ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 NRF ERC

ROSAEC MEMO
2011-13
September 5, 2011

Hoare logic for multistaged programs

Kwangkeun Yi
Seoul National University
kwang@ropas.snu.ac.kr

Cristian Gherghina
National University of Singapore
cristian@comp.nus.edu.sg

September 5, 2011

Abstract

In this work we present a straight forward extension of Hoare logic that caters for multistaged programs. We have chosen a minimalist support language which allowed us to focus on the issues pertaining to the staged features. Similarly, the support logic is a simple, staged, first order logic with equality assertions. We allow the equality assertions to contain descriptions of staged code in the form of pre post conditions which are in turn expressed as formulas in our logic. We formulate Hoare rules for each of the language constructs. Furthermore we prove the rules sound with respect to an intuitive semantic of our staged language. The formalization is done in Coq, proofs are available at: http://www.comp.nus.edu.sg/ cristian/projects

1 Introduction

Several previous works focused on easing the way for verification of multistaged programs: the main approach in recent proposals consists of translating such programs to equivalent unstaged programs for which traditional verification systems can be applied. In this work we propose a direct approach of verification for staged programs. We propose an extension of Hoare logic that is catered for an imperative language with staging constructs.

2 Support language syntax

For the simplicity of the development we dismissed orthogonal language features and focused on a minimal yet complete support language. Our language is a while language with operations for handling staged constructs: a box operator for creating stages (boxes) which are stored in memory, an unbox command for exposing the staged code and a run command that will unbox and execute the content of a box. The syntax is described in Figure 1.

We have several observations about the language first and foremost boxes are first class elements, they can be created, stored, moved without restrictions and eventually executed. Also note that the box operator is decorated with a specification (pre/post condition) that describes the resulting box, the actual form and well-formed constraints imposed on this specification will be described in the following sections.

3 Small-step operational semantics

We define a program state representation as a pair of memory and program. The program denotes the remaining commands to be executed while the memory describes the values of the visible variables. The memory is thus a partial function from variables to values. As values, we allow for numerical constants and box values which are basically sequences of staged code.

$$
\begin{array}{rcll}
cmd & ::= & cmd_1 \ ; \ cmd_2 & sequence \\
 & | & Skip & nop \\
 & | & v = exp & assign \\
 & | & Unbox\ v & unbox \\
 & | & Run\ v & run \\
 & | & While\ v\ cmd & while \\
 & & & \\
exp & ::= & const \mid v \mid Box(cmd, spec) &
\end{array}
$$

Figure 1: Language syntax

$$
\begin{array}{rcll}
val & ::= & vData\ of\ Nat & constants \\
 & | & vBox\ of\ cmd & box\ containing\ cmd \\
 & & & \\
\rho & ::= & var \hookrightarrow val & memory \\
 & & & \\
\sigma & ::= & (\rho * cmd) & state
\end{array}
$$

Figure 2: Program state

Before defining the small step semantics for our language we also need to introduce some helper functions: expression evaluators and consistency checkers. These support functions will model the way in which an expression is evaluated in this language.

evalBox is a partial function from program states and expressions to values. It is focused on the evaluation of a box construct. The language requires that a box operation evaluate all unbox commands within the boxed code before the box value is constructed. An unbox command requires the unboxed variable to contain a box value. The evaluation consists in replacing the unbox command with the box content. Therefore the *evalBox* function will recursively traverse the code and replace all unbox commands with the corresponding code. *evalExpr* is a partial function that takes a memory state and an expression and tries to evaluate it to a value.

Secondly we define a *box_cl* check that verifies if a boxed code is free of any unbox operations if so the box can appear as an argument to a run command.

With these definitions in place we can describe the small step semantics of our language. Note that the expected behaviour for a multistaged program is that unbox commands can not be executed. Unboxings are restricted to levels strictly greater than 0 where they can only be evaluated/eliminated by *evalBox* . Thus a machine encountering such a command should block. Therefore the semantics should not contain a step for unbox commands. However for simplicity of the Hoare rule presentation we parameterize our semantics with a flag that will enable in exceptional situations the execution of a unbox command.

