Circus to CSP

Artur Oliveira Gomes

June 26, 2017

Contents

1 Abstract Syntax Trees

Both Z and Circus AST are found here.

```
module AST where
```

```
--
-- $Id: AST.hs,v 1.58 2005-03-26 13:07:43 marku Exp $
--
-- This module defines Abstract Syntax Trees for Z terms
-- (expressions, predicates, schemas etc.).
-- These abstract syntax trees are also used for the *result* of
-- evaluating Z terms.
--
-- There are often several semantically equivalent data structures for
```

```
-- representing a given result, each with different space usage and ability
-- to perform various operations efficiently.  For example, the result of
-- evaluating a set comprehension expression (of type \power \ints) could
-- be represented by several data structures, including:
--
--       ZIntSet (Just lo) (Just hi)                      (= lo .. hi)
--       ZFSet s                      (s is defined in FiniteSets.lhs)
--       ZSetDisplay [ZInt 3, ZInt 4, complex_int_expr]
--
-- The ZIntSet one is best for contiguous ranges of integers and can even
-- handle infinite ranges (a missing endpoint); the ZFSet one is only
-- used when all elements are defined and in canonical form -- it keeps
-- elements in strictly sorted order so that common set operations can be
-- done in linear time; The ZSetDisplay structure is used for finite sets
-- that contain complex (non-canonical) elements (for example the above
-- ZSetDisplay may contain two or three elements, depending upon whether
-- the 'complex_int_expr' evaluates to 3 or 4 or something else).
--
-- Evaluation functions may use different strategies for each data
-- structure, or may coerce a given structure into their favourite.
--
-- Haskell defines == (and <, > etc.) over ZExpr structures, but this
-- is not always the same as semantic equality (=).  Eg. Is this true?
--
--             a==b   =>  a=b
--
-- According to Spivey and the Z standard, not always!  If a or b
-- is undefined, then the truth value of a=b is unknown.
-- Even more commonly, the converse is not always true, because several
-- different data structures may represent the same value.  However, when
-- both a and b are in 'canonical' (see isCanonical below) form, we have:
--
--             a==b   <=>   a=b.
--
-- Intuitively, any ZExpr that is constructed entirely from the following
-- constructors must be in a unique canonical form:
--
--    ZInt, ZGiven, ZTuple, ZFree0, ZFree1, ZFSet, ZBinding.
--
-- Free types are represented as follows.
-- Given a typical free type:   CList ::= nil | cons <<C x CList>>,
-- T is represented by the data structure:
--
--  d = ZFreeType clist
--        [ZBranch0 nil,
--         ZBranch1 cons (ZCross (...C...) (ZVar clist))]
--
-- where nil=("nil",[]), cons=("cons",[]), clist=("CList",[]).
-- Note how the first argument to ZFreeType supports recursive references.
-- After the 'unfold' stage, free types never contain any free variables.
--
-- Members of this free type are represented as:
--
--      nil       is ZFree0 nil
--
--      cons      is the function (\lambda x: c \cross d @ Free1 cons x)
--                (functions are actually represented as a ZSetComp term)
--
--      cons val  is ZFree1 cons val   (if val is in C \cross CList)
--                                     (otherwise it will be undefined)
--
-- where x is some local ZVar, c is the representation of type C
-- and d is given above.  In other words, (ZBranch0 nil) represents
-- the singleton set:   { ZFree0 nil }
```

```
--
-- Invariants
-- ==========
-- Here are the main invariants of these data structures:
--
-- * ZTuple and ZCross always have at least two members in their arg list.
-- * ZFSet only contains canonical values.
-- * If ZIntSet has both an upper and lower bound, then the lower bound
--   should be no greater than the upper.  (In fact, the empty set case
--   is normally represented as 'zemptyset', below).
-- * An empty set can be represented in many ways, but the preferred
--   representation is 'zemptyset', below).
-- * All manipulations of the argument of ZFSet should be done via
--   functions in the FiniteSets module (in case the representation
--   of those finite sets changes in the future).  Construction of a
--   new finite set should normally be done via FiniteSets.make_zfset.
--   (it will return ZSetDisplay instead if some members are not canonical).
-- * The (name,value) pairs of ZBinding terms are always sorted in
--   increasing alphabetically order, with no duplicate names.
-- * The Maybe parts of ZSetComp and ZMu are always filled in
--   after the unfold phase.  That is, they are not 'Nothing'.
-- * All schema expressions are removed during the Unfold phase.
```

1.1 Z Abstract Syntax

1.1.1 Z Given Sets

```
type GivenSet = ZVar       -- names of given sets.
type GivenValue = String   -- members of given sets are strings
type ZInt = Integer        -- If you change this, you must also change
        -- the definition of L_NUMBER in Lexer.hs
type ZFSet = [ZExpr]       -- But always manipulate via FiniteSets functions.
```

TODO: Make this a separate module, perhaps combined with `VarSet`.

1.1.2 Z Names and Decorations

```
type ZDecor = String       -- a decoration: ''', '!', '?' or '_N'
type ZVar = (String, [ZDecor]) -- all kinds of Z names
type ZName = String

make_zvar :: String -> [ZDecor] -> ZVar
make_zvar s dl = (s,dl)

decorate_zvar :: ZVar -> [ZDecor] -> ZVar
decorate_zvar (s,dl) d = (s,dl++d)

prime_zvar :: ZVar -> ZVar
prime_zvar v = decorate_zvar v ["'"]

unprime_zvar :: ZVar -> ZVar
-- Pre: is_primed_zvar v
unprime_zvar (n,["'"]) = (n,[])

string_to_zvar :: String -> ZVar
string_to_zvar s = make_zvar s []

get_zvar_name :: ZVar -> String
get_zvar_name = fst

get_zvar_decor :: ZVar -> [ZDecor]
get_zvar_decor = snd

is_unprimed_zvar :: ZVar -> Bool
is_unprimed_zvar (_,[])  = True
is_unprimed_zvar _       = False
```

```haskell
is_primed_zvar :: ZVar -> Bool
is_primed_zvar (_,["'"]) = True
is_primed_zvar _          = False

is_input_zvar :: ZVar -> Bool
is_input_zvar (_,["?"])  = True
is_input_zvar _          = False

is_output_zvar :: ZVar -> Bool
is_output_zvar (_,["!"]) = True
is_output_zvar _         = False


show_zvar :: ZVar -> String
show_zvar (s,dl) = s ++ concat dl

show_zvars :: [ZVar] -> String
show_zvars = concatMap ((' ':) . show_zvar)
```

1.1.3 Z Relations and Functions

```haskell
data ZReln   -- binary toolkit relations (all take one arg: a pair)
  = ZLessThan         -- 3 < 4
  | ZLessThanEq       -- 3 \leq 3
  | ZGreaterThan      -- 4 > 3
  | ZGreaterThanEq    -- 4 \geq 4
  | ZSubset           -- {1,2} \subset {1,2,4}
  | ZSubsetEq         -- {1,2} \subseteq {1,2}
  | ZPartition        -- {(1,{1,3}),(4,{2,4})} \partition 1..4
  | ZPrefix           -- <1,2> \prefix <1,2,3,4>
  | ZSuffix           -- <2,3> \suffix <0,1,2,3>
  | ZInSeq            -- <2,3> \inseq  <0,1,2,3,4,5>
  -- These next two should only be used within the Pretty Printer.
  -- E.g. The parser expands a \neq b into (ZNot (ZEqual a b))
  --      and that form is always used internally.
  | ZNeq
  | ZNotin
  deriving (Eq,Ord,Show)

data ZFunc1   -- prefix and postfix unary functions
        -- (These all take an argument that is not a pair)
  = ZDom      -- \dom
  | ZRan      -- \ran
  | ZSizeof   -- slash hash-symbol
  | ZBigCup   -- \bigcup
  | ZBigCap   -- \bigcap
  | ZId       -- \id     -- changed into ZSetComp by Unfold.hs
  | ZRev      -- rev
  | ZHead     -- head
  | ZLast     -- last
  | ZTail     -- tail
  | ZFront    -- front
  | ZSquash   -- squash
  | ZDCat     -- \dcat
  | ZSucc     -- succ    -- changed into ZSetComp by Unfold.hs
  | ZNegate   -- '-'
  | ZMax      -- max
  | ZMin      -- min
  | ZInv      -- '~'
  | ZStar     -- '*'
  | ZClosure  -- '+'
  | ZSum      -- an extension for 424 module 3.
  deriving (Eq,Ord,Show)


data ZFunc2   -- binary functions that take one argument: a pair
  = ZMapsto   -- \mapsto   (unfoldexpr converts this into a pair
```

```
  -- Integer operations
  | ZUpto     -- \upto
  | ZPlus     -- +
  | ZMinus    -- '-'
  | ZTimes    -- *
  | ZDiv      -- \div
  | ZMod      -- \mod
  -- Set operations
  | ZUnion    -- \cup
  | ZInter    -- \cap
  | ZSetMinus-- '\'
  -- Relation/Function operations
  | ZComp     -- \comp     (relation composition)
  | ZCirc     -- \circ     (backward relation composition)
  | ZDRes     -- \dres
  | ZRRes     -- \rres
  | ZNDRes    -- \ndres
  | ZNRRes    -- \nrres
  | ZRelImg   -- _ \limg _ \rimg
  | ZOPlus    -- \oplus    (function/relation overriding)
  -- Sequence operations
  | ZCat      -- \cat      sequence concatenation
  | ZExtract  -- \extract  = \squash (A \dres Seq)
  | ZFilter   -- \filter   = \squash (Seq \rres A)
  -- These two are not syntactically binary functions, but semantically
  -- they behave as though they are, because they take a pair as an argument.
  | ZFirst    -- first
  | ZSecond   -- second
  deriving (Eq,Ord,Show)

data ZStrange -- toolkit functions/sets that defy categorization!
  = ZIter      -- iter n R (or R^n) is curried: takes two arguments.
  | ZDisjoint -- is a set of functions of type: Index \pfun \power Elem
  deriving (Eq,Ord,Show)
```

1.1.4 Z Generators and Filters

These 'Generator or Filter' terms are used to represent the search space within quantifiers, set comprehensions, schemas. All (Include ...) terms should be expanded out before being passed to the eval... functions.

The scope of declared names is complex here. Immediately after parsing, the usual Z scope rules apply. That is, in $[x : T;\ y : U;\ P;\ Q]$ the scope of x and y includes any predicates such as P and Q, but excludes all types, T and U. This allows signatures (declarations) to be reordered with impunity.

AFTER the unfold and uniquify stages (see Unfold.hs), the scope rules are basically left to right. A variable x is in scope immediately AFTER its declaration. Note that in 'Choose x t', the t is not in the scope of the newly declared x, but following predicates and declarations are in the scope of x. Similarly for 'Evaluate x e t' — e and t are outside the scope of x. This means that one must be careful when reordering elements of a [GenFilt] not to move terms further left than the declarations the their free variables.

Note: to implement these scoping rules, a common trick that we use in several places (eg. `Eval::gen_and_filter`) is to pass around TWO environments as we recurse through a [ZGenFilt]. One environment is the environment from outside the whole list, and is used to evaluate/manipulate the type expressions, while the other environment is the internal one (which is extended as we go left to right into the list) and is used on the other expressions and predicates.

```haskell
data ZGenFilt
  = Include ZSExpr      -- Schema inclusion
  | Choose ZVar ZExpr   -- (Choose x T) means x:T
  | Check ZPred
  | Evaluate ZVar ZExpr ZExpr -- This means Let x==e | e \in t
  deriving (Eq,Ord,Show)


genfilt_names :: [ZGenFilt] -> [ZVar]
genfilt_names []                     = []
genfilt_names (Choose v _:gfs)       = v : genfilt_names gfs
genfilt_names (Check _:gfs)          = genfilt_names gfs
genfilt_names (Evaluate v _ _:gfs) = v : genfilt_names gfs
genfilt_names (Include s:gfs)
  = error ("genfilt_names called before "++show s++" expanded.")
```

1.1.5 Z Expressions

```haskell
data ZExpr
  = ---------- Basic Z values (non-set values) ----------
    ZVar ZVar             -- for non-schema names (may include decorations)
  | ZInt ZInt             -- an integer constant
  | ZGiven GivenValue     -- an element of a given set
  | ZFree0 ZVar           -- a member of a free type.
  | ZFree1 ZVar ZExpr     -- a member of a free type (with an argument)
  | ZTuple [ZExpr]        -- (a,b,c)
  | ZBinding [(ZVar,ZExpr)] -- always in sorted name order (no duplicates)
  ---------- Data structures for sets ----------
  -- These are roughly ordered by how 'large' a set they typically represent.
  | ZSetDisplay [ZExpr]    -- set displays, like {1,2,4}
  | ZSeqDisplay [ZExpr]    -- sequence displays, like <1,2,4>
  | ZFSet ZFSet            -- all elements must be in canonical form.
  | ZIntSet (Maybe ZInt) (Maybe ZInt) -- integer range with lo/hi bounds.
      --   ZIntSet (Just lo) (Just hi) means lo..hi.
      --   ZIntSet Nothing   (Just hi) means -infinity..hi.
      --   ZIntSet (Just lo) Nothing   means lo..+infinity.
      --   ZIntSet Nothing   Nothing   means \num
  | ZGenerator ZReln ZExpr -- sets that are useful for iterating through.
      -- ZGenerator r e = { x:ZUniverse | x rel e }
  | ZCross [ZExpr]          -- a \cross b \cross c
  | ZFreeType ZVar [ZBranch] -- an entire free type (all branches)
  | ZPowerSet{baseset::ZExpr, -- power set types
      is_non_empty::Bool,
      is_finite::Bool}
  | ZFuncSet{ domset::ZExpr, -- relation/function/sequence types
      ranset::ZExpr,
      is_function::Bool,
      is_total::Bool,          -- dom R = domset
      is_onto::Bool,           -- ran R = ranset
      is_one2one::Bool,        -- injective
```

```
        is_sequence::Bool,     -- dom is 1.. length s
        is_non_empty::Bool,
        is_finite::Bool}
  | ZSetComp [ZGenFilt] (Maybe ZExpr) -- set comprehensions
  | ZLambda [ZGenFilt] ZExpr          -- only for parsing (removed in Unfold)
  | ZESchema ZSExpr                   -- sets of bindings (removed in Unfold)
  | ZGivenSet GivenSet                -- an entire given set
  | ZUniverse                 -- the set of all Z values! (a unit for \cap)
  ---------- Z constructs that are not necessarily sets ----------
  | ZCall ZExpr ZExpr                 -- function call:  f a
  | ZReln ZReln                       -- binary toolkit relations
  | ZFunc1 ZFunc1                     -- unary toolkit functions
  | ZFunc2 ZFunc2                     -- binary toolkit functions
  | ZStrange ZStrange                 -- miscellaneous toolkit functions/sets.
  | ZMu [ZGenFilt] (Maybe ZExpr)      -- mu expression
  | ZELet [(ZVar,ZExpr)] ZExpr        -- let a=1;b=2 in... (removed in Unfold)
  | ZIf_Then_Else ZPred ZExpr ZExpr   -- if p then e1 else e2
  | ZSelect ZExpr ZVar                -- e.field
  | ZTheta ZSExpr                     -- \theta S (removed in Unfold)
  deriving (Eq,Ord,Show)
```

ZValue is a synonym for ZExpr, but is used for the result of evaluations, where the last group of ZExpr alternatives above are the most common kinds of results.

```
type ZValue = ZExpr
is_pair :: ZValue -> Bool
is_pair (ZTuple [_,_]) = True
is_pair _              = False

pair_fst :: ZValue -> ZValue
pair_fst (ZTuple [x,_]) = x
pair_fst _ = error "pair_fst applied to non-pair value"

pair_snd :: ZValue -> ZValue
pair_snd (ZTuple [_,y]) = y
pair_snd _ = error "pair_snd applied to non-pair value"

isZFSet :: ZExpr -> Bool
isZFSet (ZFSet _) = True
isZFSet _         = False

-- This is equivalent to (ZFSet FiniteSets.emptyset), but
-- for convenience we define it directly here.
zemptyset :: ZExpr
zemptyset = ZFSet []

-- This is the union of all Z relations:  ZUniverse <-> ZUniverse
zrelations :: ZExpr
zrelations = ZFuncSet{domset=ZUniverse,
          ranset=ZUniverse,
          is_function =False,
          is_total    =False,
          is_onto     =False,
          is_one2one  =False,
          is_sequence =False,
          is_non_empty=False,
          is_finite   =False}
```

1.1.6 Z Predicates

```
data ZPred
  = ZFalse{reason::[ZPred]}
  | ZTrue{reason::[ZPred]}
  | ZAnd ZPred ZPred
  | ZOr ZPred ZPred
  | ZImplies ZPred ZPred
  | ZIff ZPred ZPred
  | ZNot ZPred
  | ZExists [ZGenFilt] ZPred
```

```
  | ZExists_1 [ZGenFilt] ZPred
  | ZForall [ZGenFilt] ZPred
  | ZPLet [(ZVar,ZExpr)] ZPred  -- removed in Unfold
  | ZEqual ZExpr ZExpr
  | ZMember ZExpr ZExpr
  | ZPre ZSExpr                 -- removed in Unfold
  | ZPSchema ZSExpr             -- removed in Unfold
  deriving (Eq,Ord,Show)

ztrue = ZTrue{reason=[]}
zfalse = ZFalse{reason=[]}
```

1.1.7 Z Schemas

```
data ZSExpr
  = ZSchema [ZGenFilt]
  | ZSRef ZSName [ZDecor] [ZReplace]
  | ZS1 ZS1 ZSExpr              -- unary schema operators
  | ZS2 ZS2 ZSExpr ZSExpr       -- binary schema operators
  | ZSHide ZSExpr [ZVar]
  | ZSExists [ZGenFilt] ZSExpr
  | ZSExists_1 [ZGenFilt] ZSExpr
  | ZSForall [ZGenFilt] ZSExpr
  deriving (Eq,Ord,Show)

-- Note that any legal list of ZReplace's must not contain any repeated
-- first-argument ZVars.  Eg [a/b,a/c] is legal, but [b/a,c/a] is not.
-- When renaming causes names to be merged, the merged names must have
-- the same type.
data ZReplace
  = ZRename ZVar ZVar           -- S [yi / xi] = ZRename (ZVar xi []) (ZVar yi [])
  | ZAssign ZVar ZExpr          -- S [xi := 3] = ZAssign (ZVar xi []) (Int 3)
  deriving (Eq,Ord,Show)

data ZSName                     -- schema names including prefix.
  = ZSPlain String | ZSDelta String | ZSXi String
  deriving (Eq,Ord,Show)

data ZS1
  = ZSPre | ZSNot
  deriving (Eq,Ord,Show)

data ZS2
  = ZSAnd | ZSOr | ZSImplies | ZSIff
  | ZSProject | ZSSemi | ZSPipe
  deriving (Eq,Ord,Show)
```

1.1.8 Z Paragraphs

```
data ZPara
  = ZGivenSetDecl GivenSet        -- [XXX]
  | ZSchemaDef ZSName ZSExpr      -- \begin{schema}{XXX}...\end{schema}
                                  -- or XXX \defs [...|...]
  | ZAbbreviation ZVar ZExpr      -- XXX == expression
  | ZFreeTypeDef ZVar [ZBranch]   -- XXX ::= A | B | ...
  | ZPredicate ZPred
  | ZAxDef [ZGenFilt]             -- \begin{axdef}...\end{axdef}
  | ZGenDef [ZGenFilt]            -- \begin{gendef}...\end{gendef}
  | ZMachineDef{machName::String,   -- a state machine.
    machState::String,
    machInit::String,
    machOps::[String]}
    -- Inclusion of Circus Paragraphs
  | CircChannel [CDecl]         -- \circchannel CDecl
  | CircChanSet ZName CSExp     -- \circchanset N == CSExp
  | Process ProcDecl            -- ProcDecl
  deriving (Eq,Ord,Show)
```

```
data ZBranch                     -- E.g. given T ::= A | C <<N x T>>
  = ZBranch0 ZVar                -- the A branch is: ZBranch0 ("A",[])
  | ZBranch1 ZVar ZExpr          -- and C branch is: ZBranch1 ("C",[]) (ZCross [...])
  deriving (Eq,Ord,Show)

isBranch0 :: ZBranch -> Bool
isBranch0 (ZBranch0 _) = True
isBranch0 _            = False

type ZSpec = [ZPara]
```

Any `ZExpr`/`ZValue` that satisfies 'isCanonical' is fully evaluated into a unique form. For such terms, `==` is equivalent to semantic equality.

```
isCanonical :: ZExpr -> Bool
isCanonical (ZInt _)       = True
isCanonical (ZFSet _)      = True  -- an invariant of the system
isCanonical (ZTuple v)     = all isCanonical v
isCanonical (ZGiven _)     = True
isCanonical (ZFree0 _)     = True
isCanonical (ZFree1 _ v)   = isCanonical v
isCanonical (ZBinding bs) = all (isCanonical . snd) bs
isCanonical _              = False
```

`isDefined e` is true when e is obviously well defined (though it may be too big to compute). Any canonical value is defined, but so are some infinite sets like ℕ: `(ZIntSet (Just 0) Nothing)` When `isDefined` is false, the term may still be defined. NOTE: `isDefined` ignores type correctness. E.g. $\{1, \{1\}\}$ is treated as being defined.

```
isDefined :: ZExpr -> Bool
isDefined (ZInt _)          = True
isDefined (ZIntSet _ _)     = True
isDefined (ZFSet _)         = True  -- an invariant of the system
isDefined (ZTuple v)        = all isDefined v
isDefined (ZReln _)         = True
isDefined (ZGiven _)        = True
isDefined (ZGivenSet _)     = True
-- could add some toolkit functions here (at least the non-generic ones).
isDefined (ZSetDisplay vs)= all isDefined vs
isDefined (ZSeqDisplay vs)= all isDefined vs
isDefined (ZFree0 _)        = True
isDefined (ZFree1 _ _)      = True   -- Note (1)
isDefined (ZBinding bs)     = all (isDefined . snd) bs
isDefined v                 = False
```

Note 1: `ZFree1` terms initially only appear as the body of lambda terms. The reduction of those lambda terms checks domain membership, which includes proving definedness. So any standalone `ZFree1` term must be defined.

2 Circus Abstract Syntax

```
--------------------------------------
-------------   Circus   -------------
--    Artur Oliveira - May 2016     --
--------------------------------------
```

2.0.1 Circus Program

```
type CProgram = [ZPara]

--
-- Channel and chanset decl
--
data CDecl
  = CChan ZName                       --  no type is defined
  | CChanDecl ZName ZExpr             -- channel_name : type
  | CGenChanDecl ZName ZName ZExpr    -- generic chan decl
  deriving (Eq,Ord,Show)
```

```
{-
Channel Schema declaration is left out for now, but
one could declare it as:

\begin{schema}
\circchan c1:\nat \\
\circchan c2:\nat \\
\end{schema}

and therefore, would need to define it in terms of the Z parser
-- | SchemaExp -- left out for now

-}
```

2.0.2 Circus Channel Expression

```
data CSExp
  = CSExpr ZName                          -- a chanset decl from another chanset
  | CSEmpty                               -- Empty chanset
  | CChanSet [ZName]                      -- named chanset
  | ChanSetUnion CSExp CSExp              -- chanset union
  | ChanSetInter CSExp CSExp              -- chanset intersection
  | ChanSetDiff CSExp CSExp               -- chanset hidding chanset
  deriving (Eq,Ord,Show)
```

2.0.3 Circus Process

```
data ProcDecl
  = CProcess ZName ProcessDef             -- \circprocess N \circdef ProcDef
  | CParamProcess ZName [ZName] ProcessDef  -- \circprocess N[N^{+}] \circdef ProcDef
  | CGenProcess ZName [ZName] ProcessDef  -- \circprocess N[N^{+}] \circdef ProcDef
  deriving (Eq,Ord,Show)

data ProcessDef
  = ProcDefSpot [ZGenFilt] ProcessDef     -- Decl \circspot ProcDef
  | ProcDefIndex [ZGenFilt] ProcessDef    -- Decl \circindex ProcDef
  | ProcDef CProc                         -- Proc
  deriving (Eq,Ord,Show)

data CProc
  = CRepSeqProc [ZGenFilt] CProc          -- \Semi Decl \circspot Proc
  | CRepExtChProc [ZGenFilt] CProc        -- \Extchoice Decl \circspot Proc
  | CRepIntChProc [ZGenFilt] CProc        -- \IntChoice Decl \circspot Proc
  | CRepParalProc CSExp [ZGenFilt] CProc  -- \lpar CSExp \rpar Decl \circspot Proc
  | CRepInterlProc [ZGenFilt] CProc       -- \Interleave Decl \circspot Proc
  | CHide CProc CSExp                     -- Proc \circhide CSExp
  | CExtChoice CProc CProc                -- Proc \extchoice Proc
  | CIntChoice CProc CProc                -- Proc \intchoice Proc
  | CParParal CSExp CProc CProc           -- Proc \lpar CSExp \rpar Proc
  | CInterleave CProc CProc               -- Proc \interleave Proc
  -- | ChanProcDecl CDecl ProcessDef [ZExpr]  -- (Decl \circspot ProcDef)(Exp^{+})
  | CGenProc ZName [ZExpr]                -- N[Exp^{+}]
  | CParamProc ZName [ZExpr]             -- N(Exp^{+})
  -- | CIndexProc [ZGenFilt] ProcessDef   -- \(Decl \circindex ProcDef) \lcircindex Exp^{+} \rcirc
-- TODO
  | CProcRename ZName [Comm] [Comm]       -- Proc[N^{+}:=N^{+}] -- TODO
  | CSeq CProc CProc                      -- Proc \cirCSeq Proc
  | CSimpIndexProc ZName [ZExpr]          -- N\lcircindex Exp^{+} \rcircindex
  | CircusProc ZName                      -- N
  | ProcMain ZPara [PPar] CAction         -- \circbegin PPar*
                                          --   \circstate SchemaExp PPar*
                                          --   \circspot Action
                                          --   \circend
  | ProcStalessMain [PPar] CAction        -- \circbegin PPar*
                                          --   \circspot Action
                                          --   \circend
 deriving (Eq,Ord,Show)
```

2.0.4 Circus Name-Sets

```
data NSExp
  = NSExpEmpty                          -- \{\}
  | NSExprMult [ZName]                  -- \{N^{+}\}
  | NSExprSngl ZName                    -- N
  | NSExprParam ZName [ZExpr]           -- N(Exp)
  | NSUnion NSExp NSExp                 -- NSExp \union NSExp
  | NSIntersect NSExp NSExp             -- NSExp \intersect NSExp
  | NSHide NSExp NSExp                  -- NSExp \circhide \NSExp
  | NSBigUnion ZExpr
  deriving (Eq,Ord,Show)
```

2.0.5 Circus Actions

```
data PPar
 = ProcZPara ZPara                      -- Par
 | CParAction ZName ParAction           -- N \circdef ParAction
 | CNameSet ZName NSExp                 -- \circnameset N == NSExp
 deriving (Eq,Ord,Show)

data ParAction
 = CircusAction CAction                              -- Action
 | ParamActionDecl [ZGenFilt] ParAction -- Decl \circspot ParAction
 deriving (Eq,Ord,Show)

data CAction
 = CActionSchemaExpr ZSExpr            -- \lschexpract S \rschexpract
 | CActionCommand CCommand
 | CActionName ZName
 | CSPSkip | CSPStop | CSPChaos
 | CSPCommAction Comm CAction          -- Comm \then Action
 | CSPGuard ZPred CAction              -- Pred \circguard Action
 | CSPSeq CAction CAction              -- Action \circseq Action
 | CSPExtChoice CAction CAction        -- Action \extchoice Action
 | CSPIntChoice CAction CAction        -- Action \intchoice Action
 | CSPNSParal NSExp CSExp NSExp CAction CAction -- Action \lpar NSExp | CSExp | NSExp \rpar Action
 | CSPParal CSExp CAction CAction      -- Action \lpar CSExp \rpar Action
 | CSPNSInter NSExp NSExp CAction CAction -- Action \linter NSExp | NSExp \rinter Action
 | CSPInterleave CAction CAction       -- Action \interleave Action
 | CSPHide CAction CSExp               -- Action \circhide CSExp
 | CSPParAction ZName [ZExpr]          -- Action(Exp^{+})
 | CSPRenAction ZName CReplace         -- Action[x/y,z/n]
 | CSPRecursion ZName CAction          -- \circmu N \circspot Action
 | CSPUnfAction ZName CAction          -- N (Action)
 | CSPUnParAction [ZGenFilt] CAction ZName     -- (Decl \circspot Action) (ZName)
 | CSPRepSeq [ZGenFilt] CAction        -- \Semi Decl \circspot Action
 | CSPRepExtChoice [ZGenFilt] CAction  -- \Extchoice Decl \circspot Action
 | CSPRepIntChoice [ZGenFilt] CAction  -- \IntChoice Decl \circspot Action
 | CSPRepParalNS CSExp [ZGenFilt] NSExp CAction -- \lpar CSExp \rpar Decl \circspot \lpar NSExp \rp
 | CSPRepParal CSExp [ZGenFilt] CAction  -- \lpar CSExp \rpar Decl \circspot ction
 | CSPRepInterlNS [ZGenFilt] NSExp CAction  -- \Interleave Decl \circspot \linter NSExp \rinter Act
 | CSPRepInterl [ZGenFilt] CAction     -- \Interleave Decl \circspot  Action
  deriving (Eq,Ord,Show)
```

2.0.6 Circus Communication

```
data Comm
  = ChanComm ZName [CParameter]          -- N CParameter*
  | ChanGenComm ZName [ZExpr] [CParameter]-- N [Exp^{+}] CParameter *
  deriving (Eq,Ord,Show)


data CParameter
   = ChanInp ZName                       -- ?N
   | ChanInpPred ZName ZPred             -- ?N : Pred
   | ChanOutExp ZExpr                    -- !Exp
```

```
  | ChanDotExp ZExpr                      -- .Exp
  deriving (Eq,Ord,Show)
```

2.0.7 Circus Commands

```
data CCommand
  = CAssign [ZVar] [ZExpr]                -- N^{+} := Exp^{+}
  | CIf CGActions                         -- \circif GActions \cirfi
  | CVarDecl [ZGenFilt] CAction           -- \circvar Decl \circspot Action
  | CValDecl [ZGenFilt] CAction           -- \circval Decl \circspot Action
  | CResDecl [ZGenFilt] CAction           -- \circres Decl \circspot Action
  | CVResDecl [ZGenFilt] CAction          -- \circvres Decl \circspot Action
  | CAssumpt [ZName] ZPred ZPred          -- N^{+} \prefixcolon [Pred,Pred]
  | CAssumpt1 [ZName] ZPred               -- N^{+} \prefixcolon [Pred]
  | CPrefix ZPred ZPred                   -- \prefixcolon [Pred,Pred]
  | CPrefix1 ZPred                        -- \prefixcolon [Pred]
  | CommandBrace ZPred                    -- \{Pred\}
  | CommandBracket ZPred                  -- [Pred]
  deriving (Eq,Ord,Show)

data CGActions
 = CircGAction ZPred CAction                   -- Pred \circthen Action
 | CircThenElse CGActions CGActions      -- CGActions \circelse GActions
 -- | CircElse ParAction     -- \circelse CAction
 deriving (Eq,Ord,Show)

data CReplace
  = CRename [ZVar] [ZVar]          -- A[yi / xi] = CRename (ZVar xi []) (ZVar yi [])
  | CRenameAssign [ZVar] [ZVar]    -- A[yi := xi] = CRenameAssign (ZVar xi []) (ZVar yi [])
  deriving (Eq,Ord,Show)
```

2.1 Environments

Used during traversal/evaluation of terms

Environments contain stacks (lists), with new bound variables being pushed onto the front of the list.

The environment also stores information about how large the search space is, and how hard we want to search:

- `search_space` starts at 1, and is multiplied by the size of the type sets as we search inside [ZGenFilt] lists.
- If `search_space` gets larger than `max_search_space`, we stop searching (and return a search space error).
- If we try to generate a finite set larger than `max_set_size`, we return a setsize error.

```
type SearchSpace = [(ZVar,Int)]   -- the max number of choices for each var.
type GlobalDefs  = [(ZVar,ZExpr)]

data Env =
    Env{search_space::Integer,
  search_vars::SearchSpace, -- search_space = product of these nums
  max_search_space::Integer,
  max_set_size::Integer,
  global_values::GlobalDefs,
  local_values::[(ZVar,ZExpr)]
  --avoid_variables::VarSet   TODO: add later?
       }
    deriving Show

empty_env :: GlobalDefs -> Env
empty_env gdefs =
    Env{search_space=1,
  search_vars=[],
  max_search_space=100000,
  max_set_size=1000,
  global_values=gdefs,
  local_values=[]
  --avoid_variables=vs
```

```
        }

-- an environment for temporary evaluations.
-- Smaller search space, no names defined.
dummy_eval_env = (empty_env []){max_search_space=10000}


set_max_search_space :: Integer -> Env -> Env
set_max_search_space i env = env{max_search_space=i}

set_max_set_size :: Integer -> Env -> Env
set_max_set_size i env = env{max_set_size=i}

envPushLocal :: ZVar -> ZExpr -> Env -> Env
envPushLocal v val env = env{local_values = (v,val) : local_values env}

envPushLocals :: [(ZVar,ZExpr)] -> Env -> Env
envPushLocals vs env = env{local_values = vs ++ local_values env}

envIsLocal  :: Env -> ZVar -> Bool
envIsLocal env v = v `elem` (map fst (local_values env))

-- schema names are undecorated global names whose value is a schema?
-- TODO: check out what the Z standard says.
envIsSchema :: Env -> String -> Bool
envIsSchema env v =
    not (null [0 | (n,ZESchema _) <- global_values env, n==string_to_zvar v])

envLookupLocal :: (Monad m) => ZVar -> Env -> m ZValue
envLookupLocal v env =
    case lookup v (local_values env) of
   Just e  -> return e
   Nothing -> fail ("unknown local variable: " ++ show_zvar v)

envLookupGlobal :: (Monad m) => ZVar -> Env -> m ZValue
envLookupGlobal v env =
    case lookup v (global_values env) of
   Just e  -> return e
   Nothing -> fail ("unknown global variable: " ++ show_zvar v)

envLookupVar :: (Monad m) => ZVar -> Env -> m ZValue
envLookupVar v env =
    case lookup v (local_values env) of
   Just e  -> return e
   Nothing -> case lookup v (global_values env) of
       Just e  -> return e
       Nothing -> fail ("unknown variable: " ++ show_zvar v)
```

2.2 Visitor Classes for Z Terms

```
data ZTerm
    = ZExpr ZExpr
    | ZPred ZPred
    | ZSExpr ZSExpr
    | ZNull
    deriving (Eq,Ord,Show)


-- This class extends monad to have the standard features
-- we expect while evaluating/manipulating Z terms.
-- It supports a standard notion of `environment',
-- which maintains a mapping from names to ZExpr, plus
-- other flags etc.  The environment is extended by the
-- local names as the traversal goes inside binders (like forall).
--
-- TODO: can we build in the notion of uniquify-variables?
--       eg.
-- uniquify_expr env (ZSetComp gf (Just e)) = ZSetComp gf2 (Just e2)
```

```haskell
--     where
--     (gf2, env2, sub) = uniquify_gfs env gf
--     e2 = substitute sub env2 (uniquify_expr env2 e)

class (Monad m) => Visitor m where
    -- these methods define what the visitor does!
    visitExpr      :: ZExpr -> m ZExpr
    visitPred      :: ZPred -> m ZPred
    visitSExpr     :: ZSExpr -> m ZSExpr
    visitBranch    :: ZBranch -> m ZBranch
    visitBinder    :: [ZGenFilt] -> ZTerm -> m ([ZGenFilt],ZTerm,Env)
    visitGenFilt   :: ZGenFilt -> m ZGenFilt
    visitTerm      :: ZTerm -> m ZTerm
    visitCDecl      :: CDecl -> m CDecl
    -- visitPara ??

    -- Methods for manipulating the environment,
    -- which includes a mapping from names to expressions.
    lookupLocal  :: ZVar -> m ZExpr   -- lookup locals only
    lookupGlobal :: ZVar -> m ZExpr   -- lookup globals only
    lookupVar    :: ZVar -> m ZExpr   -- lookup locals, then globals
    -- methods for pushing local variables.
    pushLocal   :: ZVar -> ZExpr -> m ()
    pushLocals  :: [(ZVar,ZExpr)] -> m ()
    pushGenFilt :: ZGenFilt -> m ()
    pushBinder  :: [ZGenFilt] -> m ()
    currEnv     :: m Env          -- returns the current environment
    setEnv      :: Env -> m ()    -- changes to use the given environment
         -- (It is generally better to use withEnv)
    withEnv     :: Env -> m a -> m a  -- uses the given environment
    localEnv    :: m a -> m a     -- uses the current env then discards it

    ----------------- Default Implementations --------------------
    -- The default visitors just recurse through the term
    -- Instances will override some cases of these, like this:
    --    myvisitExpr (ZVar v) = ...             (special processing)
    --    myvisitExpr e        = traverseExpr e  (handle all other cases)
    visitExpr = traverseExpr
    visitPred = traversePred
    visitSExpr = traverseSExpr
    visitBranch = traverseBranch
    visitBinder = traverseBinder
    visitGenFilt = traverseGenFilt
    visitTerm = traverseTerm
    visitCDecl = traverseCDecl

    -- Default environment implementations.
    -- Minimum defs required are: currEnv and setEnv.
    lookupLocal v  = currEnv >>= envLookupLocal v
    lookupGlobal v = currEnv >>= envLookupGlobal v
    lookupVar v    = currEnv >>= envLookupVar v
    pushLocal v t  = currEnv >>= (setEnv . envPushLocal v t)
    pushLocals vs  = currEnv >>= (setEnv . envPushLocals vs)
    pushGenFilt    = pushGFType
    pushBinder     = mapM_ pushGenFilt
    withEnv e m =
      do  origenv <- currEnv
          setEnv e
          res <- m
          setEnv origenv
          return res
    localEnv m = do {env <- currEnv; withEnv env m}


-- auxiliary functions for visitors
pushGFType :: Visitor m => ZGenFilt -> m ()
pushGFType (Evaluate v e t) = pushLocal v t
pushGFType (Choose v t) = pushLocal v t
pushGFType _ = return ()
```

2.2.1 Default Traversal Functions

The following `traverse*` functions are useful defaults for visitor methods. They recurse through Z terms, invoking the VISITOR methods at each level (NOT the `traverse*` functions!).

This gives an inheritance-like effect, which allows instances of the Visitor class to define a method M which overrides just the few cases it is interested in, then call one of these `traverse*` functions to handle the remaining cases (subterms within those cases will invoke M, not just `traverse*`). Thus the effective visitor method will be the fixed-point of traverse overridden by M etc.

