Describing Semantics of Programing Languages using the Ott Tool

Alwin W. Zulehner

JKU Linz

November 6, 2013

Table of contents

Introduction to Operational semantics

- Describe meaning of program mathematically
- Interpret a program as sequence of computational steps
- Return value of functional program
- Can be used to reason about programming languages
- Used first time to define semantics of Algol 68
- Two different levels of detail:
  - Big-step operational semantics
  - Small-step operational semantics

Big Step Operational Semantics

- Also known as natural semantics
- Can be seen as relations over configurations
- Not possible to define concurrency, ...
- Example: Arithmetic expressions

$$\frac{< a_1, \sigma > \rightarrow i_1 \qquad < a_2, \sigma > \rightarrow i_2}{< a_1 + a_2, \sigma > \rightarrow i_1 +_{int} i_2}$$

$$\frac{< a_1, \sigma > \rightarrow i_1 \qquad < a_2, \sigma > \rightarrow i_2}{< a_1 / a_2, \sigma > \rightarrow i_1 /_{int} i_2} \quad \text{if} i_2 \neq 0$$

Small Step Operational Semantics

- Idea: describing behaviour of parts of program
- Set of inference rules
- More rules than in big step operational semantics
- Possibility to describe concurrency and order of computations
- Example: Arithmetic expressions

$$\frac{< a_1, \sigma > \rightarrow < a_1', \sigma >}{< a_1 / a_2, \sigma > \rightarrow < a_1' / a_2, \sigma >}$$

$$\frac{< a_2, \sigma > \rightarrow < a_2', \sigma >}{< a_1 / a_2, \sigma > \rightarrow < a_1 / a_2', \sigma >}$$

$$< i_1 / i_2, \sigma > \rightarrow < i_1' /_{int} i_2', \sigma > \qquad \text{if} i_2 \neq 0$$

CBV λ: A Simple Programming Language

- A simple Call-By-Value λ Calculus
- A term can either be
  - A variable: x
  - A λ expression: $\lambda x.t$
  - An application: $t_1 t_2$
- Not possible to store values, just replace variables in terms through λ expressions
- CBV λ is independent of types

Table of contents

Introduction to the Ott tool

- Developed by Francesco Zappa Nardelli, Peter Sewell, and Scott Owens
- Tool for writing definitions of programming languages and calculi
- Written in OCaml
- Define syntax and semantics in ASCII notation
- Pretty Print with LaTeX
- Produce definitions for reasoning back ends
  - Coq
  - Isabelle
  - HOL
- Generate types for OCaml as starting point for building an interpreter

Defining the Abstract Syntax of CBV λ

```
grammar
t :: 't_' ::=
   | test x                    ::   :: Var
   | \ x . t                   ::   :: Lam
   | t t'                      ::   :: App
   | ( t )                     :: S :: Paren
   | { t / x } t'              :: M :: Tsub

v :: 'v_' ::=
   | \ x . t                   ::   :: Lam

subrules
   v <:: t
```

Reduction Rules of CBV λ

```
defn
t1 --> t2 :: ::reduce:: '' by

------------------------------- :: ax_app
(\x.t12) v2 --> {v2/x}t12

t1 --> t1'
---------------- :: ctx_app_fun
t1 t --> t1' t

t1 --> t1'
---------------- :: ctx_app_arg
v t1 --> v t1'
```

Printing Definitions with LaTeX

- Generate LaTeXoutput with

```
bin/ott -i input.ott -o output.tex
```

- Insert comments with $\{\{com\ldots\}\}$
- Override default typesetting with $\{\{tex\ldots\}\}$
- Override header and footer with $embed\{\{tex-wrap-pre\ldots\}\}$ and $embed\{\{tex-wrap-post\ldots\}\}$
- Additional command line parameters
  - -tex_show_meta <true>
  - -tex_show_categories <false>
  - -tex_colour <true>
  - -tex_wrap <true>
  - -tex_name_prefix <string>

Example I

```
metavar termvar, x ::=
  {{ tex \mathit{[[termvar]]} }}

grammar
t :: 't_' ::=                                {{ com term     }}
  | x                  ::   ::  Var          {{ com variable }}
  | \ x . t            ::   ::  Lam          {{ com lambda   }}
  | t t'               ::   ::  App          {{ com app      }}
  | ( t )              ::  S::  Paren
  | { t / x } t'       ::  M::  Tsub

v :: 'v_' ::=                                {{ com value    }}
  | \ x . t            ::   ::  Lam          {{ com lambda   }}

terminals :: 'terminals_' ::=
```

Example II

```
   | \          ::   :: lambda   {{ tex \lambda }}
   | -->        ::   :: red      {{ tex \longrightarrow }}

subrules
   v <:: t

defns
Jop :: '' ::=

 defn
 t1 --> t2 :: ::reduce::''
                {{ com [[t1]] reduces to [[t2]]}} by

   ------------------------------------  :: ax_app
```