The necessity for this extra semantic will be made clear when discussing the assign rule. In short such a lax semantics ($\rightsquigarrow_f$ *alse*) is useful in establishing the correctness of the specifications given by the user for the box operations. However note that in the default semantics ,$\rightsquigarrow_t$ *rue*, unboxes are not executed.

$$
\begin{array}{lcll}
evalBox\ \rho\ (Unbox\ v) & \leadsto & c & if\ \rho\ v\ =\ vBox\ c \\
evalBox\ \rho\ (s_1;\ s_2) & \leadsto & (c_1; c_2) & if\ (evalBox\ \rho\ s_1) \leadsto c_1 \wedge (evalBox\ \rho\ s_2) \leadsto c_2 \\
evalBox\ \rho\ (while\ v\ c) & \leadsto & while\ v\ c_1 & if\ evalBox\ \rho\ c\ \leadsto\ c_1 \\
evalBox\ \rho\ (Run\ v) & \leadsto & (Run\ v) & \\
evalBox\ \rho\ Skip & \leadsto & Skip & \\
evalBox\ \rho\ (v\ =\ e) & \leadsto & v\ =\ e &
\end{array}
$$

$$
\begin{array}{lcl}
evalExpr\ \rho\ ct & \leadsto & vData\ ct \\
evalExpr\ \rho\ v & \leadsto & \rho\ v \\
evalExpr\ \rho\ Box\ k\ s & \leadsto & vBox\ (evalBox\ \rho\ v)
\end{array}
$$

Figure 3: Evaluators

$$
\begin{array}{lcll}
(\rho,\ Skip;\ k) & \leadsto_b & (\rho,\ k) & \\
(\rho,\ (c1; c2);\ k) & \leadsto_b & (\rho,\ c1;\ (c2;\ k)) & \\
(\rho,\ v = e;\ k) & \leadsto_b & (\rho[v \mapsto nv],\ k) & if\ evalExpr\ \rho\ e = nv \\
(\rho,\ run\ v;\ k) & \leadsto_b & (\rho,\ c;\ k) & if\ \rho\ v\ =\ vBox\ c \wedge box\_cl\ c \\
(\rho,\ (while\ v\ c);\ k) & \leadsto_b & (\rho,\ (c\ ;\ while\ v\ c); k) & if\ \rho\ v =\ vData\ 0 \\
(\rho,\ (while\ v\ c);\ k) & \leadsto_b & (\rho,\ k) & if\ \rho\ v \neq\ vData\ 0 \\
(\rho,\ Unbox\ v;\ k) & \leadsto_{false} & (\rho,\ c;\ k) & if\ \rho\ v\ =\ vBox\ c
\end{array}
$$

Figure 4: Small Step Semantics

In both cases the run command can be executed only if the variable on which it is applied stores a box that is closed, this well-formed conditions requires a box to be free of unbox commands before it is run. All other side conditions for the semantic steps are standard for an imperative while language.

4 Hoare tuples

We have chosen first order logic with equalities as the support logic for the Hoare tuples. All program states will be abstracted into formulas in this logic. The syntax of the formulas can be summarized as follows:

$$
\begin{array}{rcl}
\Phi & ::= & False \mid True \mid v \Downarrow lvalue \\
 & \mid & \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \rightarrow \Phi_2 \mid \neg\Phi \mid \exists\ v.\Phi \mid \forall\ v.\Phi
\end{array}
$$

In this logic the basic atoms are assertions on the abstractions of variable values. Depending on the concrete value of the variable, the abstracted variable values can be either constants or box specifications. An abstraction of a box value consists of a triple of: precondition, postcondition and boolean value indicating if the box is free of unbox operations and hence the content can be run.

$$
\begin{array}{rcl}
lvalue & ::= & lvData\ ct \\
 & \mid & lvBox\ (\Phi_{pre},\ \Phi_{post},\ bool)
\end{array}
$$

As the formulas encode constraints on memory states, in the proof mechanization we define the formulas as functions from the states to booleans, such that a formula is satisfied by a memory state iff the application of the formula to the state yields true. In this way a satisfiability check is translated to a function application.