The goal of this design is that when the data structures change (adding/removing/changing cases), then updating the traversal* functions here should update ALL traversals within Jaza. (The code that does something specific with the changed cases will still need updating manually within each traversal, but this is usually a small fraction of the possible cases).

These default traversal methods extend the environment by pushing the TYPE expression of each local variable.

WARNING: `traverseSExpr` currently does nothing. This implies that: all schema inclusions are ignored as ZGenFilt lists are being processed, which means that inner terms will not have the right environment. This is not a problem once all schema expressions have been unfolded. This problem will be fixable (if necessary) after typechecking is implemented.

```
traverseExpr e@(ZVar _) = return e
traverseExpr e@(ZInt _) = return e
traverseExpr e@(ZGiven _) = return e
traverseExpr e@(ZFree0 _) = return e
traverseExpr (ZFree1 n e) =
    do  e2 <- visitExpr e
        return (ZFree1 n e2)
traverseExpr (ZTuple es) =
    do  es2 <- mapM visitExpr es
        return (ZTuple es2)
traverseExpr (ZBinding ves) =
    do  ves2 <- mapM traverseZVarExpr ves
        return (ZBinding ves2)
traverseExpr (ZSetDisplay es) =
    do  es2 <- mapM visitExpr es
        return (ZSetDisplay es2)
traverseExpr (ZSeqDisplay es) =
    do  es2 <- mapM visitExpr es
        return (ZSeqDisplay es2)
traverseExpr e@(ZFSet vals) = return e
traverseExpr e@(ZIntSet lo hi) = return e
traverseExpr (ZGenerator r e) =
    do  e2 <- visitExpr e
        return (ZGenerator r e2)
traverseExpr (ZCross es) =
    do  es2 <- mapM visitExpr es
        return (ZCross es2)
traverseExpr e@(ZFreeType name bs) =
    do  bs2 <- localEnv (pushLocal name e >> mapM visitBranch bs)
        return (ZFreeType name bs2)
traverseExpr e@ZPowerSet{} =
    do  base2 <- visitExpr (baseset e)
        return e{baseset=base2}
traverseExpr e@ZFuncSet{} =
    do  dom2 <- visitExpr (domset e)
        ran2 <- visitExpr (ranset e)
        return e{domset=dom2, ranset=ran2}
traverseExpr (ZSetComp gfs (Just e)) =
    do  (gfs2,ZExpr e2,_) <- visitBinder gfs (ZExpr e)
        return (ZSetComp gfs2 (Just e2))
traverseExpr (ZLambda gfs e) =
    do  (gfs2,ZExpr e2,_) <- visitBinder gfs (ZExpr e)
        return (ZLambda gfs2 e2)
traverseExpr (ZESchema se) =
    do  se2 <- visitSExpr se
        return (ZESchema se2)
traverseExpr e@(ZGivenSet _) = return e
traverseExpr e@ZUniverse = return e
traverseExpr (ZCall f e) =
    do  f2 <- visitExpr f
        e2 <- visitExpr e
```

```
        return (ZCall f2 e2)
traverseExpr e@(ZReln rel) = return e
traverseExpr e@(ZFunc1 f)  = return e
traverseExpr e@(ZFunc2 f)  = return e
traverseExpr e@(ZStrange _) = return e
traverseExpr (ZMu gfs (Just e)) =
    do  (gfs2,ZExpr e2,_) <- visitBinder gfs (ZExpr e)
        return (ZMu gfs2 (Just e2))
traverseExpr (ZELet defs e) =
    do  defs2 <- mapM traverseZVarExpr defs
        e2 <- visitExpr e
        return (ZELet defs2 e2)
traverseExpr (ZIf_Then_Else p thn els) =
    do  p2 <- visitPred p
        thn2 <- visitExpr thn
        els2 <- visitExpr els
        return (ZIf_Then_Else p2 thn2 els2)
traverseExpr (ZSelect e v) =
    do  e2 <- visitExpr e
        return (ZSelect e2 v)
traverseExpr (ZTheta se) =
    do  se2 <- visitSExpr se
        return (ZTheta se2)


-- helper functions
traverseZVarExpr (v,e) =
    do  e2 <- visitExpr e
        return (v,e2)


traverseMaybeExpr Nothing =
    return Nothing
traverseMaybeExpr (Just e) =
    do  e2 <- visitExpr e
        return (Just e2)


traversePred e@ZFalse{} = return e
traversePred e@ZTrue{} = return e
traversePred (ZAnd p q) =
  do  p2 <- visitPred p
      q2 <- visitPred q
      return (ZAnd p2 q2)
traversePred (ZOr p q) =
  do  p2 <- visitPred p
      q2 <- visitPred q
      return (ZOr p2 q2)
traversePred (ZImplies p q) =
  do  p2 <- visitPred p
      q2 <- visitPred q
      return (ZImplies p2 q2)
traversePred (ZIff p q) =
  do  p2 <- visitPred p
      q2 <- visitPred q
      return (ZIff p2 q2)
traversePred (ZNot p) =
  do  p2 <- visitPred p
      return (ZNot p2)
traversePred (ZExists gfs p) =
  do  (gfs2,ZPred p2,_) <- visitBinder gfs (ZPred p)
      return (ZExists gfs2 p2)
traversePred (ZExists_1 gfs p) =
  do  (gfs2,ZPred p2,_) <- visitBinder gfs (ZPred p)
      return (ZExists_1 gfs2 p2)
traversePred (ZForall gfs p) =
  do  (gfs2,ZPred p2,_) <- visitBinder gfs (ZPred p)
      return (ZForall gfs2 p2)
traversePred (ZPLet defs p) =
  do  defs2 <- mapM traverseZVarExpr defs
```

```
      p2 <- visitPred p
      return (ZPLet defs2 p2)
traversePred (ZEqual p q) =
  do  p2 <- visitExpr p
      q2 <- visitExpr q
      return (ZEqual p2 q2)
traversePred (ZMember p q) =
  do  p2 <- visitExpr p
      q2 <- visitExpr q
      return (ZMember p2 q2)
traversePred (ZPre se) =
  do  se2 <- visitSExpr se
      return (ZPre se2)
traversePred (ZPSchema se) =
  do  se2 <- visitSExpr se
      return (ZPSchema se2)


-- instances should override this.
-- (not necessary if the terms they are visiting have already
--  had all schema expressions unfolded).
traverseSExpr se = fail "traverseSExpr is not implemented"


traverseBranch e@(ZBranch0 _) =
    return e
traverseBranch (ZBranch1 name e) =
  do  e2 <- visitExpr e
      return (ZBranch1 name e2)


-- The default traversal for binders obeys the Jaza (post-unfold)
-- scope rules: the scope of a declared variable starts immediately
-- after the declaration (so includes following declaration types).
traverseGenFilt (Choose v t) =
  do  t2 <- visitExpr t
      pushLocal v t2
      return (Choose v t2)
traverseGenFilt (Check p) =
  do  p2 <- visitPred p
      return (Check p2)
traverseGenFilt (Evaluate v e t) =
  do  e2 <- visitExpr e
      t2 <- visitExpr t
      pushLocal v t2
      return (Evaluate v e2 t2)
traverseGenFilt (Include p) =
    fail "traverseGenFilt should not see schema inclusions"


traverseBinder gfs term =
    localEnv trav2
    where
    trav2 = do gfs2 <- mapM visitGenFilt gfs
               term2 <- visitTerm term
               env <- currEnv
               return (gfs2,term2,env)


traverseTerm (ZExpr e)  = visitExpr e >>= (return . ZExpr)
traverseTerm (ZPred p)  = visitPred p >>= (return . ZPred)
traverseTerm (ZSExpr e) = visitSExpr e >>= (return . ZSExpr)
traverseTerm (ZNull)    = return ZNull
```

2.2.2 Circus Traversal

```
traverseCDecl cd = fail "traverseCDecl is not implemented"
--traverseCDecl (CChan v) = visitCDecl v >>= (return . CChan)
```

```
--traverseCDecl (CChanDecl v e ) = visitCDecl v e >>= (return . CChanDecl)
--traverseCDecl (CMultChanDecl v e ) = visitCDecl v e >>= (return . CMultChanDecl)
--traverseCDecl (CGenChanDecl  v1 v2 e ) = visitCDecl v1 v2 e >>= (return . CGenChanDecl)
```

Substitution

3 Substitution

Defines substitution-related functions over Z terms. These functions should be applied only to unfolded terms (so ZESchema, ZTheta expressions etc. are not handled here).

Exports ZExpr and ZPred as instances of SubsTerm, which is a type class containing functions for performing substitution, determining free variables, etc.

Note that 'substitute sub vs term' takes a set of variables, vs, as well as the substitution, sub. This varset must include all free variables of the entire term that the substituted term will be placed inside (including free vars of 'term' itself), plus any bound variables that 'term' is within the scope of. This allows the substitute function to preserve the 'no-repeated-bound-vars' invariant.

```
(
  avoid_variables,
  choose_fresh_var,
  def_U_NAME,
  def_U_prefix,
  diff_varset,
  empty_varset,
  free_var_CAction,
  free_var_ZExpr,
  free_var_ZGenFilt,
  free_var_ZPred,
  free_vars,     -- Hugs does not export this automatically
  fvars_expr,
  fvars_genfilt,
  fvars_pred,
  get_vars_ZExpr,
  in_varset,
  inter_varset,
  isPrefixOf,
  join_name,
  make_subinfo,
  -- rename_actions_loc_var,
  rename_bndvars,
  rename_lhsvars,   -- only for use by Unfold really.
  show_varset,
  sub_CAction,
  sub_expr,
  sub_genfilt,
  sub_genfilt2,
  sub_ParAction,
  sub_pred,
  subs_add,
  subs_avoid,
  subs_domain,
  subs_range,
  subs_remove,
  subs_sub,
  subseteq_varset,
  SubsTerm,
  substitute,    -- Hugs does not export this automatically
  Substitution,
  union_varset,
  union_varsets,
  uniquify,      -- Restores the no-repeated-bound-vars invariant
  VarSet,
  varset,
  varset_from_zvars,
  varset_to_zvars
)
where
```

```
import AST
import FiniteSets
import Data.Char

type Substitution = [(ZVar,ZExpr)]

-- Optional Precondition checking
-- Define pre f msg val = val to turn this off.
pre f msg val = val
pre False msg val = error ("Precondition Error: " ++ msg)
pre True  msg val = val

class SubsTerm t where
  substitute :: Substitution -> VarSet -> t -> t
  free_vars  :: t -> VarSet    -- result is all ZVar's
  uniquify   :: VarSet -> t -> t
  uniquify   = substitute []

instance SubsTerm ZExpr where
  substitute = presubstitute sub_expr
  free_vars  = fvars_expr

instance SubsTerm ZPred where
  substitute = presubstitute sub_pred
  free_vars  = fvars_pred

presubstitute f sub vs term =
    pre ((termvars `diff_varset` domvars) `subseteq_varset` vs)
        ("subs does not include all free vars: " ++ argmsg)
        (f (make_subinfo sub (union_varsets (vs:ranvars))) term)
    where
    ranvars = map (free_vars . snd) sub
    domvars = varset_from_zvars (map fst sub)
    termvars = free_vars term
    argmsg = "\n\t" ++ show term ++
             "\n\t" ++ show sub ++
             "\n\t{" ++ show_varset vs ++ "}"
```

3.1 *VarSet* ADT

To get more typechecking, here we create a copy of the *FinSet* ADT, restricted to handling just (*ZVar* _) terms.

```
newtype VarSet = VarSet FinSet    -- but containing only (ZVar _) terms.
                 deriving (Eq,Show)

-- Now we promote all the relevant FinSet operations to VarSet.
varset   :: [ZExpr] -> VarSet
varset vs
  = if bad == [] then VarSet (set vs) else error "non-vars in varset"
  where
  bad = filter (not . isZVar) vs
  isZVar (ZVar _) = True
  isZVar _        = False

varset_from_zvars  :: [ZVar] -> VarSet
varset_from_zvars = VarSet . set . map ZVar

zvars_from_zexpr (ZVar x) = [x]
zvars_from_zexpr _ = []

varset_to_zvars  :: VarSet -> [ZVar]
varset_to_zvars (VarSet (x:xs)) = zvars_from_zexpr x ++ (varset_to_zvars (VarSet xs))
varset_to_zvars empty_varset = []

empty_varset = VarSet emptyset

union_varsets :: [VarSet] -> VarSet
union_varsets vs = VarSet (gen_union [s | VarSet s <- vs])
```

```
-- binary operations
union_varset    (VarSet a) (VarSet b) = VarSet (union a b)
inter_varset    (VarSet a) (VarSet b) = VarSet (inter a b)
diff_varset     (VarSet a) (VarSet b) = VarSet (diff a b)
subseteq_varset (VarSet a) (VarSet b) = subset a b

in_varset        v         (VarSet b) = v `mem` b
show_varset     (VarSet vs) = show_zvars [v | ZVar v <- vs]
```

3.2 *SubstitutionInfo* ADT

It is convenient to pass around more information than just the substitution, so we pass around this SubstitutionInfo type, which contains the substitution, plus the set of variables which must be avoided when choosing new local variables. This 'avoid' set must contain:

- all free variables of the entire term that surrounds the term that substitute is being applied to (usually none, because most complete terms have no free vars). Note: This is slightly stronger than necessary – it could be just the free vars minus the domain of the substitution.
- all outer bound variables of the entire term (so that the substitution preserves the uniquify invariant – no repeated bound variable names on any path into the term)
- all free variables in the range of the substitution (because we must avoid capturing these)

3.3 Substitution – Manipulating sets

```
type SubstitutionInfo = (Substitution, VarSet)
```

```
make_subinfo :: Substitution -> VarSet -> SubstitutionInfo
make_subinfo sub vs = (sub, vs)
```

```
subs_sub :: SubstitutionInfo -> Substitution
subs_sub (sub,_) = sub
```

```
subs_domain :: SubstitutionInfo -> [ZVar]
subs_domain (sub,_) = map fst sub
```

```
subs_range :: SubstitutionInfo -> [ZExpr]
subs_range (sub,_) = map snd sub
```

```
subs_avoid :: SubstitutionInfo -> VarSet
subs_avoid (_,vs) = vs
```

```
subs_add :: SubstitutionInfo -> (ZVar,ZExpr) -> SubstitutionInfo
subs_add (sub,vs) (x,e) =
    ((x,e):sub, vs `union_varset` extras)
    where
    extras  = varset_from_zvars [x] `union_varset` free_vars e
```

```
subs_remove :: SubstitutionInfo -> ZVar -> SubstitutionInfo
subs_remove (sub,vs) x = (filter (\ (v,_) -> v /= x) sub, vs)
```

3.4 Substitution for Expressions

```

sub_expr :: SubstitutionInfo -> ZExpr -> ZExpr
sub_expr subs e@(ZUniverse)   = e
sub_expr subs e@(ZVar v)      = maybe e id (lookup v (fst subs))
sub_expr subs e@(ZGiven _)    = e
sub_expr subs e@(ZGivenSet _) = e
sub_expr subs e@(ZInt _)      = e
sub_expr subs (ZGenerator r e) = ZGenerator r (sub_expr subs e)
sub_expr subs e@(ZPowerSet{}) = e{baseset=sub_expr subs (baseset e)}
sub_expr subs e@(ZFuncSet{})  = e{domset=sub_expr subs (domset e),
                                  ranset=sub_expr subs (ranset e)}
sub_expr subs (ZCross es)     = ZCross (map (sub_expr subs) es)
sub_expr subs (ZTuple es)     = ZTuple (map (sub_expr subs) es)
sub_expr subs (ZCall e1 e2)   = ZCall (sub_expr subs e1) (sub_expr subs e2)
sub_expr subs (ZSetDisplay es) = ZSetDisplay (map (sub_expr subs) es)
sub_expr subs (ZSeqDisplay es) = ZSeqDisplay (map (sub_expr subs) es)
sub_expr subs (ZSetComp gfs (Just e)) = ZSetComp gfs2 (Just e2)
  where
  (gfs2,e2) = sub_genfilt sub_expr subs gfs e
sub_expr subs (ZLambda gfs e) = ZLambda gfs2 e2
  where
  (gfs2,e2) = sub_genfilt sub_expr subs gfs e
sub_expr subs (ZMu gfs (Just e)) = ZMu gfs2 (Just e2)
  where
  (gfs2,e2) = sub_genfilt sub_expr subs gfs e
--sub_expr subs (ZELet defs e)  = ZELet defs2 e2
--  where
--  (defs2, e2) = sub_letdef sub_expr subs defs e
sub_expr subs (ZIf_Then_Else p e1 e2) = ZIf_Then_Else p' e1' e2'
  where
  p'  = sub_pred subs p
  e1' = sub_expr subs e1
  e2' = sub_expr subs e2
sub_expr subs (ZSelect e v) = ZSelect (sub_expr subs e) v
  -- Note that e.v = (\lambda [u:U;v:V] @ v) e      (when e:[u:U;v:V])
  --               = \{ u:U; v:V @ (\lblot u==u,v==v \rblot, v) \}
  --    Field names:                         ^    ^
  --    Variable names: ^    ^                  ^    ^         ^
  -- This makes it clear that v is local to this set comprehension,
  -- so is not free within 'e.v' and should not be renamed!
sub_expr subs e@(ZReln _)     = e
sub_expr subs e@(ZFunc1 _)    = e
sub_expr subs e@(ZFunc2 _)    = e
sub_expr subs e@(ZStrange _)  = e
sub_expr subs e@(ZFSet _)     = e   -- contains no vars at all
sub_expr subs e@(ZIntSet _ _) = e
sub_expr subs (ZBinding bs)   = ZBinding [(v,sub_expr subs e)|(v,e) <- bs]
sub_expr subs e@(ZFree0 _)    = e
sub_expr subs (ZFree1 n e)    = ZFree1 n (sub_expr subs e)
sub_expr subs e@(ZFreeType _ _) = e   -- has no free variables
sub_expr subs e = error ("substitute should not see: " ++ show e)
```

3.5 Substitution for Predicates

```
sub_pred :: SubstitutionInfo -> ZPred -> ZPred
sub_pred subs p@(ZFalse{})      = p
sub_pred subs p@(ZTrue{})       = p
sub_pred subs (ZAnd p1 p2)      = ZAnd (sub_pred subs p1) (sub_pred subs p2)
sub_pred subs (ZOr p1 p2)       = ZOr  (sub_pred subs p1) (sub_pred subs p2)
sub_pred subs (ZImplies p1 p2) = ZImplies (sub_pred subs p1) (sub_pred subs p2)
sub_pred subs (ZIff p1 p2)      = ZIff (sub_pred subs p1) (sub_pred subs p2)
sub_pred subs (ZNot p)          = ZNot (sub_pred subs p)
sub_pred subs (ZExists gfs p)   = ZExists gfs2 p2
  where
```

```
  (gfs2,p2) = sub_genfilt sub_pred subs gfs p
sub_pred subs (ZExists_1 gfs p)= ZExists_1 gfs2 p2
  where
  (gfs2,p2) = sub_genfilt sub_pred subs gfs p
sub_pred subs (ZForall gfs p)  = ZForall gfs2 p2
  where
  (gfs2,p2) = sub_genfilt sub_pred subs gfs p
--sub_pred subs (ZPLet defs p)   = ZPLet defs2 p2
--  where
--  (defs2, p2) = sub_letdef sub_pred subs defs p
sub_pred subs (ZEqual e1 e2)  = ZEqual (sub_expr subs e1) (sub_expr subs e2)
sub_pred subs (ZMember e1 e2) = ZMember (sub_expr subs e1) (sub_expr subs e2)
sub_pred subs p = error ("substitute should not see: " ++ show p)
```

3.5.1 Substitution for Circus Actions

```
sub_ParAction :: SubstitutionInfo -> ParAction -> ParAction
sub_ParAction subs (CircusAction vCAction) = (CircusAction (sub_CAction subs vCAction))
sub_ParAction subs (ParamActionDecl vZGenFilt_lst vParAction)
  = (ParamActionDecl vZGenFilt_lst2 vParAction2)
  where
    (vZGenFilt_lst2,vParAction2) = sub_genfilt sub_ParAction subs vZGenFilt_lst vParAction
```

```
sub_CAction :: SubstitutionInfo -> CAction -> CAction
sub_CAction subs (CActionCommand c)
  = (CActionCommand (sub_CCommand subs c))
sub_CAction subs (CSPCommAction cc c)
  = (CSPCommAction (sub_Comm subs cc) (sub_CAction subs c))
sub_CAction subs (CSPGuard p c)
  = (CSPGuard (sub_pred subs p) (sub_CAction subs c))
sub_CAction subs (CSPSeq ca cb)
  = (CSPSeq (sub_CAction subs ca) (sub_CAction subs cb))
sub_CAction subs (CSPExtChoice ca cb)
  = (CSPExtChoice (sub_CAction subs ca) (sub_CAction subs cb))
sub_CAction subs (CSPIntChoice ca cb)
  = (CSPIntChoice (sub_CAction subs ca) (sub_CAction subs cb))
sub_CAction subs (CSPNSParal ns1 cs ns2 ca cb)
  = (CSPNSParal ns1 cs ns2 (sub_CAction subs ca) (sub_CAction subs cb))
sub_CAction subs (CSPParal cs ca cb)
  = (CSPParal cs (sub_CAction subs ca) (sub_CAction subs cb))
sub_CAction subs (CSPNSInter ns1 ns2 ca cb)
  = (CSPNSInter ns1 ns2 (sub_CAction subs ca) (sub_CAction subs cb))
sub_CAction subs (CSPInterleave ca cb)
  = (CSPInterleave (sub_CAction subs ca) (sub_CAction subs cb))
sub_CAction subs (CSPHide c cs)
  = (CSPHide (sub_CAction subs c) cs)
sub_CAction subs (CSPParAction nm xp)
  = (CSPParAction nm xp)
sub_CAction subs (CSPRenAction nm cr)
  = (CSPRenAction nm cr)
sub_CAction subs (CSPRecursion nm c)
  = (CSPRecursion nm (sub_CAction subs c))
sub_CAction subs (CSPUnParAction lst c nm)
  = (CSPUnParAction lst (sub_CAction subs c) nm)
sub_CAction subs (CSPRepSeq lst c)
  = (CSPRepSeq lst (sub_CAction subs c))
sub_CAction subs (CSPRepExtChoice lst c)
  = (CSPRepExtChoice lst (sub_CAction subs c))
sub_CAction subs (CSPRepIntChoice lst c)
  = (CSPRepIntChoice lst (sub_CAction subs c))
sub_CAction subs (CSPRepParalNS cs lst ns c)
  = (CSPRepParalNS cs lst ns (sub_CAction subs c))
sub_CAction subs (CSPRepParal cs lst c)
  = (CSPRepParal cs lst (sub_CAction subs c))
sub_CAction subs (CSPRepInterlNS lst ns c)
  = (CSPRepInterlNS lst ns (sub_CAction subs c))
sub_CAction subs (CSPRepInterl lst c)
  = (CSPRepInterl lst (sub_CAction subs c))
```

```
sub_CAction subs x = x
```

3.5.2 Substitution for Circus Communication

```
-- I still need to work on the substitution starting from the function sub_Comm
-- so we can have substitution over Circus Actions and CircusPar.
-- This is not yet compiled, as I'm still working on it.

sub_Comm :: SubstitutionInfo -> Comm -> Comm
sub_Comm subs (ChanComm vZName vCParameter_lst) = (ChanComm vZName (map (sub_CParameter subs) vCPar
sub_Comm subs (ChanGenComm vZName vZExpr_lst vCParameter_lst) = (ChanGenComm vZName (map (sub_expr
```

```
sub_CParameter :: SubstitutionInfo -> CParameter -> CParameter
sub_CParameter subs (ChanInp vZName) = (ChanInp vZName)
sub_CParameter subs (ChanInpPred vZName vZPred) = (ChanInpPred vZName (sub_pred subs vZPred))
sub_CParameter subs (ChanOutExp vZExpr) = (ChanOutExp (sub_expr subs vZExpr))
sub_CParameter subs (ChanDotExp vZExpr) = (ChanDotExp (sub_expr subs vZExpr))
```

3.5.3 Substitution for Circus Commands

```
-- sub_expr subs (ZSetComp gfs (Just e)) = ZSetComp gfs2 (Just e2)
-- (gfs2,e2) = sub_genfilt sub_expr subs gfs e
sub_CCommand :: SubstitutionInfo -> CCommand -> CCommand
sub_CCommand subs (CAssign vZVar_lst vZExpr_lst) = (CAssign vZVar_lst (map (sub_expr subs) vZExpr_l
sub_CCommand subs (CIf vCGActions) = (CIf (sub_CGActions subs vCGActions))
sub_CCommand subs (CVarDecl vZGenFilt_lst vCAction)
  = (CVarDecl vZGenFilt_lst2 vCAction2)
  where
    (vZGenFilt_lst2,vCAction2) = sub_genfilt sub_CAction subs vZGenFilt_lst vCAction
sub_CCommand subs (CAssumpt vZName_lst v1ZPred v2ZPred) = (CAssumpt vZName_lst (sub_pred subs v1ZPr
sub_CCommand subs (CAssumpt1 vZName_lst vZPred) = (CAssumpt1 vZName_lst (sub_pred subs vZPred))
sub_CCommand subs (CPrefix v1ZPred v2ZPred) = (CPrefix (sub_pred subs v1ZPred) (sub_pred subs v2ZPr
sub_CCommand subs (CPrefix1 vZPred) = (CPrefix1 (sub_pred subs vZPred))
sub_CCommand subs (CommandBrace vZPred) = (CommandBrace (sub_pred subs vZPred))
sub_CCommand subs (CommandBracket vZPred) = (CommandBracket (sub_pred subs vZPred))
sub_CCommand subs (CValDecl vZGenFilt_lst vCAction)
  = (CValDecl vZGenFilt_lst2 vCAction2)
  where
    (vZGenFilt_lst2,vCAction2) = sub_genfilt sub_CAction subs vZGenFilt_lst vCAction
sub_CCommand subs (CResDecl vZGenFilt_lst vCAction)
  = (CResDecl vZGenFilt_lst2 vCAction2)
  where
    (vZGenFilt_lst2,vCAction2) = sub_genfilt sub_CAction subs vZGenFilt_lst vCAction
sub_CCommand subs (CVResDecl vZGenFilt_lst vCAction)
  = (CVResDecl vZGenFilt_lst2 vCAction2)
  where
    (vZGenFilt_lst2,vCAction2) = sub_genfilt sub_CAction subs vZGenFilt_lst vCAction
```

```
sub_CGActions :: SubstitutionInfo -> CGActions  -> CGActions
sub_CGActions subs (CircGAction vZPred vCAction)
  = (CircGAction (sub_pred subs vZPred) (sub_CAction subs vCAction))
sub_CGActions subs (CircThenElse (CircGAction vZPred vCAction) v2CGActions)
  = (CircThenElse (CircGAction (sub_pred subs vZPred) (sub_CAction subs vCAction)) (sub_CGActions s
-- sub_CGActions subs (CircElse vParAction) = (CircElse vParAction)
```

```
sub_CReplace :: SubstitutionInfo -> CReplace -> CReplace
sub_CReplace subs (CRename v1ZVar_lst v2ZVar_lst) = (CRename v1ZVar_lst v2ZVar_lst)
sub_CReplace subs (CRenameAssign v1ZVar_lst v2ZVar_lst) = (CRenameAssign v1ZVar_lst v2ZVar_lst)
```

```

--sub_letdef :: (SubstitutionInfo -> term -> term)
--              -> SubstitutionInfo -> [(ZVar,ZExpr)] -> VarSet -> term
--              -> ([(ZVar,ZExpr)], term)
--sub_letdef subfunc subs0 defs0 t_vars t
--  = (zip lhs2 rhs2, subfunc subs2 t)
```

```
--  where
--  (lhs,rhs) = unzip defs0
--  subs1 = subs0 `subs_remove` lhs
--  dont_capture = subs_range_vars subs1
--  clash = varset_from_zvars lhs `inter_varset` dont_capture
--  inuse = t_vars `union_varset` dont_capture
--  (lhs2, extrasubs) = rename_lhsvars clash inuse lhs
--  subs2 = subs1 `subs_union` extrasubs
--  rhs2  = map (sub_expr subs0) rhs
```

```

-- rename_lhsvars clash inuse vars
-- This chooses new names for each v in vars that is also in clash.
-- The new names are chosen to avoid inuse.
rename_lhsvars :: VarSet -> VarSet -> [ZVar] -> ([ZVar], Substitution)
rename_lhsvars (VarSet []) inuse lhs = (lhs,[]) -- optimize the common case
rename_lhsvars clash inuse [] = ([], [])
rename_lhsvars clash inuse (v:vs)
  | ZVar v `in_varset` clash = (v2:vs2, (v,ZVar v2):subs2)
  | otherwise                = (v:vs2,  subs2)
  where
  (vs2, subs2) = rename_lhsvars clash inuse2 vs
  v2 = choose_fresh_var inuse (get_zvar_name v)
  inuse2 = varset_from_zvars [v2] `union_varset` inuse
```

3.6 Substitution

This is the most complex part of substitution. The scope rules for [*ZGenFilt*] lists are fairly subtle (see AST.hs) and on top of those, we have to do a substitution, being careful (as usual!) to rename any of the bound variables that might capture variables in the range of the substitution. This is enough to make life exciting...

The '*subfunc*' argument is either sub_pred or sub_expr. It is passed as a parameter so that this function can work on [*ZGenFilt*] lists that are followed by either kind of term. (An earlier version used the type class 'substitute', and avoided having this parameter, but GHC 4.02 did not like that).

```

sub_genfilt :: (SubstitutionInfo -> term -> term)
               -> SubstitutionInfo -> [ZGenFilt] -> term
               -> ([ZGenFilt], term)
sub_genfilt subfunc subs0 gfs0 t =
    (gfs, subfunc finalsubs t)
    where
    (gfs,finalsubs) = sub_genfilt2 subs0 gfs0

sub_genfilt2 :: SubstitutionInfo -> [ZGenFilt]
             -> ([ZGenFilt], SubstitutionInfo)
sub_genfilt2 subs0 [] =
    ([], subs0)
sub_genfilt2 subs0 (Evaluate x e t:gfs0) =
    (Evaluate x2 e2 t2 : gfs, subs)
    where
    e2 = sub_expr subs0 e
    t2 = sub_expr subs0 t
    subs1 = subs0 `subs_remove` x
    (x2,subs2) =
        if ZVar x `in_varset` subs_avoid subs1
           then (fresh, subs1 `subs_add` (x,ZVar fresh))
           else (x,subs1)
    fresh = choose_fresh_var (subs_avoid subs1) (get_zvar_name x)
    (gfs, subs) = sub_genfilt2 subs2 gfs0
sub_genfilt2 subs0 (Choose x e:gfs0) =
    (Choose x2 (sub_expr subs0 e) : gfs, subs)
    where
    subs1 = subs0 `subs_remove` x
    (x2,subs2) =
        if ZVar x `in_varset` subs_avoid subs1
           then (fresh, subs1 `subs_add` (x,ZVar fresh))
           else (x,subs1)
    fresh = choose_fresh_var (subs_avoid subs1) (get_zvar_name x)
```

```
    (gfs, subs) = sub_genfilt2 subs2 gfs0
sub_genfilt2 subs0 (Check p:gfs0) =
    (Check (sub_pred subs0 p) : gfs, subs)
    where
    (gfs, subs) = sub_genfilt2 subs0 gfs0
```

This renames any bound variables that are in '*clash*', to avoid capture problems. (It only renames the defining occurrence of the variables, not all the places where they are used, but it returns a substitution which will do that when it is applied later). To ensure that the new variable name is fresh, it is chosen to not conflict with any of the variable in 'inuse'.

This function could almost be implemented using *map*, but we use a recursive defn so that as each fresh variable is chosen, it can be added to the set of '*inuse*' variables.

```

rename_bndvars :: VarSet -> VarSet -> [ZGenFilt] -> ([ZGenFilt],Substitution)
rename_bndvars (VarSet []) _ gfs = (gfs,[]) -- optimize a common case
rename_bndvars clash inuse [] = ([],[])
rename_bndvars clash inuse (c@(Evaluate v e t):gfs0)
  | ZVar v `in_varset` clash = (Evaluate v2 e t:gfs, (v,ZVar v2):subs)
  | otherwise                = (c:gfs_easy, subs_easy)
  where
  (gfs, subs)           = rename_bndvars clash inuse2 gfs0
  (gfs_easy, subs_easy) = rename_bndvars clash inuse gfs0
  v2 = choose_fresh_var inuse (get_zvar_name v)
  inuse2 = varset_from_zvars [v2] `union_varset` inuse
rename_bndvars clash inuse (c@(Choose v e):gfs0)
  | ZVar v `in_varset` clash = (Choose v2 e:gfs, (v,ZVar v2):subs)
  | otherwise                = (c:gfs, subs)
  where
  (gfs, subs) = rename_bndvars clash inuse2 gfs0
  v2 = choose_fresh_var inuse (get_zvar_name v)
  inuse2 = varset_from_zvars [v2] `union_varset` inuse
rename_bndvars clash inuse (c@(Check _):gfs0)
  = (c:gfs, subs)
  where
  (gfs, subs) = rename_bndvars clash inuse gfs0
```

3.7 Free Variables for *Expressions*

```
free_var_ZExpr :: ZExpr -> [ZVar]
free_var_ZExpr x = varset_to_zvars $ free_vars x
```

```

-- TODO: an more efficient algorithm might be to keep track
--   of the bound vars on the way in, and only generate those
--   that are not in that set.  This is what Zeta does, and it
--   might produce less garbage.
fvars_expr :: ZExpr -> VarSet
fvars_expr ZUniverse         = empty_varset
fvars_expr e@(ZVar v)        = varset [e]
fvars_expr (ZGiven _)        = empty_varset
fvars_expr (ZGivenSet _)     = empty_varset
fvars_expr (ZInt _)          = empty_varset
fvars_expr (ZGenerator r e)  = fvars_expr e
fvars_expr (ZPowerSet{baseset=e})
                             = fvars_expr e
fvars_expr (ZFuncSet{domset=e1,ranset=e2})
                             = fvars_expr e1 `union_varset` fvars_expr e2
fvars_expr (ZCross es)       = union_varsets (map fvars_expr es)
fvars_expr (ZTuple es)       = union_varsets (map fvars_expr es)
fvars_expr (ZCall e1 e2)     = fvars_expr e1 `union_varset` fvars_expr e2
fvars_expr (ZSetDisplay es)  = union_varsets (map fvars_expr es)
fvars_expr (ZSeqDisplay es)  = union_varsets (map fvars_expr es)
fvars_expr (ZSetComp gfs (Just e))
                             = fvars_genfilt gfs (fvars_expr e)
fvars_expr (ZLambda gfs e)   = fvars_genfilt gfs (fvars_expr e)
fvars_expr (ZMu gfs (Just e))=fvars_genfilt gfs (fvars_expr e)
fvars_expr (ZELet defs e)
```

```
  = rhsvars `union_varset` (fvars_expr e `diff_varset` bndvarset)
  where
  (bndvars, rhss) = unzip defs
  bndvarset = varset (map ZVar bndvars)
  rhsvars = union_varsets (map fvars_expr rhss)
fvars_expr (ZIf_Then_Else p e1 e2)
  = fvars_pred p `union_varset` fvars_expr e1 `union_varset` fvars_expr e2

fvars_expr (ZSelect e v)     = fvars_expr e
fvars_expr (ZReln _)         = empty_varset
fvars_expr (ZFunc1 _)        = empty_varset
fvars_expr (ZFunc2 _)        = empty_varset
fvars_expr (ZStrange _)      = empty_varset
fvars_expr (ZFSet _)         = empty_varset
fvars_expr (ZIntSet _ _)     = empty_varset
fvars_expr (ZBinding bs)     = union_varsets (map (fvars_expr . snd) bs)
fvars_expr (ZFree0 _)        = empty_varset
fvars_expr (ZFree1 n e)      = fvars_expr e
fvars_expr (ZFreeType _ _)   = empty_varset  -- has no free variables
fvars_expr e  = error ("free_vars should not see: " ++ show e)
```

3.8 Free Variables for *Predicates*

```
free_var_ZPred :: ZPred -> [ZVar]
free_var_ZPred x = varset_to_zvars $ free_vars x
```

```
fvars_pred :: ZPred -> VarSet
fvars_pred (ZFalse{})        = empty_varset
fvars_pred (ZTrue{})         = empty_varset
fvars_pred (ZAnd p1 p2)      = fvars_pred p1 `union_varset` fvars_pred p2
fvars_pred (ZOr p1 p2)       = fvars_pred p1 `union_varset` fvars_pred p2
fvars_pred (ZImplies p1 p2) = fvars_pred p1 `union_varset` fvars_pred p2
fvars_pred (ZIff p1 p2)      = fvars_pred p1 `union_varset` fvars_pred p2
fvars_pred (ZNot p)          = fvars_pred p
fvars_pred (ZExists gfs p)   = fvars_genfilt gfs (fvars_pred p)
fvars_pred (ZExists_1 gfs p)= fvars_genfilt gfs (fvars_pred p)
fvars_pred (ZForall gfs p)   = fvars_genfilt gfs (fvars_pred p)
fvars_pred (ZPLet defs p)
  = rhsvars `union_varset` (fvars_pred p `diff_varset` bndvarset)
  where
  (bndvars, rhss) = unzip defs
  bndvarset = varset (map ZVar bndvars)
  rhsvars = union_varsets (map fvars_expr rhss)
fvars_pred (ZEqual e1 e2)    = fvars_expr e1 `union_varset` fvars_expr e2
fvars_pred (ZMember e1 e2)   = fvars_expr e1 `union_varset` fvars_expr e2
fvars_pred p = error ("freevars should not see: " ++ show p)
```

```
free_var_ZGenFilt :: [ZGenFilt] -> (t -> [ZVar]) -> t -> [ZVar]

free_var_ZGenFilt lst f e = varset_to_zvars $ (fvars_genfilt lst (varset_from_zvars $ f e))
```

```


fvars_genfilt :: [ZGenFilt] -> VarSet -> VarSet
fvars_genfilt gfs inner = foldr adjust inner gfs
    where
    adjust (Choose x t) inner =
        (inner `diff_varset` varset_from_zvars [x])
         `union_varset` free_vars t
    adjust (Check p) inner =
        inner `union_varset` free_vars p
    adjust (Evaluate x e t) inner =
        (inner `diff_varset` varset_from_zvars [x])
          `union_varset` (free_vars e `union_varset` free_vars t)
```

