Circus to CSP

Artur Oliveira Gomes

February 8, 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

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

```
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.

```
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

```
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
 | 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
  | CAssumpt [ZName] ZPred ZPred              -- N^{+} \prefixcolon [Pred,Pred]
  | CAssumpt1 [ZName] ZPred                   -- N^{+} \prefixcolon [Pred,Pred]
  | CPrefix ZPred ZPred                       -- \prefixcolon [Pred,Pred]
  | CPrefix1 ZPred                            -- \prefixcolon [Pred]
  | CommandBrace ZPred                        -- \{Pred\}
  | CommandBracket ZPred                      -- [Pred]
  | CValDecl [ZGenFilt] CAction               -- \circval Decl \circspot Action
  | CResDecl [ZGenFilt] CAction               -- \circres Decl \circspot Action
  | CVResDecl [ZGenFilt] CAction              -- \circvres Decl \circspot Action
  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)
--     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)
```

3 Mapping Functions - Stateless Circus

Mapping Omega Functions from Circus to Circus

3.1 Stateless Circus - Actions

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

is written in Haskell as:

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

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

is written in Haskell as:

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

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

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):xs)) a)
  = make_get_com lxs (rename_vars_CAction (CSPCommAction (ChanComm c ((ChanDotExp e):xs)) (omega_pr
  where lxs = 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):xs)) a)
  = omega_CAction (CSPCommAction (ChanComm c ((ChanDotExp e):xs)) a)
```

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

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 = 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:

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

is written in Haskell as:

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

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

where

$$x \in wrtV(A)$$

is written in Haskell as:

```
omega_CAction (CSPCommAction (ChanComm c [ChanInpPred (x:xs) 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 = concat (map get_ZVar_st (free_var_ZPred p))
```

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

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))
```

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

is written in Haskell as:

```
omega_CAction (CSPExtChoice ca cb)
  = make_get_com lsx (rename_vars_CAction (CSPExtChoice (omega_prime_CAction ca) (omega_prime_CActi
   where
    lsx = (map fst (nub (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 = (map fst (nub (free_var_CAction a1))) `union` (map fst (nub (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:

```haskell
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))
```

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

is written in Haskell as:

```haskell
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 (x1,[])]) act)
  = (CSPRepParalNS (CSExpr cs) [Choose (x,[]) (ZSetDisplay lsx)]
          (NSExprParam ns [ZVar (x1,[])])
          (omega_CAction act))
```

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

```haskell
omega_CAction (CActionCommand (CAssign varls valls))
  = make_get_com (map fst varls) (rename_vars_CAction (make_set_com varls valls CSPSkip))
```

$$
\begin{array}{l}
\Omega_A(\mathbf{if}\ g(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow A\ \mathbf{fi}) \mathrel{\widehat=} \\
\quad get.v_0?vv_0 \longrightarrow \ldots \longrightarrow get.v_n?vv_n \longrightarrow \\
\quad get.l_0?vl_0 \longrightarrow \ldots \longrightarrow get.l_m?vl_m \longrightarrow \\
\quad \mathbf{if}\ g(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow \Omega'_A(A)\ \mathbf{fi}
\end{array}
$$

```haskell
omega_CAction (CActionCommand (CIf (CircGAction g a)))
  = make_get_com lsx (rename_vars_CAction (CActionCommand
             (CIf (CircGAction g (omega_prime_CAction a)))))
  where
   lsx = (map fst (nub (free_var_ZPred g)))
```

$$\Omega_A\left(\begin{array}{l}\mathbf{if}\ g_0(v_0,...,v_n,l_0,...,l_m) \longrightarrow A_0\\ \qquad [\!]\ ...\\ \qquad [\!]\ g_n(v_0,...,v_n,l_0,...,l_m) \longrightarrow A_n\\ \mathbf{fi}\end{array}\right) \mathrel{\widehat=}$$

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

