From CCS(-ish) to CSP(-ish)

Andrew Butterfield
(based on material by Gerard Ekembe Ngondi)
© 2020

September 25, 2020

Contents

Chapter 1

Application

This program mechanises the CCS-CSP translations being explored by Gerard Ekembe Ngondi in his work as a Marie-Curie ALECS Fellow at Trinity College Dublin.

Key references:

- [GEN] Working document by Gerard Ekembe Ngondi.
- [CC] R. Milner, "Communication and Concurrency".

1.1 Main Program

```
Copyright  Andrew Buttefield (c) 2017--18

LICENSE: BSD3, see file LICENSE at reasonEq root
```

```
module Main where

import Syntax
import Examples
import Semantics
import Translate

import Debug.Trace
dbg msg x = trace (msg++show x) x
pdbg nm x = dbg ('@':nm++":\n") x
```

1.1.1 Version

```
progName = "ccs2csp"
version = "0.0.1.0"
name_version = progName++" "++version
```

1.1.2 Mainline

```
main :: IO ()
main
  = do putStrLn name_version
       putStrLn "Nothing to see here yet."
       putStrLn "Suggest you use ghci"
       putStrLn "stack ghci src/Examples.lhs"
       putStrLn "\n, Goodbye."
```

Chapter 2

Libraries

2.1 Syntax

```
Copyright  Andrew Buttefield (c) 2017

LICENSE: BSD3, see file LICENSE at reasonEq root
```

```
module Syntax where

import Data.Set (Set)
import qualified Data.Set as S

--import Debug.Trace
--dbg msg x = trace (msg++show x) x
```

We making barring part of a name:

```
data Name = Std String | Bar String deriving (Eq,Ord,Read)

instance Show Name where
  show (Std s)  =  s
  show (Bar s)  =  s ++ "-bar"

bar :: Name -> Name
bar (Std s)  =  Bar s
bar (Bar s)  =  Std s
```

```
data Index
  = None
  | One Int
  | Two Int Int
  deriving (Eq,Ord,Read)

instance Show Index where
  show None       =  ""
  show (One i)    =  show i
  show (Two i j)  =  show (One i) ++ ";" ++ show (One j)
```

```
type Event = (Name, Index)

event :: Name -> Event
event nm = (nm,None)

ievent :: Name -> Int -> Event
ievent nm i = (nm,One i)

i2event :: Name -> Int -> Int -> Event
i2event nm i j -- reorder indices so first <= second
  | i > j       =  (nm,Two j i)
  | otherwise   =  (nm,Two i j)

showEvent :: Event -> String
showEvent (nm,i) = show nm ++ show i

evtbar :: Event -> Event
evtbar (nm,i) = (bar nm,i)
```

```
data Prefix
  = T             -- tau
  | Evt Event     -- a or a-bar
  | T' String     -- t[a|a-bar]
```

```
  deriving (Eq,Ord,Read)

instance Show Prefix where
  show T = "t"
  show (Evt (n,i)) = show n ++ show i
  show (T' n) = show T ++ "["++n++"|"++n++"-bar]"

pfxbar :: Prefix -> Prefix
pfxbar (Evt e)  =  Evt $ evtbar e
pfxbar pfx      =  pfx
```

```
type RenFun = [(String,String)]
```