3.9 Free Variables for *Circus* actions

```
free_var_CAction :: CAction -> [ZVar]
free_var_CAction (CActionSchemaExpr x) = []
free_var_CAction (CActionCommand c) = (free_var_comnd c)
free_var_CAction (CActionName nm) = []
free_var_CAction (CSPSkip) = []
free_var_CAction (CSPStop) = []
free_var_CAction (CSPChaos) = []
free_var_CAction (CSPCommAction (ChanComm com xs) c) = (get_chan_var xs)++(free_var_CAction c)
free_var_CAction (CSPGuard p c) = (free_var_ZPred p)++(free_var_CAction c)
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 (CSPIntChoice ca cb) = (free_var_CAction ca)++(free_var_CAction cb)
free_var_CAction (CSPNSParal ns1 cs ns2 ca cb) = (free_var_CAction ca)++(free_var_CAction cb)
free_var_CAction (CSPParal cs ca cb) = (free_var_CAction ca)++(free_var_CAction cb)
free_var_CAction (CSPNSInter ns1 ns2 ca cb) = (free_var_CAction ca)++(free_var_CAction cb)
free_var_CAction (CSPInterleave ca cb) = (free_var_CAction ca)++(free_var_CAction cb)
free_var_CAction (CSPHide c cs) = (free_var_CAction c)
free_var_CAction (CSPParAction nm xp) = []
free_var_CAction (CSPRenAction nm cr) = []
free_var_CAction (CSPRecursion nm c) = (free_var_CAction c)
free_var_CAction (CSPUnParAction lst c nm) =  free_var_ZGenFilt lst free_var_CAction c
free_var_CAction (CSPRepSeq lst c) =  free_var_ZGenFilt lst free_var_CAction c
free_var_CAction (CSPRepExtChoice lst c) =  free_var_ZGenFilt lst free_var_CAction c
free_var_CAction (CSPRepIntChoice lst c) =  free_var_ZGenFilt lst free_var_CAction c
free_var_CAction (CSPRepParalNS cs lst ns c) =  free_var_ZGenFilt lst free_var_CAction c
free_var_CAction (CSPRepParal cs lst c) =  free_var_ZGenFilt lst free_var_CAction c
free_var_CAction (CSPRepInterlNS lst ns c) =  free_var_ZGenFilt lst free_var_CAction c
free_var_CAction (CSPRepInterl lst c) =  free_var_ZGenFilt lst free_var_CAction c
```

```
get_chan_var :: [CParameter] -> [ZVar]
get_chan_var [] = []
get_chan_var [ChanDotExp (ZVar (x,_))] = [(x,[])]
get_chan_var [ChanOutExp (ZVar (x,_))] = [(x,[])]
get_chan_var [_] = []
get_chan_var ((ChanDotExp (ZVar (x,_))):xs) = [(x,[])]++(get_chan_var xs)
get_chan_var ((ChanOutExp (ZVar (x,_))):xs) = [(x,[])]++(get_chan_var xs)
get_chan_var (_:xs) = (get_chan_var xs)
```

3.10 Free Variables for *Circus* commands

```
free_var_comnd (CAssign v e)
 = v
free_var_comnd (CIf ga)
 = free_var_if ga
free_var_comnd (CVarDecl lst c)
 =  free_var_ZGenFilt lst free_var_CAction c
free_var_comnd (CAssumpt n p1 p2)
 = []
free_var_comnd (CAssumpt1 n p)
 = []
free_var_comnd (CPrefix p1 p2)
 = []
free_var_comnd (CPrefix1 p)
 = []
free_var_comnd (CommandBrace z)
 = (free_var_ZPred z)
free_var_comnd (CommandBracket z)
 = (free_var_ZPred z)
free_var_comnd (CValDecl lst c)
 =  free_var_ZGenFilt lst free_var_CAction c
free_var_comnd (CResDecl lst c)
 =  free_var_ZGenFilt lst free_var_CAction c
free_var_comnd (CVResDecl lst c)
 =  free_var_ZGenFilt lst free_var_CAction c
```

```
free_var_if (CircGAction p a)
 = (free_var_ZPred p)++(free_var_CAction a)
free_var_if (CircThenElse (CircGAction p a) gb)
 = (free_var_ZPred p)++(free_var_CAction a)++(free_var_if gb)
-- free_var_if (CircElse (CircusAction a))
--  = (free_var_CAction a)
-- free_var_if (CircElse (ParamActionDecl x (CircusAction a)))
--  =  free_var_ZGenFilt x free_var_CAction a
```

3.11 Fresh Variables generator

```
-- returns a ZVar that does not appear in the given list of zvars.
choose_fresh_var :: VarSet -> String -> ZVar
choose_fresh_var vs s
  = head [v|d <- decors, let v = make_zvar s d, not (ZVar v `in_varset` vs)]

decors0 = [[d] | d <- ["_1","_2","_3","_4","_5","_6","_7","_8","_9"]]
decors  = [[]] ++ [d2++d | d2 <- decors, d <- decors0]


avoid_variables :: Env -> VarSet
avoid_variables = varset_from_zvars . map fst . local_values
```

3.12 New features for *Circus*

Renaming local variables of an action

```
rename_ZGenFilt_loc_var (Choose (va,x) e)
  = (Choose ((join_name (join_name "lv" va) newt),x) e)
  where
        newt = (def_U_NAME $ get_vars_ZExpr e)
rename_ZGenFilt_loc_var _ = error "local var translation not defined for this"
```

Then we also make a function that returns all the local variable definitions

```
list_ZGenFilt_loc_var (Choose (va,x) e)
  = [((va,x),e)]
list_ZGenFilt_loc_var _ = []
list_ZGenFilt_loc_var' (Choose (va,x) e)
  = [ZVar (va,x)]
list_ZGenFilt_loc_var' _ = []
```

```
-- rename_actions_loc_var (CParAction a (ParamActionDecl vZGenFilt_lst vCAction))
--    = (CParAction a (ParamActionDecl vZGenFilt_lst2 vCAction2))
--    where
--      locVar = map list_ZGenFilt_loc_var vZGenFilt_lst
--      renLocVar = map rename_ZGenFilt_loc_var vZGenFilt_lst
--      newLocVar = map list_ZGenFilt_loc_var' renLocVar
--      subs = mergeTwoLists locVar newLocVar
--      (vZGenFilt_lst2,vCAction2) = sub_genfilt sub_ParAction subs vZGenFilt_lst vCAction
-- rename_actions_loc_var _ = []

mergeTwoLists [] [] = []
mergeTwoLists [x] [y] = [(x,y)]
mergeTwoLists (x:xs) (y:ys) = [(x,y)] ++ (mergeTwoLists xs ys)
mergeTwoLists _ _ = error "Error whilst mergin two lists"


def_U_NAME x = ("U_"++(map toUpper (take 3 x)))
def_U_prefix x = (map toUpper (take 3 x))
```

```
get_vars_ZExpr :: ZExpr -> ZName
-- get_vars_ZExpr (ZVar (('\\':'\\':xs),x)) = (map toUpper (take 1 xs)) ++ (drop 1 xs)
get_vars_ZExpr (ZVar (a,x))
  = strip a "\92"
```

```
get_vars_ZExpr (ZCall (ZVar ("\\power",[])) (ZVar (c,[])))
  = "P"++get_vars_ZExpr (ZVar (c,[]))
```

```
join_name n v = n ++ "_" ++ v
```

```

-- | The 'stripPrefix' function drops the given prefix from a list.
-- It returns 'Nothing' if the list did not start with the prefix
-- given, or 'Just' the list after the prefix, if it does.
--
-- > stripPrefix "foo" "foobar" == Just "bar"
-- > stripPrefix "foo" "foo" == Just ""
-- > stripPrefix "foo" "barfoo" == Nothing
-- > stripPrefix "foo" "barfoobaz" == Nothing
stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]
stripPrefix [] ys = Just ys
stripPrefix (x:xs) (y:ys)
 | x == y = stripPrefix xs ys
stripPrefix _ _ = Nothing

strip str x
  | x `isPrefixOf` str = drop 1 str
  | otherwise = str
    where Just restOfString = stripPrefix x str
```

```
isPrefixOf [] _ = True
isPrefixOf _ [] = False
isPrefixOf (x : xs) (y : ys) = (x == y && isPrefixOf xs ys)
```

4 Mapping Functions - Stateless Circus

Mapping Omega Functions from Circus to Circus

4.1 Stateless Circus - Actions

$$\Omega_A(\mathbf{Skip}) \widehat{=} \mathbf{Skip}$$
$$\Omega_A(\mathbf{Stop}) \widehat{=} \mathbf{Stop}$$
$$\Omega_A(\mathbf{Chaos}) \widehat{=} \mathbf{Chaos}$$

is written in Haskell as:

```
omega_CAction :: CAction -> CAction
omega_CAction CSPSkip = CSPSkip
omega_CAction CSPStop = CSPStop
omega_CAction CSPChaos = CSPChaos
```

$$\Omega_A(c \longrightarrow A) \widehat{=} c \longrightarrow \Omega_A(A)$$

is written in Haskell as:

```
omega_CAction (CSPCommAction (ChanComm c []) a)
  = (CSPCommAction (ChanComm c []) (omega_CAction a))
```

$$\begin{aligned}\Omega_A(c.e(v_0, \ldots, v_n, l_0, \ldots, l_m) \longrightarrow A) \widehat{=}\\ &get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow\\ &get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow\\ &c.e(vv_0, \ldots, vv_n, vl_0, \ldots, vl_m) \longrightarrow \Omega'_A(A)\end{aligned}$$

where

$$FV(e) = (v_0, \ldots, v_n, l_0, \ldots, l_m)$$

is written in Haskell as:

```
omega_CAction (CSPCommAction (ChanComm c [ChanDotExp e]) a)
  = make_get_com lxs (rename_vars_CAction (CSPCommAction (ChanComm c [ChanDotExp e]) (omega_prime_C
  where lxs = remdups $ concat (map get_ZVar_st (free_var_ZExpr e))
omega_CAction (CSPCommAction (ChanComm c ((ChanDotExp e):xs)) a)
  = make_get_com lxs (rename_vars_CAction (CSPCommAction (ChanComm c ((ChanDotExp e):xs)) (omega_pr
  where lxs = remdups $ concat (map get_ZVar_st (free_var_ZExpr e))
```

$$
\begin{aligned}
&\Omega_A(c!e(v_0,\ldots,v_n,l_0,\ldots,l_m) \longrightarrow A) \mathrel{\widehat=} \\
&\qquad c.e(v_0,\ldots,v_n,l_0,\ldots,l_m) \longrightarrow A
\end{aligned}
$$

```
omega_CAction (CSPCommAction (ChanComm c [ChanOutExp e]) a)
  = omega_CAction (CSPCommAction (ChanComm c [ChanDotExp e]) a)
omega_CAction (CSPCommAction (ChanComm c ((ChanOutExp e):xs)) a)
  = omega_CAction (CSPCommAction (ChanComm c ((ChanDotExp e):xs)) a)
```

$$
\begin{aligned}
&\Omega_A(g(v_0,\ldots,v_n,l_0,\ldots,l_m) \longrightarrow A) \mathrel{\widehat=} \\
&\qquad get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow \\
&\qquad get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow \\
&\qquad g(vv_0,\ldots,vv_n,vl_0,\ldots,vl_m) \mathbin{\&} \Omega'_A(A)
\end{aligned}
$$

is written in Haskell as:

```
omega_CAction (CSPGuard g a)
  = make_get_com lxs (rename_vars_CAction (CSPGuard (rename_ZPred g) (omega_prime_CAction a)))
  where lxs = remdups $ concat (map get_ZVar_st (free_var_ZPred g))
```

I'm considering $x?k \neq x?k : P$ and I'm making the translation straightforward:

$$
\begin{aligned}
&\Omega_A(c?x \longrightarrow A) \mathrel{\widehat=} \\
&\qquad c?x \longrightarrow \Omega'_A(A)
\end{aligned}
$$

is written in Haskell as:

```
omega_CAction (CSPCommAction (ChanComm c [ChanInp (x:xs)]) a)
  = (CSPCommAction (ChanComm c [ChanInp (x:xs)]) (omega_CAction a))
```

$$
\begin{aligned}
&\Omega_A(c?x : P(x, v_0,\ldots,v_n,l_0,\ldots,l_m) \longrightarrow A) \mathrel{\widehat=} \\
&\qquad get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow \\
&\qquad get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow \\
&\qquad c?x : P(x, vv_0,\ldots,vv_n,vl_0,\ldots,vl_m) \longrightarrow \Omega'_A(A)
\end{aligned}
$$

where

$$x \in wrtV(A)$$

is written in Haskell as:

```
omega_CAction (CSPCommAction (ChanComm c [ChanInpPred x p]) a)
  = case not (elem x (getWrtV(a))) of
    True -> make_get_com lsx (rename_vars_CAction (CSPCommAction
             (ChanComm c [ChanInpPred x p])
                 (omega_prime_CAction a)))
    _  -> (CSPCommAction (ChanComm c [ChanInpPred x p]) a)
  where lsx = remdups $ concat (map get_ZVar_st (free_var_ZPred p))
```

$$\Omega_A(A_1 \mathbin{;} A_2) \mathrel{\widehat=} \Omega_A(A_1) \mathbin{;} \Omega_A(A_2)$$

is written in Haskell as:

```
omega_CAction (CSPSeq ca cb)
  = (CSPSeq (omega_CAction ca) (omega_CAction cb))
```

$$\Omega_A(A_1 \sqcap A_2) \mathrel{\widehat=} \Omega_A(A_1) \sqcap \Omega_A(A_2)$$

is written in Haskell as:

```
omega_CAction (CSPIntChoice ca cb)
  = (CSPIntChoice (omega_CAction ca) (omega_CAction cb))
```

$$\begin{array}{l}
\Omega_A(A_1 \mathbin{\Box} A_2) \mathrel{\widehat=} \\
\quad get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow \\
\quad get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow \\
\qquad (\Omega'_A(A_1) \mathbin{\Box} \Omega'_A(A_2))
\end{array}$$

is written in Haskell as:

```
omega_CAction (CSPExtChoice ca cb)
  = make_get_com lsx (CSPExtChoice (omega_prime_CAction ca) (omega_prime_CAction cb))
   where
    lsx = remdups $ concat $ map get_ZVar_st $ free_var_CAction (CSPExtChoice ca cb)
```

$$\begin{array}{l}
\Omega_A(A1 \llbracket\, ns1 \mid cs \mid ns2 \,\rrbracket\, A2) \mathrel{\widehat=} \\
\quad get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow \\
\quad get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
(\ \Omega'_A(A_1)\ ;\ terminate \longrightarrow \mathbf{Skip}\ ) \\
\llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\
MemoryMerge(\{v0 \mapsto vv0, \ldots\}, LEFT)
\end{array}\right) \setminus MEM_I \\
\llbracket \{\} \mid cs \mid \{\} \rrbracket \\
\left(\begin{array}{l}
(\ \Omega'_A(A_2)\ ;\ terminate \longrightarrow \mathbf{Skip}\ ) \\
\llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\
MemoryMerge(\{v0 \mapsto vv0, \ldots\}, RIGHT)
\end{array}\right) \setminus MEM_I
\end{array}\right) \\
\llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\
Merge
\end{array}\right) \\
\qquad \setminus \{\!|\ mleft, mright\ |\!\}
\end{array}$$

```
-- omega_CAction (CSPNSParal ns1 cs ns2 a1 a2)
--   = make_get_com lsx (rename_vars_CAction (CSPHide
--    (CSPNSParal NSExpEmpty (CSExpr "MEMi") NSExpEmpty
--      (CSPNSParal NSExpEmpty cs NSExpEmpty
--       (CSPHide
--        (CSPNSParal NSExpEmpty (CSExpr "MEMi") NSExpEmpty
--         (CSPSeq a1 (CSPCommAction (ChanComm "terminate" []) CSPSkip))
--         (CSPParAction "MemoryMerge"
--          [ZSetDisplay [],
--                 ZVar ("LEFT",[])]))
--        (CSExpr "MEMi"))
--       (CSPHide
--        (CSPNSParal NSExpEmpty (CSExpr "MEMi") NSExpEmpty
--         (CSPSeq a2 (CSPCommAction (ChanComm "terminate" []) CSPSkip))
--         (CSPParAction "MemoryMerge"
--          [ZSetDisplay [],
--                 ZVar ("RIGHT",[])]))
--        (CSExpr "MEMi")))
--       (CActionName "Merge"))
--       (CChanSet ["mleft","mright"])))
--    where
--     lsx = union (map fst (remdups (free_var_CAction a1))) (map fst (remdups (free_var_CAction a2
```

$$\Omega_A(\mathop{;}\ x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Omega_A(A(v_1)\ ;\ \ldots ;\ A(v_n))$$

is written in Haskell as:

```
omega_CAction (CSPRepSeq [Choose (x,[]) (ZSeqDisplay xs)] (CSPParAction act [ZVar (x1,[])]))
  = case x == x1 of
    True -> omega_CAction (rep_CSPRepSeq act xs)
    _  -> (CSPRepSeq [Choose (x,[]) (ZSeqDisplay xs)]
          (CSPParAction act [ZVar (x1,[])]))
omega_CAction (CSPRepSeq [Choose (x,[]) v] act)
  = (CSPRepSeq [Choose (x,[]) v] (omega_CAction act))
```

$$\Omega_A(\Box\, x : \langle v_1, ..., v_n\rangle \bullet A(x)) \mathrel{\widehat=} \Omega_A(A(v_1) \Box \ldots \Box A(v_n))$$

is written in Haskell as:

```
omega_CAction (CSPRepExtChoice [Choose (x,[]) (ZSeqDisplay xs)] (CSPParAction act [ZVar (x1,[])]))
  = case x == x1 of
    True -> omega_CAction (rep_CSPRepExtChoice act xs)
    _  -> (CSPRepExtChoice [Choose (x,[]) (ZSeqDisplay xs)]
          (CSPParAction act [ZVar (x1,[])]))
omega_CAction (CSPRepExtChoice [Choose (x,[]) v] act)
  = (CSPRepExtChoice [Choose (x,[]) v] (omega_CAction act))
```

$$\Omega_A(\sqcap\, x : \langle v_1, ..., v_n\rangle \bullet A(x)) \mathrel{\widehat=} \Omega_A(A(v_1) \sqcap \ldots \sqcap A(v_n))$$

is written in Haskell as:

```
omega_CAction (CSPRepIntChoice [Choose (x,[]) (ZSeqDisplay xs)]
          (CSPParAction act [ZVar (x1,[])]))
  = case x == x1 of
    True -> omega_CAction(rep_CSPRepIntChoice act xs)
    _  -> (CSPRepIntChoice [Choose (x,[]) (ZSeqDisplay xs)]
          (CSPParAction act [ZVar (x1,[])]))
omega_CAction (CSPRepIntChoice [Choose (x,[]) v] act)
  = (CSPRepIntChoice [Choose (x,[]) v] (omega_CAction act))
```

$$\Omega_A(\llbracket cs \rrbracket\, x : \langle v_1, ..., v_n\rangle \bullet \llbracket ns(x) \rrbracket\, A(x)) \mathrel{\widehat=} \left(\begin{array}{l} A(v_1) \\ \llbracket ns(v_1) \mid cs \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\}\rrbracket \\ \left(\ldots \left(\begin{array}{l} \Omega_A(A(v_n - 1)) \\ \llbracket ns(v_n - 1) \mid cs \mid ns(v_n)\rrbracket \\ A(v_n) \end{array} \right) \right) \end{array} \right)$$

is written in Haskell as:

```
omega_CAction (CSPRepParalNS (CSExpr cs) [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (CSPParAction a [ZVar (x2,[])]))
  = case (x == x1) && (x == x2) of
    True -> omega_CAction (rep_CSPRepParalNS a cs ns x lsx)
    _  -> (CSPRepParalNS (CSExpr cs) [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (CSPParAction a [ZVar (x2,[])]))
omega_CAction (CSPRepParalNS (CSExpr cs) [Choose (x,[]) (ZSetDisplay lsx)] (NSExprParam ns [ZVar (x
  = (CSPRepParalNS (CSExpr cs) [Choose (x,[]) (ZSetDisplay lsx)] (NSExprParam ns [ZVar (x1,[])]) (o
```

$$\Omega_A(\mathbf{val}\ Decl \bullet P) \mathrel{\widehat=} \mathbf{val}\ Decl \bullet \Omega_A(P)$$

is written in Haskell as:

```
omega_CAction (CActionCommand (CValDecl xs a))
  = (CActionCommand (CValDecl xs (omega_CAction a)))
```

$$
\begin{array}{l}
\Omega_A\left(x_0, \ldots, x_n := e_0\left(\begin{array}{l} v_0, ..., v_n, \\ l_0, ..., l_m \end{array}\right), \ldots, e_n\left(\begin{array}{l} v_0, ..., v_n, \\ l_0, ..., l_m \end{array}\right)\right) \mathrel{\widehat=} \\
\quad get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow \\
\quad get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow \\
\quad set.x_0!e_0(vv_0, ..., vv_n, vl_0, ..., vl_m) \longrightarrow \\
\quad \ldots \longrightarrow \\
\quad set.x_n!e_n(vv_0, ..., vv_n, vl_0, ..., vl_m) \longrightarrow \mathbf{Skip}
\end{array}
$$

```
omega_CAction (CActionCommand (CAssign varls valls))
  = make_get_com (remdups $ concat (map get_ZVar_st varls))  (make_set_com omega_CAction varls (map
```

$$\Omega_A(A \setminus cs) \mathrel{\widehat=} \Omega_A(A) \setminus cs$$

is written in Haskell as:

```
omega_CAction (CSPHide a cs) = (CSPHide (omega_CAction a) cs)
```

$$
\begin{array}{l}
\Omega_A\left(\begin{array}{l}
\mathbf{if}\ g_0(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow A_0 \\
\qquad [\!]\ \ldots \\
\qquad [\!]\ g_n(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow A_n \\
\mathbf{fi}
\end{array}\right) \mathrel{\widehat=} \\
\quad get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow \\
\quad get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow \\
\quad \mathbf{if}\ g_0(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow \Omega'_A(A_0) \\
\qquad [\!]\ \ldots \\
\qquad [\!]\ g_n(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow \Omega'_A(A_n) \\
\quad \mathbf{fi}
\end{array}
$$

```
omega_CAction (CActionCommand (CIf gax))
  = make_get_com lsx (CActionCommand (CIf (mk_guard_pair omega_prime_CAction (rename_guard_pair lsx
  where
   gpair = get_guard_pair gax
   lsx = concat (map get_ZVar_st (remdups (concat (map free_var_ZPred (map fst gpair)))))
```

$$\Omega_A(\mu X \bullet A(X)) \mathrel{\widehat=} \mu X \bullet \Omega_A(A(X))$$

is written in Haskell as:

```
omega_CAction (CSPRecursion x c) = (CSPRecursion x (omega_CAction c))
```

$$
\begin{array}{l}
\Omega_A(|||\ x : \langle v_1, ..., v_n\rangle \bullet A(x)) \mathrel{\widehat=} \\
\quad \left(\begin{array}{l}
A(v_1) \\
[\![ns(v_1) \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\}]\!] \\
\left(\ldots\left(\begin{array}{l}
\Omega_A(A(v_n - 1)) \\
[\![ns(v_n - 1) \mid ns(v_n)]\!] \\
A(v_n)
\end{array}\right)\right)
\end{array}\right)
\end{array}
$$

is written in Haskell as:

```
omega_CAction (CSPRepInterlNS [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (CSPParAction a [ZVar (x2,[])]))
  = case (x == x1) && (x == x2) of
    True -> omega_CAction (rep_CSPRepInterlNS a ns x lsx)
    _  ->  (CSPRepInterlNS [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (CSPParAction a [ZVar (x2,[])]))
omega_CAction (CSPRepInterlNS [Choose (x,[]) (ZSetDisplay lsx)]
```

```
        (NSExprParam ns [ZVar (x1,[])])
        act)
  = (CSPRepInterlNS [Choose (x,[]) (ZSetDisplay lsx)]
        (NSExprParam ns [ZVar (x1,[])])
        (omega_CAction act))
```

$$\Omega_A(\{g\}) \mathrel{\widehat=} :[g, true]$$

```
omega_CAction (CActionCommand (CommandBrace g))
  = omega_CAction (CActionCommand (CPrefix g (ZTrue {reason = []})))
```

$$\Omega_A([g]) \mathrel{\widehat=} :[g]$$

```
omega_CAction (CActionCommand (CommandBracket g))
  = omega_CAction (CActionCommand (CPrefix1 g))
```

$$\Omega_A(A[old_1, ..., old_n := new_1, ..., new_n) \mathrel{\widehat=} \\ A[new_1, ..., new_n / old_1, ..., old_n)$$

```
omega_CAction (CSPRenAction a (CRenameAssign left right))
  = (CSPRenAction a (CRename right left))
```

In order to pattern match any other *Circus* construct not mentioned here, we propagate the *omega_CAction* function to the remainder of the constructs.

```
omega_CAction (CActionSchemaExpr vZSExpr) = (CActionSchemaExpr vZSExpr)
-- omega_CAction (CActionCommand vCCommand) = (CActionCommand vCCommand)
omega_CAction (CActionName vZName) = (CActionName vZName)
omega_CAction (CSPCommAction vComm vCAction) = (CSPCommAction vComm (omega_CAction vCAction))
-- omega_CAction (CSPGuard vZPred vCAction) = (CSPGuard vZPred (omega_CAction vCAction))
-- omega_CAction (CSPSeq v1CAction v2CAction) = (CSPSeq (omega_CAction (omega_CAction v1CAction)) (
-- omega_CAction (CSPExtChoice v1CAction v2CAction) = (CSPExtChoice (omega_CAction v1CAction) (omeg
-- omega_CAction (CSPIntChoice v1CAction v2CAction) = (CSPIntChoice (omega_CAction v1CAction) (omeg
omega_CAction (CSPNSParal v1NSExp vCSExp v2NSExp v1CAction v2CAction) = (CSPNSParal v1NSExp vCSExp
omega_CAction (CSPParal vCSExp v1CAction v2CAction) = (CSPParal vCSExp (omega_CAction v1CAction) (o
omega_CAction (CSPNSInter v1NSExp v2NSExp v1CAction v2CAction) = (CSPNSInter v1NSExp v2NSExp (omega
omega_CAction (CSPInterleave v1CAction v2CAction) = (CSPInterleave (omega_CAction v1CAction) (omega
-- omega_CAction (CSPHide vCAction vCSExp) = (CSPHide (omega_CAction vCAction) vCSExp)
omega_CAction (CSPParAction vZName vZExpr_lst) = (CSPParAction vZName vZExpr_lst)
omega_CAction (CSPRenAction vZName vCReplace) = (CSPRenAction vZName vCReplace)
-- omega_CAction (CSPRecursion vZName vCAction) = (CSPRecursion vZName (omega_CAction vCAction))
omega_CAction (CSPUnfAction vZName vCAction) = (CSPUnfAction vZName (omega_CAction vCAction))
omega_CAction (CSPUnParAction vZGenFilt_lst vCAction vZName) = (CSPUnParAction vZGenFilt_lst (omega
omega_CAction (CSPRepSeq vZGenFilt_lst vCAction) = (CSPRepSeq vZGenFilt_lst (omega_CAction vCAction
omega_CAction (CSPRepExtChoice vZGenFilt_lst vCAction) = (CSPRepExtChoice vZGenFilt_lst (omega_CAct
omega_CAction (CSPRepIntChoice vZGenFilt_lst vCAction) = (CSPRepIntChoice vZGenFilt_lst (omega_CAct
omega_CAction (CSPRepParalNS vCSExp vZGenFilt_lst vNSExp vCAction) = (CSPRepParalNS vCSExp vZGenFil
omega_CAction (CSPRepParal vCSExp vZGenFilt_lst vCAction) = (CSPRepParal vCSExp vZGenFilt_lst (omeg
omega_CAction (CSPRepInterlNS vZGenFilt_lst vNSExp vCAction) = (CSPRepInterlNS vZGenFilt_lst vNSExp
omega_CAction (CSPRepInterl vZGenFilt_lst vCAction) = (CSPRepInterl vZGenFilt_lst (omega_CAction vC
omega_CAction x = x
```

4.2 Definitions of Ω'_A

$$\Omega'_A(\mathbf{Skip}) \mathrel{\widehat=} \mathbf{Skip}$$

$$\Omega'_A(\mathbf{Stop}) \mathrel{\widehat=} \mathbf{Stop}$$

$$\Omega'_A(\mathbf{Chaos}) \mathrel{\widehat=} \mathbf{Chaos}$$

is written in Haskell as:

```
omega_prime_CAction :: CAction -> CAction
omega_prime_CAction CSPSkip = CSPSkip
omega_prime_CAction CSPStop = CSPStop
omega_prime_CAction CSPChaos = CSPChaos
```

$$\Omega'_A(c \longrightarrow A) \mathrel{\widehat=} c \longrightarrow \Omega'_A(A)$$

is written in Haskell as:

```
omega_prime_CAction (CSPCommAction (ChanComm c []) a)
  = (CSPCommAction (ChanComm c []) (omega_prime_CAction a))
```

$$\Omega'_A(c.e \longrightarrow A) \mathrel{\widehat=} c(vv_0, ..., vv_n, vl_0, ..., vl_m) \longrightarrow \Omega'_A(A)$$

is written in Haskell as:

```
omega_prime_CAction (CSPCommAction (ChanComm c [ChanDotExp e]) a)
  = (CSPCommAction (ChanComm c [ChanDotExp (rename_ZExpr e)]) (omega_prime_CAction a))
```

$$\begin{array}{l}\Omega'_A(c!e \longrightarrow A) \mathrel{\widehat=} \\ \qquad c.e \longrightarrow A\end{array}$$

```
omega_prime_CAction (CSPCommAction (ChanComm c [ChanOutExp e]) a)
  = omega_prime_CAction (CSPCommAction (ChanComm c [ChanDotExp e]) a)
omega_prime_CAction (CSPCommAction (ChanComm c ((ChanOutExp e):xs)) a)
  = omega_prime_CAction (CSPCommAction (ChanComm c ((ChanDotExp e):xs)) a)
```

$$\begin{array}{l}\Omega'_A(g \longrightarrow A) \mathrel{\widehat=} \\ \qquad g \mathbin{\&} \Omega'_A(A)\end{array}$$

is written in Haskell as:

```
omega_prime_CAction (CSPGuard g a)
  = (CSPGuard (rename_ZPred g) (omega_prime_CAction a))
```

I'm considering $x?k \neq x?k : P$ and I'm making the translation straightforward:

$$\begin{array}{l}\Omega'_A(c?x \longrightarrow A) \mathrel{\widehat=} \\ \qquad c?x \longrightarrow \Omega'_A(A)\end{array}$$

is written in Haskell as:

```
omega_prime_CAction (CSPCommAction (ChanComm c [ChanInp (x:xs)]) a)
  = (CSPCommAction (ChanComm c [ChanInp (x:xs)]) (omega_prime_CAction a))
```

$$\begin{array}{l}\Omega'_A(c?x : P \longrightarrow A) \mathrel{\widehat=} \\ \qquad c?x : P \longrightarrow \Omega'_A(A)\end{array}$$

where

$$x \in wrtV(A)$$

is written in Haskell as:

```
omega_prime_CAction (CSPCommAction (ChanComm c [ChanInpPred x p]) a)
  = (CSPCommAction (ChanComm c [ChanInpPred x p])
                  (omega_prime_CAction a))
```

$$\Omega'_A(A_1 \; ; \; A_2) \mathrel{\widehat=} \Omega'_A(A_1) \; ; \; \Omega_A(A_2)$$

is written in Haskell as:

```
omega_prime_CAction (CSPSeq ca cb)
  = (CSPSeq (omega_prime_CAction ca) (omega_CAction cb))
```

$$\Omega'_A(A_1 \sqcap A_2) \mathrel{\widehat=} \Omega'_A(A_1) \sqcap \Omega'_A(A_2)$$

is written in Haskell as:

```
omega_prime_CAction (CSPIntChoice ca cb)
  = (CSPIntChoice (omega_prime_CAction ca) (omega_prime_CAction cb))
```

$$\begin{array}{l} \Omega'_A(A_1 \mathrel{\Box} A_2) \mathrel{\widehat=} \\ \qquad (\Omega'_A(A_1) \mathrel{\Box} \Omega'_A(A_2)) \end{array}$$

is written in Haskell as:

```
omega_prime_CAction (CSPExtChoice ca cb)
  = (CSPExtChoice (omega_prime_CAction ca) (omega_prime_CAction cb))
```

$$\begin{array}{l} \Omega'_A(A1 \lbrack\!\lbrack \, ns1 \mid cs \mid ns2 \, \rbrack\!\rbrack \, A2) \mathrel{\widehat=} \\ \quad get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow \\ \quad get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow \\ \qquad \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} (\; \Omega'_A(A_1) \; ; \; terminate \longrightarrow \mathbf{Skip} \;) \\ \lbrack\!\lbrack \{\} \mid MEM_I \mid \{\} \rbrack\!\rbrack \\ MemoryMerge(\{v0 \mapsto vv0, \ldots\}, LEFT) \end{array} \right) \setminus MEM_I \\ \lbrack\!\lbrack \{\} \mid cs \mid \{\} \rbrack\!\rbrack \\ \left(\begin{array}{l} (\; \Omega'_A(A_2) \; ; \; terminate \longrightarrow \mathbf{Skip} \;) \\ \lbrack\!\lbrack \{\} \mid MEM_I \mid \{\} \rbrack\!\rbrack \\ MemoryMerge(\{v0 \mapsto vv0, \ldots\}, RIGHT) \end{array} \right) \setminus MEM_I \end{array} \right) \\ \lbrack\!\lbrack \{\} \mid MEM_I \mid \{\} \rbrack\!\rbrack \\ Merge \end{array} \right) \\ \qquad \setminus \{\!| \, mleft, mright \, |\!\} \end{array}$$

```
-- omega_prime_CAction (CSPNSParal ns1 cs ns2 a1 a2)
--   = make_get_com lsx (rename_vars_CAction (CSPHide
--    (CSPNSParal NSExpEmpty (CSExpr "MEMi") NSExpEmpty
--      (CSPNSParal NSExpEmpty cs NSExpEmpty
--       (CSPHide
--        (CSPNSParal NSExpEmpty (CSExpr "MEMi") NSExpEmpty
--         (CSPSeq a1 (CSPCommAction (ChanComm "terminate" []) CSPSkip))
--         (CSPParAction "MemoryMerge"
--          [ZSetDisplay [],
--                 ZVar ("LEFT",[])]))
--        (CSExpr "MEMi"))
--       (CSPHide
--        (CSPNSParal NSExpEmpty (CSExpr "MEMi") NSExpEmpty
--         (CSPSeq a2 (CSPCommAction (ChanComm "terminate" []) CSPSkip))
--         (CSPParAction "MemoryMerge"
--          [ZSetDisplay [],
--                 ZVar ("RIGHT",[])]))
--        (CSExpr "MEMi")))
--       (CActionName "Merge"))
--       (CChanSet ["mleft","mright"])))
--    where
--     lsx = union (map fst (remdups (free_var_CAction a1))) (map fst (remdups (free_var_CAction a2
```

$$\Omega'_A(\mathbin{\raise0.6ex\hbox{$\scriptstyle\circ$}\!\!\!;} \; x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Omega'_A(A(v_1) \; ; \; \ldots ; \; A(v_n))$$

is written in Haskell as:

```
omega_prime_CAction (CSPRepSeq [Choose (x,[]) (ZSeqDisplay xs)] (CSPParAction act [ZVar (x1,[])]))
  = case x == x1 of
    True -> omega_prime_CAction (rep_CSPRepSeq act xs)
    _  -> (CSPRepSeq [Choose (x,[]) (ZSeqDisplay xs)]
          (CSPParAction act [ZVar (x1,[])]))
omega_prime_CAction (CSPRepSeq [Choose (x,[]) v] act)
  = (CSPRepSeq [Choose (x,[]) v] (omega_prime_CAction act))
```

$$\Omega'_A(\Box\, x : \langle v_1, ..., v_n\rangle \bullet A(x)) \mathrel{\widehat=} \Omega'_A(A(v_1) \Box \ldots \Box A(v_n))$$

is written in Haskell as:

```
omega_prime_CAction (CSPRepExtChoice [Choose (x,[]) (ZSeqDisplay xs)] (CSPParAction act [ZVar (x1,[
  = case x == x1 of
    True -> omega_prime_CAction (rep_CSPRepExtChoice act xs)
    _  -> (CSPRepExtChoice [Choose (x,[]) (ZSeqDisplay xs)]
          (CSPParAction act [ZVar (x1,[])]))
omega_prime_CAction (CSPRepExtChoice [Choose (x,[]) v] act)
  = (CSPRepExtChoice [Choose (x,[]) v] (omega_prime_CAction act))
```

$$\Omega'_A(\bigsqcap x : \langle v_1, ..., v_n\rangle \bullet A(x)) \mathrel{\widehat=} \Omega'_A(A(v_1) \sqcap \ldots \sqcap A(v_n))$$

is written in Haskell as:

```
omega_prime_CAction (CSPRepIntChoice [Choose (x,[]) (ZSeqDisplay xs)]
          (CSPParAction act [ZVar (x1,[])]))
  = case x == x1 of
    True -> omega_prime_CAction(rep_CSPRepIntChoice act xs)
    _  -> (CSPRepIntChoice [Choose (x,[]) (ZSeqDisplay xs)]
          (CSPParAction act [ZVar (x1,[])]))
omega_prime_CAction (CSPRepIntChoice [Choose (x,[]) v] act)
  = (CSPRepIntChoice [Choose (x,[]) v] (omega_prime_CAction act))
```

$$\Omega'_A(\llbracket cs \rrbracket\, x : \langle v_1, ..., v_n\rangle \bullet \llbracket ns(x) \rrbracket\, A(x)) \mathrel{\widehat=}
\left(\begin{array}{l}
A(v_1) \\
\llbracket ns(v_1) \mid cs \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\}\rrbracket \\
\left(\ldots \left(\begin{array}{l}
\Omega'_A(A(v_n - 1)) \\
\llbracket ns(v_n - 1) \mid cs \mid ns(v_n)\rrbracket \\
A(v_n)
\end{array}\right) \right)
\end{array}\right)$$

is written in Haskell as:

```
omega_prime_CAction (CSPRepParalNS (CSExpr cs) [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (CSPParAction a [ZVar (x2,[])]))
  = case (x == x1) && (x == x2) of
    True -> omega_prime_CAction (rep_CSPRepParalNS a cs ns x lsx)
    _  -> (CSPRepParalNS (CSExpr cs) [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (CSPParAction a [ZVar (x2,[])]))
omega_prime_CAction (CSPRepParalNS (CSExpr cs) [Choose (x,[]) (ZSetDisplay lsx)] (NSExprParam ns [Z
  = (CSPRepParalNS (CSExpr cs) [Choose (x,[]) (ZSetDisplay lsx)] (NSExprParam ns [ZVar (x1,[])]) (o
```

$$\Omega'_A(\mathbf{val}\ Decl \bullet P) \mathrel{\widehat=} \mathbf{val}\ Decl \bullet \Omega'_A(P)$$

is written in Haskell as:

```
omega_prime_CAction (CActionCommand (CValDecl xs a))
  = (CActionCommand (CValDecl xs (omega_prime_CAction a)))
```

$$\Omega'_A \left(\; x_0, \ldots, x_n := e_0, \ldots, e_n \; \right) \mathrel{\widehat=} \\ \quad set.x_0!e_0 \longrightarrow \\ \quad \ldots \longrightarrow \\ \quad set.x_n!e_n \longrightarrow \mathbf{Skip}$$

```
omega_prime_CAction (CActionCommand (CAssign varls valls))
  =  (make_set_com omega_prime_CAction varls valls CSPSkip)
```

$$\Omega'_A(A \setminus cs) \mathrel{\widehat=} \Omega'_A(A) \setminus cs$$

is written in Haskell as:

```
omega_prime_CAction (CSPHide a cs) = (CSPHide (omega_prime_CAction a) cs)
```

$$\Omega'_A \left(\begin{array}{l} \mathbf{if}\ g_0 \longrightarrow A_0 \\ \quad [\!]\ \ldots \\ \quad [\!]\, g_n \longrightarrow A_n \\ \mathbf{fi} \end{array} \right) \mathrel{\widehat=} \\ \quad \begin{array}{l} \mathbf{if}\ g_0 \longrightarrow \Omega'_A(A_0) \\ \quad [\!]\ \ldots \\ \quad [\!]\, g_n \longrightarrow \Omega'_A(A_n) \\ \mathbf{fi} \end{array}$$

```
omega_prime_CAction (CActionCommand (CIf glx))
  = (CActionCommand (CIf (mk_guard_pair omega_prime_CAction guard_pair)))
  where
   guard_pair = get_guard_pair glx
```

$$\Omega'_A(\mu X \bullet A(X)) \mathrel{\widehat=} \mu X \bullet \Omega'_A(A(X))$$

is written in Haskell as:

```
omega_prime_CAction (CSPRecursion x c) = (CSPRecursion x (omega_prime_CAction c))
```

$$\Omega'_A(|||\ x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \\ \quad \left(\begin{array}{l} A(v_1) \\ [\![ns(v_1) \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\}]\!] \\ \left(\ldots \left(\begin{array}{l} \Omega'_A(A(v_n - 1)) \\ [\![ns(v_n - 1) \mid ns(v_n)]\!] \\ A(v_n) \end{array} \right) \right) \end{array} \right)$$

is written in Haskell as:

```
omega_prime_CAction (CSPRepInterlNS [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (CSPParAction a [ZVar (x2,[])]))
  = case (x == x1) && (x == x2) of
    True -> omega_prime_CAction (rep_CSPRepInterlNS a ns x lsx)
    _  -> (CSPRepInterlNS [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (CSPParAction a [ZVar (x2,[])]))
omega_prime_CAction (CSPRepInterlNS [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          act)
  = (CSPRepInterlNS [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (omega_prime_CAction act))
```

