

Theory exercises

Program verification with types and logic (NWI-IMC060)

Week 4

1. Consider the following syntax of a simple programming language:

$$e ::= x \mid \mathbf{true} \mid \mathbf{false} \mid (e_1, e_2) \\ \mid \mathbf{let} (x, y) = e_1 \mathbf{in} e_2 \mid \mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2$$

This language has a construct (e_1, e_2) that creates a tuple and $\mathbf{let} (x, y) = e_1 \mathbf{in} e_2$ that “matches” on a tuple. For example, one can write the following program:

$$\mathbf{let} (x, y) = (\mathbf{true}, \mathbf{false}) \mathbf{in} (\mathbf{if} x \mathbf{then} y \mathbf{else} \mathbf{false})$$

After the let-expression, the variable x will be bound to \mathbf{true} and y will be bound to \mathbf{false} . The result of the program is thus \mathbf{false} .

The values of the language are the expressions described by the following grammar:

$$v ::= \mathbf{true} \mid \mathbf{false} \mid (v_1, v_2)$$

- (a) The big-step semantics $e \Downarrow v$ for the fragment of the language without tuples is formally defined as follows:

$$\frac{}{\mathbf{true} \Downarrow \mathbf{true}} \quad \frac{}{\mathbf{false} \Downarrow \mathbf{false}} \\ \frac{e_0 \Downarrow \mathbf{true} \quad e_1 \Downarrow v_1}{\mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2 \Downarrow v_1} \quad \frac{e_0 \Downarrow \mathbf{false} \quad e_2 \Downarrow v_2}{\mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2 \Downarrow v_2}$$

Complete the big-step semantics to also include the rules for (e_1, e_2) and $\mathbf{let} (x, y) = e_1 \mathbf{in} e_2$. Explain how your semantics corresponds to the intuitive semantics. You may assume you have the usual substitution function $\mathbf{subst} x v e$ at your disposal.

- (b) The small-step semantics (without evaluation contexts) $e_1 \Rightarrow e_2$ for the fragment of the language without tuples is formally defined as follows:

$$\frac{e_0 \Rightarrow e'_0}{\mathbf{if} e_0 \mathbf{then} e_1 \mathbf{else} e_2 \Rightarrow \mathbf{if} e'_0 \mathbf{then} e_1 \mathbf{else} e_2} \\ \frac{}{\mathbf{if} \mathbf{true} \mathbf{then} e_1 \mathbf{else} e_2 \Rightarrow e_1} \quad \frac{}{\mathbf{if} \mathbf{false} \mathbf{then} e_1 \mathbf{else} e_2 \Rightarrow e_2}$$

Complete the small-step semantics to also include the rules for (e_1, e_2) and $\mathbf{let} (x, y) = e_1 \mathbf{in} e_2$. Explain how your semantics corresponds to the intuitive semantics. You may assume you have the usual substitution function $\mathbf{subst} x v e$ at your disposal.

2. Consider the syntax and small-step semantics of a small language with an “either” construct.

```

Inductive val :=
| VBool : bool -> val
| VNat : nat -> val.

Inductive expr :=
| EVal : val -> expr
| EEither : expr -> expr -> expr
| EIf : expr -> expr -> expr -> expr.

Inductive head_step : expr -> expr -> Prop :=
| Either_head_step_l e1 e2 :
  head_step (EEither e1 e2) e1
| Either_head_step_r e1 e2 :
  head_step (EEither e1 e2) e2
| If_head_step_true e2 e3 :
  head_step (EIf (EVal (VBool true)) e2 e3) e2
| If_head_step_false e2 e3 :
  head_step (EIf (EVal (VBool false)) e2 e3) e3.

Inductive step : expr -> expr -> Prop :=
| do_head_step e e' :
  head_step e e' -> step e e'
| If_step e1 e1' e2 e3 :
  step e1 e1' -> step (EIf e1 e2 e3) (EIf e1' e2 e3).

Inductive steps : expr -> expr -> Prop :=
| steps_refl e :
  steps e e
| steps_step e1 e2 e3 :
  step e1 e2 -> steps e2 e3 -> steps e1 e3.

```

- (a) Explain the intuitive semantics of the “either” construct. Your answer should involve an example program. You should give the possible return values of your example program and explain how the operational semantics assigns these return values. You may write your example program in pseudo syntax, *i.e.*, you do not have to use Coq syntax.
- (b) Define an interpreter for this language as:

```

Fixpoint interp (e : expr) : list (option val)

```

The `list` type is used to account for non-determinism. The interpreter should return all possible return values as elements of the list. The list should be considered as a set, *i.e.*, the order and duplicates are irrelevant. The `option` type is used to account for expressions that get stuck. You should give your answer in Coq syntax. You are allowed to use standard functions on lists, such as `app : list A -> list A -> list A` and:

```

Fixpoint flat_map {A B} (f : A -> list B) (l : list A) : list B :=
  match l with
  | [] => []
  | x :: l => app (f x) (flat_map f l)
  end.

```

- (c) State the correctness of the interpreter w.r.t. the small-step operational semantics. Your correctness lemma(s) should account for both the `Some` and `None` cases. You are allowed to

use standard predicates on lists, such as `In : A -> list A -> Prop`. You should give your answer in Coq syntax.