Mapping Staterich Circus Processes into Stateless Processes in Haskell: Isabelle Theories and Proofs using Haskabelle

By Artur Oliveira Gomes

February 9, 2017

Contents

1 Circus and Z Abstract Syntax Tree

theory *AST* **imports** *Prelude* **begin**

1.1 Z Given Sets

type-synonym *GivenValue* = *string*

type-synonym *ZInt* = *int*

1.2 Z Names and Decorations

type-synonym *ZDecor* = *string*

type-synonym *ZVar* = *string* ∗ (*ZDecor list*)

type-synonym *GivenSet* = *ZVar*

type-synonym *ZName* = *string*

1.3 Z Expressions and Predicates

datatype *ZGenFilt* = *Choose ZVar ZExpr*
| *Check ZPred*
| *Evaluate ZVar ZExpr ZExpr*
and *ZExpr* = *ZVar ZVar*
| *ZInt ZInt*
| *ZTuple ZExpr list*
| *ZBinding* (*ZVar* ∗ *ZExpr*) *list*
| *ZSetDisplay ZExpr list*
| *ZSeqDisplay ZExpr list*
| *ZCross ZExpr list*
| *ZSetComp ZGenFilt list ZExpr option*
| *ZCall ZExpr ZExpr*
and *ZPred* = *ZFalse ZPred list*
| *ZTrue ZPred list*
| *ZAnd ZPred ZPred*
| *ZPSchema ZSExpr*
and *ZSExpr* = *ZSchema ZGenFilt list*

primrec *reason* :: *ZPred* ⇒ *ZPred list*
where
reason (*ZTrue x*) = *x*
| *reason* (*ZFalse x*) = *x*

primrec *update-reason* :: *ZPred list* ⇒ *ZPred* ⇒ *ZPred*
where
update-reason x (*ZTrue* -) = (*ZTrue x*)
| *update-reason x* (*ZFalse* -) = (*ZFalse x*)

type-synonym *ZFSet = ZExpr list*

datatype *ZSName = ZSPlain string*
| *ZSDelta string*
| *ZSXi string*

datatype *CSExp = CSExpr ZName*
| *CSEmpty*
| *CChanSet ZName list*
| *ChanSetUnion CSExp CSExp*
| *ChanSetInter CSExp CSExp*
| *ChanSetDiff CSExp CSExp*

datatype *NSExp = NSExpEmpty*
| *NSExprMult ZName list*
| *NSExprSngl ZName*
| *NSExprParam ZName ZExpr list*
| *NSUnion NSExp NSExp*
| *NSIntersect NSExp NSExp*
| *NSHide NSExp NSExp*
| *NSBigUnion ZExpr*

datatype *CParameter = ChanInp ZName*
| *ChanInpPred ZName ZPred*
| *ChanOutExp ZExpr*
| *ChanDotExp ZExpr*

datatype *Comm = ChanComm ZName CParameter list*
| *ChanGenComm ZName ZExpr list CParameter list*

1.4 Circus Actions

datatype *CAction = CSPSkip*
| *CSPStop*
| *CSPChaos*
| *CSPCommAction Comm CAction*
| *CSPSeq CAction CAction*
| *CSPExtChoice CAction CAction*

datatype *ParAction = CircusAction CAction*
| *ParamActionDecl ZGenFilt list ParAction*

1.5 Z and Circus Paragraphs

datatype *ZPara = ZSchemaDef ZSName ZSExpr*
| *Process ProcDecl*
and *ProcDecl = CProcess ZName ProcessDef*
| *CParamProcess ZName ZName list ProcessDef*

```
        | CGenProcess ZName ZName list ProcessDef
and   ProcessDef = ProcDefSpot ZGenFilt list ProcessDef
        | ProcDef CProc
and   CProc = ProcMain ZPara PPar list CAction
        | ProcStalessMain PPar list CAction
and   PPar = ProcZPara ZPara
        | CParAction ZName ParAction
        | CNameSet ZName NSExp

type-synonym ZSpec = ZPara list

type-synonym CProgram = ZPara list

end
```

2 Auxiliary Definitions for Omega functions

```
theory OmegaDefs imports AST Prelude begin
```

Concatenation of two names - Used for the CSP renaming of Process+StateVariables

```
fun join-name
where
  join-name n v = (n @ (''-'' @ v))
```