Example III

$$(\backslash x.t12)\ v2 \longrightarrow \{v2/x\}t12$$

$$\frac{t1 \longrightarrow t1'}{t1\ t \longrightarrow t1'\ t}\ ::\ ctx\_app\_fun$$

$$\frac{t1 \longrightarrow t1'}{v\ t1 \longrightarrow v\ t1'}\ ::\ ctx\_app\_arg$$

Output

$termvar,\ x$

t	$::=$		term
	$\mid$	x	variable
	$\mid$	$\lambda x.t$	lambda
	$\mid$	$t\ t'$	app
	$\mid$	(t) S	

v	$::=$		value
	$\mid$	$\lambda x.t$	lambda

$\boxed{t_1 \longrightarrow t_2}$ t_1 reduces to t_2

$$\frac{}{(\lambda x.t_{12})\ v_2 \longrightarrow \{v_2/x\}t_{12}}\ \text{AX\_APP}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t \longrightarrow t_1'\ t}\ \text{CTX\_APP\_FUN}$$

$$\frac{t_1 \longrightarrow t_1'}{v\ t_1 \longrightarrow v\ t_1'}\ \text{CTX\_APP\_ARG}$$

```
Definition rules:          3 good    0 bad
Definition rule clauses: 5 good    0 bad
```

Reasoning about Semantics using Isabelle/HOL

- Generate Isabelle/HOL output with

```
bin/ott -i input.ott -o output.thy
```

- Define types of meta variables
- Define bindings of variables
- Define substitutions
- Isabelle syntax support: *{{isasyn [[t]]\<bullet>[[t']] }} {{ isaprec 50 }}*
- Additional command line parameters
  - -isabelle_primrec <true>
  - -isabelle_inductive <true>
  - -isa_syntax <false>
  - -isa_generate_lemmas <false>

Example I

```
metavar termvar, x ::=
{{ isa string}} {{ coq nat}}
{{ hol string}} {{ coq-equality }}

grammar
t :: 't_' ::=
   | x                  ::   :: Var
   | \ x . t            ::   :: Lam        (+ bind x in t +)
   | t t'               ::   :: App
   | ( t )              :: S:: Paren       {{ icho [[t]]  }}
   | { t / x } t'       :: M:: Tsub
    {{ icho (tsubst_t [[t]] [[x]] [[t']])}}

v :: 'v_' ::=
   | \ x . t            ::   :: Lam
```

Example II

```
subrules
  v <:: t

substitutions
  single t x :: tsubst

defns
Jop :: '' ::=

 defn
 t1 --> t2 :: ::reduce::'' by

    ------------------------------ :: ax_app
    (\x.t12) v2 --> {v2/x}t12
```

Example III

$$\frac{t1 \longrightarrow t1'}{t1\ t \longrightarrow t1'\ t} :: \text{ctx\_app\_fun}$$

$$\frac{t1 \longrightarrow t1'}{v\ t1 \longrightarrow v\ t1'} :: \text{ctx\_app\_arg}$$

Output: out.v I

```
(* generated by Ott 0.21.2 from: tests/test10.4.ott
   *)

Require Import Arith.
Require Import Bool.
Require Import List.


Definition termvar := nat.
Lemma eq_termvar: forall (x y : termvar), {x = y} +
   {x <> y}.
Proof.
  decide equality; auto with ott_coq_equality arith.
Defined.
Hint Resolve eq_termvar : ott_coq_equality.
```

Output: out.v II

```
Inductive t : Set :=
  | t_Var (x:termvar)
  | t_Lam (x:termvar) (t5:t)
  | t_App (t5:t) (t':t).

(** subrules *)
Definition is_v_of_t (t_6:t) : Prop :=
   match t_6 with
    | (t_Var x) => False
    | (t_Lam x t5) => (True)
    | (t_App t5 t') => False
end.

(** library functions *)
```

Output: out.v III

```
Fixpoint list_mem A (eq:forall a b:A,{a=b}+{a<>b}) (
    x:A) (l:list A) {struct l} : bool :=
  match l with
  | nil => false
  | cons h t => if eq h x then true else list_mem A
      eq x t
end.
Implicit Arguments list_mem.


(** substitutions *)
Fixpoint tsubst_t (t_6:t) (x5:termvar) (t__7:t) {
    struct t__7} : t :=
  match t__7 with
  | (t_Var x) => (if eq_termvar x x5 then t_6 else (
      t_Var x))
```