$\Omega'_A(\{g\}) \mathrel{\widehat=} :[g, true]$

```
omega_prime_CAction (CActionCommand (CommandBrace g))
  = omega_prime_CAction (CActionCommand (CPrefix g (ZTrue {reason = []})))
```

$\Omega'_A([g]) \mathrel{\widehat=} :[g]$

```
omega_prime_CAction (CActionCommand (CommandBracket g))
  = omega_prime_CAction (CActionCommand (CPrefix1 g))
```

$\Omega'_A(A[old_1, ..., old_n := new_1, ..., new_n) \mathrel{\widehat=}$
$\quad A[new_1, ..., new_n / old_1, ..., old_n)$

```
omega_prime_CAction (CSPRenAction a (CRenameAssign left right))
  = (CSPRenAction a (CRename right left))
```

In order to pattern match any other *Circus* construct not mentioned here, we propagate the *omega_CAction* function to the remainder of the constructs.

```
omega_prime_CAction (CActionSchemaExpr vZSExpr) = (CActionSchemaExpr vZSExpr)
-- omega_prime_CAction (CActionCommand vCCommand) = (CActionCommand vCCommand)
omega_prime_CAction (CActionName vZName) = (CActionName vZName)
omega_prime_CAction (CSPCommAction vComm vCAction) = (CSPCommAction vComm (omega_prime_CAction vCAc
-- omega_prime_CAction (CSPGuard vZPred vCAction) = (CSPGuard vZPred (omega_prime_CAction vCAction)
-- omega_prime_CAction (CSPSeq v1CAction v2CAction) = (CSPSeq (omega_prime_CAction (omega_prime_CAc
-- omega_prime_CAction (CSPExtChoice v1CAction v2CAction) = (CSPExtChoice (omega_prime_CAction v1CA
-- omega_prime_CAction (CSPIntChoice v1CAction v2CAction) = (CSPIntChoice (omega_prime_CAction v1CA
omega_prime_CAction (CSPNSParal v1NSExp vCSExp v2NSExp v1CAction v2CAction) = (CSPNSParal v1NSExp v
omega_prime_CAction (CSPParal vCSExp v1CAction v2CAction) = (CSPParal vCSExp (omega_prime_CAction v
omega_prime_CAction (CSPNSInter v1NSExp v2NSExp v1CAction v2CAction) = (CSPNSInter v1NSExp v2NSExp
omega_prime_CAction (CSPInterleave v1CAction v2CAction) = (CSPInterleave (omega_prime_CAction v1CAc
-- omega_prime_CAction (CSPHide vCAction vCSExp) = (CSPHide (omega_prime_CAction vCAction) vCSExp)
omega_prime_CAction (CSPParAction vZName vZExpr_lst) = (CSPParAction vZName vZExpr_lst)
omega_prime_CAction (CSPRenAction vZName vCReplace) = (CSPRenAction vZName vCReplace)
-- omega_prime_CAction (CSPRecursion vZName vCAction) = (CSPRecursion vZName (omega_prime_CAction v
omega_prime_CAction (CSPUnfAction vZName vCAction) = (CSPUnfAction vZName (omega_prime_CAction vCAc
omega_prime_CAction (CSPUnParAction vZGenFilt_lst vCAction vZName) = (CSPUnParAction vZGenFilt_lst
omega_prime_CAction (CSPRepSeq vZGenFilt_lst vCAction) = (CSPRepSeq vZGenFilt_lst (omega_prime_CAct
omega_prime_CAction (CSPRepExtChoice vZGenFilt_lst vCAction) = (CSPRepExtChoice vZGenFilt_lst (omeg
omega_prime_CAction (CSPRepIntChoice vZGenFilt_lst vCAction) = (CSPRepIntChoice vZGenFilt_lst (omeg
omega_prime_CAction (CSPRepParalNS vCSExp vZGenFilt_lst vNSExp vCAction) = (CSPRepParalNS vCSExp vZ
omega_prime_CAction (CSPRepParal vCSExp vZGenFilt_lst vCAction) = (CSPRepParal vCSExp vZGenFilt_lst
omega_prime_CAction (CSPRepInterlNS vZGenFilt_lst vNSExp vCAction) = (CSPRepInterlNS vZGenFilt_lst
omega_prime_CAction (CSPRepInterl vZGenFilt_lst vCAction) = (CSPRepInterl vZGenFilt_lst (omega_prim
omega_prime_CAction x = x
```

5 Circus Refinement Laws

Law 1 (var-exp-par)

$(\mathbf{var}\ \ d : T \bullet A1) \llbracket\, ns1 \mid cs \mid ns2 \,\rrbracket\, A2$
$=$
$(\mathbf{var}\ \ d : T \bullet A1 \llbracket\, ns1 \mid cs \mid ns2 \,\rrbracket\, A2)$

- : *From D24.1* – $\{d, d'\} \cap FV(A2) = \varnothing$
- : *From Oliveira's Thesis:* $x \notin FV(A_2) \cup ns_1 \cup ns_2$

```
crl_var_exp_par :: CAction -> Refinement CAction
crl_var_exp_par e@(CSPNSParal ns1 cs ns2 (CActionCommand (CVarDecl [(Choose (d,[]) t)] a1)) a2)
  = Done{orig = Just e, refined = Just ref, proviso = [p1]}
    where
      ref = (CActionCommand (CVarDecl [(Choose (d,[]) t)] (CSPNSParal ns1 cs ns2 a1 a2)))
      p1 = (ZEqual (ZCall (ZVar ("\\cap",[])) (ZTuple [ZSetDisplay [ZVar (d,[]),ZVar (d,["'"])],ZSe
crl_var_exp_par _ = None
```

Law 2 (var-exp-par-2)

$$(\mathbf{var}\ \ d : T \bullet A1) \llbracket ns1 \mid cs \mid ns2 \rrbracket (\mathbf{var}\ \ d : T \bullet A2)$$
$$=$$
$$(\mathbf{var}\ \ d : T \bullet A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2)$$

```
crl_var_exp_par2 :: CAction -> Refinement CAction
crl_var_exp_par2 e@(CSPNSParal ns1 cs ns2
                    (CActionCommand (CVarDecl d1 a1))
                    (CActionCommand (CVarDecl d2 a2)))
  = case (d1 == d2) of
      True -> Done{orig = Just e, refined = Just (CActionCommand (CVarDecl d1
                      (CSPNSParal ns1 cs ns2 a1 a2))),
          proviso = []}
      False -> None
crl_var_exp_par2 _ = None
```

Law 3 (var-exp-rec)

$$\mu X \bullet (\mathbf{var}\, x : T \bullet F(X))$$
$$=$$
$$\mathbf{var}\, x : T \bullet (\mu X \bullet F(X))$$

provided ***x is initiated before use in F***

```
crl_var_exp_rec :: CAction -> Refinement CAction
crl_var_exp_rec e@(CSPRecursion mX (CActionCommand (CValDecl [Choose (x,[]) (ZVar (t,[]))] (CSPUnfA
  = case (mX == mX1) of
      True -> Done{orig = Just e, refined = Just (CActionCommand (CValDecl [Choose (x,[]) (ZVar (t,
      False -> None
crl_var_exp_rec _ = None
```

Law 4 (Variable block/Sequence—extension*)

$$A1\, ;\ \ (\mathbf{var}\ \ x : T \bullet A2)\, ;\ \ A3$$
$$=$$
$$(\mathbf{var}\ \ x : T \bullet A1\, ;\ \ \ A2\, ;\ \ A3)$$

provided $x \notin FV(A1) \cup FV(A3)$

```
crl_variableBlockSequenceExtension :: CAction -> Refinement CAction
crl_variableBlockSequenceExtension
      e@(CSPSeq (CSPSeq a1
            (CActionCommand (CVarDecl [(Choose (x,[]) t)] a2))) a3)
  =  Done{orig = Just e, refined = Just (CActionCommand (CVarDecl [(Choose (x,[]) t)] (CSPSeq (CSPS
  where
    prov = (ZMember (ZSetDisplay [ZVar (x,[])]) (ZCall (ZVar ("\\cup",[])) (ZTuple [ZSetDisplay
$ zvar_to_zexpr (free_var_CAction a1),ZSetDisplay $ zvar_to_zexpr (free_var_CAction a3)])))
crl_variableBlockSequenceExtension
      e@(CSPSeq a1
            (CActionCommand (CVarDecl [(Choose (x,[]) t)] a2)))
  =  Done{orig = Just e, refined = Just (CActionCommand (CVarDecl [(Choose (x,[]) t)] (CSPSeq a1 a2
  where
    prov = (ZMember (ZSetDisplay [ZVar (x,[])]) (ZCall (ZVar ("\\cup",[])) (ZTuple [ZSetDisplay
$ zvar_to_zexpr (free_var_CAction a1)])))
crl_variableBlockSequenceExtension _ = None
```

Law 5 (Variable block introduction*)

$$A = \mathbf{var}\ x : T \bullet A$$

provided $x \notin FV(A)$

```
crl_variableBlockIntroduction :: CAction -> ZVar -> ZExpr -> Refinement CAction
crl_variableBlockIntroduction a x t
  =  Done{orig = Just a, refined = Just (CActionCommand (CVarDecl [(Choose x t)] a)),
          proviso=[prov]}
  where
    prov = (ZNot (ZMember (ZVar x) (ZSetDisplay $ zvar_to_zexpr (free_var_CAction a))))
crl_variableBlockIntroduction _ _ _ = None
```

```
crl_variableBlockIntroduction_backwards :: CAction -> Refinement CAction
crl_variableBlockIntroduction_backwards e@(CActionCommand (CVarDecl [(Choose (x,[]) t)] a))
  =  Done{orig = Just e, refined = Just a, proviso = [prov]}
  where
  prov = (ZNot (ZMember (ZVar (x,[])) (ZSetDisplay $ zvar_to_zexpr (free_var_CAction a))))

crl_variableBlockIntroduction_backwards e@(CSPCommAction (ChanComm x f) (CActionCommand (CValDecl [
  =  Done{orig = Just e, refined = Just ref, proviso = [prov]}
  where
    ref = (CActionCommand (CValDecl [Choose (y,[]) (ZVar (t,[]))] (CSPCommAction (ChanComm x f) a))
    prov = (ZNot (ZMember (ZVar (y,[])) (ZSetDisplay $ zvar_to_zexpr (free_var_CAction (CSPCommActi
crl_variableBlockIntroduction_backwards _ = None
```

Law 6 (Variable block introduction—2*)

$$\mathbf{var}\ x : T_1;\ y : T_2 \bullet A = \mathbf{var}\ y : T_2 \bullet A$$

provided $x \notin FV(A)$

```
crl_variableBlockIntroduction2_backwards e@(CActionCommand (CVarDecl ((Choose (x,[]) t):xs) a))
  =  Done{orig = Just e, refined = Just (CActionCommand (CVarDecl xs a)), proviso = [prov]}
  where
    prov = (ZNot (ZMember (ZVar (x,[])) (ZSetDisplay $ zvar_to_zexpr (free_var_CAction a))))
crl_variableBlockIntroduction2_backwards _ = None
```

Law 7 (join—blocks)

$$\mathbf{var}\ x : T_1 \bullet \mathbf{var}\ y : T_2 \bullet A$$
$$=$$
$$\mathbf{var}\ x : T_1;\ y : T_2 \bullet A$$

```
crl_joinBlocks :: CAction -> Refinement CAction
crl_joinBlocks e@(CActionCommand (CVarDecl [(Choose x t1)] (CActionCommand (CVarDecl [(Choose y t2)
  =  Done{orig = Just e, refined = Just (CActionCommand (CVarDecl [(Choose x t1),(Choose y t2)] a))
          proviso=[]}
crl_joinBlocks _ = None
```

Law 8 (Sequence unit)

$$(A)\mathbf{Skip}\ ;\ A = A$$
$$(B)A = A\ ;\ \mathbf{Skip}$$

```
crl_seqSkipUnit_a :: CAction -> Refinement CAction
crl_seqSkipUnit_a e@(CSPSeq CSPSkip a) =  Done{orig = Just e, refined = Just a, proviso=[]}
crl_seqSkipUnit_a _ = None
crl_seqSkipUnit_b e@(CSPSeq a CSPSkip) =  Done{orig = Just e, refined = Just a, proviso=[]}
crl_seqSkipUnit_b _ = None
-- crl_seqSkipUnit_b :: CAction -> Refinement CAction
-- crl_seqSkipUnit_b a =  Done{orig = Just a, refined = Just (CSPSeq a CSPSkip),proviso=[]}
```

Law 9 (Recursion unfold)

$$\mu X \bullet F(X)$$
$$=$$
$$F(\mu X \bullet F(X))$$

```
crl_recUnfold :: CAction -> Refinement CAction
crl_recUnfold e@(CSPRecursion x1 (CSPUnfAction f (CActionName x2)))
  = case (x1 == x2) of
      True -> Done{orig = Just e, refined = Just (CSPUnfAction f (CSPRecursion x1 (CSPUnfAction f (
      False -> None
crl_recUnfold _ = None
```

Law 10 (Parallelism composition introduction 1*)

$$c \longrightarrow A$$
$$=$$
$$(c \longrightarrow A \llbracket ns1 \mid \{\!| c |\!\} \mid ns2 \rrbracket c \longrightarrow Skip)$$

$$c.e \longrightarrow A$$
$$=$$
$$(c.e \longrightarrow A \llbracket ns1 \mid \{\!| c |\!\} \mid ns2 \rrbracket c.e \longrightarrow Skip)$$

provided

- $c \notin usedC(A)$
- $wrtV(A) \subseteq ns1$

```
crl_parallelismIntroduction1b :: CAction -> NSExp -> [ZName] -> NSExp -> Refinement CAction
crl_parallelismIntroduction1b
  ei@(CSPCommAction (ChanComm c
      [ChanDotExp e]) a)
      (NSExprMult ns1) cs (NSExprMult ns2)
  =  Done{orig = Just ei,
          refined = Just (CSPNSParal (NSExprMult ns1) (CChanSet cs) (NSExprMult ns2)
                (CSPCommAction (ChanComm c [ChanDotExp e]) a)
                (CSPCommAction (ChanComm c [ChanDotExp e]) CSPSkip)),
          proviso=[p1,p2]}
    where
      p1 = (ZNot (ZMember (ZVar (c,[])) (ZTuple [ZSetDisplay (usedC a)])))
      p2 = (ZMember (ZTuple [ZSetDisplay (getWrtV a),ZSetDisplay $ zname_to_zexpr ns1]) (ZVar ("\\s
crl_parallelismIntroduction1b _ _ _ _ = None

crl_parallelismIntroduction1a :: CAction -> NSExp -> [ZName] -> NSExp -> Refinement CAction
crl_parallelismIntroduction1a
    ei@(CSPCommAction (ChanComm c e) a)
        (NSExprMult ns1) cs (NSExprMult ns2)
  =  Done{orig = Just ei, refined = Just (CSPNSParal (NSExprMult ns1) (CChanSet cs) (NSExprMult ns2
                  (CSPCommAction (ChanComm c e) a)
                  (CSPCommAction (ChanComm c e) CSPSkip)),proviso=[p1,p2]}
    where
      p1 = (ZNot (ZMember (ZVar (c,[]))  (ZTuple [ZSetDisplay (usedC a)])))
      p2 = (ZMember (ZTuple [ZSetDisplay (getWrtV a),ZSetDisplay $ zname_to_zexpr ns1]) (ZVar ("\\s
crl_parallelismIntroduction1a _ _ _ _ = None
```

Law 11 (Channel extension 1)

$$A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2$$
$$=$$
$$A1 \llbracket ns1 \mid cs \cup \{\!| c |\!\} \mid ns2 \rrbracket A2$$

provided $c \notin usedC(A1) \cup usedC(A2)$

```
crl_chanExt1 :: CAction -> ZName -> Refinement CAction
crl_chanExt1 e@(CSPNSParal ns1 (CChanSet cs) ns2 a1 a2) c
```

```
  =  Done{orig = Just e, refined = Just (CSPNSParal ns1 (ChanSetUnion
                      (CChanSet cs) (CChanSet [c])) ns2 a1 a2),
                      proviso=[p1]}
    where
      p1 = (ZNot (ZMember (ZVar (c,[])) (ZCall (ZVar ("\\cup",[])) (ZTuple [ZSetDisplay (usedC a1),
crl_chanExt1 _ _ = None
```

Law 12 (Hiding expansion 2*)

$$A \setminus cs$$
$$=$$
$$A \setminus (cs \cup \{\!| c |\!\})$$

provided $c \notin usedC(A)$

```
crl_hidingExpansion2 :: CAction -> ZName -> Refinement CAction
crl_hidingExpansion2 e@(CSPHide a (CChanSet cs)) c
  = Done{orig = Just e, refined = Just (CSPHide a (ChanSetUnion (CChanSet cs) (CChanSet [c]))),
                      proviso=[p1]}
    where
      p1 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay (usedC a))))
crl_hidingExpansion2 _ _ = None
```

Law 13 (Prefix/Hiding*)

$$(c \longrightarrow Skip) \setminus \{\!| c |\!\} = \mathbf{Skip}$$

$$(c.e \longrightarrow Skip) \setminus \{\!| c |\!\} = \mathbf{Skip}$$

```
crl_prefixHiding :: CAction -> Refinement CAction
crl_prefixHiding
  e@(CSPHide (CSPCommAction (ChanComm c _) CSPSkip) (CChanSet [c1]))
  = case (c == c1) of
      True        -> Done{orig = Just e, refined = Just CSPSkip,proviso=[]}
      otherwise   -> None
crl_prefixHiding _ = None
```

Law 14 (Hiding Identity*)

$$A \setminus cs = A$$

provided $cs \cap usedC(A) = \varnothing$

```
crl_hidingIdentity :: CAction -> Refinement CAction
crl_hidingIdentity e@(CSPHide a (CChanSet cs))
  = Done{orig = Just e, refined = Just a, proviso=[p1]}
    where
      p1 = (ZEqual (ZCall (ZVar ("\\cap",[])) (ZTuple [ZSetDisplay $zname_to_zexpr cs,ZSetDisplay (
crl_hidingIdentity _ = None
```

Law 15 (Parallelism composition/External choice—exchange)

$$(A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2) \Box (B1 \llbracket ns1 \mid cs \mid ns2 \rrbracket B2)$$
$$=$$
$$(A1 \Box B1) \llbracket ns1 \mid cs \mid ns2 \rrbracket (A2 \Box B2)$$

provided $A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket B2 = A2 \llbracket ns1 \mid cs \mid ns2 \rrbracket B1 = Stop$

```
crl_parExtChoiceExchange :: CAction -> Refinement CAction
crl_parExtChoiceExchange
    e@(CSPExtChoice
      (CSPNSParal ns1 cs ns2 a1 a2)
      (CSPNSParal ns11 cs1 ns21 b1 b2))
  = case pred of
      True ->  Done{orig = Just e, refined = Just ref, proviso=[]}
```

```
      False -> None
      where
        astop = (CSPNSParal ns1 cs ns2 a1 b2) == (CSPNSParal ns2 cs ns1 a2 b1)
        ref = (CSPNSParal ns1 cs ns2
                  (CSPExtChoice a1 b1)
                  (CSPExtChoice a2 b2))
        pred = (ns1 == ns11 && cs1 == cs && ns2 == ns21) && astop
crl_parExtChoiceExchange _ = None
```

Law 16 (External choice unit)

$$Stop \Box A = A$$

```
crl_extChoiceStopUnit :: CAction -> Refinement CAction
crl_extChoiceStopUnit e@(CSPExtChoice CSPStop a)
  = Done{orig = Just e, refined = Just a,proviso=[]}
crl_extChoiceStopUnit _
  = None
```

Law 17 (Hiding/External choice—distribution*)

$$(A1 \Box A2) \setminus cs$$
$$=$$
$$(A1 \setminus cs) \Box (A2 \setminus cs)$$

provided $(initials(A1) \cup initials(A2)) \cap cs = \varnothing$

```
crl_hidingExternalChoiceDistribution :: CAction -> Refinement CAction
crl_hidingExternalChoiceDistribution
    e@(CSPHide (CSPExtChoice a1 a2) (CChanSet cs))
  = Done{orig = Just e, refined = Just ref, proviso=[p1]}
    where
      p1 = (ZEqual (ZCall (ZVar ("\\cap",[])) (ZTuple [ZCall (ZVar ("\\cup",[])) (ZTuple [ZSetDispl
      ref = (CSPExtChoice
                (CSPHide a1 (CChanSet cs))
                (CSPHide a2 (CChanSet cs)))


crl_hidingExternalChoiceDistribution _ = None
```

Law 18 (Parallelism composition Deadlocked 1*)

$$(c1 \longrightarrow A1) \llbracket ns1 \mid cs \mid ns2 \rrbracket (c2 \longrightarrow A2)$$
$$=$$
$$\mathbf{Stop}$$
$$=$$
$$\mathbf{Stop} \llbracket ns1 \mid cs \mid ns2 \rrbracket (c2 \longrightarrow A2)$$

provided

- $c1 \neq c2$
- $\{c1, c2\} \subseteq cs$

```
crl_parallelismDeadlocked1 :: CAction -> Refinement CAction
crl_parallelismDeadlocked1
    e@(CSPNSParal ns1 (CChanSet cs) ns2
          (CSPCommAction (ChanComm c1 x) a1)
          (CSPCommAction (ChanComm c2 y) a2))
  =  Done{orig = Just e, refined = Just ref, proviso=[p1,p2]}
    where
      p1 = ZNot (ZEqual (ZVar (c1,[])) (ZVar (c2,[])))
      p2 = (ZMember (ZTuple [ZSetDisplay [ZVar (c1,[]),ZVar (c2,[])],ZSetDisplay $ zname_to_zexpr c
      ref = (CSPNSParal ns1 (CChanSet cs) ns2
                CSPStop
```

```
                (CSPCommAction (ChanComm c2 y) a2))
crl_parallelismDeadlocked1 e@(CSPNSParal ns1 (CChanSet cs) ns2
                              CSPStop (CSPCommAction c2 a2))
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = CSPStop
crl_parallelismDeadlocked1 _ = None
```

Law 19 (Sequence zero)

$$\mathbf{Stop} \; ; \; A = \mathbf{Stop}$$

```
crl_seqStopZero :: CAction -> Refinement CAction
crl_seqStopZero e@(CSPSeq CSPStop a)
  =  Done{orig = Just e, refined = Just CSPStop,proviso=[]}
crl_seqStopZero _ = None
```

Law 20 (Communication/Parallelism composition—distribution)

$$(c!e \longrightarrow A1) \llbracket ns1 \mid cs \mid ns2 \rrbracket (c?x \longrightarrow A2(x))$$
$$=$$
$$c.e \longrightarrow (A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2(e))$$

provided

- $c \in cs$
- $x \notin FV(A2)$.

TODO: implement proviso

```
crl_communicationParallelismDistribution :: CAction -> Refinement CAction
crl_communicationParallelismDistribution
    ei@(CSPNSParal ns1 (CChanSet cs) ns2
        (CSPCommAction (ChanComm c [ChanOutExp e]) a1)
        (CSPCommAction (ChanComm c1 [ChanInp x1])
          (CSPParAction a2 [ZVar (x,[])])))
  = case pred of
      True  -> Done{orig = Just ei, refined = Just ref, proviso=[p1,p2]}
      False -> None
    where
      p1 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs)))
      p2 = (ZNot (ZMember (ZVar (x,[])) (ZSetDisplay $ zvar_to_zexpr(free_var_CAction (CSPParActio
      ref = (CSPCommAction (ChanComm c [ChanDotExp e])
                  (CSPNSParal ns1 (CChanSet cs) ns2 a1 (CSPParAction a2 [e])))
      pred = (c == c1 && x == x1)
crl_communicationParallelismDistribution _ = None
```

Law 21 (Channel extension 3*)

$$(A1 \llbracket ns1 \mid cs1 \mid ns2 \rrbracket A2(e)) \setminus cs2$$
$$=$$
$$((c!e \longrightarrow A1) \llbracket ns1 \mid cs1 \mid ns2 \rrbracket (c?x \longrightarrow A2(x))) \setminus cs2$$

provided

- $c \in cs1$
- $c \in cs2$
- $x \notin FV(A2)$

TODO: implement proviso

```
crl_channelExtension3 ei@(CSPHide
          (CSPNSParal ns1 (CChanSet cs1) ns2 a1 (CActionCommand (CVarDecl [Choose (e,[]) t1] mact))
  =  Done{orig = Just ei, refined = Just ref, proviso=[p1,p2,p3]}
    where
      p1 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs1)))
      p2 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs2)))
      p3 = (ZNot (ZMember (ZVar (x,[])) (ZSetDisplay $ zvar_to_zexpr(free_var_CAction (CActionComma
      ref = (CSPHide
                (CSPNSParal ns1 (CChanSet cs1) ns2
                    (CSPCommAction (ChanComm c [ChanOutExp (ZVar (e,[]))]) a1)
                    (CSPCommAction (ChanComm c [ChanInp x])
                        (CActionCommand (CVarDecl [Choose (x,[]) t1] mact))))
                (CChanSet cs2))
crl_channelExtension3 _ _ _= None
```

```
crl_channelExtension3_backwards ei@(CSPHide (CSPNSParal ns1 (CChanSet cs1) ns2 (CSPCommAction (Chan
  = case pred of
      True -> Done{orig = Just ei, refined = Just ref, proviso=[p1,p2,p3]}
      False -> None
    where
       p1 = (ZNot (ZMember (ZVar (c1,[])) (ZSetDisplay $ zname_to_zexpr cs1)))
       p2 = (ZNot (ZMember (ZVar (c2,[])) (ZSetDisplay $ zname_to_zexpr cs2)))
       p3 = (ZNot (ZMember (ZVar (x,[])) (ZSetDisplay $ zvar_to_zexpr(free_var_CAction (CSPParActio
       pred = (c1 == c2) && (x == x1)
       ref = (CSPHide (CSPNSParal ns1 (CChanSet cs1) ns2 a1 (CSPParAction a2 [e])) (CChanSet cs2))
crl_channelExtension3_backwards _ = None
```

Law 22 (Channel extension 4*)

$$(A1 \llbracket ns1 \mid cs1 \mid ns2 \rrbracket A2) \setminus cs2$$
$$=$$
$$((c \longrightarrow A1) \llbracket ns1 \mid cs1 \mid ns2 \rrbracket (c \longrightarrow A2)) \setminus cs2$$
$$(A1 \llbracket ns1 \mid cs1 \mid ns2 \rrbracket A2) \setminus cs2$$
$$=$$
$$((c.e \longrightarrow A1) \llbracket ns1 \mid cs1 \mid ns2 \rrbracket (c.e \longrightarrow A2)) \setminus cs2$$

provided

- $c \in cs1$
- $c \in cs2$

```
crl_channelExtension4 ei@(CSPHide (CSPNSParal ns1 (CChanSet cs1) ns2 a1 a2) (CChanSet cs2)) (ChanCo
  =  Done{orig = Just ei, refined = Just ref, proviso=[p1,p2]}
    where
       p1 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs1)))
       p2 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs2)))
       ref = (CSPHide (CSPNSParal ns1 (CChanSet cs1) ns2 (CSPCommAction (ChanComm c [ChanOutExp (e)
crl_channelExtension4 ei@(CSPHide (CSPNSParal ns1 (CChanSet cs1) ns2 a1 a2)
                                (CChanSet cs2)) (ChanComm c e)
  = Done{orig = Just ei, refined = Just ref, proviso=[p1,p2]}
    where
      p1 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs1)))
      p2 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs2)))
      ref = (CSPHide (CSPNSParal ns1 (CChanSet cs1) ns2
                          (CSPCommAction (ChanComm c e) a1)
                           (CSPCommAction (ChanComm c e) a2))
                        (CChanSet cs2))
crl_channelExtension4 _ _ = None
```

Law 26 (prom-var-state)

```
crl_promVarState :: CProc -> Refinement CProc
crl_promVarState
  e@(ProcMain
      (ZSchemaDef (ZSPlain st) s)
      [CParAction l (CircusAction (CActionCommand (CValDecl [Choose (x,[]) (ZVar (t,[]))] a)))]
```

```
      (CActionCommand (CValDecl [Choose (x1,[]) (ZVar (t1,[]))] ma)))
  = case (x==x1 && t == t1) of
        True -> Done{orig = Just e, refined = Just ref, proviso=[]}
        False -> None
    where
      ref = (ProcMain
                   (ZSchemaDef (ZSPlain st) (ZS2 ZSAnd s (ZSchema [Choose (x,[]) (ZVar (t,[]))])))
                   [CParAction l (CircusAction a)] ma)

crl_promVarState _ = None
```

Law 27 (prom-var-state-2)

```
crl_promVarState2 :: CProc -> ZSName -> Refinement CProc
crl_promVarState2
    e@(ProcStalessMain
      [CParAction lact (ParamActionDecl [(Choose v t)] act)]
      (CActionCommand (CVarDecl [Choose v1 t1] mact))) st
  = case (v==v1 && t == t1) of
      True -> Done{orig = Just e, refined = Just ref, proviso=[]}
      False -> None
      where
        ref = (ProcMain (ZSchemaDef st (ZSchema [Choose v1 t1]))
                  [CParAction lact act] mact)
crl_promVarState2
    e@(ProcStalessMain
      [CParAction lact (ParamActionDecl ((Choose v t):xs) act)]
      (CActionCommand (CVarDecl ((Choose v1 t1):ys) mact))) st
  = case (v==v1 && t == t1) of
      True -> Done{orig = Just e, refined = Just ref, proviso=[]}
      False -> None
      where
        ref = (ProcMain (ZSchemaDef st (ZSchema [Choose v1 t1]))
                  [CParAction lact (ParamActionDecl xs act)]
                  (CActionCommand (CVarDecl ys mact)))
```

Law 23 (Parallelism composition unit*)

$$\mathbf{Skip} \llbracket ns1 \mid cs \mid ns2 \rrbracket \mathbf{Skip} = \mathbf{Skip}$$

```
crl_parallelismUnit1 :: CAction -> Refinement CAction
crl_parallelismUnit1 e@(CSPNSParal _ _ _ CSPSkip CSPSkip)
  =  Done{orig = Just e, refined = Just CSPSkip, proviso=[]}
crl_parallelismUnit1 _ = None
```

Law 24 (Parallelism composition/Interleaving Equivalence*)

$$A1 \,|\!\![\, ns2 \mid ns2 \,]\!\!|\, A2$$
$$=$$
$$A1 \llbracket ns2 \mid \varnothing \mid ns2 \rrbracket A2$$

```
crl_parallInterlEquiv :: CAction -> Refinement CAction
crl_parallInterlEquiv e@(CSPNSInter ns1 ns2 a1 a2)
  =  Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPNSParal ns1 CSEmpty ns2 a1 a2)
crl_parallInterlEquiv _ = None
-- crl_parallInterlEquiv_backwards :: CAction -> Refinement CAction
-- crl_parallInterlEquiv_backwards e@(CSPNSParal ns1 CSEmpty ns2 a1 a2)
--   = Done{orig = Just e, refined = Just ref, proviso=[]}
--     where
--       ref = (CSPNSInter ns1 ns2 a1 a2)
-- crl_parallInterlEquiv_backwards _ = None
```

Law 25 (Hiding/Sequence—distribution*)

$$(A1 ;\ \ A2) \setminus cs$$
$$=$$
$$(A1 \setminus cs) ;\ \ (A2 \setminus cs)$$

```
crl_hidingSequenceDistribution :: CAction -> Refinement CAction
crl_hidingSequenceDistribution e@(CSPHide (CSPSeq a1 a2) cs)
    = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPSeq (CSPHide a1 cs) (CSPHide a2 cs))
crl_hidingSequenceDistribution _ = None
```

Law 26 (Guard/Sequence—associativity)

$$(g \mathbin{\&} A1) \;;\; A2$$
$$=$$
$$g \mathbin{\&} (A1 \;;\; A2)$$

```
crl_guardSeqAssoc :: CAction -> Refinement CAction
crl_guardSeqAssoc e@(CSPSeq (CSPGuard g a1) a2)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where ref = (CSPGuard g (CSPSeq a1 a2))
crl_guardSeqAssoc _ = None
```

Law 27 (Input prefix/Parallelism composition—distribution 2*)

$$c?x \longrightarrow (A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2)$$
$$=$$
$$(c?x \longrightarrow A1) \llbracket ns1 \mid cs \mid ns2 \rrbracket A2$$

provided

- $c \notin cs$
- $x \notin usedV(A2)$
- $initials(A2) \subseteq cs$
- $A2$ *is deterministic*

TODO: implement proviso

```
crl_inputPrefixParallelismDistribution2 :: CAction -> Refinement CAction
crl_inputPrefixParallelismDistribution2
      e@(CSPCommAction (ChanComm c [ChanInp x])
          (CSPNSParal ns1 (CChanSet cs) ns2 a1 a2))
  = Done{orig = Just e, refined = Just ref, proviso=[p1,p2,p3]}
    where
      ref = (CSPNSParal ns1 (CChanSet cs) ns2 (CSPCommAction (ChanComm c [ChanInp x]) a1) a2)
      p1 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs)))
      p2 = (ZNot (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr(usedV a2))))
      p3 = (ZMember (ZTuple [ZSetDisplay (initials a2),ZSetDisplay (zname_to_zexpr cs)]) (ZVar ("\\
      p4 = "a2 is deterministic"
crl_inputPrefixParallelismDistribution2 _ = None
```

Law 28 (Prefix/*Skip)**

$$c \longrightarrow A = (c \longrightarrow \mathbf{Skip}) \;;\; A$$

$$c.e \longrightarrow A = (c.e \longrightarrow \mathbf{Skip}) \;;\; A$$

```
-- crl_prefixSkip :: CAction -> Refinement CAction
-- crl_prefixSkip e@(CSPCommAction (ChanComm c [ChanDotExp x]) a)
--   = Done{orig = Just e, refined = Just ref, proviso=[]}
--     where
--       ref = (CSPSeq (CSPCommAction
--                (ChanComm c [ChanDotExp x]) CSPSkip) a)
-- crl_prefixSkip e@(CSPCommAction c a)
--   = Done{orig = Just e, refined = Just ref, proviso=[]}
--     where
```

```
--        ref = (CSPSeq (CSPCommAction c CSPSkip) a)
-- crl_prefixSkip _ = None

-- 9/04/18
-- I had to make these two auxiliary functions in order to
-- put the crl_prefixSkip_backwards function working properly.
-- Basically, it searches for a CSPSkip within a prefixed action
-- and then removes the CSPSkip replacing it with a2, the RHS of
-- a CSPSeq action.
endPrefWithSkip (CSPCommAction c CSPSkip) = True
endPrefWithSkip (CSPCommAction c c1) = endPrefWithSkip c1
endPrefWithSkip _ = False

remPrefSkip a2 (CSPCommAction c CSPSkip) = (CSPCommAction c a2)
remPrefSkip a2 (CSPCommAction c c1) = (CSPCommAction c (remPrefSkip a2 c1))

crl_prefixSkip_backwards :: CAction -> Refinement CAction
crl_prefixSkip_backwards e@(CSPSeq (CSPCommAction (ChanComm c [ChanDotExp x]) CSPSkip) a)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPCommAction (ChanComm c [ChanDotExp x]) a)
crl_prefixSkip_backwards e@(CSPSeq a1 a2)
  | endPrefWithSkip a1
      = Done{orig = Just e, refined = Just ref, proviso=[]}
  | otherwise = None
    where
      ref = remPrefSkip a2 a1
crl_prefixSkip_backwards _ = None
```

