Step2Core translation guidelines

Written by:
Anna Kukuy & Bodnya Valeriya
TABLE OF CONTENTS:

DECLARATIONS .................................................. 3
TYPES .............................................................. 6
EXPRESSIONS .................................................... 8
MODULE TRANSITION SYSTEM ............................. 9
MODULE COMPOSITION ................................. 10
Declarations

Step:

3 types of declaration exist in STEP:
   1. Type declarations
   2. Macro declarations
   3. Variable declarations

1. **Type declarations:**
Type declarations are used to define abbreviation for compound types

Step:

type-decl -> t-decl | enum-decl | u-type
  t-decl -> id, type
  enum-decl -> id, id+
  u-type -> id
  type -> id | basety | array | channel | range+ | type+ | record

Types can be declared in:
   a. Global declaration (Module System level)
      ModuleSystem -> name?, decl*, module+

   b. Basic and abstract module definitions
      module-description -> exported-transitions?, env-assumption?,
                          decl?, initial?, transition+

   c. Expressions of module compositions
      module-renaming -> module-composition, decl+, substitution+
      module-restriction -> module-composition, decl+, restriction+
      module-augmentation -> module-composition, decl+, expns

Core:

TYPE -> TypeDef+
TypeDef -> ATOM, (Range | Scalarset | Enum | ATOM))

Types can be declared only in global program space

Global -> HOLD_PREVIOUS?, CONST?, TYPE?, VAR?, DEFINE?

Step To Core Type Declarations:
Since types in Step are globally visible (no matter where they were defined), the translation is trivial.
2. Macro declarations:

Step:

mac-decl -> id, typed-ids?, type?, expn

Macros can be declared in:
   Global declaration (Module System level)
   ModuleSystem -> name?, decl*, module+

Core:

DEFINE -> Def+
Def -> ATOM, Expr

Macros can be declared in:
   a. Global declaration
      Global -> HOLD_PREVIOUS?, CONST?, TYPE?, VAR?, DEFINE?
   b. Module definitions
      Module -> ATOM, Params?, DEFINE?, VAR?, COJOIN?, Comment?,
                  (Modcombin | Transition+)

Step To Core Macro Declarations:
   The translation is trivial

3. Variable declarations:

Step:

system-var-decl -> mode, ids, type, expn?, init_val?
mode -> local | external-in | external-out | in | out

<table>
<thead>
<tr>
<th></th>
<th>in</th>
<th>out</th>
<th>local</th>
</tr>
</thead>
<tbody>
<tr>
<td>external</td>
<td>Visible and changeable by others modules, but not by module himself, cannot have where clause</td>
<td>Visible and changeable by others modules</td>
<td>-----------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>Visible by others modules, constant</td>
<td>Visible but not changeable by others modules</td>
<td>Not visible</td>
</tr>
</tbody>
</table>
Variables can be declared only in modules (no global variables)

a. Basic and abstract module definitions
   module-description -> exported-transitions?, env-assumption?, decl?, initial?, transition+

b. Expressions of module compositions
   module-renaming -> module-composition, decl+, substitution+
   module-restriction -> module-composition, decl+, restriction+
   module-augmentation -> module-composition, decl+, expns

Core:

VAR -> VarDef+
VarDef -> ATOM, Typename, INITVAL?

Types can be declared in:

a. Global declaration
   Global -> HOLD_PREVIOUS?, CONST?, TYPE?, VAR?, DEFINE?

b. Module definitions
   Module ->ATOM, Params?, DEFINE?, VAR?, COJOIN?,
   Comment?, (Modcombin | Transition+)

- **HOLD_PREVIOUS** defines the functionality of an assignment for global variables. If it presents, then for all variables that are not mentioned in the assignment, their previous values are retained. If it does not present then for all variables that are not mentioned in the assignment, random values are chosen from their domain.

- Variables defined in modules are local.