$$P, Q \quad ::= \quad 0 \mid \alpha.P \mid P|Q \mid P + Q \mid P \setminus L \mid P[f] \mid X \mid \mu X \bullet P$$

```
data CCS
  = Zero
  | Pfx Prefix CCS
  | Sum [CCS]
  | Par [CCS]
  | Rstr [Event] CCS
  | Ren RenFun CCS
  | PVar String
  | Rec String CCS
  deriving (Eq,Ord,Read)

-- f s2s Zero
-- f s2s (Pfx pfx ccs)
-- f s2s (Sum ccss)
-- f s2s (Par ccss)
-- f s2s (Rstr es ccs)
-- f s2s (Ren s2s ccs)
-- f s2s (PVar s)
-- f s2s (Rec s ccs)
-- f s2s ccs
```

```
instance Show CCS where
  showsPrec p Zero  = showString "0"
  showsPrec p (Pfx pfx Zero) = showString $ show pfx
  showsPrec p (Pfx pfx ccs)
    = showParen (p > pPfx) $
        showString (show pfx) .
        showString "." .
        showsPrec pPfx ccs
  showsPrec p (Sum []) = showsPrec p Zero
  showsPrec p (Sum [ccs]) = showsPrec p ccs
  showsPrec p (Sum (ccs:ccss))
    = showParen (p > pSum) $
        showsPrec pSum' ccs .
        showSum pSum' ccss
  showsPrec p (Par []) = showsPrec p Zero
  showsPrec p (Par [ccs]) = showsPrec p ccs
  showsPrec p (Par (ccs:ccss))
    = showParen (p > pPar) $
        showsPrec pPar' ccs .
        showPar pPar' ccss
  showsPrec p (Rstr es ccs)
    = showParen (p > pRstr) $
```

```
        showsPrec pRstr' ccs .
        showString "\\" .
        showEvents es
  showsPrec p (Ren s2s ccs)
    = showParen (p > pRen) $
        showsPrec pRen' ccs .
        showString "[" .
        showRenFun s2s .
        showString "]"
  showsPrec p (PVar nm) = showString nm
  showsPrec p (Rec nm ccs)
    = showParen True $
        showString "mu " .
        showString nm .
        showString " @ " .
        showsPrec 0 ccs
```

```
-- Comm+Conc, p44
-- tightest: {Ren,Rstr}, Pfx, Par, Sum :loosest

pSum = 2; pSum' = pSum+1
pPar = 4; pPar' = pPar+1
pPfx = 6; pPfx' = pPfx+1
pRen = 8; pRen' = pRen+1
pRstr = pRen; pRstr' = pRstr+1

showSum p [] = id
showSum p (ccs:ccss)
  = showString " + " .
    showsPrec p ccs .
    showSum p ccss

showPar p [] = id
showPar p (ccs:ccss)
  = showString " | " .
    showsPrec p ccs .
    showPar p ccss

showEvents [] = id
showEvents [e] = showString $ showEvent e
showEvents (e:es)
  = showString "{" .
    showString (showEvent e) .
    showString "," .
    showEvents' es .
    showString "}"

showEvents' [] = id
showEvents' [e] = showString $ showEvent e
showEvents' (e:es)
  = showString (showEvent e) .
    showString "," .
    showEvents' es

showRenFun [] = showString ""
showRenFun [ee] = showEE ee
showRenFun (ee:ees)
  = showEE ee .
    showString "," .
    showRenFun ees
```

```
showEE (e1,e2) = showString e1 .
                 showString "/" .
                 showString e2
```

Smart Builders:

```
csum :: [CCS] -> CCS
csum [] = Zero
csum [ccs] = ccs
csum ccss = Sum ccss

cpar :: [CCS] -> CCS
cpar [] = Zero
cpar [ccs] = ccs
cpar ccss = Par ccss


rstr :: [Event] -> CCS -> CCS
rstr [] ccs = ccs
rstr es ccs = Rstr es ccs

endo :: Eq a => [(a,a)] -> a -> a
endo [] a = a
endo ((a1,a2):as) a
 | a == a1     =  a2
 | otherwise   =  endo as a
```

Summaries:

```
prefixesOf :: CCS -> Set Prefix
prefixesOf (Pfx pfx ccs)   =  S.singleton pfx `S.union` prefixesOf ccs
prefixesOf (Sum ccss)      =  S.unions $ map prefixesOf $ ccss
prefixesOf (Par ccss)      =  S.unions $ map prefixesOf $ ccss
prefixesOf (Rstr ss ccs)   =  prefixesOf ccs
prefixesOf (Ren s2s ccs)   =  prefixesOf $ doRename (endo s2s) ccs
prefixesOf (Rec s ccs)     =  prefixesOf ccs
prefixesOf _               =  S.empty
```