Law 29 (Prefix/Parallelism composition—distribution)

$$c \longrightarrow (A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2)$$
$$=$$
$$(c \longrightarrow A1) \llbracket ns1 \mid cs \cup \{\!| c |\!\} \mid ns2 \rrbracket (c \longrightarrow A2)$$
$$c.e \longrightarrow (A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2)$$
$$=$$
$$(c.e \longrightarrow A1) \llbracket ns1 \mid cs \cup \{\!| c |\!\} \mid ns2 \rrbracket (c.e \longrightarrow A2)$$

provided $c \notin usedC(A1) \cup usedC(A2)$ *or* $c \in cs$

```
crl_prefixParDist :: CAction -> Refinement CAction
crl_prefixParDist e@(CSPCommAction (ChanComm c [])
                    (CSPNSParal ns1 (CChanSet cs) ns2 a1 a2))
  = Done{orig = Just e, refined = Just ref, proviso=[ZAnd p1 p2]}
    where
      ref = (CSPNSParal
                    ns1 (ChanSetUnion (CChanSet cs) (CChanSet [c])) ns2
                    (CSPCommAction (ChanComm c []) a1)
                    (CSPCommAction (ChanComm c []) a2))
      p1 = (ZNot (ZMember (ZVar (c,[])) (ZCall (ZVar ("\\cup",[])) (ZTuple [ZSetDisplay $ zvar_to_z
      p2 = (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs))
crl_prefixParDist ei@(CSPCommAction (ChanComm c [ChanDotExp e])
                    (CSPNSParal ns1 (CChanSet cs) ns2 a1 a2))
 = Done{orig = Just ei, refined = Just ref, proviso=[ZAnd p1 p2]}
    where
      ref = (CSPNSParal ns1 (ChanSetUnion (CChanSet cs) (CChanSet [c])) ns2
                    (CSPCommAction (ChanComm c [ChanDotExp e]) a1)
                    (CSPCommAction (ChanComm c [ChanDotExp e]) a2))
      p1 = (ZNot (ZMember (ZVar (c,[])) (ZCall (ZVar ("\\cup",[])) (ZTuple [ZSetDisplay $ zvar_to_z
      p2 = (ZMember (ZVar (c,[])) (ZSetDisplay $ zname_to_zexpr cs))
crl_prefixParDist _ = None
```

Law 30 (External choice/Sequence—distribution 2*)

$$((g1 \,\&\, A1) \,\Box\, (g2 \,\&\, A2))\,;\; B$$
$$=$$
$$((g1 \,\&\, A1)\,;\; B) \,\Box\, ((g2 \,\&\, A2)\,;\; B)$$

provided $g1 \Rightarrow \neg\, g2$ TODO: implement proviso

```
crl_externalChoiceSequenceDistribution2 :: CAction -> Refinement CAction
crl_externalChoiceSequenceDistribution2
      e@(CSPSeq (CSPExtChoice (CSPGuard g1 a1) (CSPGuard g2 a2)) b)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPExtChoice (CSPSeq (CSPGuard g1 a1) b)
                  (CSPSeq (CSPGuard g2 a2) b))
crl_externalChoiceSequenceDistribution2 _ = None
```

Law 31 (True guard)

$$\mathbf{True} \mathbin{\&} A = A$$

```
crl_trueGuard :: CAction -> Refinement CAction
crl_trueGuard e@(CSPGuard ZTrue{reason=[]} a)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = a
crl_trueGuard _ = None
```

Law 32 (False guard)

$$\mathbf{False} \mathbin{\&} A = \mathbf{Stop}$$

```
crl_falseGuard :: CAction -> Refinement CAction
crl_falseGuard e@(CSPGuard ZFalse{reason=[]} _)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = CSPStop
crl_falseGuard _ = None
```

Law 33 (Hiding/*Chaos*—distribution*)

$$\mathbf{Chaos} \setminus cs = \mathbf{Chaos}$$

```
crl_hidingChaos :: CAction -> Refinement CAction
crl_hidingChaos e@(CSPHide CSPChaos _)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = CSPChaos
crl_hidingChaos _ = None
```

Law 40 (Sequence zero 2*) **Law 34 (Sequence zero 2*)**

$$\mathbf{Chaos} \mathbin{;} A = \mathbf{Chaos}$$

```
crl_seqChaosZero :: CAction -> Refinement CAction
crl_seqChaosZero e@(CSPSeq CSPChaos _)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = CSPChaos
crl_seqChaosZero _ = None
```

Law 41 (Parallelism composition Zero*) **Law 35 (Parallelism composition Zero*)**

$$Chaos \llbracket ns1 \mid cs \mid ns2 \rrbracket A = Chaos$$

```
crl_parallelismZero :: CAction -> Refinement CAction
crl_parallelismZero e@(CSPNSParal _ _ _ CSPChaos _)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = CSPChaos
crl_parallelismZero _ = None
```

Law 36 (Internal choice/Parallelism composition Distribution*)

$$(A1 \sqcap A2) \llbracket ns1 \mid cs \mid ns2 \rrbracket A3$$
$$=$$
$$(A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A3) \sqcap (A2 \llbracket ns1 \mid cs \mid ns2 \rrbracket A3)$$

```
crl_internalChoiceParallelismDistribution :: CAction -> Refinement CAction
crl_internalChoiceParallelismDistribution e@(CSPNSParal ns1 cs ns2 (CSPIntChoice a1 a2) a3)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPIntChoice (CSPNSParal ns1 cs ns2 a1 a3) (CSPNSParal ns1 cs ns2 a2 a3))
crl_internalChoiceParallelismDistribution _ = None
```

Law 37 (Sequence/Internal choice—distribution*)

$$A_1 ; (A_2 \sqcap A_3) = (A_1 ; A_2) \sqcap (A_1 ; A_3)$$

```
--Law 43 (Sequence/Internal choice distribution)
crl_sequenceInternalChoiceDistribution :: CAction -> Refinement CAction
crl_sequenceInternalChoiceDistribution e@(CSPSeq a1 (CSPIntChoice a2 a3))
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPIntChoice (CSPSeq a1 a2) (CSPSeq a1 a3))
crl_sequenceInternalChoiceDistribution _ = None
```

Law 38 (Hiding/Parallelism composition—distribution*)

$$(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2) \setminus cs_2 = (A_1 \setminus cs_2) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (A_2 \setminus cs_2)$$

provided $cs_1 \cap cs_2 = \varnothing$

```
crl_hidingParallelismDistribution :: CAction -> Refinement CAction
crl_hidingParallelismDistribution
    e@(CSPHide (CSPNSParal ns1 (CChanSet cs1) ns2 a1 a2) (CChanSet cs2))
  = Done{orig = Just e, refined = Just ref, proviso=[prov]}
    where
      ref = (CSPNSParal ns1 (CChanSet cs1) ns2 (CSPHide a1 (CChanSet cs1)) (CSPHide a2 (CChanSet cs
      prov = (ZEqual (ZCall (ZVar ("\\cap",[])) (ZTuple [ZSetDisplay $ zname_to_zexpr cs1,ZSetDispl
crl_hidingParallelismDistribution _ = None
```

Law 39 (Hiding combination)

$$(A \setminus cs1) \setminus cs2$$
$$=$$
$$A \setminus (cs1 \cup cs2)$$

```
crl_hidingCombination :: CAction -> Refinement CAction
crl_hidingCombination e@(CSPHide (CSPHide a cs1) cs2)
  = Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPHide a (ChanSetUnion cs1 cs2))
crl_hidingCombination _ = None
```

Law 40 (Assignment Removal*)

$$x := e ; A(x) = A(e)$$

provided

- *x is not free in $A(e)$*

TODO: implement proviso

```
crl_assignmentRemoval :: CAction -> Refinement CAction
crl_assignmentRemoval ei@(CSPSeq
                              (CActionCommand (CAssign [(x,[])] [ZVar (e,[])]))
                              (CActionCommand (CValDecl [Choose (x1,[]) (ZVar (t,[]))] a)))
  = case x == x1 of
      True -> Done{orig = Just ei, refined = Just ref, proviso=[prov]}
      _ -> None
    where
      ref = (CActionCommand (CValDecl [Choose (e,[]) (ZVar (t,[]))] a))
      prov = (ZNot (ZMember (ZVar (x,[])) (ZSetDisplay $ zvar_to_zexpr(free_var_CAction ref))))
crl_assignmentRemoval _ = None
```

Law 41 (Assignment Removal)

$$x := e\ ;\ A(x) = A(e)$$

provided

- x *is not free in* $A(e)$

TODO: implement proviso

```
crl_innocuousAssignment :: CAction -> Refinement CAction
crl_innocuousAssignment e@(CActionCommand (CAssign [(x1,[])] [ZVar (x2,[])]))
  = case (x1 == x2) of
      True -> Done{orig = Just e, refined = Just ref, proviso=[]}
      False -> None
    where
      ref = CSPSkip
crl_innocuousAssignment _ = None
```

Law 42 (Variable Substitution*)

$$\mathbf{var}\ x \bullet A(x) = \mathbf{var}\ y \bullet A(y)$$

provided x *is not free in* A y *is not free in* A

TODO: implement proviso

```
crl_variableSubstitution2 :: CAction -> ZName -> Refinement CAction
crl_variableSubstitution2
    e@(CActionCommand (CVarDecl [Include (ZSRef (ZSPlain x) [] [])]
              (CSPParAction a [ZVar (x1,[])]))) y
  =  Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CActionCommand (CVarDecl [Include (ZSRef (ZSPlain y) [] [])]
        (CSPParAction a [ZVar (y,[])])))
crl_variableSubstitution2 _ _ = None
```

Law 43 (Input Prefix/Sequence Distribution*)

$$(c?x \longrightarrow A1)\ ;\ A2$$
$$=$$
$$c?x \longrightarrow (A1\ ;\ A2)$$

provided

- $x \notin FV(A2)$

TODO: implement proviso

```
crl_inputPrefixSequenceDistribution :: CAction -> Refinement CAction
crl_inputPrefixSequenceDistribution
    e@(CSPSeq (CSPCommAction (ChanComm c [ChanInp x]) a1) a2 )
  =  Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPCommAction (ChanComm c [ChanInp x]) (CSPSeq a1 a2))
crl_inputPrefixSequenceDistribution _ = None
```

Law 44 (Input Prefix/Hiding Identity*)

$$(c?x \longrightarrow A1) \setminus cs$$
$$=$$
$$c?x \longrightarrow (A1 \setminus cs2)$$

provided

- $x \notin FV(A2)$

TODO: implement proviso

```
crl_inputPrefixHidIdentity :: CAction -> Refinement CAction
crl_inputPrefixHidIdentity
    e@(CSPHide (CSPCommAction (ChanComm c [ChanInp x]) a1) (CChanSet cs))
  =  Done{orig = Just e, refined = Just ref, proviso=[prov]}
    where
      ref = (CSPCommAction (ChanComm c [ChanInp x]) (CSPHide a1 (CChanSet cs)))
      prov = (ZNot (ZMember (ZVar (x,[])) (ZSetDisplay (zname_to_zexpr cs))))
crl_inputPrefixHidIdentity _ = None
```

Law 45 (Guard/Parallelism composition—distribution*)

$$(g \mathbin{\&} A1) \llbracket ns1 \mid cs \mid ns2 \rrbracket A2$$
$$=$$
$$g \mathbin{\&} (A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2)$$

provided

- $initials(A2) \subseteq cs$

```
crl_guardParDist :: CAction -> Refinement CAction
crl_guardParDist
    e@(CSPNSParal ns1 (CChanSet cs) ns2 (CSPGuard g a1) a2)
  =  Done{orig = Just e, refined = Just ref, proviso=[prov]}
    where
      ref = (CSPGuard g (CSPNSParal ns1 (CChanSet cs) ns2 a1 a2))
      prov = (ZMember (ZTuple [ZSetDisplay (initials a2),ZSetDisplay (zname_to_zexpr cs)]) (ZVar ("
crl_guardParDist
    e@(CSPNSParal ns1 (CChanSet cs) ns2 a1 (CSPGuard g a2))
  =  Done{orig = Just e, refined = Just ref, proviso=[prov]}
    where
      ref = (CSPGuard g (CSPNSParal ns1 (CChanSet cs) ns2 a1 a2))
      prov = (ZMember (ZTuple [ZSetDisplay (initials a1),ZSetDisplay (zname_to_zexpr cs)]) (ZVar ("
crl_guardParDist _ = None
```

Law 46 (Internal choice/Hiding composition Distribution)

$$(A1 \sqcap A2) \setminus cs$$
$$=$$
$$(A1 \setminus cs) \sqcap (A2 \setminus cs)$$

```
crl_internalChoiceHidingDistribution :: CAction -> Refinement CAction
crl_internalChoiceHidingDistribution
    e@(CSPHide (CSPIntChoice a1 a2) cs)
  =  Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPIntChoice (CSPHide a1 cs) (CSPHide a2 cs))
crl_internalChoiceHidingDistribution _ = None
```

Law 47 (Assignment *Skip*)

$$\mathbf{var}\ x \bullet x := e$$
$$=$$
$$\mathbf{var}\ x \bullet \mathbf{Skip}$$

```
crl_assignmentSkip :: CAction -> Refinement CAction
crl_assignmentSkip
    ei@(CActionCommand (CValDecl
        [Include (ZSRef (ZSPlain x) [] [])]
        (CActionCommand (CAssign [(x1,[])] [ZVar (e,[])]))))
  =  Done{orig = Just ei, refined = Just ref, proviso=[]}
    where
      ref = (CActionCommand (CValDecl
        [Include (ZSRef (ZSPlain x) [] [])] CSPSkip))
crl_assignmentSkip _ = None
```

For testing purposes **Law 48 (Guard combination)**

$$g_1 \& (g_2 \& A) = (g_1 \wedge g_2) \& A$$

```
crl_guardComb :: CAction -> Refinement CAction
crl_guardComb e@(CSPGuard g1 (CSPGuard g2 c))
  =  Done{orig = Just e, refined = Just ref, proviso=[]}
    where
      ref = (CSPGuard (ZAnd g1 g2) c)
crl_guardComb _ = None
```

5.1 Auxiliary functions from Oliveira's PhD thesis

Function for $usedC(A)$

```
usedC :: CAction -> [ZExpr]
usedC (CSPCommAction x c) = [getCommName x] ++ usedC c
usedC (CSPGuard _ c) = usedC c
usedC (CSPSeq ca cb) = (usedC ca) ++ (usedC cb)
usedC (CSPExtChoice ca cb) = (usedC ca) ++ (usedC cb)
usedC (CSPIntChoice ca cb) = (usedC ca) ++ (usedC cb)
usedC (CSPNSParal _ _ _ ca cb) = (usedC ca) ++ (usedC cb)
usedC (CSPParal _ ca cb) = (usedC ca) ++ (usedC cb)
usedC (CSPNSInter _ _ ca cb) = (usedC ca) ++ (usedC cb)
usedC (CSPInterleave ca cb) = (usedC ca) ++ (usedC cb)
usedC (CSPHide c _) = usedC c
usedC (CSPRecursion _ c) = usedC c
usedC (CSPRepSeq _ c) = usedC c
usedC (CSPRepExtChoice _ c) = usedC c
usedC (CSPRepIntChoice _ c) = usedC c
usedC (CSPRepParalNS _ _ _ c) = usedC c
usedC (CSPRepParal _ _ c) = usedC c
usedC (CSPRepInterlNS _ _ c) = usedC c
usedC (CSPRepInterl _ c) = usedC c
usedC _ = []
```

Function for $usedV(A)$

```
usedV :: CAction -> [a]
usedV (CSPCommAction x c) = [] ++ usedV c
usedV (CSPGuard _ c) = usedV c
usedV (CSPSeq ca cb) = (usedV ca) ++ (usedV cb)
usedV (CSPExtChoice ca cb) = (usedV ca) ++ (usedV cb)
usedV (CSPIntChoice ca cb) = (usedV ca) ++ (usedV cb)
usedV (CSPNSParal _ _ _ ca cb) = (usedV ca) ++ (usedV cb)
usedV (CSPParal _ ca cb) = (usedV ca) ++ (usedV cb)
usedV (CSPNSInter _ _ ca cb) = (usedV ca) ++ (usedV cb)
usedV (CSPInterleave ca cb) = (usedV ca) ++ (usedV cb)
usedV (CSPHide c _) = usedV c
usedV (CSPRecursion _ c) = usedV c
usedV (CSPRepSeq _ c) = usedV c
usedV (CSPRepExtChoice _ c) = usedV c
usedV (CSPRepIntChoice _ c) = usedV c
usedV (CSPRepParalNS _ _ _ c) = usedV c
usedV (CSPRepParal _ _ c) = usedV c
usedV (CSPRepInterlNS _ _ c) = usedV c
```

```
usedV (CSPRepInterl _ c) = usedV c
usedV _ = []
```

```
getCommName :: Comm -> ZExpr
getCommName (ChanComm n _) = ZVar (n,[])
getCommName (ChanGenComm n _ _) = ZVar (n,[])
```

Function used for *initials*

```
initials :: CAction -> [ZExpr]
initials (CSPCommAction x c) = [getCommName x]
initials (CSPGuard _ c) = initials c
initials (CSPSeq ca cb) = (initials ca)
initials (CSPExtChoice ca cb) = (initials ca) ++ (initials cb)
initials (CSPIntChoice ca cb) = (initials ca) ++ (initials cb)
initials (CSPNSParal _ _ _ ca cb) = (initials ca) ++ (initials cb)
initials (CSPParal _ ca cb) = (initials ca) ++ (initials cb)
initials (CSPNSInter _ _ ca cb) = (initials ca) ++ (initials cb)
initials (CSPInterleave ca cb) = (initials ca) ++ (initials cb)
initials (CSPHide c _) = initials c
initials (CSPRecursion _ c) = initials c
initials (CSPRepSeq _ c) = initials c
initials (CSPRepExtChoice _ c) = initials c
initials (CSPRepIntChoice _ c) = initials c
initials (CSPRepParalNS _ _ _ c) = initials c
initials (CSPRepParal _ _ c) = initials c
initials (CSPRepInterlNS _ _ c) = initials c
initials (CSPRepInterl _ c) = initials c
initials CSPSkip = [ZVar ("tick",[])]
initials _ = []
```

```
--trace (CSPCommAction (ChanComm x _) c) =
--  [[],[x],map (x:) (trace c)]
--trace (CSPCommAction (ChanGenComm x _ _) c) =
--  [[],[x],map (x:) (trace c)]
--trace (CSPSeq ca cb) = (trace ca)
--trace (CSPExtChoice ca cb) = (trace ca) ++ (trace cb)
data CSPOp = Com Char CSPOp | Seq CSPOp CSPOp | ExtCh CSPOp CSPOp | CSPSkp
trace :: CSPOp -> [[Char]]
trace (Com a p) = map (a:) (trace p)
trace (Seq a b) = (trace a) ++ (trace b)
trace (ExtCh a b) = (trace a)++ (trace b)
trace (CSPSkp)
 = [[]]
```

Function used for *deterministic*

```
deterministic :: CAction -> Maybe [Char]
deterministic (CSPCommAction x c) = deterministic c
deterministic (CSPGuard _ c) = deterministic c
deterministic (CSPSeq ca cb) =
  case (da == Nothing)
   && (da == db)
    of
        false -> Just "Deterministic"
  where
    da = (deterministic ca)
    db = (deterministic cb)

deterministic (CSPExtChoice ca cb) =
  case (da == Nothing)
   && (da == db)
    of
        false -> Just "Deterministic"
  where
    da = (deterministic ca)
    db = (deterministic cb)

deterministic (CSPIntChoice ca cb) = Nothing
deterministic (CSPNSParal _ _ _ ca cb) = Nothing
```

```
deterministic (CSPParal _ ca cb) =  Nothing
deterministic (CSPNSInter _ _ ca cb) =
  case (da == Nothing)
   && (da == db)
    of
        false -> Just "Deterministic"
  where
    da = (deterministic ca)
    db = (deterministic cb)

deterministic (CSPInterleave ca cb) =
  case (da == Nothing)
   && (da == db)
    of
        false -> Just "Deterministic"
  where
    da = (deterministic ca)
    db = (deterministic cb)
deterministic (CSPHide c _) = Nothing
deterministic (CSPRecursion _ c) = deterministic c
deterministic (CSPRepSeq _ c) = deterministic c
deterministic (CSPRepExtChoice _ c) = deterministic c
deterministic (CSPRepIntChoice _ c) = Nothing
deterministic (CSPRepParalNS _ _ _ c) = Nothing
deterministic (CSPRepParal _ _ c) = Nothing
deterministic (CSPRepInterlNS _ _ c) = deterministic c
deterministic (CSPRepInterl _ c) = deterministic c
deterministic CSPSkip = Just "Deterministic"
deterministic _ = Just "Undefined"
```

```
isDeterministic :: CAction -> [Char]
isDeterministic a
  = case x of
      Just "Deterministic" -> "Deterministic"
      Just x               -> "undefined"
      Nothing              -> "Non-deterministic"
    where x = (deterministic a)
```

5.2 Mechanism for applying the refinement laws

First I'm listing all the refinement laws currently available. Then I'm putting it as the variable "reflaws".

```
-- Description of each function:

-- For Circus Actions:
-- crl_assignmentRemoval :: CAction -> Refinement CAction
-- crl_assignmentSkip :: CAction -> Refinement CAction
-- crl_chanExt1 :: CAction -> ZName -> Refinement CAction
-- crl_communicationParallelismDistribution :: CAction -> CAction
-- crl_extChoiceSeqDist :: CAction -> CAction
-- crl_extChoiceStopUnit :: CAction -> CAction
-- crl_externalChoiceSequenceDistribution2 :: CAction -> Refinement CAction
-- crl_falseGuard :: CAction -> Refinement CAction
-- crl_guardParDist :: CAction -> Refinement CAction
-- crl_guardSeqAssoc :: CAction -> Refinement CAction
-- crl_hidingChaos :: CAction -> Refinement CAction
-- crl_hidingCombination :: CAction -> Refinement CAction
-- crl_hidingExpansion2 :: CAction -> ZName -> Refinement CAction
-- crl_hidingExternalChoiceDistribution :: CAction -> CAction
-- crl_hidingIdentity :: CAction -> Refinement CAction
-- crl_hidingParallelismDistribution :: CAction -> Refinement CAction
-- crl_hidingSequenceDistribution :: CAction -> Refinement CAction
-- crl_innocuousAssignment :: CAction -> Refinement CAction
-- crl_inputPrefixHidIdentity :: CAction -> Refinement CAction
-- crl_inputPrefixParallelismDistribution2 :: CAction -> Refinement CAction
-- crl_inputPrefixSequenceDistribution :: CAction -> Refinement CAction
-- crl_internalChoiceHidingDistribution :: CAction -> Refinement CAction
-- crl_internalChoiceParallelismDistribution :: CAction -> Refinement CAction
```

```
-- crl_joinBlocks :: CAction -> Refinement CAction
-- crl_parallelismDeadlocked1 :: CAction -> CAction
-- crl_parallelismDeadlocked3 :: CAction -> Refinement CAction
-- crl_parallelismExternalChoiceDistribution :: CAction -> CAction
-- crl_parallelismExternalChoiceExpansion :: CAction -> Comm -> CAction -> Refinement CAction
-- crl_parallelismIntroduction1a :: CAction -> NSExp -> [ZName] -> NSExp -> Refinement CAction
-- crl_parallelismIntroduction1b :: CAction -> NSExp -> [ZName] -> NSExp -> Refinement CAction
-- crl_parallelismUnit1 :: CAction -> Refinement CAction
-- crl_parallelismZero :: CAction -> Refinement CAction
-- crl_parallInterlEquiv :: CAction -> Refinement CAction
-- crl_parallInterlEquiv_backwards :: CAction -> Refinement CAction
-- crl_parExtChoiceExchange :: CAction -> Refinement CAction
-- crl_parSeqStep :: CAction -> Refinement CAction
-- crl_prefixHiding :: CAction -> Refinement CAction
-- crl_prefixParDist :: CAction -> Refinement CAction
-- crl_prefixSkip :: CAction -> Refinement CAction
-- crl_recUnfold :: CAction -> Refinement CAction
-- crl_seqChaosZero :: CAction -> Refinement CAction
-- crl_seqSkipUnit_a :: CAction -> Refinement CAction
-- crl_seqSkipUnit_b :: CAction -> Refinement CAction
-- crl_seqStopZero :: CAction -> CAction
-- crl_sequenceInternalChoiceDistribution :: CAction -> Refinement CAction
-- crl_trueGuard :: CAction -> Refinement CAction
-- crl_var_exp_par :: CAction -> Refinement CAction
-- crl_var_exp_par2 :: CAction -> Refinement CAction
-- crl_var_exp_rec :: CAction -> Refinement CAction
-- crl_variableBlockIntroduction :: CAction -> ZVar -> ZExpr -> Refinement CAction
-- crl_variableBlockSequenceExtension :: CAction -> Refinement CAction
-- crl_variableSubstitution2 :: CAction -> ZName -> Refinement CAction

-- For Circus Processes:
-- crl_promVarState :: CProc -> Refinement CProc
-- crl_promVarState2 :: CProc -> ZSName -> Refinement CProc
reflawsCAction :: [CAction -> Refinement CAction]
reflawsCAction
        = [crl_assignmentRemoval,
            -- ,
            -- crl_chanExt1,
            -- crl_hidingExpansion2,
            -- crl_parallelismIntroduction1a,
            -- crl_parallelismIntroduction1a_backwards,
            -- crl_parallelismIntroduction1b,
            -- crl_parallelismIntroduction1b_backwards,
            -- crl_parallInterlEquiv_backwards,
            -- crl_promVarState,
            -- crl_promVarState2,
            -- crl_var_exp_par,
            -- crl_variableBlockIntroduction,
            -- crl_variableSubstitution2
            crl_assignmentSkip,
            crl_communicationParallelismDistribution,
            crl_extChoiceStopUnit,
            crl_externalChoiceSequenceDistribution2,
            crl_falseGuard,
           crl_guardParDist,
            crl_guardComb,
             crl_guardSeqAssoc,
            crl_hidingChaos,
            crl_hidingCombination,
            crl_hidingExternalChoiceDistribution,
            crl_hidingIdentity,
            crl_hidingParallelismDistribution,
            crl_hidingSequenceDistribution,
            crl_innocuousAssignment,
            crl_inputPrefixHidIdentity,
            crl_inputPrefixParallelismDistribution2,
            crl_inputPrefixSequenceDistribution,
            crl_internalChoiceHidingDistribution,
            crl_internalChoiceParallelismDistribution,
            crl_joinBlocks,
```

```
              crl_parallelismDeadlocked1,
              crl_parallelismUnit1,
              crl_parallelismZero,
              crl_parallInterlEquiv,
              crl_parExtChoiceExchange,
              crl_prefixHiding,
              crl_prefixParDist,
              -- crl_prefixSkip, -- This one is going into an infinite loop with crl_seqSkipUnit_a
              crl_prefixSkip_backwards, -- this one fixes the probl above
              crl_recUnfold,
              crl_seqChaosZero,
              crl_seqSkipUnit_a,
              crl_seqSkipUnit_b,
              crl_seqStopZero,
              crl_sequenceInternalChoiceDistribution,
              crl_trueGuard,
              crl_var_exp_par2,
              crl_var_exp_rec,
              crl_variableBlockIntroduction_backwards,
              crl_variableBlockSequenceExtension
            ]
-- reflawsCProc = [crl_promVarState, crl_promVarState2]
```

I'm defining a type for the result of the refinement. It can either be *None*, when the refinement is not applied to that specification, or, it can be *Done*{*refined* :: *t*, *proviso* :: [*Bool*]} with a list of provisos to be proved true, where *t* can either be used for a *CProc* or a *CAction*. We then will write those provisos in a text file so it can be used in a theorem prover, like Isabelle/HOL.

```
data Refinement t = None
                  | Done{orig :: Maybe t,
                         refined :: Maybe t,
                         proviso :: [ZPred]
                    } deriving (Eq,Show)
```

Then I'm starting to implement the mechanism itself. Basically, it will try to apply the refinement laws one by one until a result *RefinementCAction* is returned.

```
type RFun t = t -> Refinement t

applyCAction :: (RFun CAction) -> (RFun CAction)
applyCAction r e@(CActionCommand (CIf g))
 = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActionsIf r g of
          (a,b) ->
              Done{orig = Just e, refined = Just (CActionCommand (CIf a)), proviso=b}
          _ -> None
{-
applyCAction r e@(CSPSeq a1 a2) = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [a1,a2]  of
            [a1', a2'] ->
                Done{orig = Just e,
                    refined = Just (CSPSeq (isRefined a1 a1') (isRefined a2 a2')),
                    proviso=(get_proviso a1')++(get_proviso a2')}
            _ -> None
-}

applyCAction r e@(CActionCommand (CVarDecl gf c))
 = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
          [c'] ->
              Done{orig = Just e, refined = Just (CActionCommand (CVarDecl gf (isRefined c c'))), p
          _ -> None

applyCAction r e@(CActionCommand (CValDecl gf c))
```

```
 = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
          [c'] ->
              Done{orig = Just e, refined = Just (CActionCommand (CValDecl gf (isRefined c c'))), p
          _ -> None

applyCAction r e@(CActionCommand (CResDecl gf c))
 = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
          [c'] ->
              Done{orig = Just e, refined = Just (CActionCommand (CResDecl gf (isRefined c c'))), p
          _ -> None

applyCAction r e@(CActionCommand (CVResDecl gf c))
 = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
          [c'] ->
              Done{orig = Just e, refined = Just (CActionCommand (CVResDecl gf (isRefined c c'))),
          _ -> None

applyCAction r e@(CSPCommAction (ChanComm com xs) c)
 = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
          [c'] ->
              Done{orig = Just e, refined = Just (CSPCommAction (ChanComm com xs) (isRefined c c'))
          _ -> None

applyCAction r e@(CSPGuard p c) = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
          [c'] ->
              Done{orig = Just e, refined = Just (CSPGuard p (isRefined c c')), proviso=(get_provis
          _ -> None

applyCAction r e@(CSPSeq a1 a2) = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [a1,a2]  of
            [a1', a2'] ->
                Done{orig = Just e,
                    refined = Just (CSPSeq (isRefined a1 a1') (isRefined a2 a2')),
                    proviso=(get_proviso a1')++(get_proviso a2')}
            _ -> None

applyCAction r e@(CSPExtChoice a1 a2) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [a1,a2]  of
            [a1', a2'] ->
                Done{orig = Just e,
                    refined = Just (CSPExtChoice (isRefined a1 a1') (isRefined a2 a2')),
                    proviso=(get_proviso a1')++(get_proviso a2')}
            _ -> None

applyCAction r e@(CSPIntChoice a1 a2) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [a1,a2]  of
            [a1', a2'] ->
                Done{orig = Just e,
                    refined = Just (CSPIntChoice (isRefined a1 a1') (isRefined a2 a2')),
```

```
                    proviso=(get_proviso a1')++(get_proviso a2')}
            _ -> None

applyCAction r e@(CSPParal cs a1 a2) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [a1,a2]  of
            [a1', a2'] ->
                Done{orig = Just e,
                    refined = Just (CSPParal cs (isRefined a1 a1') (isRefined a2 a2')),
                    proviso=(get_proviso a1')++(get_proviso a2')}
            _ -> None

applyCAction r e@(CSPNSParal ns1 cs ns2 a1 a2)
 = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [a1,a2]  of
            [a1', a2'] ->
                Done{orig = Just e,
                    refined = Just (CSPNSParal ns1 cs ns2 (isRefined a1 a1') (isRefined a2 a2')),
                    proviso=(get_proviso a1')++(get_proviso a2')}
            _ -> None

applyCAction r e@(CSPNSInter ns1 ns2 a1 a2) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [a1,a2]  of
            [a1', a2'] ->
                Done{orig = Just e,
                    refined = Just (CSPNSInter ns1 ns2 (isRefined a1 a1') (isRefined a2 a2')),
                    proviso=(get_proviso a1')++(get_proviso a2')}
            _ -> None

applyCAction r e@(CSPInterleave a1 a2) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [a1,a2]  of
            [a1', a2'] ->
                Done{orig = Just e,
                    refined = Just (CSPInterleave (isRefined a1 a1') (isRefined a2 a2')),
                    proviso=(get_proviso a1')++(get_proviso a2')}
            _ -> None
applyCAction r e@(CSPHide c cs) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
          [c'] ->
              Done{orig = Just e, refined = Just (CSPHide (isRefined c c') cs), proviso=(get_provis
          _ -> None

applyCAction r e@(CSPUnfAction nm c) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
          [c'] ->
              Done{orig = Just e, refined = Just (CSPUnfAction nm (isRefined c c')), proviso=(get_p
          _ -> None

applyCAction r e@(CSPRecursion nm c) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
          [c'] ->
              Done{orig = Just e, refined = Just (CSPRecursion nm (isRefined c c')), proviso=(get_p
          _ -> None

applyCAction r e@(CSPUnParAction lst c nm) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
```

```
      -> case applyCActions r [c]  of
           [c'] ->
               Done{orig = Just e, refined = Just (CSPUnParAction lst (isRefined c c') nm), proviso=
           _ -> None

applyCAction r e@(CSPRepSeq lst c) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
           [c'] ->
               Done{orig = Just e, refined = Just (CSPRepSeq lst (isRefined c c')), proviso=(get_pro
           _ -> None

applyCAction r e@(CSPRepExtChoice lst c) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
           [c'] ->
               Done{orig = Just e, refined = Just (CSPRepExtChoice lst (isRefined c c')), proviso=(g
           _ -> None

applyCAction r e@(CSPRepIntChoice lst c) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
           [c'] ->
               Done{orig = Just e, refined = Just (CSPRepIntChoice lst (isRefined c c')), proviso=(g
           _ -> None

applyCAction r e@(CSPRepParalNS cs lst ns c) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
           [c'] ->
               Done{orig = Just e, refined = Just (CSPRepParalNS cs lst ns (isRefined c c')), provis
           _ -> None

applyCAction r e@(CSPRepParal cs lst c) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
           [c'] ->
               Done{orig = Just e, refined = Just (CSPRepParal cs lst (isRefined c c')), proviso=(ge
           _ -> None
applyCAction r e@(CSPRepInterlNS lst ns c) = case r e of
     r'@( Done{orig = _or, refined = _re, proviso=_pr} )-> r'
     None
      -> case applyCActions r [c]  of
           [c'] ->
               Done{orig = Just e, refined = Just (CSPRepInterlNS lst ns (isRefined c c')), proviso=
           _  -> None

applyCAction r e@(CSPRepInterl lst c) = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None
      -> case applyCActions r [c]  of
           [c'] ->
               Done{orig = Just e, refined = Just (CSPRepInterl lst (isRefined c c')), proviso=(get_
           _ -> None
applyCAction r e
 = case r e of
     r'@(Done{orig = _or, refined = _re, proviso=_pr}) -> r'
     None ->  None


-- Applies a refinement law into a list of actions.
applyCActions :: RFun CAction -> [CAction] -> [Refinement CAction]
applyCActions r [] = []
applyCActions r [e]
 = [applyCAction r e]
```

```
applyCActions r (e:es)
 = (applyCAction r e):(applyCActions r es)

applyCActionsIf :: RFun CAction -> CGActions -> (CGActions,[ZPred])
applyCActionsIf r (CircGAction zp ca)
  = ((CircGAction zp (isRefined ca ca')), get_proviso ca')
  where ca' = (applyCAction r ca)
applyCActionsIf r (CircThenElse ga gb)
 = ((CircThenElse ga' gb'),prova++provb)
  where (ga',prova) = (applyCActionsIf r ga)
        (gb',provb) = (applyCActionsIf r gb)


---
-- This will control if something was refined or not
isRefined :: CAction-> Refinement CAction -> CAction
isRefined a b
  = case b of
      (Done{orig=_, refined=Just e', proviso=z}) -> e'
      None -> a
isRefined' :: CAction-> Maybe CAction -> CAction
isRefined' a b
  = case b of
      Just e' -> e'
      Nothing -> a
```

5.3 The automated refinement tool

```
crefine :: [RFun CAction]
                 -> [RFun CAction]
                 -> CAction
                 -> [Refinement CAction]
                 -> [Refinement CAction]
crefine lst [r] e steps =
    reverse (results:steps)
    where
      results = applyCAction r e

crefine lst (r:rs) e steps =
    case rsx of
      ei@(Done{orig=Just a, refined=Just e', proviso=z}) ->
        case a==e' of
          True -> crefine lst rs e steps
          False -> crefine lst lst e' (ei:steps)
      None -> crefine lst rs e steps
    where rsx = applyCAction r e

refine :: [RFun CAction] -> CAction -> [Refinement CAction]
refine f g = crefine f f g []
```

```
getRef ::  [Refinement CAction] -> Maybe CAction
getRef [] = Nothing
getRef [e@(Done{orig=x, refined=y, proviso=z})] = y
getRef [None] = Nothing
getRef xs = Just $ get_refined (last xs)
```

5.4 Testing the tool

```
runStepRefinement :: CAction -> [Refinement CAction]
runStepRefinement x = refine reflawsCAction x

runRefinement :: CAction -> Maybe CAction
runRefinement x = getRef $ refine reflawsCAction x

refineCAction :: CAction -> CAction
refineCAction x = get_refined $ last (refine reflawsCAction x)
```

5.5 Printing the Refinement Steps

First we get the bits from the *Refinement* record

```
get_orig :: Refinement CAction -> CAction
get_orig (Done{orig=Just a,refined=_,proviso=_}) = a
get_refined :: Refinement CAction -> CAction
get_refined (Done{orig=_,refined=Just b,proviso=_}) = b
get_proviso :: Refinement CAction -> [ZPred]
get_proviso None = []
get_proviso (Done{orig=_,refined=_,proviso=c}) = c
```

Then we define some printing functions, so the refinement can look better on screen.

```
print_proviso [] = "none"
print_proviso [c] = show c
print_proviso (c:cs) = (show c) ++ "] and also [" ++ (print_proviso cs)

print_ref_head c = "LHS\n=\n"
                  ++ (show (get_orig c)) ++"\n\n=   provided["
                  ++ (print_proviso (get_proviso c)) ++"]\n\n"
                  ++ (show (get_refined c)) ++"\n\n"
print_ref_steps c = "=   provided["
                  ++ (print_proviso (get_proviso c)) ++"]\n\n"
                  ++ (show (get_refined c)) ++ "\n\n"
```

Finally, we define a *print_ref* function which prints out on screen the refinement. We can take that to a file with *print_file_ref*.

```
print_ref [] = "No refinement was performed\n"
print_ref [None] = "No refinement was performed\n"
print_ref [x] = print_ref_head x
print_ref (x:xs) = print_ref_head x ++ (print_ref' xs)
print_ref' [] = "=\nRHS"
print_ref' [x] = print_ref_steps x
print_ref' (x:xs) = print_ref_steps x ++ (print_ref' xs)
print_file_ref fname example = writeFile fname $ print_ref $runStepRefinement example
```