Step To Core Variable Declarations:

<table>
<thead>
<tr>
<th>Step</th>
<th>external in</th>
<th>In</th>
<th>external out</th>
<th>Out</th>
<th>local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>global variable + adding &quot;var'=var&quot; in assignment field of ALL transitions inside the module where the &quot;int&quot; variable was defined.</td>
<td>global constant</td>
<td>global variable</td>
<td>global variable + adding &quot;var'=var&quot; in assignment field of ALL transitions outside the module where the &quot;out&quot; variable was defined.</td>
<td>Usual module variables, since Core local variables are by default not visible from outside the module</td>
</tr>
</tbody>
</table>

**HOLD_PREVIOUS** must be set by default in CDL program
Types

Step:

type-decl -> t-decl | enum-decl | u-type
  t-decl -> id, type
  enum-decl -> id, id+
  u-type -> id
type -> id | basety | array | channel | range+ | type+ | record
  basety -> bool | int | rat | real
  array -> type, type
  range -> expn, expn
  record -> record-entry+
  record-entry -> id, type

Types used for variable declarations
  system-var-decl -> mode, ids, type, expn?

Core:

TypeDef -> ATOM, (Range | Scalarset | Enum | ATOM)
  Range -> Expr, Expr
  Scalarset -> Expr
  Enum -> ATOM+
  Typename -> ArrayType | BOOL | INTEGER | ATOM
  ArrayType -> Expr, Typename

  • ATOM in Typename element is name of type defined in TypeDef element.
  • Types used for variable declarations
    VarDef -> ATOM, Typename, INITVAL?

Step To Core:

Almost all types’ translation is trivial except for:
  • Rational and real types that can’t be simulated in Core.
  • Tuple type will be translated using the following transformation:

    a: int*bool*int
    #1 a
    in Core

    a_step_tuple_1:int
    a_step_tuple_2:bool
    a_step_tuple_3:int

This way happens tuple rejection and update:
Step          Core
#a 1=..      a_tuple_core_def_1=..

Step          Core
@a 1 := ..     a_step_tuple_1:= ..

- Record type will be translated using the following transformation:

In STeP:

a: { id1:int id2:bool id3:int }

in Core

a_step_tuple_id1:int
a_step_tuple_id2:bool
a_step_tuple_id3:int

This way happens record rejection and update:
Step          Core
#a id1=..      a_step_tuple_id1=..

Step          Core
@a id1 := ..     a_step_tuple_id1:= ..
Expressions

Step:

\[
\text{expn} \rightarrow \text{bool-const} \mid \text{int-const} \mid \text{id} \mid \text{array-ref} \mid \text{let} \mid \text{primed} \mid \langle \text{expn}, \text{infix-op}, \text{expn} \rangle \mid \langle \text{prefix-op}, \text{expn} \rangle \\
\text{infix-op} \rightarrow \text{BOOLEAN-OPS} \mid \text{ARITH-OPS} \mid \text{PRED-OPS} \\
\text{BOOLEAN-OPS} \rightarrow \text{AND} \mid \text{OR} \mid \text{IMPL} \mid \text{EQUIV} \\
\text{ARITH-OPS} \rightarrow \text{PLUS} \mid \text{TIMES} \mid \text{MINUS} \mid \text{MOD} \mid \text{DIV} \\
\text{PRED-OPS} \rightarrow \text{EQ} \mid \text{NOTEQ} \mid \text{GT} \mid \text{LT} \mid \text{GE} \mid \text{LE} \\
\text{prefix-op} \rightarrow \text{NEG} \mid \text{MINUS} \mid \text{CHANNEL-OPS}
\]

Core:

\[
\text{Expr} \rightarrow \text{Constant} \mid \text{PLUS} \mid \text{MINUS} \mid \text{DIVIDE} \mid \text{TIMES} \mid \text{MOD} \mid \text{EQUAL} \mid \text{NOTEQUAL} \mid \text{LE} \mid \text{GE} \mid \text{LT} \mid \text{GT} \mid \text{Neatomset} \mid \text{OR} \mid \text{AND} \mid \text{NOT} \mid \text{PAR_EXPR} \\
\text{Constant} \rightarrow \text{Genconst} \mid \text{Boolconst} \\
\text{Genconst} \rightarrow \text{Atom} \mid \text{TaggedAtom} \mid \text{Number} \\
\text{TaggedAtom} \rightarrow \text{Atom} \\
\text{Atom} \rightarrow \text{ATOM} \mid \text{ARRAY}
\]

Step To Core:

1. Infix operations
   - Almost all mentioned infix operations have trivial translation to Core (in fact, Step has some additional infix operations, e.g. temporal, but we cannot translate them, so we have ignored them).

