond-nz(E,C)` and `cond-nz(E,C1,C2)`.

Subset inclusion corresponds to a construct. We will use the name of the included set to form the module name.

Example: `E: Exp ::= Var | Integer | Boolean | ...` will mean that the abstract syntax part of integers used as expressions is stored at `Cons/Exp/Integer/ABS`.

Variants should be avoided when they can be derived from other constructs. Example: the command `cond-nz(E,C)` can be derived as `cond-nz(E,C,skip)`.

Individual Constructs: Static Semantics

The static semantics of a set of constructs specifies the relevant sets of states *State* (and the final states *Final*).

The static semantics of each construct is stored under directory CHK. Example, for `Cmd` is stored at `Cons/Cmd/CHK`. It specifies the inclusion of `Cmd` in *State* and that of the `void` type in *Final*.

Recall: only *big-step* semantics, no intermediate states.

Each set of computed values in the dynamic semantics has a corresponding set of types in the static semantics (with type `void` representing the empty set of values).

A construct is *well-formed* when all its components are well-formed and their types consistent (you will see more about this later).

The *order of conditions* in rules is irrelevant mathematically but important for Prolog (conditions are checked left-to-right)

Each set of constructs requires the specification of a separate transition relation.

Individual Constructs: Dynamic Semantics

The dynamic semantics of a set of constructs specifies the relevant sets of states *State* (and the final states *Final*).

The static semantics of each construct is stored under directory RUN. Example, for *Cmd* is stored at *Cons/Cmd/RUN*. It specifies the inclusion of *Cmd* in *State* and the inclusion of its set of computed values in *Final*.

In contrast to the static semantics, the dynamic semantics of a construct might not include that of all its components. *Why?*

Generally involves *small-step* computations where intermediate states are needed (this means several rules per constructor).

Overlapping rules give rise to *non-deterministic* computations. Their *order* can affect their Prolog execution. *Is this the case for non-overlapping?*

Example: abstract syntax

Consider the following abstract syntax (subset of some language):

C: Cmd ::= skip | seq(Cmd,Cmd) | cond-nz(Exp,Cmd)

E: Exp ::= Integer | Var | assign-seq(Var,Exp) |
app(Op,Arg)

VAR Var ::= Id

O Op ::= + | - | * | div | mod | ord | < | =< |
> | >= | = | \=

ARG Arg ::= Exp | tup-seq(Exp,Exp)

The modular abstract syntax would be:

see Cmd/skip, Cmd/seq, Cmd/con-nz

see Exp/Integer, Exp/Var, Exp/assign-seq,
Exp/app

see Var/Id

see Op/plus, Op/minus, Op/times, Op/div,
Op/mod, Op/ord, Op/lt, Op/leq,
Op/gt, Op/geq, Op/eq, Op/neq

see Arg/Exp, Arg/tup-seq

Example: Files

With (some of) the files containing the following info:

In Cmd/ABS

```
C:Cmd
```

In Cmd/CHK

```
State ::= Cmd
Final ::= void
```

In Cmd/RUN

```
State ::= Cmd
Final ::= skip
```

In Cmd/seq/ABS

```
Cmd ::= seq(Cmd,Cmd)
```

In Cmd/seq/RUN

```
      C1 --{...}-> C1'
-----
seq(C1,C2):Cmd --{...}-> seq(C1',C2)

seq(skip,C2):Cmd --->C2
```

Note that files such as Const/Integer/RUN will not need to be created since the constructs are treated as computed values in abstract syntax trees.

Example: Prolog translation

The modular abstract syntax for modules M_1, \dots, M_n is grouped together in prolog by the command

```
:- ['M1', ..., 'Mn'].
```

which causes the modules to be loaded.

The abstract syntax of, for example, Cmd/seq/ABS is translated as:

```
cmd(seq(Cmd1,Cmd2)):- cmd(Cmd1), cmd(Cmd2).
```

The different Prolog rules for Cmd are grouped together by the command:

```
:- multifile cmd/1.
```

which allows rules of a predicate to appear in separate files.

Primes such as E' and E'_2 are transcribed as $E_$ and E_2 .

An overview of the abstract syntax can be obtained with listing/1:
listing[cmd/1,exp/1,op/1].

The dynamic semantics of, for example, Cmd/seq/RUN is translated as:

```
cmd(seq(C1,C2)) ---X--> cmd(seq(C_1,C2)) :-
      C1 ---X--> C_1.
```

```
cmd(seq(skip,C2)) ---U--> C2 :- unobs(U).
```

```
:- ensure_loaded('Cmd/skip/...').
```

Example: Prolog translation (Cont)

The following transition rule (which would appear in `Var/Id/RUN`):

$$\frac{U = \{\text{env:Env}, \dots\}, \text{Env}(I) = V}{I:\text{Var} \text{ --U--> } V}$$

would be translated into Prolog as:

```
id_var(I) ---U--> =
  member(U, [env=Env|_]),
  lookup(I, Env, V),
  unobs(U).
```