```
omega_CAction (CActionCommand (CIf (CircThenElse gl glx)))
  = make_get_com lsx (rename_vars_CAction (CActionCommand (CIf (mk_guard_pair guard_pair))))
  where
   guard_pair = get_guard_pair (CircThenElse gl glx)
   lsx = map fst (remdups $ concat $ map free_var_ZPred (map fst guard_pair))
```

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

is written in Haskell as:

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

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

is written in Haskell as:

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

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

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

is written in Haskell as:

```
omega_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))
omega_CAction x = x
```

NOTE: Besides the transformation rules for $[g]$ and g, the remaining transformation rules from page 91 of the D24.1 document, were not yet implemented.

3.2 Definitions of Ω'_A

```
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))
```

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

is written in Haskell as:

```
omega_prime_CAction (CSPCommAction (ChanComm c x) a)
  = (CSPCommAction (ChanComm c x) (omega_prime_CAction a))
```

$$\Omega'_A(g(v_0, \ldots, v_n, l_0, \ldots, l_m) \longrightarrow A) \mathrel{\widehat{=}} \\ g(vv_0, \ldots, vv_n, vl_0, \ldots, vl_m) \mathbin{\&} \Omega'_A(A)$$

is written in Haskell as:

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

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

is written in Haskell as:

```
omega_prime_CAction (CSPExtChoice ca cb)
  = make_get_com lsx (CSPExtChoice (omega_prime_CAction ca) (omega_prime_CAction cb))
   where
    lsx = concat (map get_ZVar_st (remdups (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_prime_CAction (CSPNSParal ns1 cs ns2 a1 a2)
  = make_get_com lsx (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 = concat (map get_ZVar_st ((free_var_CAction a1) `union` (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_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_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))
```

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

is written in Haskell as:

```
omega_prime_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_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 [Z
  = 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,[])]) (omega_prime_CAction (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,[])])
          (omega_prime_CAction act))
```

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

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

$$\Omega'_A(\mathbf{if}\ g(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow A\ \mathbf{fi}) \mathrel{\widehat=} \mathbf{if}\ g(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow \Omega'_A(A)\ \mathbf{fi}$$

```
omega_prime_CAction (CActionCommand (CIf (CircGAction g a)))
  = (CActionCommand (CIf (CircGAction g (omega_prime_CAction a))))
```

$$
\Omega'_A\left(\begin{array}{l}\mathbf{if}\ g_0(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow A_0\\ \qquad \square \dots\\ \qquad \square\, g_n(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow A_n\\ \mathbf{fi}\end{array}\right) \mathrel{\widehat=}
\begin{array}{l}\mathbf{if}\ g_0(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow \Omega'_A(A_0)\\ \qquad \square \dots\\ \qquad \square\, g_n(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow \Omega'_A(A_n)\\ \mathbf{fi}\end{array}
$$

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

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

is written in Haskell as:

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

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

is written in Haskell as:

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

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

is written in Haskell as:

```haskell
omega_prime_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_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))
```

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

```haskell
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))
```

```
omega_prime_CAction (CActionName n)
  = (CActionName n)
```

```
omega_prime_CAction x
  = error ("Not defined for Omega'"++ show x)
```

3.3 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 $omega_C Action$ 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 $omega_C Action$ 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) )
```

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

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)
```

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

```
get_guard_pair :: CGActions -> [(ZPred, CAction)]
get_guard_pair (CircThenElse (CircGAction g2 a2) (CircGAction g3 a3))
  = [(g2,a2),(g3,a3)]
get_guard_pair (CircThenElse (CircGAction g1 a1) glx)
  = [(g1,a1)]++(get_guard_pair glx)
```

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

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",[]
```

4 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)