2. Prefix operations
   - NEG to NOT and MINUS to MINUS.
   - There is no need to translate channel operations to Core, since they belong to SPL.

3. array-ref is ARAY, id is ATOM, primed(expn) is TypedAtom (all three are from type constant in Core).
Module Transition System

Module systems allow a hierarchical representation of transition systems.

Step:
The basic block of the system is transition module.
module -> basic-module | module-definition | (abstract-module, module-composition)

There are 3 types of modules in Step:
1. Basic module – consists of parameters, interface that describes its interaction with the environment and the body that describes its actions (transitions).
   basic-module -> name, params?, module-description
   params -> param+
   param -> id, type
   module-description -> exported-transitions?, env-assumption?, decl?, initial?, transition+

2. Abstract module has interface only (used for recursive declarations and case distinctions)
3. Complex modules constructed from simpler ones by module expressions.
   module-definition -> name, params?, module-composition

Core:
Module -> ATOM, Params?, DEFINE?, VAR?, COJOIN?, Comment?, (Modcombin | Transition+)

Core has only one type of module that can include combination of modules or transitions.
Module Composition

The most difficult in the module translation is the difference in the module composition between the languages.

Step:

Complex modules constructed from simpler ones by (recursive) module expressions, allowing the description of hierarchical systems of unbounded depth.
Module expressions refer to instances of parameterized modules defined earlier by name.

\[
\begin{align*}
\text{module-composition} & \to \text{(module-expn, module-expn)} \mid \text{module-expn} \\
\text{module-expn} & \to \text{module-instance} \mid \text{module-renaming} \mid \text{module-hiding} \mid \\
& \mid \text{module-augmentation} \mid \text{module-restriction} \mid \text{module-casedistinction} \\
\text{module-instance} & \to \text{name, name, expns?} \\
\text{module-renaming} & \to \text{module-composition, decl+, substitution+} \\
\text{module-hiding} & \to \text{module-composition, name} \\
\text{module-restriction} & \to \text{module-composition, decl+, restriction+} \\
\text{module-augmentation} & \to \text{module-composition, decl+, expns} \\
\text{module-casedistinction} & \to \text{case+} \\
\text{case} & \to \text{expn, module-expn} \\
\text{substitution} & \to \text{var-substitution} \mid \text{trans-substitution} \\
\text{var-substitution} & \to \text{name, expn} \\
\text{trans-substitution} & \to \text{name, name} \\
\text{restriction} & \to \text{id, id, expn, expn}
\end{align*}
\]

1. Renaming allows renaming of some module variables or transitions.
2. Hiding used to make some module-variables local.
   - Parameters are transformed to locals
   - Transitions are removed from export list
3. Restriction is mechanism replacing old parameters with new ones
4. Augmentation is mechanism giving as the possibility to add to the module interface new output variables given as function of existing ones. All transitions are augmented to update of the new variables and constraint on them is added to initial condition.
5. Casedistinction – choosing interface for a module according to the case.
6. Module composition between module expressions is synchrony for exported transitions with the same name and asynchrony for the rest

Core:

\[
\begin{align*}
\text{Modcombin} & \to \text{INST} \mid \text{ASYNC} \mid \text{SYNC} \mid \text{PART} \\
\text{INST} & \to \text{ATOM, Modparams?, Renames?} \\
\text{ASYNC} & \to \text{Modcombin, Modcombin} \\
\text{SYNC} & \to \text{Modcombin, Modcombin} \\
\text{PART} & \to \text{Modcombin, Pair+, Modcombin}
\end{align*}
\]
There are 4 types of module compositions:

1. Instantiation of module (with possible renaming of transitions)
2. Asynchrony – all transitions composed asynchrony
4. Partial synchronization – only listed transitions are synchronized

Renaming exists only for transitions

Step To Core:

The module translation will be performed before the translation of types and variables, so in this step the translated program will still contain such Step definitions, like “in”, “out”, “exported” and “local”.