Output: out.v IV

```
  | (t_Lam x t5) => t_Lam x (if list_mem eq_termvar
      x5 (cons x nil) then t5 else (tsubst_t t_6 x5
      t5))
  | (t_App t5 t') => t_App (tsubst_t t_6 x5 t5) (
      tsubst_t t_6 x5 t')
end.

(** definitions *)

(* defns Jop *)
Inductive reduce : t -> t -> Prop :=    (* defn
    reduce *)
 | ax_app : forall (x:termvar) (t12 v2:t),
     is_v_of_t v2 ->
     reduce (t_App (t_Lam x t12) v2) (tsubst_t
        v2 x t12 )
```

Output: out.v V

```
| ctx_app_fun : forall (t1 t_5 t1':t),
    reduce t1 t1' ->
    reduce (t_App t1 t_5) (t_App t1' t_5)
| ctx_app_arg : forall (v5 t1 t1':t),
    is_v_of_t v5 ->
    reduce t1 t1' ->
    reduce (t_App v5 t1) (t_App v5 t1').
```

Executeable Code with OCaml

- Generate OCaml Types

```
bin/ott -i input.ott -o output.ml
```

- No concrete variables in semantic definitions
- Generate OCaml types of abstract syntax, auxiliary functions, subrules and substitutions
- *NO* implementation of semantic rules
- Just a parser and no interpreter
- Additional command line parameters
  - -parse <string> ":nontermroot: term"
  - -fast_parse <false>
  - -signal_parse_errors <false>
  - -ocaml_include_terminals <false> (experimental!)

Example I

```
% all
metavar termvar, x ::=    {{ com  term variable }}
{{ isa string}} {{ coq nat}} {{ hol string}} {{ coq-
    equality }}
{{ ocaml int}} {{ lex alphanum}} {{ tex \mathit{[[
    termvar]]} }}

grammar
t :: 't_' ::=                                    {{ com
    term     }}
  | x                  ::   :: Var
    {{ com variable}}
  | \ x . t            ::   :: Lam (+ bind x in t +)
    {{ com lambda  }}
  | t t'               ::   :: App
```

Example II

```
    {{ com app      }}
  | ( t )           :: S:: Paren
    {{ icho [[t]]   }}
  | { t / x } t'    :: M:: Tsub
        {{ icho (tsubst_t [[t]] [[x]] [[t']]) }}

v :: 'v_' ::=   {{ com value    }}
  | \ x . t       ::   :: Lam {{ com lambda   }}

terminals :: 'terminals_' ::=
  | \             ::   :: lambda  {{ tex \lambda }}
  | -->           ::   :: red     {{ tex \
    longrightarrow }}

subrules
  v <:: t
```

Example III

```
substitutions
  single t x :: tsubst

defns
Jop :: '' ::=

 defn
 t1 --> t2 :: ::reduce::'' {{ com [[t1]] reduces to
    [[t2]]}} by

    ------------------------------------ :: ax_app
    (\x.t12) v2 -->  {v2/x}t12

    t1 --> t1'
```

Example IV

$$\frac{}{\texttt{t1 t} \longrightarrow \texttt{t1' t}} :: \texttt{ctx\_app\_fun}$$

$$\frac{\texttt{t1} \longrightarrow \texttt{t1'}}{\texttt{v t1} \longrightarrow \texttt{v t1'}} :: \texttt{ctx\_app\_arg}$$

Output: out.ml I

```
(* generated by Ott 0.21.2 from: tests/test10.7.ott
   *)

type termvar = int (* term variable *)

type
t =  (* term *)
   T_Var of termvar (* variable *)
 | T_Lam of termvar * t (* lambda *)
 | T_App of t * t (* app *)


(** subrules *)
let is_v_of_t (t5:t) : bool =
  match t5 with
```

Output: out.ml II

```
  | (T_Var x) -> false
  | (T_Lam (x,t)) -> (true)
  | (T_App (t,t')) -> false

(** substitutions *)
let rec tsubst_t (t5:t) (x5:termvar) (t_6:t) : t =
  match t_6 with
  | (T_Var x) -> (if x=x5 then t5 else (T_Var x))
  | (T_Lam (x,t)) -> T_Lam (x,(if List.mem x5 ([x])
     then t else (tsubst_t t5 x5 t)))
  | (T_App (t,t')) -> T_App ((tsubst_t t5 x5 t),(
     tsubst_t t5 x5 t'))

(** definitions *)
```

Code Generation from Isabelle/HOL

- Developed at:
  - University of Cambridge(Larry Paulson)
  - Technische Universitaet Muenchen(Tobias Nipkow)
  - Universite Paris-Sud (Makarius Wenzel)
- Isabelle can be used to prove properties
- Code generation for SML, OCaml, Haskell and Scala

Example: Amortised Queue I

```
theory AQueue
imports Main
begin
datatype 'a queue = AQueue "'a list" "'a list"

definition empty :: "'a queue" where
  "empty = AQueue [] []"

primrec enqueue :: "'a => 'a queue => 'a queue"
    where
  "enqueue x (AQueue xs ys) = AQueue (x # xs) ys"

fun dequeue :: "'a queue => 'a option \<times> 'a
    queue" where
  "dequeue (AQueue [] []) = (None, AQueue [] [])"
```