2.1 Manipulating Lists

```
fun inListVar
where
  inListVar x Nil = False
| inListVar x [va] = (case x = va of
                        True ⇒ True
                      | - ⇒ False)
| inListVar x (va # vst) = (case x = va of
                              True ⇒ True
                            | - ⇒ inListVar x vst)


fun delete-from-list
where
  delete-from-list x Nil = Nil
| delete-from-list x [v] = (case x = v of
                              True ⇒ Nil
                            | False ⇒ [v])
| delete-from-list x (v # va) = (case x = v of
                                   True ⇒ delete-from-list x va
                                 | False ⇒ (v # (delete-from-list x va)))


fun setminus
```

```
where
  setminus Nil - = Nil
| setminus (v # va) Nil = (v # va)
| setminus (v # va) (b # vb) = ((delete-from-list b (v # va)) @ (setminus (v #
va) vb))


fun member
where
  member x Nil = False
| member x (b # y) = (if x = b then True else member x y)

fun intersect
where
  intersect Nil y = Nil
| intersect (a # x) y = (if member a y then a # (intersect x y)
                         else intersect x y)

fun union
where
  union Nil y = y
| union (a # x) y = (if (member a y) then (union x y)
                     else a # (union x y))


fun elem-by :: 'a ⇒ 'a list ⇒ bool
where
  elem-by - Nil = False
| elem-by y (x # xs) = (y = x | elem-by y xs)


fun isPrefixOf
where
  isPrefixOf Nil - = True
| isPrefixOf - Nil = False
| isPrefixOf (x # xs) (y # ys) = (x = y & isPrefixOf xs ys)

fun remdups
where
  remdups Nil = Nil
| remdups (x # xs) = (if (member x xs) then remdups xs
                      else x # remdups xs)

fun subset
where
  subset xs ys = list-all (% arg0 . member arg0 ys) xs
```

2.2 Free Variables functions

fun *free-var-ZGenFilt*
where
free-var-ZGenFilt (*Choose v e*) = [*v*]
| *free-var-ZGenFilt* (*Check p*) = *Nil*
| *free-var-ZGenFilt* (*Evaluate v e1 e2*) = *Nil*

fun *free-var-ZPred* :: *ZPred* ⇒ *ZVar list*
where
free-var-ZPred (*ZFalse p*) = *Nil*
| *free-var-ZPred* (*ZTrue p*) = *Nil*
| *free-var-ZPred* (*ZAnd a b*) = (*free-var-ZPred a* @ *free-var-ZPred b*)
| *free-var-ZPred x* = *Nil*

fun *fvs*
where
fvs f Nil = *Nil*
| *fvs f* (*e* # *es*) = (*f e* @ (*fvs f es*))

function (*sequential*) *free-var-ZExpr* :: *ZExpr* ⇒ *ZVar list*
where
free-var-ZExpr (*ZVar v*) = [*v*]
| *free-var-ZExpr* (*ZInt c*) = *Nil*
| *free-var-ZExpr* (*ZSetDisplay exls*) = *fvs free-var-ZExpr exls*
| *free-var-ZExpr* (*ZSeqDisplay exls*) = *fvs free-var-ZExpr exls*
| *free-var-ZExpr* (*ZCall ex ex2*) = *free-var-ZExpr ex2*
| *free-var-ZExpr* - = *Nil*
by *pat-completeness auto*

fun *free-var-CAction* :: *CAction* ⇒ *ZVar list*
where
free-var-CAction CSPSkip = *Nil*
| *free-var-CAction CSPStop* = *Nil*
| *free-var-CAction CSPChaos* = *Nil*
| *free-var-CAction* (*CSPSeq ca cb*) = ((*free-var-CAction ca*) @ (*free-var-CAction cb*))
| *free-var-CAction* (*CSPExtChoice ca cb*) = ((*free-var-CAction ca*) @ (*free-var-CAction cb*))
| *free-var-CAction* (*CSPCommAction v va*) = *Nil*