Testing area

```

-- Usage:
-- you can type
-- $ print_file_ref "ref_steps.txt" cexample2
-- And it will write the refinement of cexample2 into the ref_steps.txt file.

cexample = (CSPNSParal NSExpEmpty (CChanSet ["c1","c2"]) NSExpEmpty (CSPGuard (ZMember (ZTuple [ZVa
(CActionName "a1")) (CActionName "a2"))
cexample2 = (CSPGuard (ZMember (ZTuple [ZVar ("v1",[]),ZInt 0]) (ZVar (">",[])))  (CSPNSParal NSExp
(CActionName "a1")) (CActionName "a2")))
cexample3 = (CActionCommand (CValDecl [Choose ("b",[]) (ZSetComp [Choose ("x",[]) (ZVar ("BINDING",
cexample4 = (CActionCommand (CValDecl [Choose ("b",[]) (ZSetComp [Choose ("x",[]) (ZVar ("BINDING",
cexample5= (CSPInterleave (CSPCommAction (ChanComm "tick" []) (CActionCommand (CAssign [("sv_SysClo
```

6 Mapping Functions - Circus to CSP

Mapping Functions - Circus to CSP

```
module MappingFunCircusCSP
(
  mapping_Circus
)
where
import AST
import Subs
import CRL
import FormatterToCSP
import Data.List
import Data.Text hiding (map,concat)
```

```
import Data.Char hiding (toUpper, toTitle)
import MappingFunStatelessCircus
import OmegaDefs


showexpr = zexpr_string (pinfo_extz 80)
```

6.1 Mapping Circus Actions

NOTE: *CActionSchemaExpr* is not yet implemented.

```
mapping_CAction :: ZName -> [ZPara] -> CAction -> ZName
mapping_CAction procn spec (CActionCommand cc)
  = "("++mapping_CCommand procn spec cc++")"
mapping_CAction procn spec (CActionName zn)
  = zn
mapping_CAction procn spec (CSPUnfAction x (CActionName v))
  = x ++"("++v++")"
--mapping_CAction procn spec (CActionSchemaExpr zse)
--  = undefined
```

$$\Upsilon_A(c?x : P \longrightarrow A) \mathrel{\widehat=} \texttt{c?x :\{x | x <- } \delta(c), \Upsilon_{\mathbb{B}}(P(x))\} \texttt{ -> } \Upsilon_A(A)$$

```
mapping_CAction procn spec (CSPCommAction (ChanComm c [ChanInpPred x p]) a)
  = case np of
    "true" -> c ++ "?"++ x ++ " : { x | x <- "++ (get_c_chan_type spec c (get_chan_list spec)) ++ "
    _ -> c ++ "?"++ x ++ " : { x | x <- "++ (get_c_chan_type spec c (get_chan_list spec)) ++ ", "++
    where
      np = (mapping_predicate (get_delta_names1 spec) p)
```

$$\Upsilon_A(c?x \longrightarrow A) \mathrel{\widehat=} \texttt{c?x -> } \Upsilon_A(A)$$

```
mapping_CAction procn spec (CSPCommAction (ChanComm c [ChanInp x]) a)
  = (get_channel_name spec (ChanComm c [ChanInp x]))
    ++ " -> "
    ++ mapping_CAction procn spec (a)
```

$$\Upsilon_A(c!v \longrightarrow A) \mathrel{\widehat=} \texttt{c!v -> } \Upsilon_A(A)$$

```
mapping_CAction procn spec (CSPCommAction (ChanComm c [ChanOutExp (ZVar (x,[]))]) a)
  = (get_channel_name spec (ChanComm c [ChanOutExp (ZVar (x,[]))]))
    ++ " -> "
    ++ mapping_CAction procn spec (a) ++ ""

mapping_CAction procn spec (CSPCommAction (ChanComm c lst) a)
  = (get_channel_name spec (ChanComm c lst))
    ++ " -> "
    ++ mapping_CAction procn spec (a) ++ ""
```

$$\Upsilon_A(c \longrightarrow A) \mathrel{\widehat=} \texttt{c -> } \Upsilon_A(A)$$

```
mapping_CAction procn spec (CSPCommAction c a)
  = (get_channel_name spec c)
    ++ " -> "
    ++ mapping_CAction procn spec (a) ++ ""
```

$$\Upsilon_A(A \mathbin{\Box} B) \mathrel{\widehat=} \Upsilon_A(A) \texttt{ [] } \Upsilon_A(B)$$

```
mapping_CAction procn spec (CSPExtChoice a b)
  = "( " ++ mapping_CAction procn spec (a)
    ++ "\n\t\t [] "
    ++ mapping_CAction procn spec (b) ++ ")"
```

$$\Upsilon_A(g\ \&\ A) \mathrel{\widehat=} \Upsilon_{\mathbb{B}}(g)\ \ \texttt{\&}\ \ \Upsilon_A(A)$$

```
mapping_CAction procn spec (CSPGuard g ca)
 -- I'm using the True Guard
  -- and False Guard laws directly
  -- into the translation.
  = case guard of
    "true" -> (mapping_CAction procn spec ca) -- True Law (true & A = A)
    "false" -> "STOP"                         -- False Law (false & A = Stop)
    _ -> "( " ++ guard ++ " & " ++ (mapping_CAction procn spec ca) ++ " )"
  where guard = (mapping_predicate (get_delta_names1 spec) g)
```

$$\Upsilon_A(A \setminus cs) \mathrel{\widehat=} \Upsilon_A(A) \setminus \Upsilon_{\mathbb{P}cs}(cs)$$

```
mapping_CAction procn spec (CSPHide a cs)
  = "( " ++ mapping_CAction procn spec (a)
    ++  "\\"
    ++ mapping_predicate_cs (cs) ++ " )"
```

$$\Upsilon_A(A \sqcap B) \mathrel{\widehat=} \Upsilon_A(A)\ \texttt{|\~{}|}\ \Upsilon_A(B)$$

```
mapping_CAction procn spec (CSPIntChoice a b)
  = "( " ++ mapping_CAction procn spec (a)
    ++ " |~| "
    ++ mapping_CAction procn spec (b) ++ " )"
```

```
mapping_CAction procn spec (CSPInterleave ca cb)
   = "( " ++ mapping_CAction procn spec (ca)
     ++ "\n\t\t ||| "
     ++ mapping_CAction procn spec (cb) ++ " )"
```

$$\Upsilon_A(A \mid [ns1 \mid ns2] \mid B) \mathrel{\widehat=} \Upsilon_A(A)\ \texttt{|||}\ \Upsilon_A(B)$$

```
mapping_CAction procn spec (CSPNSInter ns1 ns2 a b)
  = "( " ++ mapping_CAction procn spec (a)
    ++ "\n\t\t|||"
    ++ mapping_CAction procn spec (b) ++ " )"
```

$$\Upsilon_A(A \llbracket ns1 \mid cs \mid ns2 \rrbracket B) \mathrel{\widehat=} \Upsilon_A(A)\ \texttt{[|}\ \Upsilon_{\mathbb{P}cs}(cs)\ \texttt{|]}\ \Upsilon_A(B)$$

```
mapping_CAction procn spec (CSPNSParal ns1 cs ns2 a b)
  = "( " ++ mapping_CAction procn spec (a)
    ++ "\n\t\t [| "
    ++ mapping_predicate_cs (cs)
    ++ " |] \n\t\t"
    ++ mapping_CAction procn spec (b) ++ " )"
```

```
mapping_CAction procn spec (CSPParAction zn xl)
  = zn ++ "(" ++ concat (map (mapping_ZExpr (get_delta_names1 spec)) xl) ++ ")"
```

$$\Upsilon(\mu X \bullet A(X)) \mathrel{\widehat=} \texttt{let Arec =} \Upsilon_A(A(A_{rec}))\texttt{within Arec}$$

```
mapping_CAction procn spec (CSPRecursion x a)
  = "( " ++ "let "
    ++ x
    ++ " = "
    ++ mapping_CAction procn spec (a)
    ++ " within "
    ++ x ++ " )"
```

$\Upsilon_A(\Box\, x : S \bullet A) \mathrel{\widehat=} \texttt{[] x :}\Upsilon_{\mathbb{P}}(S)\,\texttt{@}\,\Upsilon_A(A)$

```
mapping_CAction procn spec (CSPRepExtChoice [(Choose (x,[]) s)] a)
  = "( " ++ "[] "
    ++  x
    ++ " : "
    ++ (mapping_ZExpr (get_delta_names1 spec) s)
    ++ " @\n\t\t\t "
    ++ mapping_CAction procn spec (a) ++ " )"
```

$\Upsilon_A(\sqcap\, x : S \bullet A) \mathrel{\widehat=} \texttt{|\textasciitilde| x :}\Upsilon_{\mathbb{P}}(S)\,\texttt{@}\,\Upsilon_A(A)$

```
mapping_CAction procn spec (CSPRepIntChoice [(Choose (x,[]) s)] a)
  = "( " ++ "|~| "
    ++  x
    ++ " : "
    ++ (mapping_ZExpr (get_delta_names1 spec) s)
    ++ " @\n\t\t\t "
    ++ mapping_CAction procn spec (a) ++ " )"
```

$\Upsilon_A(|||\, x : S \bullet [\![\varnothing]\!]\, A) \mathrel{\widehat=} \texttt{||| x:}\Upsilon_{\mathbb{P}}(S)\,\texttt{@}\,\Upsilon_A(A)$

```
mapping_CAction procn spec (CSPRepInterlNS [(Choose (x,[]) s)] NSExpEmpty a)
  = "( " ++ "||| "
    ++  x
    ++ " : "
    ++ (mapping_ZExpr (get_delta_names1 spec) s)
    ++ " @ "
    ++ mapping_CAction procn spec (a) ++ " )"
```

$\Upsilon_A([\![cs]\!]\, x : S \bullet [\![\varnothing]\!]\, A) \mathrel{\widehat=} \texttt{[|}\Upsilon_{\mathbb{P}cs}(cs)\texttt{|] x :}\Upsilon_{\mathbb{P}}(S)\,\texttt{@}\,\Upsilon_A(A)$

```
mapping_CAction procn spec (CSPRepParalNS cs [(Choose (x,[]) s)] NSExpEmpty a)
  = "( " ++ "[| "
    ++ mapping_predicate_cs (cs)
    ++ " |] "
    ++  x
    ++ " : "
    ++ (mapping_ZExpr (get_delta_names1 spec) s)
    ++ " @ "
    ++ mapping_CAction procn spec (a) ++ " )"
```

$\Upsilon_A(;\ x : S \bullet A) \mathrel{\widehat=} \texttt{; x :}\Upsilon_{seq}(S)\,\texttt{@}\,\Upsilon_A(A)$

```
mapping_CAction procn spec (CSPRepSeq [(Choose (x,[]) s)] a)
  = "( " ++ "; "
    ++  show x
    ++ " : "
    ++ (mapping_ZExpr (get_delta_names1 spec) s)
    ++ " @ "
    ++ mapping_CAction procn spec (a) ++ " )"
```

$\Upsilon_A(A\ ;\ B) \mathrel{\widehat{=}} \Upsilon_A(A) ; \Upsilon_A(B)$

```
mapping_CAction procn spec (CSPSeq a b)
  = "( " ++ mapping_CAction procn spec (a)
    ++ " ; "
    ++ mapping_CAction procn spec (b) ++ " )"
```

$\Upsilon_A(\mathbf{Skip}) \mathrel{\widehat{=}} \mathtt{SKIP}$

```
mapping_CAction procn spec (CSPSkip)
  = "SKIP"
```

$\Upsilon_A(\mathbf{Stop}) \mathrel{\widehat{=}} \mathtt{STOP}$

```
mapping_CAction procn spec (CSPStop)
  = "STOP"
```

$\Upsilon_A(\mathbf{Chaos}) \mathrel{\widehat{=}} \mathtt{CHAOS}$

```
mapping_CAction procn spec (CSPChaos)
  = "CHAOS"
```

```
mapping_CAction procn spec x
  = fail ("not implemented by mapping_CAction: " ++ show x)
```

6.2 Mapping Circus Commands

NOTE: *CAssumpt*, *CommandBrace*, *CommandBracket* not implemented yet

```
mapping_CCommand :: ZName -> [ZPara] -> CCommand -> ZName
mapping_CCommand procn spec (CAssign (x:xs) (y:ys))
  = error ("Assignments are not available in CSP")
mapping_CCommand procn spec (CAssumpt (x:xs) zpa zpb)
  = error ("Assumptions are not available in CSP")
mapping_CCommand procn spec (CIf cga)
  = mapping_CGActions procn spec cga
-- mapping_CCommand procn spec (CommandBrace zp)
--    = undefined
-- mapping_CCommand procn spec (CommandBracket zp)
--    = undefined
-- mapping_CCommand procn spec (CResDecl (x:xs) ca)
--    = undefined

mapping_CCommand procn spec (CValDecl [Choose ("b",[]) (ZSetComp [Choose ("x",[]) (ZVar ("BINDING",
 = "let "++ restr
    ++"\n\twithin"
    ++"\n\t\t|~| "++ bnd ++" @ Memorise("++(mapping_CAction procn spec ca)++", "++restn++")\n"
    where
      znames = (get_delta_names1 spec)
      ztypes = remdups $ map select_type_zname znames
      restr = mk_charll_to_charl "\n" $ map (mk_restrict spec znames) ztypes
      bnd = mk_charll_to_charl ", " $ map mk_binding_list ztypes
      restn = mk_charll_to_charl ", " $ map mk_restrict_name ztypes

mapping_CCommand procn spec (CValDecl (x:xs) ca)
   = ""
-- mapping_CCommand procn spec (CVResDecl (x:xs) ca)
--    = undefined
mapping_CCommand procn spec x
  = fail ("not implemented by mapping_CCommand: " ++ show x)
```

6.3 Mapping Circus Guarded Actions

```
mapping_CGActions :: ZName -> [ZPara] -> CGActions -> ZName
mapping_CGActions procn spec (CircThenElse cga1 cga2)
  = (mapping_CGActions procn spec cga1) ++ " [] " ++ (mapping_CGActions procn spec cga2)
mapping_CGActions procn spec (CircGAction zp ca)
  = (mapping_predicate (get_delta_names1 spec) zp) ++ " & " ++ (mapping_CAction procn spec ca)
```

6.4 Mapping Channel Communication

```
mapping_Comm :: ZName -> [ZPara] -> Comm -> String
mapping_Comm procn spec (ChanComm zn xs)
  = zn ++ (mapString (mapping_CParameter procn) spec xs)
mapping_Comm procn spec (ChanGenComm zn xs ys)
  = error ("Assumptions are not yet implemented")
```

```
mapString :: (t1 -> t -> String) -> t1 -> [t] -> String
mapString f s [] = ""
mapString f s [x] = (f s x)
mapString f s (x:xs) = (f s x) ++ (mapString f s xs)
```

```
mapping_CParameter :: ZName -> [ZPara] -> CParameter -> ZName
mapping_CParameter procn spec (ChanInp zn)
  = zn
mapping_CParameter procn spec (ChanInpPred zn zp)
 = zn ++ (mapping_predicate (get_delta_names1 spec) zp)
mapping_CParameter procn spec (ChanOutExp ze)
  = mapping_CParameter procn spec (ChanDotExp ze)
mapping_CParameter procn spec (ChanDotExp ze)
  = "."++(mapping_ZExpr (get_delta_names1 spec) ze)
```

6.5 Mapping Circus Namesets

```

-- mapping_NSExp procn spec (NSExpEmpty)
--    = undefined
-- mapping_NSExp procn spec (NSExprMult (x:xs))
--    = undefined
-- mapping_NSExp procn spec (NSExprSngl zn)
--    = undefined
-- mapping_NSExp procn spec (NSHide nse1 nse2)
--    = undefined
-- mapping_NSExp procn spec (NSIntersect nse1 nse2)
--    = undefined
-- mapping_NSExp procn spec (NSUnion nse1 nse2)
--    = undefined
mapping_NSExp procn spec x
  = fail ("not implemented by mapping_NSExp: " ++ show x)
```

7 Mapping Functions from Circus to CSP - Based on D24.1 - COMPASS

7.1 Mapping Functions for Predicates

```
mapping_predicate :: [ZName] -> ZPred -> String
-- NOt sure what "if then  else" is about
-- mapping_predicate lst (ZIf_Then_Else b x1 x2)
--    = "if " ++ (mapping_predicate lst b) ++
--      " then  " ++ (mapping_predicate lst x1) ++
--      " else " ++ (mapping_predicate lst x2)
mapping_predicate lst ( (ZMember (ZTuple [a,b]) (ZVar ("\\geq",[]))))
  = (mapping_ZExpr lst a) ++ " >= " ++ (mapping_ZExpr lst b)
mapping_predicate lst ( (ZMember (ZTuple [a,b]) (ZVar (">",[]))))
```

```
  = (mapping_ZExpr lst a) ++ " > " ++ (mapping_ZExpr lst b)
mapping_predicate lst ( (ZMember (ZTuple [a,b]) (ZVar ("\\leq",[]))))
  = (mapping_ZExpr lst a) ++ " <= " ++ (mapping_ZExpr lst b)
mapping_predicate lst ( (ZMember (ZTuple [a,b]) (ZVar ("<",[]))))
  = (mapping_ZExpr lst a) ++ " < " ++ (mapping_ZExpr lst b)
mapping_predicate lst ( (ZNot (ZEqual a b)))
  = (mapping_ZExpr lst a) ++ " != " ++ (mapping_ZExpr lst b)
mapping_predicate lst ( (ZEqual a b))
  = (mapping_ZExpr lst a) ++ " == " ++ (mapping_ZExpr lst b)
mapping_predicate lst (ZOr a b)
  = (mapping_predicate lst a) ++ " or " ++ (mapping_predicate lst b)
mapping_predicate lst (ZAnd a b)
  = (mapping_predicate lst a) ++ " and " ++ (mapping_predicate lst b)
mapping_predicate lst ( (ZNot b))
  = "not " ++ (mapping_predicate lst b)
mapping_predicate lst (ZPSchema (ZSRef (ZSPlain "\\true") [] []))
  = "true"
mapping_predicate lst (ZPSchema (ZSRef (ZSPlain "\\false") [] []))
  = "false"
mapping_predicate lst (ZTrue{reason=[]})
  = "true"
mapping_predicate lst (ZFalse{reason=[]})
  = "false"
mapping_predicate lst (ZMember (ZVar (x,[])) (ZCall (ZVar ("\\delta",[])) (ZVar (n,[]))))
  = "type"++(lastN 3 x)++"("++n++")"
mapping_predicate lst (ZMember a b)
  = "member("++(mapping_ZExpr lst a)++","++(mapping_ZExpr lst b)++")"
mapping_predicate lst x
  = fail ("not implemented by mapping_predicate: " ++ show x)
```

7.2 Mapping Function for Channel Set Expressions

```
mapping_predicate_cs :: CSExp -> String
{-
The following one is not very well accepted by FDR as it may introduce differente type channels and
For instance,

Couldn't match expected type Event with actual type Int=>Event
    In the expression: getCurrentTime
    In the expression: {tick, getCurrentTime}
    In the statement of a comprehension: c <- {tick, getCurrentTime}
    Relevant variable types:
        getCurrentTime :: Int=>Event
        tick :: Event
        HDMachine :: Proc
        SysClock :: Proc

I think it would be rather correct if we define it as {| x,y,z|}

-}
mapping_predicate_cs (cs)
  -- = "Union({{| c |} | c <- "++ (mapping_set_cs_exp cs) ++" })"
  = (mapping_set_cs_exp cs)
mapping_set_cs_exp (CChanSet x)
  = "{| "++(mapping_ZExpr_def x)++" |}"
mapping_set_cs_exp (CSExpr x)
  = x
mapping_set_cs_exp (ChanSetUnion a b)
  = "union("++ (mapping_set_cs_exp a)++","++ (mapping_set_cs_exp b) ++")"
mapping_set_cs_exp (ChanSetInter a b)
  = "inter("++ (mapping_set_cs_exp a)++","++ (mapping_set_cs_exp b) ++")"
mapping_set_cs_exp (ChanSetDiff a b)
  = "diff("++ (mapping_set_cs_exp a)++","++ (mapping_set_cs_exp b) ++")"
mapping_set_cs_exp x
  = fail ("not implemented by mapping_set_cs_exp: " ++ show x)
```

7.3 Mapping Function for Sequence Expressions

The mapping function for sequence expressions is defined as follows:

```
get_channel_name :: [ZPara] -> Comm -> ZName

get_channel_name spec (ChanComm "mget" [ChanDotExp (ZVar (x,[])),ChanInp v1])
  = "\n\t\tmget."++x++"?"++v1++":(type"++(lastN 3 x)++"("++x++"))"
get_channel_name spec (ChanComm "mset" ((ChanDotExp (ZVar (x,[]))):xs))
  = "\n\t\tmset."++x++".("++(lastN 3 x)++(get_channel_name_cont spec xs)++")"
get_channel_name spec (ChanComm x y)
  = x++(get_channel_name_cont spec y)
get_channel_name spec (ChanGenComm _ _ _)
  = ""
```

```
get_channel_name_cont spec [] = ""
get_channel_name_cont spec [(ChanOutExp v)]
  = get_channel_name_cont spec [(ChanDotExp v)]
get_channel_name_cont spec [(ChanDotExp v)]
  = "."++(mapping_ZExpr (get_delta_names1 spec) v)
get_channel_name_cont spec [(ChanInp v)]
  = "?"++v
get_channel_name_cont spec [(ChanInpPred v x)]
  = "?"++v++":"++(mapping_predicate (get_delta_names1 spec) x)
get_channel_name_cont spec ((ChanOutExp v) : xs)
  = get_channel_name_cont spec ((ChanDotExp v) : xs)
get_channel_name_cont spec ((ChanDotExp v) : xs)
  = "."++(mapping_ZExpr (get_delta_names1 spec) v)++(get_channel_name_cont spec xs)
get_channel_name_cont spec ((ChanInp v) : xs)
  = "?"++v++(get_channel_name_cont spec xs)
get_channel_name_cont spec ((ChanInpPred v x) : xs)
  = "?"++v++":"++(mapping_predicate (get_delta_names1 spec) x)++(get_channel_name_cont spec xs)
```

```
get_c_chan_type :: [ZPara] -> ZName -> [CDecl] -> String
get_c_chan_type spec c [(CChanDecl a b)]
  = case a == c of
      True -> mapping_ZExpr (get_delta_names1 spec) b
      False -> error "Channel not found"
get_c_chan_type spec c ((CChanDecl a b):xs)
  = case a == c of
      True -> mapping_ZExpr (get_delta_names1 spec) b
      False -> get_c_chan_type spec c xs
get_c_chan_type spec c (_:xs)
  = get_c_chan_type spec c xs
get_c_chan_type spec c []
  = error "No channel was found"
```

```
get_chan_list [CircChannel x] = x
get_chan_list ((CircChannel x):xs) = x ++ (get_chan_list xs)
get_chan_list (_:xs) = (get_chan_list xs)
get_chan_list _ = []
```

```
mapping_ZTuple [ZVar ("\\nat",_)] = "NatValue"
mapping_ZTuple [ZVar ("\\nat_1",_)] = "NatValue"
-- mapping_ZTuple [ZVar (v,_)] = "value("++v++")"
mapping_ZTuple [ZVar (v,_)] = v
mapping_ZTuple [ZInt x] = show (fromIntegral x)
mapping_ZTuple ((ZVar (v,_)):xs) = (v) ++ "," ++ (mapping_ZTuple xs)
mapping_ZTuple ((ZInt x):xs) = (show (fromIntegral x)) ++ "," ++ (mapping_ZTuple xs)
mapping_ZTuple _ = ""
```

```
mapping_ZCross [ZVar ("\\int",_)] = "Int"
mapping_ZCross [ZVar (v,_)] = v
mapping_ZCross ((ZVar (v,_)):xs) = (v) ++ "." ++ (mapping_ZCross xs)
mapping_ZCross _ = ""
```

```
-- aux functions
mapping_ZExpr_def :: [ZName] -> String
```

```
mapping_ZExpr_def [x] = x
mapping_ZExpr_def (x:xs) = x++","++(mapping_ZExpr_def xs)
```

```
mapping_ZExpr_def_f f [x] = (f x)
mapping_ZExpr_def_f f (x:xs) = (f x)++","++(mapping_ZExpr_def_f f xs)
```

7.4 Mapping Function for Expressions

```
mapping_ZExpr :: [ZName] -> ZExpr -> String

mapping_ZExpr lst (ZVar ("\\emptyset",[])) = "{}"
mapping_ZExpr lst (ZVar ("\\int",[])) = "Int"
-- mapping_ZExpr lst (ZVar (a,_)) = a
mapping_ZExpr lst (ZInt m) = show(fromIntegral m)
mapping_ZExpr lst (ZVar (a,_))
  | (inListVar a lst) = "value"++(lastN 3 a)++"(v_"++a++")"
  | (is_ZVar_v_st a) = "value"++(lastN 3 a)++"("++a++")"
  | otherwise = a
mapping_ZExpr lst (ZBinding _) = ""
mapping_ZExpr lst (ZCall (ZSeqDisplay x) _) = "<"++(mapping_ZExpr_def_f showexpr x)++">"
mapping_ZExpr lst (ZCall (ZVar ("*",[])) (ZTuple [n,m])) = "("++mapping_ZExpr lst (n) ++ " * " ++ m
mapping_ZExpr lst (ZCall (ZVar ("+",[])) (ZTuple [n,m])) = "("++mapping_ZExpr lst (n) ++ " + " ++ m
mapping_ZExpr lst (ZCall (ZVar ("-",[])) (ZTuple [n,m])) = "("++mapping_ZExpr lst (n) ++ " - " ++ m
mapping_ZExpr lst (ZCall (ZVar ("\\035",[])) a) = "\035(" ++ mapping_ZExpr lst (a)++")"
mapping_ZExpr lst (ZCall (ZVar ("\\\035",[])) a) = "card("++(mapping_ZExpr lst a)++")"
mapping_ZExpr lst (ZCall (ZVar ("\\bigcap",[])) (ZTuple [a,b])) = "Inter("++(mapping_ZExpr lst a)++
mapping_ZExpr lst (ZCall (ZVar ("\\bigcup",[])) (ZTuple [a,b])) = "Union("++(mapping_ZExpr lst a)++
mapping_ZExpr lst (ZCall (ZVar ("\\cap",[])) (ZTuple [a,b])) = "inter("++(mapping_ZExpr lst a)++","
mapping_ZExpr lst (ZCall (ZVar ("\\cat",[])) (ZTuple [a,b])) = mapping_ZExpr lst (a)++"^"++mapping_
mapping_ZExpr lst (ZCall (ZVar ("\\cup",[])) (ZTuple [a,b])) = "union("++(mapping_ZExpr lst a)++","
mapping_ZExpr lst (ZCall (ZVar ("\\dcat",[])) s) = "concat("++mapping_ZExpr lst (s)++")"
mapping_ZExpr lst (ZCall (ZVar ("\\div",[])) (ZTuple [n,m])) = "("++mapping_ZExpr lst (n) ++ " / "
mapping_ZExpr lst (ZCall (ZVar ("\\dom",[])) a) = "dom("++(mapping_ZExpr lst a)++")"
mapping_ZExpr lst (ZCall (ZVar ("\\mod",[])) (ZTuple [n,m])) = mapping_ZExpr lst (n) ++ " % " ++ ma
mapping_ZExpr lst (ZCall (ZVar ("\\negate",[])) n) = "- " ++ mapping_ZExpr lst (n)
mapping_ZExpr lst (ZCall (ZVar ("\\oplus",[])) (ZTuple [ZVar (b,[]),ZSetDisplay [ZCall (ZVar ("\\ma
mapping_ZExpr lst (ZCall (ZVar ("\\power",[])) a) ="Set("++(mapping_ZExpr lst a)++")"
mapping_ZExpr lst (ZCall (ZVar ("\\ran",[])) a) = "set("++(mapping_ZExpr lst a)++")"
mapping_ZExpr lst (ZCall (ZVar ("\\seq",[])) a) = "Seq("++(mapping_ZExpr lst a)++")"
mapping_ZExpr lst (ZCall (ZVar ("\\setminus",[])) (ZTuple [a,b])) = "diff("++(mapping_ZExpr lst a)+
mapping_ZExpr lst (ZCall (ZVar ("head",[])) s) = "head("++mapping_ZExpr lst (s)++")"
mapping_ZExpr lst (ZCall (ZVar ("tail",[])) s) = "tail("++mapping_ZExpr lst (s)++")"
mapping_ZExpr lst (ZCall (ZVar (b,[])) (ZVar (n,[]))) = "apply("++b++","++n++")"
mapping_ZExpr lst (ZCall (ZVar ("\\upto",[])) (ZTuple [a,b]))
 = "{"++(mapping_ZExpr lst a)++".."++(mapping_ZExpr lst b)++"}"
mapping_ZExpr lst (ZCross ls) = mapping_ZCross ls
mapping_ZExpr lst (ZELet _ _) = ""
mapping_ZExpr lst (ZESchema _) = ""
mapping_ZExpr lst (ZFree0 _) = ""
mapping_ZExpr lst (ZFree1 _ _) = ""
mapping_ZExpr lst (ZFreeType _ _) = ""
mapping_ZExpr lst (ZFSet _) = ""
mapping_ZExpr lst (ZFunc1 _) = ""
mapping_ZExpr lst (ZFunc2 _) = ""
mapping_ZExpr lst (ZFuncSet _ _ _ _ _ _ _ _ _) = ""
mapping_ZExpr lst (ZGenerator _ _) = ""
mapping_ZExpr lst (ZGiven _) = ""
mapping_ZExpr lst (ZGivenSet _) = ""
mapping_ZExpr lst (ZIf_Then_Else _ _ _) = ""
mapping_ZExpr lst (ZIntSet _ _) = ""
mapping_ZExpr lst (ZLambda _ _) = ""
mapping_ZExpr lst (ZMu _ _) = ""
mapping_ZExpr lst (ZPowerSet _ _ _) = ""
mapping_ZExpr lst (ZReln _) = ""
mapping_ZExpr lst (ZSelect _ _) = ""
mapping_ZExpr lst (ZSeqDisplay []) = "<>"
mapping_ZExpr lst (ZSeqDisplay _) = ""
mapping_ZExpr lst (ZSetComp _ _ ) = ""
```

```
mapping_ZExpr lst (ZSetDisplay [ZCall (ZVar ("\\upto",[])) (ZTuple [a,b])]) = "{"++(show a)++".."++
mapping_ZExpr lst (ZSetDisplay x) = "{"++(mk_charll_to_charl ", " $ (map (mapping_ZExpr lst) x))++"
mapping_ZExpr lst (ZStrange _) = ""
mapping_ZExpr lst (ZTheta _) = ""
mapping_ZExpr lst (ZTuple ls) = "("++mapping_ZTuple ls ++ ")"
mapping_ZExpr lst (ZUniverse) = ""
mapping_ZExpr lst x = fail ("not implemented by mapping_ZExpr: " ++ show x)
```

8 Misc functions – File: DefSets.lhs

Functions used for manipulating lists (Z Sets and sequences, as well as calculating the provisos from the Circus Refinement laws)

Auxiliary function to propagate *get* communication through the variables and local variables of an action.

$$\begin{array}{l} make\_get\_com\ (v_0, \ldots, v_n, l_0, \ldots, l_m)\ A \mathrel{\widehat=} \\ \qquad get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow \\ \qquad get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow A \end{array}$$

```
make_get_com :: [ZName] -> CAction -> CAction
make_get_com [x] c
  = (CSPCommAction (ChanComm "mget"
    [ChanDotExp (ZVar (x,[])),ChanInp ("v_"++x)]) c)
make_get_com (x:xs) c
  = (CSPCommAction (ChanComm "mget"
    [ChanDotExp (ZVar (x,[])),ChanInp ("v_"++x)]) (make_get_com xs c))
make_get_com x c = c
```

```
make_set_com :: (CAction -> CAction) -> [ZVar] -> [ZExpr] -> CAction -> CAction
make_set_com f [(x,_)] [y] c
  = (CSPCommAction (ChanComm "mset"
    [ChanDotExp (ZVar (x,[])),ChanDotExp y]) (f c))
make_set_com f ((x,_):xs) (y:ys) c
  = (CSPCommAction (ChanComm "mset"
     [ChanDotExp (ZVar (x,[])),ChanDotExp y]) (make_set_com f xs ys c))
```

The function *get_guard_pair* transform *CircGAction* constructs into a list of tuples (*ZPred*, *CAction*)

```
get_guard_pair :: CGActions -> [(ZPred, CAction)]
get_guard_pair (CircGAction g2 a2)
  = [(g2,a2)]
get_guard_pair (CircThenElse (CircGAction g2 a2) glx)
  = ((g2,a2):(get_guard_pair glx))
```

The function *rename_guard_pair* will rename the guards to *v_* prefix of free variables.

```
rename_guard_pair :: [ZName] -> [(ZPred, CAction)] -> [(ZPred, CAction)]
rename_guard_pair sub [(a,b)]
  = [((substitute (mk_sub_list sub) (free_vars a) a),b)]
rename_guard_pair sub ((a,b):xs) = [((substitute (mk_sub_list sub) (free_vars a) a),b)]++(rename_gu
```

The function *mk_guard_pair* transforms a list of tuples (*ZPred*, *CAction*) and produces *CircThenElse* pieces according to the size of the list.

```
mk_guard_pair :: (CAction -> CAction) -> [(ZPred, CAction)] -> CGActions
mk_guard_pair f [(g,a)] = (CircGAction g (f a))
mk_guard_pair f ((g,a):ls) = (CircThenElse (CircGAction g (f a)) (mk_guard_pair f ls))
```

The function *mk_sub_list* will make a list of substitution variables to *v_* prefix.

```
mk_sub_list :: [ZName] -> [((ZName,[t0]),ZExpr)]
mk_sub_list [] = []
mk_sub_list [x] = [((x,[]),(ZVar ("v_"++x,[])))]
mk_sub_list (x:xs) = [((x,[]),(ZVar ("v_"++x,[])))]++(mk_sub_list xs)
```

8.1 Prototype of $wrtV(A)$, from D24.1.

Prototype of $wrtV(A)$, from D24.1.

```
-- TODO: Need to do it
getWrtV xs = []
```

```
rename_ZPred (ZFalse{reason=a})
 = (ZFalse{reason=a})
rename_ZPred (ZTrue{reason=a})
 = (ZTrue{reason=a})
rename_ZPred (ZAnd p1 p2)
 = (ZAnd (rename_ZPred p1) (rename_ZPred p2))
rename_ZPred (ZOr p1 p2)
 = (ZOr (rename_ZPred p1) (rename_ZPred p2))
rename_ZPred (ZImplies p1 p2)
 = (ZImplies (rename_ZPred p1) (rename_ZPred p2))
rename_ZPred (ZIff p1 p2)
 = (ZIff (rename_ZPred p1) (rename_ZPred p2))
rename_ZPred (ZNot p)
 = (ZNot (rename_ZPred p))
rename_ZPred (ZExists lst1 p)
 = (ZExists lst1 (rename_ZPred p))
rename_ZPred (ZExists_1 lst1 p)
 = (ZExists_1 lst1 (rename_ZPred p))
rename_ZPred (ZForall lst1 p)
 = (ZForall lst1 (rename_ZPred p))
rename_ZPred (ZPLet varxp p)
 = (ZPLet varxp (rename_ZPred p))
rename_ZPred (ZEqual xpr1 xpr2)
 = (ZEqual (rename_ZExpr xpr1) (rename_ZExpr xpr2))
rename_ZPred (ZMember xpr1 xpr2)
 = (ZMember (rename_ZExpr xpr1) (rename_ZExpr xpr2))
rename_ZPred (ZPre sp)
 = (ZPre sp)
rename_ZPred (ZPSchema sp)
 = (ZPSchema sp)
```

```
rename_vars_CReplace (CRename zvarls1 zvarls)
 = (CRename zvarls1 zvarls)
rename_vars_CReplace (CRenameAssign zvarls1 zvarls)
 = (CRenameAssign zvarls1 zvarls)
```

```
inListVar x []
 = False
inListVar x [va]
 = case x == va of
  True -> True
  _ -> False
inListVar x (va:vst)
 = case x == va of
  True -> True
  _ -> inListVar x vst
```

8.2 Auxiliary functions for the definition of Ω_A

The use of Isabelle/HOL made me rethink of what was being produced with the functions below. First, a *CSPParAction*, $A(x)$, does not need to call *omega_CAction* again, as it does not change anything, so I removed it when a list of parameters x is a singleton. Then, I realised that I don't need to call *omega_CAction* at all in any of the *rep_* functions as that function is called for the result of any *rep_* function. Finally, I don't need to carry the triple with the state variable names/types.