Example: Amortised Queue II

```
  | "dequeue (AQueue xs (y # ys)) = (Some y, AQueue
    xs ys)"
  | "dequeue (AQueue xs []) = (case rev xs of y # ys
     => (Some y, AQueue [] ys))"


export_code empty dequeue enqueue in Haskell
module_name Example file "examples/"

end
```

Example: Generated Code I

```
structure Example : sig
  val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
  val rev : 'a list -> 'a list
  datatype 'a queue = AQueue of 'a list * 'a list
  val empty : 'a queue
  val dequeue : 'a queue -> 'a option * 'a queue
  val enqueue : 'a -> 'a queue -> 'a queue
end = struct

fun fold f (x :: xs) s = fold f xs (f x s)
  | fold f [] s = s;

fun rev xs = fold (fn a => fn b => a :: b) xs [];

datatype 'a queue = AQueue of 'a list * 'a list;
```

Example: Generated Code II

```
val empty : 'a queue = AQueue ([], []);

fun dequeue (AQueue ([], [])) = (NONE, AQueue ([],
    []))
  | dequeue (AQueue (xs, y :: ys)) = (SOME y, AQueue
      (xs, ys))
  | dequeue (AQueue (v :: va, [])) =
    let
      val y :: ys = rev (v :: va);
    in
      (SOME y, AQueue ([], ys))
    end;

fun enqueue x (AQueue (xs, ys)) = AQueue (x :: xs,
    ys);
```

Example: Generated Code III

```
end; (*struct Example*)
```

Semantics in Maude MSOS Tool

- Extension of *Full Maude*
- Define modular operational Semantics
- Code execution, but no code export
- Easy and intuitive syntax

Example: Simple Programming Language I

```
(msos SIMPLE-LANGUAGE is
 Exp .
 Id .

 Env = (Id, Int) Map .

 Exp ::= let Id = Int in Exp end
       | Exp sum Exp
       | Int
       | Id .

 Label = { env : Env, ... } .

                                  Exp1 -{...}-> Exp'1
 -- ------------------------------------------------------------------
```

Example: Simple Programming Language II

$$\frac{(\mathsf{Exp1\ sum\ Exp2}) : \mathsf{Exp} \ -\{\ldots\}-> \mathsf{Exp'1\ sum\ Exp2} \quad . \qquad \mathsf{Exp2} \ -\{\ldots\}-> \mathsf{Exp'2}}{(\mathsf{Int\ sum\ Exp2}) : \mathsf{Exp} \ -\{\ldots\}-> \mathsf{Int\ sum\ Exp'2} \quad .}$$

$$\frac{\mathsf{Int3} := \mathsf{Int1} + \mathsf{Int2}}{(\mathsf{Int1\ sum\ Int2}) : \mathsf{Exp} \longrightarrow \mathsf{Int3} \quad .}$$

$$\mathsf{Env'} := (\mathsf{Id} \ |-> \mathsf{Int}) \ / \ \mathsf{Env}, \quad \mathsf{Exp} \ -\{\mathsf{env} = \mathsf{Env'}, \ \ldots\}-> \mathsf{Exp'}$$

Example: Simple Programming Language III

```
--
    --------------------------------------------------------------------

(let Id = Int in Exp end) : Exp -{env = Env,
    ...}->
  (let Id = Int in Exp' end) .

(let Id = Int in Int' end) : Exp --> Int' .

    Int := lookup (Id, Env)
-- ---------------------------------
Id : Exp -{env = Env}-> Int .

sosm)
```

Conclusions

- Ott is convenient for generating informal and formal definitions
- Parts of functionality can be left out
- Approximately 10 demos on developers website
- Difficult for beginners because there is hardly any documentation
- Compatibility issues with the current version of Isabelle
- Maude MSOS is a standalone system, but code cannot be exported

Live Demo

Ressources

- Ott manual
  http://www.cl.cam.ac.uk/ pes20/ott/ott_manual_0.21.2.html
- P. Sewell et al. *Ott: Effective tool support for the working semanticist*. doi:10.1017/S0956796809990293
- Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications - A Formal Introduction*, John Wiley & Sons, 1992.
- Fabricio Chalub and Christiano Braga. *Maude MSOS Tool*
  http://www2.ic.uff.br/ cbraga/losd/maude-msos-tool/mmt-manual.pdf
- Florian Haftmann with contributions from Lukas Bulwahn. *Code generation from Isabelle/HOL theories*
  http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/codeg
- Tobias Nipkov. *Programming and Proving in Isabelle/HOL*
  http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2013/doc/p
  prove.pdf