2.3 Production of Get and Set

We have to create signals with gets and sets carrying the values of the state variables before an action can occur when we translate from state-rich Circus processes into stateless ones.

```
fun make-get-com :: ZName list ⇒ CAction ⇒ CAction
where
  make-get-com [x] c = (CSPCommAction (ChanComm ''mget'' [ChanDotExp
(ZVar (x, Nil)), ChanInp (''v-'' @ x)]) c)
| make-get-com (x # xs) c = (CSPCommAction (ChanComm ''mget'' [ChanDotExp
(ZVar (x, Nil)), ChanInp (''v-'' @ x)]) (make-get-com xs c))
| make-get-com x c = c


fun make-set-com :: (CAction ⇒ CAction) ⇒ ZVar list ⇒ ZExpr list ⇒ CAction
⇒ CAction
where
 make-set-com f [(x, -)] [y] c = (CSPCommAction (ChanComm ''mset'' [ChanDotExp
(ZVar (x, Nil)), ChanOutExp y]) (f c))
| make-set-com f ((x, -) # xs) (y # ys) c = (CSPCommAction (ChanComm
''mset'' [ChanDotExp (ZVar (x, Nil)), ChanOutExp y]) (make-set-com f xs ys c))
| make-set-com f - - c = (f c)
```

2.4 Production of WrtV

```
fun getWrtV
where
  getWrtV xs = Nil
```

2.5 Renaming Variables

```
fun get-ZVar-st
where
  get-ZVar-st (a, x) = (case (isPrefixOf ''st-var-'' a) of
                          True ⇒ [a]
                        | False ⇒ Nil)
fun is-ZVar-st
where
  is-ZVar-st a = isPrefixOf ''st-var-'' a

fun rename-ZPred
where
  rename-ZPred (ZFalse a) = (ZFalse a)
| rename-ZPred (ZTrue a) = (ZTrue a)
| rename-ZPred (ZAnd p1 p2) = (ZAnd (rename-ZPred p1) (rename-ZPred p2))
| rename-ZPred (ZPSchema sp) = (ZPSchema sp)

fun rename-ZExpr
where
  rename-ZExpr (ZVar (va, x)) = (case (is-ZVar-st va) of
                                   True ⇒ (ZVar (''v-'' @ va, x))
                                 | False ⇒ (ZVar (va, x)))
| rename-ZExpr (ZInt zi) = (ZInt zi)
| rename-ZExpr (ZCall xpr1 xpr2) = (ZCall (rename-ZExpr xpr1) (rename-ZExpr
xpr2))
```

```
| rename-ZExpr x = x


fun bindingsVar
where
  bindingsVar Nil = Nil
| bindingsVar [((va, x), b)] = (case (is-ZVar-st va) of
                                  True ⇒ [((''v-'' @ va, x), (rename-ZExpr b))]
                                | False ⇒ [((va, x), (rename-ZExpr b))])
| bindingsVar (((va, x), b) # xs) = (case (is-ZVar-st va) of
                                       True ⇒ [((''v-'' @ va, x), (rename-ZExpr b))] @
(bindingsVar xs)
                                     | False ⇒ [((va, x), (rename-ZExpr b))] @
(bindingsVar xs))


fun rename-vars-CAction
where
  rename-vars-CAction CSPSkip = CSPSkip
| rename-vars-CAction CSPStop = CSPStop
| rename-vars-CAction CSPChaos = CSPChaos
| rename-vars-CAction (CSPSeq a1 a2) = (CSPSeq (rename-vars-CAction a1)
(rename-vars-CAction a2))
| rename-vars-CAction (CSPExtChoice a1 a2) = (CSPExtChoice (rename-vars-CAction
a1) (rename-vars-CAction a2))
| rename-vars-CAction x = x



fun getFV
where
  getFV xs = Nil


end
```

3 Omega Function Definitions

```
theory MappingFunStatelessCircus imports AST OmegaDefs Prelude begin

fun omega-CAction :: CAction ⇒ CAction and
    omega-prime-CAction :: CAction ⇒ CAction
where
  omega-CAction CSPSkip = CSPSkip
| omega-CAction CSPStop = CSPStop
| omega-CAction CSPChaos = CSPChaos
| omega-CAction (CSPSeq ca cb) = (CSPSeq (omega-CAction ca) (omega-CAction
cb))
| omega-CAction (CSPExtChoice ca cb) = (let lsx = (map fst (remdups (free-var-CAction
```

```
(CSPExtChoice ca cb))))
                                        in make-get-com lsx (rename-vars-CAction
(CSPExtChoice (omega-prime-CAction ca) (omega-prime-CAction cb))))
| omega-CAction x = x
| omega-prime-CAction CSPSkip = CSPSkip
| omega-prime-CAction CSPStop = CSPStop
| omega-prime-CAction CSPChaos = CSPChaos
| omega-prime-CAction (CSPSeq ca cb) = (CSPSeq (omega-prime-CAction ca)
(omega-prime-CAction cb))
| omega-prime-CAction x = omega-CAction x


end
```