Before introducing the Hoare tuple formalization we define the notions of : halted, can_step, safe, and guards. An execution is halted for a program state if there are no more commands to be executed. The semantics *can_step* from a given state if there exists a program state such that in exactly one step the semantics transitions from input to this state. A program state is safe if either the execution is halted for this state or if the semantics can take at least one more step. One important support for the definition of the Hoare tuple is the notion of guard. We introduce the notion of guard as follows: a formula Φ guards a code sequence k if for all memory states that satisfy Φ the code is safe. That is:

$$\frac{k{=}Skip}{halted\ \rho\ k} \qquad \frac{\exists k'\ \rho'\ \cdot\ (\rho,k){\rightsquigarrow_b}(\rho',k')}{can\_step\ \rho\ k}$$

$$\frac{\forall\ \rho'\ k'\ \cdot\ ((\rho,k){\rightsquigarrow_b^*}(\rho',k') \quad \rightarrow\ can\_step\ \rho'\ k' \vee halted\ \rho'\ k')}{safe\ b\ k\ \rho}$$

$$\frac{\forall \rho.\ (\Phi\ \rho)\ \rightarrow\ safe\ b\ k\ \rho}{guards\ b\ \Phi\ k}$$

As mentioned, the guards function describes the conditions under which a code will never get stuck. We can also introduce the *sat_post* relation that is satisfied if whenever the code finishes properly, the state in which it finished satisfies the given postcondition.

$$\frac{\forall\ \rho\ \rho' \cdot ((P\ \rho) \wedge (\rho, c) \rightsquigarrow_b^*\ (\rho', Skip)) \rightarrow Q\ \rho'}{sat\_post\ b\ c\ P\ Q}$$

Based on the guards function, we can define a Hoare tuple as a four tuple of boolean, indicating which version of the semantics to use (with or without unbox execution), formula denoting the precondition, code and formula denoting the postcondition.

$$\frac{\forall k\ \cdot\ guards\ true\ Q\ k\ \rightarrow\ guards\ b\ P\ (c;k) \wedge sat\_post\ b\ c\ P\ Q}{hoare\ b\ P\ c\ Q}$$

A Hoare tuple *hoare b P c Q* holds if for any subsequent code k that is guarded by postcondition Q, precondition P guards the sequential composition of c with k and if post condition Q is satisfied whenever c terminates. Simply put, if the program does not block while executing the code that follows c from any state that satisfies the postcondition Q then the program will not block on $c; k$ if it starts from a state that satisfies P and also if c actually terminates, then it will terminate in a state that satisfies Q.

Basically, there are two assertions enclosed within the Hoare tuple: the fact that the program will not block and that if the program terminates it will do so in a state that satisfies the postcondition.

4.1 Formula definition/satisfiability

One remaining issue is the definition of the formulas. As mentioned we represent formulas as functions from memory states to booleans which denote the satisifiability of the given formula for the input state. All the definitions apart from the assert formula are intuitive. The more complicated case is the definition of the satisfiability for the assert formula. It requires an abstraction relation between concrete values and logical values, *val2lval cval lval*. This relation will state when a given logical value is a correct abstraction of the concrete value. Note however that this is a relation and not a function. The implication is that specifications for boxes can not be inferred and this is why we require annotations to be given.

$$\frac{cv{=}\ vData\ ct \qquad lv{=}lvData\ ct}{val2lval\ \ cv\ \ lv}$$

$$\frac{\begin{array}{c}hoare\ false\ \Phi_{pre}\ c\ \Phi_{post} \qquad b\ =\ box\_cl\ c \\ lv = lvBox\ (\Phi_{pre},\ \Phi_{post},\ bool)\end{array}}{val2lval\ \ (vBox\ \ c)\ \ lv}$$

A normal expectation would be that the abstraction for the box value ($vBoxc$) is a specification Φ_{pre}, Φ_{post} that should satisfy the Hoare triple $hoare\ b\ \Phi_{pre}\ c\ \Phi_{post}$. However notice that the $val2lval$ definition is a bit more relaxed, it requires the box annotation to be a proper specification for the boxed code with respect to the lax semantics. That is, the corresponding Hoare tuple for the given specification and the boxed code holds when evaluated under the lax semantics. This version is used due to the fact that at the moment of the box creation the enclosed code might not be unbox free, therefore checking if the Hoare triple holds with respect to the default semantics will always fail for such code.