Function used to propagate *CSPRepSeq* actions

```
rep_CSPRepSeq :: ZName -> [ZExpr] -> CAction
rep_CSPRepSeq a [x]
  = (CSPParAction a [x])
rep_CSPRepSeq a (x:xs)
  = CSPSeq (CSPParAction a [x]) (rep_CSPRepSeq a xs)
```

Function used to propagate *CSPRepIntChoice* actions

```
rep_CSPRepIntChoice :: ZName -> [ZExpr] -> CAction
rep_CSPRepIntChoice a [x]
  = (CSPParAction a [x])
rep_CSPRepIntChoice a (x:xs)
  = CSPIntChoice (CSPParAction a [x]) (rep_CSPRepIntChoice a xs)
```

Function used to propagate *CSPRepExtChoice* actions

```
rep_CSPRepExtChoice :: ZName -> [ZExpr] -> CAction
rep_CSPRepExtChoice a [x]
  = (CSPParAction a [x])
rep_CSPRepExtChoice a (x:xs)
  = CSPExtChoice (CSPParAction a [x]) (rep_CSPRepExtChoice a xs)
```

Function used to propagate *CSPRepInterNS* actions

```
rep_CSPRepParalNS :: ZName -> ZName -> ZName -> String -> [ZExpr] -> CAction
rep_CSPRepParalNS a _ _ _ [x]
  = (CSPParAction a [x])
rep_CSPRepParalNS a cs ns y (x:xs)
  = (CSPNSParal (NSExprParam ns [x]) (CSExpr cs)
    (NSBigUnion (ZSetComp
           [Choose (y,[]) (ZSetDisplay xs)]
           (Just (ZCall (ZVar (ns,[])) (ZVar (y,[])))) ) )
     (CSPParAction a [x]) (rep_CSPRepParalNS a cs ns y xs) )
```

Function used to propagate *CSPRepInterNS* actions

```
rep_CSPRepInterlNS :: ZName -> ZName -> String -> [ZExpr] -> CAction
rep_CSPRepInterlNS a _ _ [x]
  = (CSPParAction a [x])
rep_CSPRepInterlNS a ns y (x:xs)
  = (CSPNSInter (NSExprParam ns [x])
    (NSBigUnion (ZSetComp
           [Choose (y,[]) (ZSetDisplay xs)]
           (Just (ZCall (ZVar (ns,[])) (ZVar (y,[])))) ) )
     (CSPParAction a [x]) (rep_CSPRepInterlNS a ns y xs) )
```

```
 -- Artur - 15/12/2016
 -- What we find below this line was taken from the Data.List module
 -- It is hard to import such package with Haskabelle, so I had
 -- to put it directly into my code.

delete_from_list x []  = []
delete_from_list x [v]
  = (case x == v of
      True -> []
      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)))

setminus [] _  = []
setminus (v : va) [] = (v : va)
setminus (v : va) (b : vb)
     = (delete_from_list b (v : va)) ++ (setminus (v : va) vb)

-- Function that takes the last n elements of a string
-- used in order to get U_TYP from sv_StateName_VarName_U_TYP
lastN :: Int -> [a] -> [a]
lastN n xs = drop (length xs - n) xs
-- From Data.List

member x [] = False
member x (b:y) = if x==b then True else member x y

intersect [] y = []
intersect (a:x) y = if member a y then a : (intersect x y) else intersect x y
```

```
union [] y = y
union (a:x) y = if (member a y) then (union x y) else a : (union x y);
-- | 'delete' @x@ removes the first occurrence of @x@ from its list argument.
-- For example,
--
-- > delete 'a' "banana" == "bnana"
--
-- It is a special case of 'deleteBy', which allows the programmer to
-- supply their own equality test.

delete                  :: (Eq a) => a -> [a] -> [a]
delete                  =  deleteBy (==)

-- | The 'deleteBy' function behaves like 'delete', but takes a
-- user-supplied equality predicate.
deleteBy                :: (a -> a -> Bool) -> a -> [a] -> [a]
deleteBy _  _ []        = []
deleteBy eq x (y:ys)    = if x `eq` y then ys else y : deleteBy eq x ys


-- Not exported:
-- Note that we keep the call to `eq` with arguments in the
-- same order as in the reference implementation
-- 'xs' is the list of things we've seen so far,
-- 'y' is the potential new element
elem_by :: (a -> a -> Bool) -> a -> [a] -> Bool
elem_by _  _ []         =  False
elem_by eq y (x:xs)     =  y `eq` x || elem_by eq y xs


splitOn :: Eq a => a -> [a] -> [[a]]
splitOn d [] = []
splitOn d s = x : splitOn d (drop 1 y) where (x,y) = span (/= d) s
```

get State variables from names

```
get_ZVar_st ((('s':'v':'_':xs),x))
 = [('s':'v':'_':xs)]
get_ZVar_st x
 = []
```

```
is_ZVar_st a = isPrefixOf "sv" a
is_ZVar_v_st a = isPrefixOf "v_sv" a
```

```
rename_ZVar (va,x)
  = case (is_st_var va) of
     True -> ("v_"++va,x)
     False -> (va,x)
rename_ZExpr (ZVar (va,x))
 = case (is_st_var va) of
   True -> (ZVar ("v_"++va,x))
   False -> (ZVar (va,x))
rename_ZExpr (ZInt zi)
 = (ZInt zi)
rename_ZExpr (ZGiven gv)
 = (ZGiven gv)
rename_ZExpr (ZFree0 va)
 = (ZFree0 va)
rename_ZExpr (ZFree1 va xpr)
 = (ZFree1 va (rename_ZExpr xpr))
rename_ZExpr (ZTuple xprlst)
 = (ZTuple (map rename_ZExpr xprlst))
rename_ZExpr (ZBinding xs)
 = (ZBinding (bindingsVar xs))
rename_ZExpr (ZSetDisplay xprlst)
 = (ZSetDisplay (map rename_ZExpr xprlst))
rename_ZExpr (ZSeqDisplay xprlst)
 = (ZSeqDisplay (map rename_ZExpr xprlst))
rename_ZExpr (ZFSet zf)
```

```
 = (ZFSet zf)
rename_ZExpr (ZIntSet i1 i2)
 = (ZIntSet i1 i2)
rename_ZExpr (ZGenerator zrl xpr)
 = (ZGenerator zrl (rename_ZExpr xpr))
rename_ZExpr (ZCross xprlst)
 = (ZCross (map rename_ZExpr xprlst))
rename_ZExpr (ZFreeType va lst1)
 = (ZFreeType va lst1)
rename_ZExpr (ZPowerSet{baseset=xpr, is_non_empty=b1, is_finite=b2})
 = (ZPowerSet{baseset=(rename_ZExpr xpr), is_non_empty=b1, is_finite=b2})
rename_ZExpr (ZFuncSet{ domset=expr1, ranset=expr2, is_function=b1, is_total=b2, is_onto=b3, is_one
 = (ZFuncSet{ domset=(rename_ZExpr expr1), ranset=(rename_ZExpr expr2), is_function=b1, is_total=b2
rename_ZExpr (ZSetComp lst1 (Just xpr))
 = (ZSetComp lst1 (Just (rename_ZExpr xpr)))
rename_ZExpr (ZSetComp lst1 Nothing)
 = (ZSetComp lst1 Nothing)
rename_ZExpr (ZLambda lst1 xpr)
 = (ZLambda lst1 (rename_ZExpr xpr))
rename_ZExpr (ZESchema zxp)
 = (ZESchema zxp)
rename_ZExpr (ZGivenSet gs)
 = (ZGivenSet gs)
rename_ZExpr (ZUniverse)
 = (ZUniverse)
rename_ZExpr (ZCall xpr1 xpr2)
 = (ZCall (rename_ZExpr xpr1) (rename_ZExpr xpr2))
rename_ZExpr (ZReln rl)
 = (ZReln rl)
rename_ZExpr (ZFunc1 f1)
 = (ZFunc1 f1)
rename_ZExpr (ZFunc2 f2)
 = (ZFunc2 f2)
rename_ZExpr (ZStrange st)
 = (ZStrange st)
rename_ZExpr (ZMu lst1 (Just xpr))
 = (ZMu lst1 (Just (rename_ZExpr xpr)))
rename_ZExpr (ZELet lst1 xpr1)
 = (ZELet (bindingsVar lst1) (rename_ZExpr xpr1))
rename_ZExpr (ZIf_Then_Else zp xpr1 xpr2)
 = (ZIf_Then_Else zp (rename_ZExpr xpr1) (rename_ZExpr xpr2))
rename_ZExpr (ZSelect xpr va)
 = (ZSelect xpr va)
rename_ZExpr (ZTheta zs)
 = (ZTheta zs)
rename_ZExpr x
 = x
```

```
bindingsVar []
 = []
bindingsVar [((va,x),b)]
 = case (is_st_var va) of
   True -> [(("v_"++va,x),(rename_ZExpr b))]
   False -> [((va,x),(rename_ZExpr b))]
bindingsVar (((va,x),b):xs)
 = case (is_st_var va) of
   True -> [(("v_"++va,x),(rename_ZExpr b))]++(bindingsVar xs)
   False -> [((va,x),(rename_ZExpr b))]++(bindingsVar xs)
```

```
rename_vars_CParameter (ChanInp zn)
 = (ChanInp zn)
rename_vars_CParameter (ChanInpPred zn zp)
 = (ChanInpPred zn (rename_ZPred zp))
rename_vars_CParameter (ChanOutExp ze)
 = (ChanOutExp (rename_ZExpr ze))
rename_vars_CParameter (ChanDotExp ze)
 = (ChanDotExp (rename_ZExpr ze))
```

```
rename_vars_Comm (ChanComm zn cpls)
```

```
 = (ChanComm zn (map rename_vars_CParameter  cpls))
rename_vars_Comm (ChanGenComm zn zexprls cpls)
 = (ChanGenComm zn (map rename_ZExpr zexprls) (map rename_vars_CParameter cpls))
```

```
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 (CActionSchemaExpr zsexp)
 = (CActionSchemaExpr zsexp)
rename_vars_CAction (CActionCommand cmd)
 = (CActionCommand (rename_vars_CCommand cmd))
rename_vars_CAction (CActionName zn)
 = (CActionName zn)
rename_vars_CAction (CSPCommAction c a)
 = (CSPCommAction (rename_vars_Comm c) (rename_vars_CAction a))
rename_vars_CAction (CSPGuard zp a)
 = (CSPGuard (rename_ZPred zp) (rename_vars_CAction a))

rename_vars_CAction (CSPIntChoice a1 a2)
 = (CSPIntChoice (rename_vars_CAction a1) (rename_vars_CAction a2))
rename_vars_CAction (CSPNSParal ns1 cs ns2 a1 a2)
 = (CSPNSParal ns1 cs ns2 (rename_vars_CAction a1) (rename_vars_CAction a2))
rename_vars_CAction (CSPParal cs a1 a2)
 = (CSPParal cs (rename_vars_CAction a1) (rename_vars_CAction a2))
rename_vars_CAction (CSPNSInter ns1 ns2 a1 a2)
 = (CSPNSInter ns1 ns2 (rename_vars_CAction a1) (rename_vars_CAction a2))
rename_vars_CAction (CSPInterleave a1 a2)
 = (CSPInterleave (rename_vars_CAction a1) (rename_vars_CAction a2))
rename_vars_CAction (CSPHide a cs)
 = (CSPHide (rename_vars_CAction a) cs)
rename_vars_CAction (CSPParAction zn zexprls)
 = (CSPParAction zn (map rename_ZExpr zexprls))
rename_vars_CAction (CSPRenAction zn crpl)
 = (CSPRenAction zn (rename_vars_CReplace crpl))
rename_vars_CAction (CSPRecursion zn a)
 = (CSPRecursion zn (rename_vars_CAction a))
rename_vars_CAction (CSPUnParAction zgf a zn)
 = (CSPUnParAction zgf (rename_vars_CAction a) zn)
rename_vars_CAction (CSPRepSeq zgf a)
 = (CSPRepSeq zgf (rename_vars_CAction a))
rename_vars_CAction (CSPRepExtChoice zgf a)
 = (CSPRepExtChoice zgf (rename_vars_CAction a))
rename_vars_CAction (CSPRepIntChoice zgf a)
 = (CSPRepIntChoice zgf (rename_vars_CAction a))
rename_vars_CAction (CSPRepParalNS cs zgf ns a)
 = (CSPRepParalNS cs zgf ns (rename_vars_CAction a))
rename_vars_CAction (CSPRepParal cs zgf a)
 = (CSPRepParal cs zgf (rename_vars_CAction a))
rename_vars_CAction (CSPRepInterlNS zgf ns a)
 = (CSPRepInterlNS zgf ns (rename_vars_CAction a))
rename_vars_CAction (CSPRepInterl zgf a)
 = (CSPRepInterl zgf (rename_vars_CAction a))
```

```
rename_vars_CCommand (CAssign zvarls1 zexprls)
 = (CAssign zvarls1 (map rename_ZExpr zexprls))
rename_vars_CCommand (CIf ga)
 = (CIf (rename_vars_CGActions ga))
rename_vars_CCommand (CVarDecl zgf a)
 = (CVarDecl zgf (rename_vars_CAction a))
rename_vars_CCommand (CAssumpt znls zp1 zp2)
 = (CAssumpt znls (rename_ZPred zp1) zp2)
```

```
rename_vars_CCommand (CAssumpt1 znls zp)
 = (CAssumpt1 znls zp)
rename_vars_CCommand (CPrefix zp1 zp2)
 = (CPrefix (rename_ZPred zp1) zp2)
rename_vars_CCommand (CPrefix1 zp)
 = (CPrefix1 zp)
rename_vars_CCommand (CommandBrace zp)
 = (CommandBrace zp)
rename_vars_CCommand (CommandBracket zp)
 = (CommandBracket zp)
rename_vars_CCommand (CValDecl zgf a)
 = (CValDecl zgf (rename_vars_CAction a))
rename_vars_CCommand (CResDecl zgf a)
 = (CResDecl zgf (rename_vars_CAction a))
rename_vars_CCommand (CVResDecl zgf a)
 = (CVResDecl zgf (rename_vars_CAction a))
```

```
rename_vars_CGActions (CircGAction zp a)
 = (CircGAction (rename_ZPred zp) (rename_vars_CAction a))
rename_vars_CGActions (CircThenElse (CircGAction zp a) cga2)
 = (CircThenElse (CircGAction (rename_ZPred zp) (rename_vars_CAction a)) (rename_vars_CGActions cga
-- rename_vars_CGActions (CircElse pa) = (CircElse pa)
```

```
remdups [] = []
remdups (x:xs) = (if (member x xs) then remdups xs else x : remdups xs)
```

8.3 Bits for FreeVariables (FV(X))

8.4 Others – No specific topic

```
subset xs ys = all (`elem` ys) xs
```

8.5 Rewritting recursive *Circus* Actions

We are translating any recursive call into *CSPRecursion* so we can rewrite the main action without an infinite loop of rewritting rules.

Firstly we define a function *isRecursive* which looks for any recursive call of a given *Circus* Action.

```
isRecursive_CAction :: ZName -> CAction -> Bool

isRecursive_CAction name (CActionCommand c)
 = isRecursive_CAction_comnd name c
isRecursive_CAction name (CActionName nm)
  | name == nm = True
  | otherwise = False
isRecursive_CAction name (CSPCommAction com c)
 = isRecursive_CAction name c
isRecursive_CAction name (CSPGuard p c)
 =isRecursive_CAction name c
isRecursive_CAction name (CSPSeq ca cb)
 = (isRecursive_CAction name ca) || (isRecursive_CAction name cb)
isRecursive_CAction name (CSPExtChoice ca cb)
 = (isRecursive_CAction name ca) || (isRecursive_CAction name cb)
isRecursive_CAction name (CSPIntChoice ca cb)
 = (isRecursive_CAction name ca) || (isRecursive_CAction name cb)
isRecursive_CAction name (CSPNSParal ns1 cs ns2 ca cb)
 = (isRecursive_CAction name ca) || (isRecursive_CAction name cb)
isRecursive_CAction name (CSPParal cs ca cb)
 = (isRecursive_CAction name ca) || (isRecursive_CAction name cb)
isRecursive_CAction name (CSPNSInter ns1 ns2 ca cb)
 = (isRecursive_CAction name ca) || (isRecursive_CAction name cb)
isRecursive_CAction name (CSPInterleave ca cb)
 = (isRecursive_CAction name ca) || (isRecursive_CAction name cb)
isRecursive_CAction name (CSPHide c cs)
 = isRecursive_CAction name c
isRecursive_CAction name (CSPRecursion n c)
```

```
  = isRecursive_CAction name c
isRecursive_CAction name (CSPUnfAction n c)
  | name == n = True
  | otherwise = False
isRecursive_CAction name (CSPUnParAction lsta c nm)
  = isRecursive_CAction name c
isRecursive_CAction name (CSPRepSeq lsta c)
  = isRecursive_CAction name c
isRecursive_CAction name (CSPRepExtChoice lsta c)
  = isRecursive_CAction name c
isRecursive_CAction name (CSPRepIntChoice lsta c)
  = isRecursive_CAction name c
isRecursive_CAction name (CSPRepParalNS cs lsta ns c)
  = isRecursive_CAction name c
isRecursive_CAction name (CSPRepParal cs lsta c)
  = isRecursive_CAction name c
isRecursive_CAction name (CSPRepInterlNS lsta ns c)
  = isRecursive_CAction name c
isRecursive_CAction name (CSPRepInterl lsta c)
  = isRecursive_CAction name c
isRecursive_CAction name (CActionSchemaExpr x)
  = False
isRecursive_CAction name (CSPSkip)
  = False
isRecursive_CAction name (CSPStop)
  = False
isRecursive_CAction name (CSPChaos)
  = False
isRecursive_CAction name (CSPParAction nm xp)
  = False
isRecursive_CAction name (CSPRenAction nm cr)
  = False
```

```
isRecursive_CAction_comnd name (CAssign v e)
  = False
isRecursive_CAction_comnd name (CIf ga)
  = (isRecursive_if name ga)
isRecursive_CAction_comnd name (CVarDecl z a)
  = isRecursive_CAction name a
isRecursive_CAction_comnd name (CAssumpt n p1 p2)
  = False
isRecursive_CAction_comnd name (CAssumpt1 n p)
  = False
isRecursive_CAction_comnd name (CPrefix p1 p2)
  = False
isRecursive_CAction_comnd name (CPrefix1 p)
  = False
isRecursive_CAction_comnd name (CommandBrace p)
  = False
isRecursive_CAction_comnd name (CommandBracket p)
  = False
isRecursive_CAction_comnd name (CValDecl z a)
  = isRecursive_CAction name a
isRecursive_CAction_comnd name (CResDecl z a)
  = isRecursive_CAction name a
isRecursive_CAction_comnd name (CVResDecl z a)
  = isRecursive_CAction name a
```

```
isRecursive_if name (CircGAction p a)
  = isRecursive_CAction name a
isRecursive_if name (CircThenElse ga gb)
  = (isRecursive_if name ga) || (isRecursive_if name gb)
```

8.5.1 Renaming the recursive call and translating it into *CSPRecursion*

We then rename the recursive call in order to make $\mu X \bullet Action \, \mathrm{seq} \, X$.

```
recursive_PPar (CParAction zn ca)
```

```
  | isRecursive_CAction zn (get_CircusAction ca)
        = (CParAction zn (makeRecursive_ParAction zn ca))
  | otherwise = (CParAction zn ca)
recursive_PPar (ProcZPara a)
  = (ProcZPara a)
recursive_PPar (CNameSet n ns)
  = (CNameSet n ns)

get_CircusAction (CircusAction ca) = ca
get_CircusAction (ParamActionDecl ls pa) = get_CircusAction pa
```

```
makeRecursive_PPar (CParAction zn pa)
  = (CParAction zn (makeRecursive_ParAction zn pa))
makeRecursive_PPar (ProcZPara a)
  = (ProcZPara a)
makeRecursive_PPar (CNameSet n ns)
  = (CNameSet n ns)
```

```
makeRecursive_ParAction name (CircusAction ca)
  = (CircusAction (makeRecursive_CAction name ca))
makeRecursive_ParAction name (ParamActionDecl ls pa)
  = (ParamActionDecl ls (makeRecursive_ParAction name pa))
```

```
makeRecursive_CAction name c =  CSPRecursion ("mu"++name) (renameRecursive_CAction name c)
```

```
renameRecursive_CAction :: ZName -> CAction -> CAction
renameRecursive_CAction name (CActionCommand c)
 = (CActionCommand (renameRecursive_CAction_comnd name c))
renameRecursive_CAction name (CActionName nm)
  | nm == name = (CActionName ("mu"++name))
  | otherwise = (CActionName nm)
renameRecursive_CAction name (CSPCommAction com c)
 = (CSPCommAction com (renameRecursive_CAction name c))
renameRecursive_CAction name (CSPGuard p c)
 = (CSPGuard p (renameRecursive_CAction name c))
renameRecursive_CAction name (CSPSeq ca cb)
 = (CSPSeq (renameRecursive_CAction name ca) (renameRecursive_CAction name cb))
renameRecursive_CAction name (CSPExtChoice ca cb)
 = (CSPExtChoice (renameRecursive_CAction name ca) (renameRecursive_CAction name cb))
renameRecursive_CAction name (CSPIntChoice ca cb)
 = (CSPIntChoice (renameRecursive_CAction name ca) (renameRecursive_CAction name cb))
renameRecursive_CAction name (CSPNSParal ns1 cs ns2 ca cb)
 = (CSPNSParal ns1 cs ns2 (renameRecursive_CAction name ca) (renameRecursive_CAction name cb))
renameRecursive_CAction name (CSPParal cs ca cb)
 = (CSPParal cs (renameRecursive_CAction name ca) (renameRecursive_CAction name cb))
renameRecursive_CAction name (CSPNSInter ns1 ns2 ca cb)
 = (CSPNSInter ns1 ns2 (renameRecursive_CAction name ca) (renameRecursive_CAction name cb))
renameRecursive_CAction name (CSPInterleave ca cb)
 = (CSPInterleave (renameRecursive_CAction name ca) (renameRecursive_CAction name cb))
renameRecursive_CAction name (CSPHide c cs)
 = (CSPHide (renameRecursive_CAction name c) cs)
renameRecursive_CAction name (CSPParAction nm xp)
  | nm == name = (CSPParAction ("mu"++nm) xp)
  | otherwise = (CSPParAction nm xp)
renameRecursive_CAction name (CSPRenAction nm cr)
 = (CSPRenAction nm cr)
renameRecursive_CAction name (CSPRecursion n c)
 = (CSPRecursion n (renameRecursive_CAction name c))
renameRecursive_CAction name (CSPRecursion n c)
 = (CSPRecursion n (renameRecursive_CAction name c))
renameRecursive_CAction name (CSPUnParAction namea c nm)
 = (CSPUnParAction namea (renameRecursive_CAction name c) nm)
renameRecursive_CAction name (CSPRepSeq namea c)
 = (CSPRepSeq namea (renameRecursive_CAction name c))
renameRecursive_CAction name (CSPRepExtChoice namea c)
 = (CSPRepExtChoice namea (renameRecursive_CAction name c))
renameRecursive_CAction name (CSPRepIntChoice namea c)
 = (CSPRepIntChoice namea (renameRecursive_CAction name c))
```

```
renameRecursive_CAction name (CSPRepParalNS cs namea ns c)
 = (CSPRepParalNS cs namea ns (renameRecursive_CAction name c))
renameRecursive_CAction name (CSPRepParal cs namea c)
 = (CSPRepParal cs namea (renameRecursive_CAction name c))
renameRecursive_CAction name (CSPRepInterlNS namea ns c)
 = (CSPRepInterlNS namea ns (renameRecursive_CAction name c))
renameRecursive_CAction name (CSPRepInterl namea c)
 = (CSPRepInterl namea (renameRecursive_CAction name c))
renameRecursive_CAction _ x = x
```

```
renameRecursive_CAction_comnd name (CAssign v e)
 = (CAssign v e)
renameRecursive_CAction_comnd name (CIf ga)
 = (CIf (renameRecursive_if name ga))
renameRecursive_CAction_comnd name (CVarDecl z a)
 = (CVarDecl z (renameRecursive_CAction name a))
renameRecursive_CAction_comnd name (CAssumpt n p1 p2)
 = (CAssumpt n p1 p2)
renameRecursive_CAction_comnd name (CAssumpt1 n p)
 = (CAssumpt1 n p)
renameRecursive_CAction_comnd name (CPrefix p1 p2)
 = (CPrefix p1 p2)
renameRecursive_CAction_comnd name (CPrefix1 p)
 = (CPrefix1 p)
renameRecursive_CAction_comnd name (CommandBrace p)
 = (CommandBrace p)
renameRecursive_CAction_comnd name (CommandBracket p)
 = (CommandBracket p)
renameRecursive_CAction_comnd name (CValDecl z a)
 = (CValDecl z (renameRecursive_CAction name a))
renameRecursive_CAction_comnd name (CResDecl z a)
 = (CResDecl z (renameRecursive_CAction name a))
renameRecursive_CAction_comnd name (CVResDecl z a)
 = (CVResDecl z (renameRecursive_CAction name a))
```

```
renameRecursive_if name (CircGAction p a)
 = (CircGAction p (renameRecursive_CAction name a))
renameRecursive_if name (CircThenElse ga gb)
 = (CircThenElse (renameRecursive_if name ga) (renameRecursive_if name gb))
-- get_if name (CircElse (CircusAction a))
--  = (CircElse (CircusAction (renameRecursive_CAction name a)))
-- get_if name (CircElse (ParamActionDecl x (CircusAction a)))
--  = (CircElse (ParamActionDecl x (CircusAction (renameRecursive_CAction name a))))
```

8.6 Expanding the main action

```
expand_action_names_PPar :: [PPar] -> PPar -> PPar
expand_action_names_PPar lst (ProcZPara zp)
  = (ProcZPara zp)
expand_action_names_PPar lst (CParAction zn pa)
  = (CParAction zn (expand_action_names_ParAction lst pa))
expand_action_names_PPar lst (CNameSet zn ns)
  = (CNameSet zn ns)
```

```
expand_action_names_ParAction :: [PPar] -> ParAction -> ParAction
expand_action_names_ParAction lst (CircusAction ca) = (CircusAction (expand_action_names_CAction ls
expand_action_names_ParAction lst (ParamActionDecl ls pa) = (ParamActionDecl ls (expand_action_name
-- Decl \circspot ParAction
```

```
expand_action_names_CAction :: [PPar] -> CAction -> CAction
expand_action_names_CAction lst (CActionSchemaExpr x)
 = (CActionSchemaExpr x)
expand_action_names_CAction lst (CActionCommand c)
 = (CActionCommand (expand_action_names_CAction_comnd lst c))
expand_action_names_CAction lst (CActionName nm)
  | (take 2 nm) == "mu" = (CActionName nm)
```

```
  | otherwise = get_action nm lst lst
expand_action_names_CAction lst (CSPSkip)
 = (CSPSkip)
expand_action_names_CAction lst (CSPStop)
 = (CSPStop)
expand_action_names_CAction lst (CSPChaos)
 = (CSPChaos)
expand_action_names_CAction lst (CSPCommAction com c)
 = (CSPCommAction com (expand_action_names_CAction lst c))
expand_action_names_CAction lst (CSPGuard p c)
 = (CSPGuard p (expand_action_names_CAction lst c))
expand_action_names_CAction lst (CSPSeq ca cb)
 = (CSPSeq (expand_action_names_CAction lst ca) (expand_action_names_CAction lst cb))
expand_action_names_CAction lst (CSPExtChoice ca cb)
 = (CSPExtChoice (expand_action_names_CAction lst ca) (expand_action_names_CAction lst cb))
expand_action_names_CAction lst (CSPIntChoice ca cb)
 = (CSPIntChoice (expand_action_names_CAction lst ca) (expand_action_names_CAction lst cb))
expand_action_names_CAction lst (CSPNSParal ns1 cs ns2 ca cb)
 = (CSPNSParal ns1 cs ns2 (expand_action_names_CAction lst ca) (expand_action_names_CAction lst cb)
expand_action_names_CAction lst (CSPParal cs ca cb)
 = (CSPParal cs (expand_action_names_CAction lst ca) (expand_action_names_CAction lst cb))
expand_action_names_CAction lst (CSPNSInter ns1 ns2 ca cb)
 = (CSPNSInter ns1 ns2 (expand_action_names_CAction lst ca) (expand_action_names_CAction lst cb))
expand_action_names_CAction lst (CSPInterleave ca cb)
 = (CSPInterleave (expand_action_names_CAction lst ca) (expand_action_names_CAction lst cb))
expand_action_names_CAction lst (CSPHide c cs)
 = (CSPHide (expand_action_names_CAction lst c) cs)
expand_action_names_CAction lst (CSPParAction nm xp)
 = (CSPParAction nm xp)
expand_action_names_CAction lst (CSPRenAction nm cr)
 = (CSPRenAction nm cr)
expand_action_names_CAction lst (CSPRecursion n (CSPSeq c (CActionName n1)))
 = case n == n1 of
   True -> (CSPRecursion n (CSPSeq (expand_action_names_CAction lst c) (CActionName n)))
   False -> (CSPRecursion n (CSPSeq (expand_action_names_CAction lst c) (CActionName n1)))
expand_action_names_CAction lst (CSPRecursion n c)
 = (CSPRecursion n (expand_action_names_CAction lst c))
expand_action_names_CAction lst (CSPUnParAction lsta c nm)
 = (CSPUnParAction lsta (expand_action_names_CAction lst c) nm)
expand_action_names_CAction lst (CSPRepSeq lsta c)
 = (CSPRepSeq lsta (expand_action_names_CAction lst c))
expand_action_names_CAction lst (CSPRepExtChoice lsta c)
 = (CSPRepExtChoice lsta (expand_action_names_CAction lst c))
expand_action_names_CAction lst (CSPRepIntChoice lsta c)
 = (CSPRepIntChoice lsta (expand_action_names_CAction lst c))
expand_action_names_CAction lst (CSPRepParalNS cs lsta ns c)
 = (CSPRepParalNS cs lsta ns (expand_action_names_CAction lst c))
expand_action_names_CAction lst (CSPRepParal cs lsta c)
 = (CSPRepParal cs lsta (expand_action_names_CAction lst c))
expand_action_names_CAction lst (CSPRepInterlNS lsta ns c)
 = (CSPRepInterlNS lsta ns (expand_action_names_CAction lst c))
expand_action_names_CAction lst (CSPRepInterl lsta c)
 = (CSPRepInterl lsta (expand_action_names_CAction lst c))
expand_action_names_CAction lst x = x
```

```
expand_action_names_CAction_comnd lst (CAssign v e)
 = (CAssign v e)
expand_action_names_CAction_comnd lst (CIf ga)
 = (CIf (get_if lst ga))
expand_action_names_CAction_comnd lst (CVarDecl z a)
 = (CVarDecl z (expand_action_names_CAction lst a))
expand_action_names_CAction_comnd lst (CAssumpt n p1 p2)
 = (CAssumpt n p1 p2)
expand_action_names_CAction_comnd lst (CAssumpt1 n p)
 = (CAssumpt1 n p)
expand_action_names_CAction_comnd lst (CPrefix p1 p2)
 = (CPrefix p1 p2)
expand_action_names_CAction_comnd lst (CPrefix1 p)
 = (CPrefix1 p)
expand_action_names_CAction_comnd lst (CommandBrace p)
```

```
 = (CommandBrace p)
expand_action_names_CAction_comnd lst (CommandBracket p)
 = (CommandBracket p)
expand_action_names_CAction_comnd lst (CValDecl z a)
 = (CValDecl z (expand_action_names_CAction lst a))
expand_action_names_CAction_comnd lst (CResDecl z a)
 = (CResDecl z (expand_action_names_CAction lst a))
expand_action_names_CAction_comnd lst (CVResDecl z a)
 = (CVResDecl z (expand_action_names_CAction lst a))
```

```
get_if lst (CircGAction p a)
 = (CircGAction p (expand_action_names_CAction lst a))
get_if lst (CircThenElse (CircGAction p a) gb)
 = (CircThenElse (CircGAction p (expand_action_names_CAction lst a)) (get_if lst gb))
-- get_if lst (CircElse (CircusAction a))
--  = (CircElse (CircusAction (expand_action_names_CAction lst a)))
-- get_if lst (CircElse (ParamActionDecl x (CircusAction a)))
--  = (CircElse (ParamActionDecl x (CircusAction (expand_action_names_CAction lst a))))
```

```
get_action _ lst [] = error "Action list is empty"
get_action name lst [(CParAction n (CircusAction a))]
  | name == n = expand_action_names_CAction lst a
  | otherwise = error ("Action "++(name)++" not found")
get_action name lst ((CParAction n (CircusAction a)):xs)
  | (name == n) = expand_action_names_CAction lst a
  | otherwise = get_action name lst xs
get_action name lst (_:xs)
  = get_action name lst xs
```

```
get_chan_param :: [CParameter] -> [ZExpr]
get_chan_param [] = []
get_chan_param [ChanDotExp (ZVar (x,_))]
 = [ZVar (x,[])]
get_chan_param [ChanOutExp (ZVar (x,_))]
 = [ZVar (x,[])]
get_chan_param [_]
 = []
get_chan_param ((ChanDotExp (ZVar (x,_))):xs)
 = [ZVar (x,[])]++(get_chan_param xs)
get_chan_param ((ChanOutExp (ZVar (x,_))):xs)
 = [ZVar (x,[])]++(get_chan_param xs)
get_chan_param (_:xs) = (get_chan_param xs)
```

```
filter_state_comp :: [(ZName, ZVar, ZExpr)] -> [ZVar]
filter_state_comp [] = []
filter_state_comp [(_, v, _)] = [v]
filter_state_comp ((_, v, _):xs) = [v]++(filter_state_comp xs)
```

```
is_st_var ('s':'v':'_':xs) = True
is_st_var _ = False
```

```
middle (a,b,c) = b
```

8.6.1 rename vars

```
rename_vars_ParAction (CircusAction ca)
  = (CircusAction (rename_vars_CAction ca))
rename_vars_ParAction (ParamActionDecl zglst pa)
  = (ParamActionDecl zglst (rename_vars_ParAction pa))
```

8.7 $[ZName]$ to $[ZExpr]$ - mainly converting to $ZVar(x, [])$

```
zname_to_zexpr [] = []
zname_to_zexpr [a] = [ZVar (a,[])]
zname_to_zexpr (a:as) = [ZVar (a,[])]++(zname_to_zexpr as)
```

8.8 [*ZVar*] to [*ZExpr*]

```
zvar_to_zexpr [] = []
zvar_to_zexpr [(a,[])] = [ZVar (a,[])]
zvar_to_zexpr ((a,[]):as) = [ZVar (a,[])]++(zvar_to_zexpr as)
```

8.9 [*ZGenFilt*] to [*ZExpr*]

```

zgenfilt_to_zexpr [] = []
zgenfilt_to_zexpr [(Choose (a,[]) t)] = [ZVar (a,[])]
zgenfilt_to_zexpr ((Choose (a,[]) t):as) = [ZVar (a,[])]++(zgenfilt_to_zexpr as)
zgenfilt_to_zexpr (_:as) = []++(zgenfilt_to_zexpr as)
```

8.9.1 rename vars

```
rename_vars_ZPara1 :: [(ZName, ZVar, ZExpr)] -> ZPara -> ZPara
rename_vars_ZPara1 lst (Process zp)
  = (Process (rename_vars_ProcDecl1 lst zp))
-- rename_vars_ZPara1 lst (ZSchemaDef n zs)
--   = (ZSchemaDef n (rename_vars_ZSExpr1 lst zs))
rename_vars_ZPara1 lst x = x
```

```
rename_vars_ZSExpr1 :: [(ZName, ZVar, ZExpr)] -> ZSExpr -> ZSExpr
rename_vars_ZSExpr1 lst (ZSchema s)
  = ZSchema (map (rename_ZGenFilt1 lst) s)
```

```
rename_vars_ProcDecl1 :: [(ZName, ZVar, ZExpr)] -> ProcDecl -> ProcDecl
rename_vars_ProcDecl1 lst (CProcess zn pd)
  = (CProcess zn (rename_vars_ProcessDef1 lst pd))
rename_vars_ProcDecl1 lst (CParamProcess zn znls pd)
  = (CParamProcess zn znls (rename_vars_ProcessDef1 lst pd))
rename_vars_ProcDecl1 lst (CGenProcess zn znls pd)
  = (CParamProcess zn znls (rename_vars_ProcessDef1 lst pd))
```

```
rename_vars_ProcessDef1 :: [(ZName, ZVar, ZExpr)] -> ProcessDef -> ProcessDef
rename_vars_ProcessDef1 lst (ProcDefSpot zgf pd)
  = (ProcDefSpot zgf (rename_vars_ProcessDef1 lst pd))
rename_vars_ProcessDef1 lst (ProcDefIndex zgf pd)
  = (ProcDefIndex zgf (rename_vars_ProcessDef1 lst pd))
rename_vars_ProcessDef1 lst (ProcDef cp)
  = (ProcDef (rename_vars_CProc1 lst cp))
```

```
rename_vars_CProc1 :: [(ZName, ZVar, ZExpr)] -> CProc -> CProc
rename_vars_CProc1 lst (CRepSeqProc zgf cp)
  = (CRepSeqProc zgf (rename_vars_CProc1 lst cp))
rename_vars_CProc1 lst (CRepExtChProc zgf cp)
  = (CRepExtChProc zgf (rename_vars_CProc1 lst cp))
rename_vars_CProc1 lst (CRepIntChProc zgf cp)
  = (CRepIntChProc zgf (rename_vars_CProc1 lst cp))
rename_vars_CProc1 lst (CRepParalProc cs zgf cp)
  = (CRepParalProc cs zgf (rename_vars_CProc1 lst cp))
rename_vars_CProc1 lst (CRepInterlProc zgf cp)
  = (CRepInterlProc zgf (rename_vars_CProc1 lst cp))
rename_vars_CProc1 lst (CHide cp cxp)
  = (CHide (rename_vars_CProc1 lst cp) cxp)
rename_vars_CProc1 lst (CExtChoice cp1 cp2)
  = (CExtChoice (rename_vars_CProc1 lst cp1) (rename_vars_CProc1 lst cp2))
rename_vars_CProc1 lst (CIntChoice cp1 cp2)
  = (CIntChoice (rename_vars_CProc1 lst cp1) (rename_vars_CProc1 lst cp2))
rename_vars_CProc1 lst (CParParal cs cp1 cp2)
  = (CParParal cs (rename_vars_CProc1 lst cp1) (rename_vars_CProc1 lst cp2))
rename_vars_CProc1 lst (CInterleave cp1 cp2)
  = (CInterleave (rename_vars_CProc1 lst cp1) (rename_vars_CProc1 lst cp2))
```

```
rename_vars_CProc1 lst (CGenProc zn zxp)
  = (CGenProc zn zxp)
rename_vars_CProc1 lst (CParamProc zn zxp)
  = (CParamProc zn zxp)
rename_vars_CProc1 lst (CProcRename zn c1 c2)
  = (CProcRename zn c1 c2)
rename_vars_CProc1 lst (CSeq cp1 cp2)
  = (CSeq (rename_vars_CProc1 lst cp1) (rename_vars_CProc1 lst cp2))
rename_vars_CProc1 lst (CSimpIndexProc zn zxp)
  = (CSimpIndexProc zn zxp)
rename_vars_CProc1 lst (CircusProc zn)
  = (CircusProc zn)
rename_vars_CProc1 lst (ProcMain zp ppl ca)
  = (ProcMain (rename_vars_ZPara1 lst zp) (map (rename_vars_PPar1 lst) ppl) (rename_vars_CAction1 l
rename_vars_CProc1 lst (ProcStalessMain ppl ca)
  = (ProcStalessMain ppl (rename_vars_CAction1 lst ca))
```