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

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

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

4.2 Bits for FreeVariables (FV(X))

4.3 Free Variables – $FV(A)$.

Need to know how to calculate for Actions.

```
getFV xs = []
```

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

```
free_var_ZExpr :: ZExpr -> [ZVar]

free_var_ZExpr(ZVar v)
 = [v]
free_var_ZExpr(ZInt c )
 = []
free_var_ZExpr(ZGiven a)
    = error "Don't know what free vars of ZGiven are right now. Check back later"

free_var_ZExpr(ZFree0 a)
    = error "Don't know what free vars of ZFree0 are right now. Check back later"

free_var_ZExpr(ZFree1 v ex)
    = error "Don't know what free vars of ZFree1 are right now. Check back later"
free_var_ZExpr(ZTuple exls )
 = fvs free_var_ZExpr exls
free_var_ZExpr(ZBinding a)
    = error "Don't know what free vars of ZBinding are right now. Check back later"
free_var_ZExpr(ZSetDisplay exls )
 = fvs free_var_ZExpr exls
free_var_ZExpr(ZSeqDisplay exls )
 = fvs free_var_ZExpr exls
free_var_ZExpr(ZFSet fs )
    = error "Don't know what free vars of ZFSet are right now. Check back later"
free_var_ZExpr(ZIntSet zi1 zi2)
```

```
     = error "Don't know what free vars of ZIntSet are right now. Check back later"
free_var_ZExpr(ZGenerator rl ex)
     = error "Don't know what free vars of ZGenerator are right now. Check back later"
free_var_ZExpr(ZCross exls )
 = fvs free_var_ZExpr exls
free_var_ZExpr(ZFreeType zv zbls)
     = error "Don't know what free vars of ZFreeType are right now. Check back later"
free_var_ZExpr(ZPowerSet{baseset=b, is_non_empty=e, is_finite=fs})
     = error "Don't know what free vars of ZPowerSet are right now. Check back later"
free_var_ZExpr(ZFuncSet{domset=d, ranset=r, is_function=f, is_total=t, is_onto=o, is_one2one=oo, is
     = error "Don't know what free vars of ZFree0 are right now. Check back later"
free_var_ZExpr(ZSetComp gfls ma)
     = error "Don't know what free vars of ZSetComp are right now. Check back later"
free_var_ZExpr(ZLambda [Choose v e] a)
 = (setminus (free_var_ZExpr a) [v])
free_var_ZExpr(ZLambda _ a)
 = []
free_var_ZExpr(ZESchema a)
     = error "Don't know what free vars of ZESchema are right now. Check back later"
free_var_ZExpr(ZGivenSet a)
     = error "Don't know what free vars of ZGivenSet are right now. Check back later"
free_var_ZExpr(ZUniverse)
     = error "Don't know what free vars of ZUniverse are right now. Check back later"
free_var_ZExpr(ZCall ex ex2)
 = free_var_ZExpr ex2 -- is this right??
free_var_ZExpr(ZReln rl )
     = error "Don't know what free vars of ZReln are right now. Check back later"
free_var_ZExpr(ZFunc1 a)
     = error "Don't know what free vars of ZFunc1 are right now. Check back later"
free_var_ZExpr(ZFunc2 a)
     = error "Don't know what free vars of ZFunc2 are right now. Check back later"
free_var_ZExpr(ZStrange zs )
     = error "Don't know what free vars of ZStrange are right now. Check back later"
free_var_ZExpr(ZMu zgls mex)
     = error "Don't know what free vars of ZMu are right now. Check back later"
free_var_ZExpr(ZELet ves pr)
     =  (setminus (free_var_ZExpr(pr)) (map fst ves)) ++ fvs free_var_ZExpr (map snd ves)
free_var_ZExpr(ZIf_Then_Else zp ex ex1)
     = error "Don't know what free vars of ZIf_Then_Else are right now. Check back later"
-- free_var_ZExpr(ZIf_Then_Else zp ex ex1)
 -- = free_var_ZPred zp ++ free_var_ZExpr ex ++ free_var_ZExpr ex1
free_var_ZExpr(ZSelect ex zv)
     = free_var_ZExpr ex ++ [zv]
free_var_ZExpr(ZTheta zsx)
     = error "Don't know what free vars of ZTheta are right now. Check back later"

free_var_ZPred :: ZPred -> [ZVar]
free_var_ZPred (ZFalse{reason=p})
     = error "Don't know what free vars of ZFalse are right now. Check back later"
free_var_ZPred (ZTrue{reason=p})
     = error "Don't know what free vars of ZTrue are right now. Check back later"
free_var_ZPred (ZAnd a b)
 = free_var_ZPred a ++ free_var_ZPred b
free_var_ZPred (ZOr a b)
 = free_var_ZPred a ++ free_var_ZPred b
free_var_ZPred (ZImplies a b)
 = free_var_ZPred a ++ free_var_ZPred b
free_var_ZPred (ZIff a b)
 = free_var_ZPred a ++ free_var_ZPred b
free_var_ZPred (ZNot a)
 = free_var_ZPred a
free_var_ZPred (ZExists [Choose v e] a)
 = (setminus (free_var_ZPred a) [v])
free_var_ZPred (ZExists ls a)
     = error "Don't know what free vars of ZExists are right now. Check back later"
free_var_ZPred (ZExists_1 [Choose v e] a)
 = (setminus (free_var_ZPred a) [v])
free_var_ZPred (ZExists_1 ls a)
     = error "Don't know what free vars of ZExists_1 are right now. Check back later"
free_var_ZPred (ZForall [Choose v e] a)
```

```
 = (setminus (free_var_ZPred a) [v])
free_var_ZPred (ZForall ls a)
    = error "Don't know what free vars of ZForall are right now. Check back later"
free_var_ZPred (ZPLet ls a )
     = error "Don't know what free vars of ZPLet are right now. Check back later"
free_var_ZPred (ZEqual expa expb)
 = free_var_ZExpr expa ++ free_var_ZExpr expb
free_var_ZPred (ZMember expa expb)
    = free_var_ZExpr expa
free_var_ZPred (ZPre zsexpr)
    = error "Don't know what free vars of ZPre are right now. Check back later"
free_var_ZPred (ZPSchema zsexpr)
    = error "Don't know what free vars of ZPSchema are right now. Check back later"
```

```
fvs f [] = []
fvs f (e:es)
 = f(e) ++ (fvs f (es))
```