The $val2lval$ relation will be used in the definition of the satisfiability condition for formulas of form $v \Downarrow lval$ as follows. A formula is said to be satisfied by a memory state ρ if:

$$\begin{array}{lcl}True\ \rho & ::= & true \\ False\ \rho & ::= & false \\ (v \Downarrow lval)\ \rho & ::= & val2lval\ (\rho\ v)\ lval \\ (\Phi_1 \wedge \Phi_2)\rho & ::= & (\Phi_1\rho) \wedge (\Phi_1\rho) \\ (\Phi_1 \vee \Phi_2)\rho & ::= & (\Phi_1\rho) \vee (\Phi_1\rho) \\ (\Phi_1 \rightarrow \Phi_2)\rho & ::= & (\Phi_1\rho) \rightarrow (\Phi_1\rho) \\ (\neg\Phi)\rho & ::= & \neg(\Phi\rho)\end{array}$$

5 Hoare rules

For each syntactic construct in the language we defined the necessary conditions under which a Hoare tuple holds. Apart from the assign rule, all of the rules have straightforward, intuitive formulations.

Lemma. (Skip)

$$\frac{}{hoare\ true\ Q\ \ Skip\ Q}$$

Lemma. (Seq)

$$\frac{hoare\ true\ P\ s_1\ Q \qquad hoare\ true\ Q\ s_2\ T}{hoare\ true\ P\ (s_1;\ s_2)\ T}$$

Lemma. (Weak)

$$\frac{hoare\ true\ P\ c\ Q \qquad P' \rightarrow P \qquad Q \rightarrow Q'}{hoare\ true\ P'\ c\ Q'}$$

Lemma. (While)

$$\frac{hoare\ true\ (P \wedge\ (v \Downarrow\ lvData\ 0))\ \ c\ \ P}{hoare\ true\ P\ \ (while\ v\ c)\ \ (P \wedge \neg(v \Downarrow\ lvData\ 0))}$$

Lemma. (Run)

$$\frac{P_1 \rightarrow\ v \Downarrow\ lvBox(P, Q, true) \qquad P_1 \rightarrow\ P}{hoare\ true\ P_1\ \ (Run\ v)\ \ Q}$$

Lemma. (Frame)

$$\frac{hoare\ true\ P\ c\ Q \qquad indep\ F\ c}{hoare\ true\ (P \wedge F)\ c\ (Q \wedge F)}$$

$Indep\ F\ c$ holds if and only if the satisfiability of formula F is not affected by the execution of c.

$$\frac{\forall \rho\ \rho' \cdot\ F\ c \to (\rho, (c; Unit)) \rightsquigarrow^{*}_{true} (\rho', \_) \to F\ \rho'}{Indep\ F\ c}$$

For the simplicity of the presentation the assign rule can be split in three subrules depending on the structure of the right hand side of the assignment.

Lemma. (Assign ct)

$$\frac{}{hoare\ true\ True\ (v = ct)\ (v \Downarrow\ lvData\ ct)}$$

Lemma. (Assign var)

$$\frac{}{hoare\ true\ (v_1 \Downarrow c)\ (v = v_1)\ (v \Downarrow c)}$$

Lemma. (Assign box)

$$\frac{hasAllBoxes\ P\ (boxVars\ c)}{hoare\ true\ P\ (v = Box\ c\ (P, Q))\ (v \Downarrow\ lvBox\ Q)}$$

At this point we can look a bit closer into the box operator specifications. We require the code to be provided with annotations for each box operation. Such annotations are pairs: box precondition and box post condition. A box precondition will need to contain the necessary constraints that the box operation succeeds. More precisely, it will need to describe constraints for the unboxings within the boxed code. The result of a box operation is a box value which in the abstract domain consists of a triplet (precondition formula, postcondition formula, boolean). Therefore a box specification will have the following type (formula * (formula *formula*boolean)).

The assign rule for box expressions has to ensure that all the variables that will be unboxed are defined and contain boxed values. We rely on the $boxVars$ function to retrieve the list of variables that appear in unbox commands in c. Furthermore we use the relation $hasAllBoxes$ to verify that all variables that are to be unfolded have a description in precondition P.

$$\frac{\forall v \in l \cdot\ \exists\ b\ Pr\ Po \cdot\ (P \to\ (v \Downarrow\ lvBox\ Pr\ Po\ b))}{hasAllBoxes\ P\ l}$$

The Assign box lemma relies on the assumption that the box specification provided in the code text is well-formed. That is we restrict the provided specifications as follows: the box in the assignment $v = Box\ c\ (P, (Pr, Po, b))$ is correctly specified if it satisfies the wf condition:

$$\frac{\forall \rho\ c_0 \cdot\ P\ \rho\ \to\ evalBox\ \rho\ c = \ c_0\ \to (hoare\ false\ Pr\ c_0\ Po) \wedge (box\_cl\ c_0 = b)}{wf\ \ c\ (P, (Pr, Po, b))}$$

Note that both val2lval and the wf relations have been using the Hoare tuple versions that allow for unbox operations to execute. This is because at the moment of the val2lval and wf applications the box value might not be fully closed. Despite this, a correct specification for a box value will need to ensure that whenever the box will be closed and implicitly the unboxes have been evaluated the resulting code will satisfy the specification.