8.9.2 Circus Actions

```
rename_vars_PPar1 :: [(ZName, ZVar, ZExpr)] -> PPar -> PPar
rename_vars_PPar1 lst (ProcZPara zp)
  = (ProcZPara zp)
rename_vars_PPar1 lst (CParAction zn pa)
  = (CParAction zn (rename_vars_ParAction1 lst pa))
rename_vars_PPar1 lst (CNameSet zn ns)
  = (CNameSet zn ns)
```

```
rename_vars_ParAction1 :: [(ZName, ZVar, ZExpr)] -> ParAction -> ParAction
rename_vars_ParAction1 lst (CircusAction ca)
  = (CircusAction (rename_vars_CAction1 lst ca))
rename_vars_ParAction1 lst (ParamActionDecl zgf pa)
  = (ParamActionDecl zgf (rename_vars_ParAction1 lst pa))
```

```
rename_vars_CAction1 :: [(ZName, ZVar, ZExpr)] -> CAction -> CAction
rename_vars_CAction1 lst (CActionSchemaExpr zsexp)
 = (CActionSchemaExpr zsexp)
rename_vars_CAction1 lst (CActionCommand cmd)
 = (CActionCommand (rename_vars_CCommand1 lst cmd))
rename_vars_CAction1 lst (CActionName zn)
 = (CActionName zn)
rename_vars_CAction1 lst (CSPSkip )
 = (CSPSkip )
rename_vars_CAction1 lst (CSPStop )
 = (CSPStop )
rename_vars_CAction1 lst (CSPChaos)
 = (CSPChaos)
rename_vars_CAction1 lst (CSPCommAction c a)
 = (CSPCommAction (rename_vars_Comm1 lst c) (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst (CSPGuard zp a)
 = (CSPGuard (rename_vars_ZPred1 lst zp) (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst (CSPSeq a1 a2)
 = (CSPSeq (rename_vars_CAction1 lst a1) (rename_vars_CAction1 lst a2))
rename_vars_CAction1 lst (CSPExtChoice a1 a2)
 = (CSPExtChoice (rename_vars_CAction1 lst a1) (rename_vars_CAction1 lst a2))
rename_vars_CAction1 lst (CSPIntChoice a1 a2)
 = (CSPIntChoice (rename_vars_CAction1 lst a1) (rename_vars_CAction1 lst a2))
rename_vars_CAction1 lst (CSPNSParal ns1 cs ns2 a1 a2)
 = (CSPNSParal ns1 cs ns2 (rename_vars_CAction1 lst a1) (rename_vars_CAction1 lst a2))
rename_vars_CAction1 lst (CSPParal cs a1 a2)
 = (CSPParal cs (rename_vars_CAction1 lst a1) (rename_vars_CAction1 lst a2))
rename_vars_CAction1 lst (CSPNSInter ns1 ns2 a1 a2)
 = (CSPNSInter ns1 ns2 (rename_vars_CAction1 lst a1) (rename_vars_CAction1 lst a2))
rename_vars_CAction1 lst (CSPInterleave a1 a2)
 = (CSPInterleave (rename_vars_CAction1 lst a1) (rename_vars_CAction1 lst a2))
rename_vars_CAction1 lst (CSPHide a cs)
 = (CSPHide (rename_vars_CAction1 lst a) cs)
rename_vars_CAction1 lst (CSPParAction zn zexprls)
 = (CSPParAction zn (map (rename_vars_ZExpr1 lst) zexprls))
```

```
rename_vars_CAction1 lst (CSPRenAction zn crpl)
 = (CSPRenAction zn (rename_vars_CReplace1 lst crpl))
rename_vars_CAction1 lst (CSPRecursion zn a)
 = (CSPRecursion zn (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst (CSPUnParAction zgf a zn)
 = (CSPUnParAction zgf (rename_vars_CAction1 lst a) zn)
rename_vars_CAction1 lst (CSPRepSeq zgf a)
 = (CSPRepSeq zgf (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst (CSPRepExtChoice zgf a)
 = (CSPRepExtChoice zgf (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst (CSPRepIntChoice zgf a)
 = (CSPRepIntChoice zgf (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst (CSPRepParalNS cs zgf ns a)
 = (CSPRepParalNS cs zgf ns (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst (CSPRepParal cs zgf a)
 = (CSPRepParal cs zgf (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst (CSPRepInterlNS zgf ns a)
 = (CSPRepInterlNS zgf ns (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst (CSPRepInterl zgf a)
 = (CSPRepInterl zgf (rename_vars_CAction1 lst a))
rename_vars_CAction1 lst x = x
```

```
rename_vars_Comm1 :: [(ZName, ZVar, ZExpr)] -> Comm -> Comm
rename_vars_Comm1 lst (ChanComm zn cpls)
 = (ChanComm zn (map (rename_vars_CParameter1 lst) cpls))
rename_vars_Comm1 lst (ChanGenComm zn zexprls cpls)
 = (ChanGenComm zn (map (rename_vars_ZExpr1 lst) zexprls) (map (rename_vars_CParameter1 lst) cpls))
```

```
rename_vars_CParameter1 :: [(ZName, ZVar, ZExpr)] -> CParameter -> CParameter
rename_vars_CParameter1 lst (ChanInp zn)
 = case (inListVar1 zn lst) of
  True -> (ChanInp (join_name (get_proc_name zn lst) zn))
  _ -> (ChanInp zn)
rename_vars_CParameter1 lst (ChanInpPred zn zp)
 = case (inListVar1 zn lst) of
  True -> (ChanInpPred (join_name (get_proc_name zn lst) zn) (rename_vars_ZPred1 lst zp))
  _ -> (ChanInpPred zn zp)
rename_vars_CParameter1 lst (ChanOutExp ze)
 = (ChanOutExp (rename_vars_ZExpr1 lst ze))
rename_vars_CParameter1 lst (ChanDotExp ze)
 = (ChanDotExp (rename_vars_ZExpr1 lst ze))
```

```
rename_vars_CCommand1 :: [(ZName, ZVar, ZExpr)] -> CCommand -> CCommand
rename_vars_CCommand1 lst (CAssign zv ze)
 = (CAssign (map (rename_vars_ZVar1 lst) zv)
            (map (rename_vars_ZExpr1 lst) ze))
rename_vars_CCommand1 lst (CIf ga)
 = (CIf (rename_vars_CGActions1 lst ga))
rename_vars_CCommand1 lst (CVarDecl zgf a)
 = (CVarDecl zgf (rename_vars_CAction1 lst a))
rename_vars_CCommand1 lst (CAssumpt znls zp1 zp2)
 = (CAssumpt znls (rename_vars_ZPred1 lst zp1) zp2)
rename_vars_CCommand1 lst (CAssumpt1 znls zp)
 = (CAssumpt1 znls zp)
rename_vars_CCommand1 lst (CPrefix zp1 zp2)
 = (CPrefix (rename_vars_ZPred1 lst zp1) zp2)
rename_vars_CCommand1 lst (CPrefix1 zp)
 = (CPrefix1 zp)
rename_vars_CCommand1 lst (CommandBrace zp)
 = (CommandBrace zp)
rename_vars_CCommand1 lst (CommandBracket zp)
 = (CommandBracket zp)
rename_vars_CCommand1 lst (CValDecl zgf a)
 = (CValDecl zgf (rename_vars_CAction1 lst a))
rename_vars_CCommand1 lst (CResDecl zgf a)
 = (CResDecl zgf (rename_vars_CAction1 lst a))
rename_vars_CCommand1 lst (CVResDecl zgf a)
 = (CVResDecl zgf (rename_vars_CAction1 lst a))
```

```
rename_vars_CGActions1 :: [(ZName, ZVar, ZExpr)] -> CGActions -> CGActions
rename_vars_CGActions1 lst (CircGAction zp a)
 = (CircGAction (rename_vars_ZPred1 lst zp) (rename_vars_CAction1 lst a))
rename_vars_CGActions1 lst (CircThenElse (CircGAction zp a) cga2)
 = (CircThenElse (CircGAction (rename_vars_ZPred1 lst zp) (rename_vars_CAction1 lst a)) (rename_var
-- rename_vars_CGActions1 lst (CircElse pa)
-- = (CircElse pa)
```

```
rename_vars_CReplace1 :: [(ZName, ZVar, ZExpr)] -> CReplace -> CReplace
rename_vars_CReplace1 lst (CRename zvarls1 zvarls)
 = (CRename zvarls1 zvarls)
rename_vars_CReplace1 lst (CRenameAssign zvarls1 zvarls)
 = (CRenameAssign zvarls1 zvarls)
```

```
bindingsVar1 lst []
 = []
bindingsVar1 lst [((va,x),b)]
 = [(((join_name (get_proc_name va lst) va),x),(rename_vars_ZExpr1 lst b))]
bindingsVar1 lst (((va,x),b):xs)
 = [(((join_name (get_proc_name va lst) va),x),(rename_vars_ZExpr1 lst b))]++(bindingsVar1 lst xs)
```

```
get_bindings_var []
 = []
get_bindings_var [((va,x),b)]
 = [va]
get_bindings_var (((va,x),b):xs)
 = va:(get_bindings_var xs)
```

```
inListVar1 :: ZName -> [(ZName, ZVar, ZExpr)] -> Bool
inListVar1 x []
 = False
inListVar1 x [(a,(va,x1),b)]
 = case x == va of
  True -> True
  _ -> False
inListVar1 x ((a,(va,x1),b):vst)
 = case x == va of
  True -> True
  _ -> inListVar1 x vst
```

```
get_proc_name :: ZName -> [(ZName, ZVar, ZExpr)] -> ZName
get_proc_name x [(a,(va,x1),b)]
 = case x == va of
  True -> a
  _ -> ""
get_proc_name x ((a,(va,x1),b):vst)
 = case x == va of
  True -> a
  _ -> get_proc_name x vst

get_var_type :: ZName -> [(ZName, ZVar, ZExpr)] -> ZExpr
get_var_type x [(a,(va,x1),b)]
 = case x == va of
  True -> b
  _ -> error "type not found whilst get_var_type"
get_var_type x ((a,(va,x1),b):vst)
 = case x == va of
  True -> b
  _ -> get_var_type x vst
```

```
rename_ZGenFilt1 lst (Include s) = (Include s)
rename_ZGenFilt1 lst (Choose (va,x) e)
  = (Choose ((join_name (join_name (join_name "sv" (get_proc_name va lst)) va) newt),x) (rename_var
    where newt = (def_U_NAME $ get_vars_ZExpr e)
rename_ZGenFilt1 lst (Check p) = (Check (rename_vars_ZPred1 lst p))
rename_ZGenFilt1 lst (Evaluate v e1 e2) = (Evaluate v (rename_vars_ZExpr1 lst e1) (rename_vars_ZExp
```

```
rename_vars_ZVar1 :: [(ZName, ZVar, ZExpr)] -> ZVar -> ZVar
rename_vars_ZVar1 lst (va,x)
 = case (inListVar1 va lst) of
    True -> ((join_name (join_name (join_name "sv" (get_proc_name va lst)) va) newt),x)
    _ -> (va,x)
    where newt = (def_U_NAME $ get_vars_ZExpr $ get_var_type va lst)
```

```
rename_vars_ZExpr1 :: [(ZName, ZVar, ZExpr)] -> ZExpr -> ZExpr
rename_vars_ZExpr1 lst (ZVar (va,x))
 = case (inListVar1 va lst) of
    True -> (ZVar
              ((join_name (join_name (join_name "sv" (get_proc_name va lst)) va) newt),x))
    _ -> (ZVar (va,x))
  where newt = (def_U_NAME $ get_vars_ZExpr $ get_var_type va lst)
rename_vars_ZExpr1 lst (ZInt zi)
 = (ZInt zi)
rename_vars_ZExpr1 lst (ZGiven gv)
 = (ZGiven gv)
rename_vars_ZExpr1 lst (ZFree0 va)
 = (ZFree0 va)
rename_vars_ZExpr1 lst (ZFree1 va xpr)
 = (ZFree1 va (rename_vars_ZExpr1 lst xpr))
rename_vars_ZExpr1 lst (ZTuple xpr)
 = (ZTuple (map (rename_vars_ZExpr1 lst) xpr))
rename_vars_ZExpr1 lst (ZBinding xs)
 = (ZBinding (bindingsVar1 lst xs))
rename_vars_ZExpr1 lst (ZSetDisplay xpr)
 = (ZSetDisplay (map (rename_vars_ZExpr1 lst) xpr))
rename_vars_ZExpr1 lst (ZSeqDisplay xpr)
 = (ZSeqDisplay (map (rename_vars_ZExpr1 lst) xpr))
rename_vars_ZExpr1 lst (ZFSet zf)
 = (ZFSet zf)
rename_vars_ZExpr1 lst (ZIntSet i1 i2)
 = (ZIntSet i1 i2)
rename_vars_ZExpr1 lst (ZGenerator zrl xpr)
 = (ZGenerator zrl (rename_vars_ZExpr1 lst xpr))
rename_vars_ZExpr1 lst (ZCross xpr)
 = (ZCross (map (rename_vars_ZExpr1 lst) xpr))
rename_vars_ZExpr1 lst (ZFreeType va pname1)
 = (ZFreeType va pname1)
rename_vars_ZExpr1 lst (ZPowerSet{baseset=xpr, is_non_empty=b1, is_finite=b2})
 = (ZPowerSet{baseset=(rename_vars_ZExpr1 lst xpr), is_non_empty=b1, is_finite=b2})
rename_vars_ZExpr1 lst (ZFuncSet{ domset=expr1, ranset=expr2, is_function=b1, is_total=b2, is_onto=
 = (ZFuncSet{ domset=(rename_vars_ZExpr1 lst expr1), ranset=(rename_vars_ZExpr1 lst expr2), is_func
rename_vars_ZExpr1 lst (ZSetComp pname1 (Just xpr))
 = (ZSetComp (map (rename_ZGenFilt1 lst) pname1) (Just (rename_vars_ZExpr1 lst xpr)))
rename_vars_ZExpr1 lst (ZSetComp pname1 Nothing)
 = (ZSetComp (map (rename_ZGenFilt1 lst) pname1) Nothing)
rename_vars_ZExpr1 lst (ZLambda pname1 xpr)
 = (ZLambda (map (rename_ZGenFilt1 lst) pname1) (rename_vars_ZExpr1 lst xpr))
rename_vars_ZExpr1 lst (ZESchema zxp)
 = (ZESchema zxp)
rename_vars_ZExpr1 lst (ZGivenSet gs)
 = (ZGivenSet gs)
rename_vars_ZExpr1 lst (ZUniverse)
 = (ZUniverse)
rename_vars_ZExpr1 lst (ZCall xpr1 xpr2)
 = (ZCall (rename_vars_ZExpr1 lst xpr1) (rename_vars_ZExpr1 lst xpr2))
rename_vars_ZExpr1 lst (ZReln rl)
 = (ZReln rl)
rename_vars_ZExpr1 lst (ZFunc1 f1)
 = (ZFunc1 f1)
rename_vars_ZExpr1 lst (ZFunc2 f2)
 = (ZFunc2 f2)
rename_vars_ZExpr1 lst (ZStrange st)
 = (ZStrange st)
rename_vars_ZExpr1 lst (ZMu pname1 (Just xpr))
 = (ZMu (map (rename_ZGenFilt1 lst) pname1) (Just (rename_vars_ZExpr1 lst xpr)))
rename_vars_ZExpr1 lst (ZELet pname1 xpr1)
```

```
 = (ZELet (bindingsVar1 lst pname1) (rename_vars_ZExpr1 lst xpr1))
rename_vars_ZExpr1 lst (ZIf_Then_Else zp xpr1 xpr2)
 = (ZIf_Then_Else zp (rename_vars_ZExpr1 lst xpr1) (rename_vars_ZExpr1 lst xpr2))
rename_vars_ZExpr1 lst (ZSelect xpr va)
 = (ZSelect xpr va)
rename_vars_ZExpr1 lst (ZTheta zs)
 = (ZTheta zs)
```

```
rename_vars_ZPred1 :: [(ZName, ZVar, ZExpr)] -> ZPred -> ZPred
rename_vars_ZPred1 lst (ZFalse{reason=a})
 = (ZFalse{reason=a})
rename_vars_ZPred1 lst (ZTrue{reason=a})
 = (ZTrue{reason=a})
rename_vars_ZPred1 lst (ZAnd p1 p2)
 = (ZAnd (rename_vars_ZPred1 lst p1) (rename_vars_ZPred1 lst p2))
rename_vars_ZPred1 lst (ZOr p1 p2)
 = (ZOr (rename_vars_ZPred1 lst p1) (rename_vars_ZPred1 lst p2))
rename_vars_ZPred1 lst (ZImplies p1 p2)
 = (ZImplies (rename_vars_ZPred1 lst p1) (rename_vars_ZPred1 lst p2))
rename_vars_ZPred1 lst (ZIff p1 p2)
 = (ZIff (rename_vars_ZPred1 lst p1) (rename_vars_ZPred1 lst p2))
rename_vars_ZPred1 lst (ZNot p)
 = (ZNot (rename_vars_ZPred1 lst p))
rename_vars_ZPred1 lst (ZExists pname1 p)
 = (ZExists pname1 (rename_vars_ZPred1 lst p))
rename_vars_ZPred1 lst (ZExists_1 lst1 p)
 = (ZExists_1 lst1 (rename_vars_ZPred1 lst p))
rename_vars_ZPred1 lst (ZForall pname1 p)
 = (ZForall pname1 (rename_vars_ZPred1 lst p))
rename_vars_ZPred1 lst (ZPLet varxp p)
 = (ZPLet varxp (rename_vars_ZPred1 lst p))
rename_vars_ZPred1 lst (ZEqual xpr1 xpr2)
 = (ZEqual (rename_vars_ZExpr1 lst xpr1) (rename_vars_ZExpr1 lst xpr2))
rename_vars_ZPred1 lst (ZMember xpr1 xpr2)
 = (ZMember (rename_vars_ZExpr1 lst xpr1) (rename_vars_ZExpr1 lst xpr2))
rename_vars_ZPred1 lst (ZPre sp)
 = (ZPre sp)
rename_vars_ZPred1 lst (ZPSchema sp)
 = (ZPSchema sp)
```

```
-- extract the delta variables in here'
get_delta_names1 [(ZFreeTypeDef ("NAME",[]) xs)]
  = get_delta_names_aux1 xs
get_delta_names1 ((ZFreeTypeDef ("NAME",[]) xs):xss)
  = (get_delta_names_aux1 xs)++(get_delta_names1 xss)
get_delta_names1 (_:xs)
  = (get_delta_names1 xs)
get_delta_names1 []
  = []
```

```
get_delta_names_aux1 [(ZBranch0 (a,[]))] = [a]
get_delta_names_aux1 ((ZBranch0 (a,[])):xs) = [a]++(get_delta_names_aux1 xs)
```

```
-- extract the delta variables in here' from the same state
get_delta_names zn [(ZFreeTypeDef ("NAME",[]) xs)]
  = get_delta_names_aux zn xs
get_delta_names zn ((ZFreeTypeDef ("NAME",[]) xs):xss)
  = (get_delta_names_aux zn xs)++(get_delta_names zn xss)
get_delta_names zn (_:xs)
  = (get_delta_names zn xs)
get_delta_names _ []
  = []
```

```
get_delta_names_aux zn [(ZBranch0 (a,[]))]
  | isPrefixOf zn a = [a]
  | otherwise = []
get_delta_names_aux zn ((ZBranch0 (a,[])):xs)
  | isPrefixOf zn a = [a]++(get_delta_names_aux zn xs)
  | otherwise = (get_delta_names_aux zn xs)
```

```
-- Make UNIVERSE datatype in CSP
mk_universe []
  = ""
mk_universe [(a,b,c,d)]
  = c++"."++d
mk_universe ((a,b,c,d):xs)
  = c++"."++d++" | "++(mk_universe xs)

-- Make subtype U_TYP = TYP.TYPE
mk_subtype []
  = ""
mk_subtype [(a,b,c,d)]
  = "subtype "++b++" = "++c++"."++d++"\n"
mk_subtype ((a,b,c,d):xs)
  = "subtype "++b++" = "++c++"."++d++"\n"++(mk_subtype xs)

-- Make value(XXX.v) function call
-- This won't be used anymore in the next commit - 21.03.17
mk_value []
  = ""
mk_value [(a,b,c,d)]
  = "value"++(lastN 3 b)++"("++c++".v) = v\n"
mk_value ((a,b,c,d):xs)
  = "value"++(lastN 3 b)++"("++c++".v) = v\n"++(mk_value xs)

-- Make type(x) function call
-- This won't be used anymore in the next commit - 21.03.17
mk_type []
  = ""
mk_type [(a,b,c,d)]
  = "type"++(lastN 3 b)++"(x) = U_"++(lastN 3 b)++"\n"
mk_type ((a,b,c,d):xs)
  = "type"++(lastN 3 b)++"(x) = U_"++(lastN 3 b)++"\n"++(mk_type xs)

-- Make tag(x) function call
mk_tag []
  = ""
mk_tag [(a,b,c,d)]
  = "tag"++(lastN 3 b)++"(x) = "++(lastN 3 b)++"\n"
mk_tag ((a,b,c,d):xs)
  = "tag"++(lastN 3 b)++"(x) = "++(lastN 3 b)++"\n"++(mk_tag xs)

-- make Memory(b_type1,b_type2,b_type3) parameters
mk_mem_param :: [(t, [Char], t1, t2)] -> [Char]
mk_mem_param [] = ""
mk_mem_param [(a,b,c,d)] = "b_"++(lastN 3 b)
mk_mem_param ((a,b,c,d):xs)
  = (mk_mem_param [(a,b,c,d)]) ++", "++ (mk_mem_param xs)

-- list of b_type parameters
mk_mem_param_lst :: [(t, [Char], t1, t2)] -> [[Char]]
mk_mem_param_lst [] = []
mk_mem_param_lst [(a,b,c,d)] = ["b_"++(lastN 3 b)]
mk_mem_param_lst ((a,b,c,d):xs)
  = (mk_mem_param_lst [(a,b,c,d)]) ++ (mk_mem_param_lst xs)

-- replace b_type by over(b_type,n,x) in case x == a
repl_mem_param_over :: [Char] -> [[Char]] -> [[Char]]

repl_mem_param_over _ [] = []
repl_mem_param_over a [x]
  | (lastN 3 x) == a  = ["over("++x++",n,x)"]
  | otherwise = [x]
repl_mem_param_over a (x:xs)
  = (repl_mem_param_over a [x]) ++ (repl_mem_param_over a xs)

-- list of b_type parameters into string of b_type1,b_type2,...
```

```
mk_charll_to_charl :: [Char] -> [[Char]] -> [Char]
mk_charll_to_charl _ [] = ""
mk_charll_to_charl sp [x] = x
mk_charll_to_charl sp (x:xs) = x++sp++(mk_charll_to_charl sp xs)

-- make mget external choices of Memory proc
mk_mget_mem_bndg :: [(t3, [Char], t4, t5)] -> [(t, [Char], t1, t2)] -> [Char]
mk_mget_mem_bndg fs []
  = ""
mk_mget_mem_bndg fs [(a,b,c,d)]
  = "([] n:dom(b_"++(lastN 3 b)++") @ mget.n!(apply(b_"++(lastN 3 b)++",n)) -> Memory("++(mk_mem_pa
mk_mget_mem_bndg fs ((a,b,c,d):xs)
  = mk_mget_mem_bndg fs [(a,b,c,d)]
  ++"\n\t[] "++mk_mget_mem_bndg fs xs


-- make mset external choices of Memory proc
mk_mset_mem_bndg fs []
  = ""
mk_mset_mem_bndg fs [(a,b,c,d)]
  = "\t[] ([] n:dom(b_"
      ++(lastN 3 b)
      ++") @ mset.n?x:type"
      ++ (lastN 3 b)
      ++"(n) -> Memory("
      ++  ( mk_charll_to_charl "," (repl_mem_param_over (lastN 3 b) (mk_mem_param_lst fs) ))
      ++"))"
mk_mset_mem_bndg fs ((a,b,c,d):xs)
  = mk_mset_mem_bndg fs [(a,b,c,d)]
  ++"\n"++mk_mset_mem_bndg fs xs


-- make subtype NAME_TYPE1, subtype...

-- first we get the names from NAME datatype
select_zname_f_zbr (ZBranch0 (n,[])) = n
select_zname_f_zbr _ = ""

--then we get the type of some name
select_type_zname n = (lastN 3 n)

-- now we filter a list of names nms of a selected type tp
filter_znames_f_type [] tp = []
filter_znames_f_type [n] tp
  | (lastN 3 n) == tp = [n]
  | otherwise = []
filter_znames_f_type (n:nms) tp
  = (filter_znames_f_type [n] tp) ++ (filter_znames_f_type nms tp)

-- with all that, we create a subtype NAME_TYPEX
lst_subtype t [] = []
lst_subtype t [z]
      | (lastN 3 z) == t = [z]
      | otherwise = []
lst_subtype t (z:zs)
      | (lastN 3 z) == t = [z] ++ (lst_subtype t zs)
      | otherwise = (lst_subtype t zs)
make_subtype_NAME zb
  = nametypels
  where
    make_subtype znls zt = "subtype NAME_"++zt++" = "++mk_charll_to_charl " | " (lst_subtype zt znl
    znames = remdups $ map select_zname_f_zbr zb
    ztypes = remdups $ map select_type_zname znames
    nametypels = mk_charll_to_charl "\n" $ map (make_subtype znames) ztypes

-- make NAME_VALUES_TYPE
mk_NAME_VALUES_TYPE n
  = "NAMES_VALUES_"++n++" = seq({seq({(n,v) | v <- type"++n++"(n)}) | n <- NAME_"++n++"})"
-- make BINDINGS_TYPE
mk_BINDINGS_TYPE n
```

```
  = "BINDINGS_"++n++" = {set(b) | b <- set(distCartProd(NAMES_VALUES_"++n++"))}"
-- make restrict functions within main action
mk_binding_list n
  = "b_"++n++" : BINDINGS_" ++ n
mk_restrict spec vlst n
    = "\t\trestrict"++n++"(bs) = dres(bs,{"++(mk_charll_to_charl ", " $ lst_subtype n vlst)++"})"
    where
      univlst = (def_universe spec)
      funivlst = remdups (filter_types_universe univlst)
      bndlst = mk_mem_param_lst funivlst

mk_restrict_name n
  = "restrict"++n++"("++"b_"++n++")"
```

```
-- extract the delta variables and types in here'
def_universe [(ZAbbreviation ("\\delta",[]) (ZSetDisplay xs))]
  = def_universe_aux xs
def_universe ((ZAbbreviation ("\\delta",[]) (ZSetDisplay xs)):xss)
  = (def_universe_aux xs)++(def_universe xss)
def_universe (_:xs)
  = (def_universe xs)
def_universe []
  = []
```

```
def_universe_aux :: [ZExpr] -> [(String, [Char], [Char], [Char])]
def_universe_aux [] = []
def_universe_aux [ZCall (ZVar ("\\mapsto",[])) (ZTuple [ZVar (b,[]),ZVar ("\\nat",[])])] = [(b,"U_N
def_universe_aux [ZCall (ZVar ("\\mapsto",[])) (ZTuple [ZVar (b,[]),ZVar (c,[])])]= [(b,(def_U_NAME
def_universe_aux ((ZCall (ZVar ("\\mapsto",[])) (ZTuple [ZVar (b,[]),ZVar ("\\nat",[])])):xs) = ((b
def_universe_aux ((ZCall (ZVar ("\\mapsto",[])) (ZTuple [ZVar (b,[]),ZVar (c,[])])):xs) = ((b,(def_
def_universe_aux [ZCall (ZVar ("\\mapsto",[])) (ZTuple [ZVar (b,[]), ZCall (ZVar ("\\power",[])) (
  = [(b,(def_U_NAME ("P"++c)), (def_U_prefix ("P"++c)), ("Set("++c++")"))]
def_universe_aux ((ZCall (ZVar ("\\mapsto",[])) (ZTuple [ZVar (b,[]), ZCall (ZVar ("\\power",[])) (
  = ((b,(def_U_NAME ("P"++c)), (def_U_prefix ("P"++c)), ("Set("++c++")")):(def_universe_aux xs))
```

```
filter_types_universe [] = []
filter_types_universe [(a,b,c,d)] = [(b,b,c,d)]
filter_types_universe ((a,b,c,d):xs) = ((b,b,c,d):(filter_types_universe xs))
```

Pieces from MappingFunStatelessCircus file

```

def_delta_mapping :: [(ZName, ZVar, ZExpr)] -> [ZExpr]
def_delta_mapping [(n,(v,[]),t)]
  = [ZCall (ZVar ("\\mapsto",[])) (ZTuple [ZVar ((join_name (join_name (join_name "sv" n) v) newt),
    where newt = (def_U_NAME $ get_vars_ZExpr t)
def_delta_mapping ((n,(v,[]),t):xs)
  = [ZCall (ZVar ("\\mapsto",[])) (ZTuple [ZVar ((join_name (join_name (join_name "sv" n) v) newt),
    ++ (def_delta_mapping xs)
    where newt = (def_U_NAME $ get_vars_ZExpr t)
def_delta_mapping [] = []
```

```
def_delta_name :: [(ZName, ZVar, ZExpr)] -> [ZBranch]
def_delta_name [(n,(v,[]),t)]
  = [ZBranch0 ((join_name (join_name (join_name "sv" n) v) newt),[])]
    where newt = (def_U_NAME $ get_vars_ZExpr t)
def_delta_name ((n,(v,[]),t):xs)
  = [ZBranch0 ((join_name (join_name (join_name "sv" n) v) newt),[])]
    ++ (def_delta_name xs)
    where newt = (def_U_NAME $ get_vars_ZExpr t)
def_delta_name [] = []
```

```
get_pre_Circ_proc :: [ZPara] -> [ZPara]
get_pre_Circ_proc ((Process cp):xs)
  = (get_pre_Circ_proc xs)
get_pre_Circ_proc (x:xs)
  = x:(get_pre_Circ_proc xs)
get_pre_Circ_proc []
  = []
```

8.10 Creating the Memory process

```
def_mem_st_Circus_aux :: [ZPara] -> [ZPara] -> [(ZName, ZVar, ZExpr)]
def_mem_st_Circus_aux spec []
  = []
def_mem_st_Circus_aux spec [x]
  = def_mem_st_CircParagraphs spec x
def_mem_st_Circus_aux spec (x:xs)
  = (def_mem_st_CircParagraphs spec x)++(def_mem_st_Circus_aux spec xs)
```

```
rename_z_schema_state spec (CProcess p (ProcDef (ProcMain (ZSchemaDef (ZSPlain n) schlst) proclst m
  = (CProcess p (ProcDef (ProcMain (ZSchemaDef (ZSPlain n) (ZSchema xs)) proclst ma)))
    where
      xs = retrive_schemas spec schlst
rename_z_schema_state spec x = x
```

```
retrive_schemas spec (ZSchema lstx) = lstx
retrive_schemas spec (ZSRef (ZSPlain nn) [] [])
  = case res of
      Just e' -> e'
      Nothing -> error "Schema definition not found!"
    where
      res = (retrieve_z_schema_state nn spec)
retrive_schemas spec (ZS1 x a)
  = (retrive_schemas spec a)
retrive_schemas spec (ZS2 ZSAnd a b)
  = (retrive_schemas spec a)++(retrive_schemas spec b)
retrive_schemas spec (ZSHide a b) = retrive_schemas spec a
retrive_schemas spec (ZSExists a b) = retrive_schemas spec b
retrive_schemas spec (ZSExists_1 a b) = retrive_schemas spec b
retrive_schemas spec (ZSForall a b) = retrive_schemas spec b
retrive_schemas spec _ = error "Schema def not implemented yet"
```

```
retrieve_z_schema_state n [(ZSchemaDef (ZSPlain nn) (ZSchema lstx))]
  | n == nn = Just lstx
  | otherwise = Nothing
retrieve_z_schema_state n [_] = Nothing
retrieve_z_schema_state n ((ZSchemaDef (ZSPlain nn) (ZSchema lstx)):xs)
  | n == nn = Just lstx
  | otherwise = retrieve_z_schema_state n xs
retrieve_z_schema_state n (_:xs) = retrieve_z_schema_state n xs
```

```
def_mem_st_CircParagraphs :: [ZPara] -> ZPara -> [(ZName, ZVar, ZExpr)]
def_mem_st_CircParagraphs spec (Process cp)
  = (def_mem_st_ProcDecl spec ncp)
    where
      ncp = rename_z_schema_state spec cp
def_mem_st_CircParagraphs spec x
  = []
```

```
def_mem_st_ProcDecl :: [ZPara] -> ProcDecl -> [(ZName, ZVar, ZExpr)]
def_mem_st_ProcDecl spec (CGenProcess zn (x:xs) pd)
  = (def_mem_st_ProcessDef spec zn pd)
def_mem_st_ProcDecl spec (CProcess zn pd)
  = (def_mem_st_ProcessDef spec zn pd)
```

```
def_mem_st_ProcessDef :: [ZPara] -> ZName -> ProcessDef -> [(ZName, ZVar, ZExpr)]
def_mem_st_ProcessDef spec name (ProcDefSpot xl pd)
  = (def_mem_st_ProcessDef spec name pd)
def_mem_st_ProcessDef spec name (ProcDefIndex xl pd)
  = (def_mem_st_ProcessDef spec name pd)
def_mem_st_ProcessDef spec name (ProcDef cp)
  = (def_mem_st_CProc spec name cp)
```

```
def_mem_st_CProc :: [ZPara] -> ZName -> CProc -> [(ZName, ZVar, ZExpr)]
def_mem_st_CProc spec name (ProcMain (ZSchemaDef n xls) (x:xs) ca)
  = (get_state_var spec name xls)
```

```
def_mem_st_CProc spec name x
  = []
```

```
get_state_var :: [ZPara] -> ZName -> ZSExpr -> [(ZName, ZVar, ZExpr)]
get_state_var spec name (ZSRef (ZSPlain nn) [] [])
  = case statev of
      Just s -> concat (map (get_state_var_aux name) s)
      Nothing -> []
    where
      statev = retrieve_z_schema_state nn spec

get_state_var spec name (ZSchema s)
  = concat (map (get_state_var_aux name) s)
get_state_var _ _ _ = []
```

```
get_state_var_aux name (Choose x y) = [(name, x, y)]
get_state_var_aux _ _ = []
```

Here I'm making the bindings for the main action.

```
filter_main_action_bind
  :: ZName -> [(ZName, ZVar, ZExpr)] ->[(ZName, ZVar, ZExpr)]
filter_main_action_bind zn [(a,b,c)]
  | zn == a = [(a,b,c)]
  | otherwise = []
filter_main_action_bind zn ((a,b,c):ls)
  | zn == a = [(a,b,c)] ++ filter_main_action_bind zn ls
  | otherwise = filter_main_action_bind zn ls

mk_main_action_bind :: [(ZName, ZVar, ZExpr)] -> CAction -> CAction
mk_main_action_bind lst ca
  | null lst = (CActionCommand (CValDecl [Choose ("b",[]) (ZSetComp [Choose ("x",[]) (ZVar ("BINDIN
  | otherwise = (CActionCommand (CValDecl [Choose ("b",[]) (ZSetComp [Choose ("x",[]) (ZVar ("BINDI
```

```
mk_inv :: [(ZName, ZVar, ZExpr)] -> [(ZName, ZVar, ZExpr)] -> ZPred
mk_inv lst [(a,(va,x),c)]
  = (ZMember (ZVar ((join_name (join_name (join_name "sv" a) va) newt),x)) (rename_vars_ZExpr1 lst
    where newt = (def_U_NAME $ get_vars_ZExpr c)
mk_inv lst ((a,b,c):xs)
  = (ZAnd (mk_inv lst xs) (mk_inv lst [(a,b,c)]))
-- mk_inv x (_:xs) = mk_inv x xs
```

Given $\{v_0, \ldots, v_n\}$, the function *make_maps_to* returns $\{v_0 \mapsto vv_o, \ldots, v_n \mapsto vv_n\}$.

```
make_maps_to :: [ZVar] -> [ZExpr]
make_maps_to [(x,[])]
  = [ZCall (ZVar ("\\mapsto",[]))
    (ZTuple [ZVar (x,[]),ZVar ("val"++x,[])])]
make_maps_to ((x,[]):xs)
  = [ZCall (ZVar ("\\mapsto",[]))
    (ZTuple [ZVar (x,[]),ZVar ("val"++x,[])])]++(make_maps_to xs)
```

TODO: this function here should somehow propagate any parameter from a replicated operator
EX: [] i: a,b,c @ x.i -¿ SKIP = x.a -¿ SKIP [] x.b -¿ SKIP [] x.c -¿ SKIP EX: [] i: a,b,c @ A(x) = A(a) [] A(b) [] A(c)

```
propagate_CSPRep (CActionSchemaExpr e) = (CActionSchemaExpr e)
propagate_CSPRep (CActionCommand c) = (CActionCommand c)
propagate_CSPRep (CActionName n) = (CActionName n)
propagate_CSPRep (CSPSkip) = (CSPSkip)
propagate_CSPRep (CSPStop ) = (CSPStop )
propagate_CSPRep (CSPChaos) = (CSPChaos)
propagate_CSPRep (CSPCommAction c a) = (CSPCommAction c (propagate_CSPRep a))
propagate_CSPRep (CSPGuard p a) = (CSPGuard p (propagate_CSPRep a))
propagate_CSPRep (CSPSeq a1 a2) = (CSPSeq (propagate_CSPRep a1) (propagate_CSPRep a2))
propagate_CSPRep (CSPExtChoice a1 a2) = (CSPExtChoice (propagate_CSPRep a1) (propagate_CSPRep a2))
propagate_CSPRep (CSPIntChoice a1 a2) = (CSPIntChoice (propagate_CSPRep a1) (propagate_CSPRep a2))
propagate_CSPRep (CSPNSParal n1 c n2 a1 a2) = (CSPNSParal n1 c n2 (propagate_CSPRep a1) (propagate_
propagate_CSPRep (CSPParal c a1 a2) = (CSPParal c (propagate_CSPRep a1) (propagate_CSPRep a2))
propagate_CSPRep (CSPNSInter n1 n2 a1 a2) = (CSPNSInter n1 n2 (propagate_CSPRep a1) (propagate_CSPR
```

```
propagate_CSPRep (CSPInterleave a1 a2) = (CSPInterleave (propagate_CSPRep a1) (propagate_CSPRep a2)
propagate_CSPRep (CSPHide a c) = (CSPHide (propagate_CSPRep a) c)
propagate_CSPRep (CSPParAction n ls) = (CSPParAction n ls)
propagate_CSPRep (CSPRenAction n r) = (CSPRenAction n r)
propagate_CSPRep (CSPRecursion n a) = (CSPRecursion n (propagate_CSPRep a))
propagate_CSPRep (CSPUnParAction ls a n) = (CSPUnParAction ls (propagate_CSPRep a) n)
propagate_CSPRep (CSPRepExtChoice ls a) = (CSPRepExtChoice ls (propagate_CSPRep a))
propagate_CSPRep (CSPRepIntChoice ls a) = (CSPRepIntChoice ls (propagate_CSPRep a))
propagate_CSPRep (CSPRepParalNS c ls n a) = (CSPRepParalNS c ls n (propagate_CSPRep a))
propagate_CSPRep (CSPRepParal c ls a) = (CSPRepParal c ls (propagate_CSPRep a))
propagate_CSPRep (CSPRepInterlNS ls n a) = (CSPRepInterlNS ls n (propagate_CSPRep a))
propagate_CSPRep (CSPRepInterl ls a) = (CSPRepInterl ls (propagate_CSPRep a))
```

```
make_memory_proc =
  CParAction "Memory" (CircusAction (CActionCommand (CVResDecl [Choose ("b",[]) (ZVar ("BINDING",[]
```