4.4 Others – No specific topic

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

```
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)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt lst))
free_var_CAction (CSPRepSeq lst c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt lst))
free_var_CAction (CSPRepExtChoice lst c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt lst))
```

```
free_var_CAction (CSPRepIntChoice lst c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt lst))
free_var_CAction (CSPRepParalNS cs lst ns c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt lst))
free_var_CAction (CSPRepParal cs lst c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt lst))
free_var_CAction (CSPRepInterlNS lst ns c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt lst))
free_var_CAction (CSPRepInterl lst c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt lst))
```

```
free_var_comnd (CAssign v e)
 = v
free_var_comnd (CIf ga)
 = free_var_if ga
free_var_comnd (CVarDecl z c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt z))
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 z c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt z))
free_var_comnd (CResDecl z c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt z))
free_var_comnd (CVResDecl z c)
 =  (setminus (free_var_CAction c) (fvs free_var_ZGenFilt z))
```

```
free_var_ZGenFilt (Include s) = []
free_var_ZGenFilt (Choose v e) = [v]
free_var_ZGenFilt (Check p) = []
free_var_ZGenFilt (Evaluate v e1 e2) = []
```

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

4.5 Expanding the main action

```
get_main_action :: [PPar] -> CAction -> CAction
get_main_action lst (CActionSchemaExpr x)
 = (CActionSchemaExpr x)
get_main_action lst (CActionCommand c)
 = (CActionCommand (get_main_action_comnd lst c))
get_main_action lst (CActionName nm)
 = get_action nm lst
get_main_action lst (CSPSkip)
 = (CSPSkip)
get_main_action lst (CSPStop)
 = (CSPStop)
get_main_action lst (CSPChaos)
 = (CSPChaos)
get_main_action lst (CSPCommAction com c)
 = (CSPCommAction com (get_main_action lst c))
```