6 Coq mechanization

All the above lemmas have been proven correct in Coq. The overall mechanization effort spans about 2000 lines of proofs. The formalization has been split in three modules. The machine description which contains the language syntax and semantic definitions together with lemmas pertaining to sequential composition of the step relation. The definitions of the formulas , the abstraction relation and the well-formed conditions are defined in the logic module. The third module Hoare rules contains the above lemmas and their proofs. Several of the facts required could be stated in a general form which can allow the reuse in other contexts. Such facts have been stated as lemmas and proven separately.

6.1 Handling box specifications

One important issue that arose in the Coq formalization was the definition of the language syntax. Allowing for specifications defined as formulas to appear in the program code proved to be a challenge due to the fact that a cyclic definition occurred: a formula is a function from memory to Prop while a memory is a function from memory to value, a value can contain code (box values) which in turn contains formulas as specifications. This constraint has been circumvented by introducing an intermediary step, the box command does not store directly the specifications, it contains an identifier which is used to retrieve the specifications from a repository. And this allows a clean decoupling of the program syntax definitions from the semantic definitions. The consequence is that the box rule requires an extra retrieve operation from a global specification repository.

6.2 Auxiliary lemmas

As mentioned, several general facts have been proven about the current language and stated as separate lemmas. Most of these lemmas pertain to a transitive closure of the step relation (the formalization of a semantic step) called stepstar and defined as follows:

```
Inductive stepstar : nat->bool->state -> state -> Prop :=
  | SeqStepStar0 : forall c b, stepstar 0 b c c
  | SeqStepStarS : forall n c c' c'' b,
      step b c c' -> stepstar n b c' c'' -> stepstar (S n) b c c''.
```

- The semantic (the step relation) is deterministic:

$$\forall\ s\ s1\ s2\ b,\ step\ b\ s\ s1\ \rightarrow\ step\ b\ s\ s2\ \rightarrow\ s1\ =\ s2$$

  With the following consequences:

  - The transitive closure of the semantic is deterministic:

$$\forall\ n\ b\ s\ s'\ s'',\ stepstar\ n\ b\ s\ s'\ \rightarrow\ stepstar\ n\ b\ s\ s''\ \rightarrow\ s'\ =\ s''$$

  - If a program finishes, then it does so in constant number of steps:

$$\forall\ n\ n'\ c\ x\ z\ b,\ stepstar\ n\ b\ (x,c)\ (z,Unit)\ \rightarrow\ stepstar\ n'\ b\ (x,c)\ (z,Unit)\ \rightarrow\ n = n'$$

- Syntactic consistency:

$$\forall\ k\ c,\ Seq\ c\ k\ =\ k\ \rightarrow\ False$$

- Backward rolling, stepping once after n steps is equivalent to stepping n+1 times:

$$\forall\ n\ c\ c'\ c''\ b,\ stepstar\ n\ b\ c\ c'\ \rightarrow\ step\ b\ c'\ c''\ \rightarrow\ stepstar\ (S\ n)\ b\ c\ c''$$

  Although it seems trivial, due to the inductive definition of stepstar this fact occurs often and is not superfluous to dismiss.