The translation rules are as follows:

1. Case Distinction

   Step:          Module “cases_module” {  
                      cases   expn1 : module-expression1,  
                                 expn2 : module-expression2,  
                                      ...  
                                 expnN : module-expressionN  
                      end cases  
              }  

   CDL:           Module module_case_”n” {  
                          COJOIN : expn1;  
                          (module-expression1)  
                      }  

                      Module module_case_”n+1” {  
                          COJOIN : expn2;  
                          (module-expression2)  
                      }  

                      Module module_case_”n+N-1” {  
                          COJOIN : expnN;  
                          (module-expressionN)  
                      }  

                      Module “cases_module” {  
                          module_case_”n” || module_case_”n+1” || ... ||  
                          module_case_”n+N-1”)  
              }
Remark – “n” is some integer value.

2. **Augmentation and Restriction**
   Augmentation and restriction are replaced by composition of original expression and instance of new module that has in the relation field the augmentation (restriction) expression.

   **Step:**
   Module System SimpleTest:

   Module simple:
   out x: int where x = 0
   Transition t1:
   ...
   EndModule
   Module sim1 = Augment( b:simple where out grants: int grants = x)

   **Core:**
   --Module System SimpleTest
   VAR
   x: int = 0
   grants: int
   Module simple( ) {
      Transition t1:
      ...
   }
   Module module_st_0() {
      Transition t:
      enable: true;
      assign: x' := x;
      relation: (grants = x);
   }
   Module sim1() {
      (simple () || (module_st_0 ()))
   }

3. **Long compositions.** To simplify the translation and reading of translation output, we decide to remove module compositions that contain 3 or more module instances (otherwise we have problem of partial synchronization). In the case we have some long composition we just translate it to composition of 2 modules, each of them is composition of 2 modules or less and so on.

   **Step:**
   Module simple1 :
   export t1
Transition t1:
   ...
EndModule

Module sim1 = b:simple1||bb:simple1||bbb:simple1

Core:
Module simple1( ) {
   Transition t1:
      ...
}
Module module_comp_0() {
   (simple1 ()|(t1,t1)|(simple1 ()))
}
Module sim1() {
   (simple1 ()|(t1,t1_simple1_t1_simple1)|(module_comp_0 ()| (t1,t1) -> t1_simple1_t1_simple1))
}

4. **Hiding.** To hide some variable or transition in some module expression EXPR, we have to find the basic module contains the variable and to make its new version (with local variable or transition) and also to make new version of all expressions in the path between the EXPR and the basic module.

**Step:**
Module simple:
   out x: int where x = 0
Transition t1:
   ...
EndModule
Module sim1 = a:simple
Module sim2 = Hide(b:sim1 x)

Core

Module simple_step_hiding_1( ) {
   VAR
   x: int
   Transition t1:
      ...
}
Module sim1_step_hiding_0() {
   simple_step_hiding_1 ()
}
Module sim2() {
   sim1_step_hiding_0 ()
}
Remark – you can see here the “z” became local, because otherwise it was moving to global space in translation time.

5. **Renaming.** To rename some variable or transition in some module expression EXPR, we have to find the basic module contains the variable and to make its new version (with renamed variable or transition) and also to make new version of all expressions in the path between the EXPR and the basic module.

**Step:**

Module System SimpleTest:
Module simple:
   export t2
   Transition t2:
   ...
EndModule
Module sim1 = s:simple
Module sim2 = Rename( b:sim1 where Transition t2 = t)

**Core:**

--Module System SimpleTest
Module simple_step_renaming_1 () {
   Transition t:
   ...
}
Module sim1_step_renaming_0 () {
   simple_step_renaming_1 ()
}
Module sim2() {
   sim1_step_renaming_0 ()
}