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

```
get_main_action_comnd lst (CAssign v e)
 = (CAssign v e)
get_main_action_comnd lst (CIf ga)
 = (CIf (get_if lst ga))
get_main_action_comnd lst (CVarDecl z a)
 = (CVarDecl z (get_main_action lst a))
get_main_action_comnd lst (CAssumpt n p1 p2)
 = (CAssumpt n p1 p2)
get_main_action_comnd lst (CAssumpt1 n p)
 = (CAssumpt1 n p)
get_main_action_comnd lst (CPrefix p1 p2)
 = (CPrefix p1 p2)
get_main_action_comnd lst (CPrefix1 p)
 = (CPrefix1 p)
get_main_action_comnd lst (CommandBrace p)
 = (CommandBrace p)
get_main_action_comnd lst (CommandBracket p)
 = (CommandBracket p)
get_main_action_comnd lst (CValDecl z a)
 = (CValDecl z (get_main_action lst a))
get_main_action_comnd lst (CResDecl z a)
 = (CResDecl z (get_main_action lst a))
get_main_action_comnd lst (CVResDecl z a)
 = (CVResDecl z (get_main_action lst a))
```

```
get_if lst (CircGAction p a)
```

```
 = (CircGAction p (get_main_action lst a))
get_if lst (CircThenElse ga gb)
 = (CircThenElse (get_if lst ga) (get_if lst gb))
get_if lst (CircElse (CircusAction a))
 = (CircElse (CircusAction (get_main_action lst a)))
get_if lst (CircElse (ParamActionDecl x (CircusAction a)))
 = (CircElse (ParamActionDecl x (CircusAction (get_main_action lst a))))
```

```
get_action _ [] = error "Action list is empty"
get_action name [(CParAction n (CircusAction a))]
 = case name == n of
   True -> a
   False -> error "Action not found"

get_action name ((CParAction n (CircusAction a)):xs)
 = case name == n of
   True -> a
   False -> get_action name xs
```

```
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)
```

```
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)
```

```
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
```

```
is_st_var ('s':'t':'_':'v':'a':'r':'_':xs) = True
is_st_var _ = False
```

```
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_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_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)
```

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

4.5.1 rename vars

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

```
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 (CSPSkip )
 = (CSPSkip )
rename_vars_CAction (CSPStop )
 = (CSPStop )
rename_vars_CAction (CSPChaos)
 = (CSPChaos)
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 (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 (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_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_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_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 cga1 cga2)
 = (CircThenElse (rename_vars_CGActions cga1) (rename_vars_CGActions cga2))
rename_vars_CGActions (CircElse pa)
 = (CircElse pa)
```

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

4.5.2 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))
```

4.5.3 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_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 zvarls1 zexprls)
 = (CAssign (map (rename_vars_ZVar1 lst) zvarls1) (map (rename_vars_ZExpr1 lst) zexprls))
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 cga1 cga2)
 = (CircThenElse (rename_vars_CGActions1 lst cga1) (rename_vars_CGActions1 lst cga2))
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
```

```
rename_ZGenFilt1 lst (Include s) = (Include s)
rename_ZGenFilt1 lst (Choose (va,x) e) = (Choose ((join_name (join_name "st_var" (get_proc_name va
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 "st_var" (get_proc_name va lst)) va),x)
  _ -> (va,x)
```

```
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 "st_var" (get_proc_name va lst)) va),x))
   _ -> (ZVar (va,x))
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_names [(ZFreeTypeDef ("NAME",[]) xs)]
  = get_delta_names_aux xs
get_delta_names ((ZFreeTypeDef ("NAME",[]) xs):xss)
  = (get_delta_names_aux xs)++(get_delta_names xss)
get_delta_names (_:xs)
  = (get_delta_names xs)
get_delta_names []
  = []
```