- Backward unrolling, n+1 steps can be decomposed as n steps followed by one step:

$$\forall\ n\ c\ c''\ b,\ stepstar\ (S\ n)\ b\ c\ c''\ \rightarrow\ exists\ c',\ (step\ b\ c'\ c''\ \wedge\ stepstar\ n\ b\ c\ c')$$

- Stepstar splitting, a sequence of steps can be split in the middle obtaining two linked sequences of steps:

$$\forall\ n1\ n2\ c1\ c3\ b,\ stepstar\ n1\ b\ c1\ c3 \rightarrow\ n2 <= n1$$
$$\rightarrow\ exists\ c2,\ stepstar\ n2\ b\ c1\ c2 \wedge\ stepstar\ (n1 - n2)\ b\ c2\ c3$$

- Stepstar splitting:

$$\forall\ n1\ n2\ c1\ c2\ c3\ b,\ stepstar\ n1\ b\ c1\ c3 \rightarrow\ stepstar\ n2\ b\ c1\ c2\ \rightarrow\ n2 < n1$$
$$\rightarrow\ stepstar\ (n1 - n2)\ b\ c2\ c3$$

- Sequential decomposition, if a step is possible for code c, then c is a sequential composition:

$$\forall\ m\ c\ s\ b,\ step\ b\ (m,c)\ s\ \rightarrow\ exists\ a,\ exists\ t,\ c = Seq\ a\ t$$

- Tail irrelevances:
  If (c;Unit) steps to a state, and (c;k) step in the same number of steps to a different state, then the two states have the same memory and the remaining code differs only at the tail.

$$\forall\ n\ k\ s\ c\ mu\ mk\ cu\ ck\ b,\ stepstar\ n\ b\ (s, Seq\ c\ k)\ (mk, ck)$$
$$\rightarrow\ stepstar\ n\ b\ (s,\ Seq\ c\ Unit)\ (mu,\ cu)\ \rightarrow\ mk = mu\ \wedge\ same\_head\ cu\ ck\ k$$

  Where:

```
  Fixpoint same_head c1 c2 t := match c1 with
| Unit => c2=t
| Seq d1 d2 => exists x1, c2= Seq d1 x1 /\ same_head d2 x1 t
| _ => False
```

  and if c executes completely in n steps leaving the memory in state mk then (c;k) after n steps has memory mk and only the code k to execute

$$\forall\ n\ k\ s\ c\ mk\ b,\ stepstar\ n\ b\ (s, Seq\ c\ Unit)(mk, Unit)\ \rightarrow\ stepstar\ n\ b\ (s,\ Seq\ c\ k)(mk,\ k)$$

  and if (c;Unit) can take n steps then (c;c2) also is able to take n steps

$$\forall\ n\ s\ c\ c2\ s1,\ stepstar\ n\ true\ (s,\ Seq\ c\ Unit)\ s1 \rightarrow\ exists\ q,\ stepstar\ n\ true\ (s,\ Seq\ c\ c2)\ q$$

- Sequence termination, if (s1;s2) terminates then s1 terminates as well:

$$\forall\ n\ s1\ s2\ s\ x\ b,\ stepstar\ n\ b\ (s,\ Seq\ s1\ s2)\ (x, Unit)$$
$$\rightarrow\ exists\ q,\ exists\ y,\ stepstar\ q\ b\ (s, Seq\ s1\ Unit)(y,\ Unit) \wedge q <= n$$

- Sequence merging:

$$\forall\ n1\ n2\ c1\ c2\ s\ q\ r\ b\ k,\ stepstar\ n1\ b\ (s, Seq\ c1\ Unit)(q, Unit) \rightarrow\ stepstar\ n2\ b\ (q, c2)(r, k)$$
$$\rightarrow\ stepstar\ (n1 + n2)\ b\ (s,\ Seq\ c1\ c2)(r, k)$$

- Tail irrelevance for step:

$$\forall\ n\ s\ c\ k\ s1\ s2\ y\ b,\ stepstar\ n\ b\ (s,\ Seq\ c\ k)\ s1 \rightarrow\ stepstar\ n\ b\ (s,\ Seq\ c\ Unit)\ s2$$
$$\rightarrow\ step\ b\ s2\ y\ \rightarrow can\_step\ b\ s1$$

- Conversions between semantics (true → false):

$$\forall\ x\ y,\ step\ true\ x\ y\ \rightarrow\ step\ false\ x\ y$$

  and

$$\forall\ n\ x\ y,\ stepstar\ n\ true\ x\ y\ \rightarrow\ stepstar\ n\ false\ x\ y$$

- Conversion between semantics (false → true):

$$\forall\ c\ s\ k\ x,\ box\_cl\ c\ = true\ \rightarrow\ step\ false\ (s,\ Seq\ c\ k)\ x\ \rightarrow\ step\ true\ (s,\ Seq\ c\ k)\ x$$