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

get State variables from names

```
get_ZVar_st ((('s':'t':'_':'v':'a':'r':'_':xs),x))
 = [('s':'t':'_':'v':'a':'r':'_':xs)]
get_ZVar_st x
 = []
```

```
get_vars_ZExpr :: ZExpr -> [ZName]
get_vars_ZExpr (ZVar (('s':'t':'_':'v':'a':'r':'_':xs),x))
 = [('s':'t':'_':'v':'a':'r':'_':xs)]
get_vars_ZExpr (ZFree1 va xpr)
 = (get_vars_ZExpr xpr)
get_vars_ZExpr (ZTuple xpr)
 = concat (map get_vars_ZExpr xpr)
get_vars_ZExpr (ZBinding xs)
 = (get_bindings_var xs)
get_vars_ZExpr (ZSetDisplay xpr)
 = concat (map get_vars_ZExpr xpr)
get_vars_ZExpr (ZSeqDisplay xpr)
 = concat (map get_vars_ZExpr xpr)
get_vars_ZExpr (ZGenerator zrl xpr)
 = (get_vars_ZExpr xpr)
get_vars_ZExpr (ZCross xpr)
```

```
 = concat (map get_vars_ZExpr xpr)
get_vars_ZExpr (ZPowerSet{baseset=xpr, is_non_empty=b1, is_finite=b2})
 = (get_vars_ZExpr xpr)
get_vars_ZExpr (ZFuncSet{ domset=expr1, ranset=expr2, is_function=b1, is_total=b2, is_onto=b3, is_o
 = (get_vars_ZExpr expr1)++(get_vars_ZExpr expr2)
get_vars_ZExpr (ZSetComp pname1 (Just xpr))
 = (get_vars_ZExpr xpr)
get_vars_ZExpr (ZLambda pname1 xpr)
 = (get_vars_ZExpr xpr)
get_vars_ZExpr (ZCall xpr1 xpr2)
 = (get_vars_ZExpr xpr1) ++(get_vars_ZExpr xpr2)
get_vars_ZExpr (ZMu pname1 (Just xpr))
 = (get_vars_ZExpr xpr)
get_vars_ZExpr (ZELet pname1 xpr1)
 = (get_bindings_var pname1)++(get_vars_ZExpr xpr1)
get_vars_ZExpr (ZIf_Then_Else zp xpr1 xpr2)
 = (get_vars_ZExpr xpr1)++(get_vars_ZExpr xpr2)
get_vars_ZExpr _ = []
```

```
get_vars_ZPred (ZAnd p1 p2)
 = ((get_vars_ZPred p1)++(get_vars_ZPred p2))
get_vars_ZPred (ZOr p1 p2)
 = ((get_vars_ZPred p1)++(get_vars_ZPred p2))
get_vars_ZPred (ZImplies p1 p2)
 = ((get_vars_ZPred p1)++(get_vars_ZPred p2))
get_vars_ZPred (ZIff p1 p2)
 = ((get_vars_ZPred p1)++(get_vars_ZPred p2))
get_vars_ZPred (ZNot p)
 = ((get_vars_ZPred p))
get_vars_ZPred (ZEqual xpr1 xpr2)
 = ( (get_vars_ZExpr xpr1)++(get_vars_ZExpr xpr2))
get_vars_ZPred (ZMember xpr1 xpr2)
 = ((get_vars_ZExpr xpr1)++(get_vars_ZExpr xpr2))
get_vars_ZPred _
 = []
```

Construction of the Universe set in CSP

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

-- def_U_NAME x = ("U_"++Data.Text.unpack(Data.Text.toUpper(Data.Text.take 3 (pack x))))
-- def_U_prefix x = (Data.Text.unpack(Data.Text.toTitle(Data.Text.take 3 (Data.Text.pack x))))

mk_universe []
  = ""
mk_universe [(a,b,c,d)]
  = c++"."++d
mk_universe ((a,b,c,d):xs)
  = c++"."++d++" | "++(mk_universe xs)

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)

mk_value []
  = ""
mk_value [(a,b,c,d)]
  = "value("++c++".v) = v\n"
mk_value ((a,b,c,d):xs)
  = "value("++c++".v) = v\n"++(mk_value xs)

mk_type []
  = ""
mk_type [(a,b,c,d)]
  = a++" then "++b
mk_type ((a,b,c,d):xs)
```

```
  = a++" then "++b++"\n\t else if x == "++(mk_type xs)

mk_tag []
  = ""
mk_tag [(a,b,c,d)]
  = a++" then "++c
mk_tag ((a,b,c,d):xs)
  = a++" then "++c++"\n\t else if x == "++(mk_tag xs)
```

```
-- 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 []
  = []
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_NAM
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_
```

```
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))
```

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


-- 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
```