Actions:

```
doRename :: (String -> String) -> CCS -> CCS
doRename s2s (Pfx pfx ccs)   =  Pfx (renPfx s2s pfx) $ doRename s2s ccs
doRename s2s (Sum ccss)      =  Sum $ map (doRename s2s) ccss
doRename s2s (Par ccss)      =  Par $ map (doRename s2s) ccss
doRename s2s (Rstr es ccs)   =  Rstr (map (renEvent s2s) es) $ doRename s2s ccs
doRename s2s (Ren s2s' ccs)  =  doRename s2s (doRename (endo s2s') ccs)
doRename s2s (Rec s ccs)     =  Rec s $ doRename s2s ccs
doRename _   ccs             = ccs

renPfx :: (String -> String) -> Prefix -> Prefix
renPfx _ T           =  T
renPfx s2s (T' s)    =  T' $ s2s s
renPfx s2s (Evt e)   =  Evt $ renEvent s2s e

renEvent :: (String -> String) -> Event -> Event
renEvent s2s (nm,i)   =  (renName s2s nm,i)

renName :: (String -> String) -> Name -> Name
renName s2s (Std nm)   =  Std $ s2s nm
renName s2s (Bar nm)   =  Bar $ s2s nm
```

2.2 Translate

```
Copyright  Andrew Butterfield (c) 2017

LICENSE: BSD3, see file LICENSE at reasonEq root
```

```
module Translate where
import Data.Maybe
import Data.Set (Set)
import qualified Data.Set as S
import Data.Map (Map)
import qualified Data.Map as M
import Syntax

--import Debug.Trace
--dbg msg x = trace (msg++show x) x
```

2.2.1 Control

This generic control code belongs in a distinct module.

```
paramwalk :: (i -> i) -> (i -> a -> b) -> i -> [a] -> [b]
paramwalk _ _ _ [] = []
paramwalk upd f i (a:as) = f i a : paramwalk upd f (upd i) as

paramileave :: (i -> a -> (b,i)) -> i -> [a] -> ([b],i)
paramileave _ i [] = ([],i)
paramileave f i (a:as)
  = let (a',i1)   =  f i a
        (as',i')  =  paramileave f i1 as
    in (a':as',i')
```

2.2.2 Pre-Indexing

Here we attach single indices to every standard or barred event, numbered from 0 upwards. Currently we fail if tagged-taus are found.

```
indexNames :: CCS -> CCS
indexNames = fst . iFrom 0

iFrom i (Pfx  pfx ccs) = (Pfx (iPfx i pfx) ccs',i')
  where (ccs',i') = iFrom (i+1) ccs
iFrom i (Sum ccss) = (Sum ccss',i')
  where (ccss',i') = paramileave iFrom i ccss
iFrom i (Par ccss) = (Par ccss',i')
  where (ccss',i') = paramileave iFrom i ccss
iFrom i (Rstr es ccs) = (Rstr es ccs',i')
  where (ccs',i') = iFrom i ccs
iFrom i (Ren pfn ccs) = (Ren pfn ccs',i')
  where (ccs',i') = iFrom i ccs
iFrom i (Rec nm ccs) = (Rec nm ccs',i')
  where (ccs',i') = iFrom i ccs
iFrom i ccs = (ccs,i)

iPfx :: Int -> Prefix -> Prefix
iPfx i T = T
iPfx i (Evt e) = Evt (ePfx i e)
iPfx i pfx@(T' _) = error ("pre-indexing CCS term with tagged-tau "++show pfx)
```

```
ePfx :: Int -> Event -> Event
ePfx i (nm,_) = (nm,One i)
```

Given a CCS term, return a mapping from events to the set of indices associated with each event.

```
type IxMap = Map Name (Set Index)
indexMap :: CCS -> IxMap
indexMap = iMap M.empty

iMap imap (Pfx (Evt (nm,i)) ccs)  =  iMap imap' ccs
                       where imap'  =  insMapping nm i imap
iMap imap (Pfx  _ ccs)            =  iMap imap ccs
iMap imap (Sum ccss)              =  iSeqMap imap ccss
iMap imap (Par ccss)              =  iSeqMap imap ccss
iMap imap (Rstr es ccs)           =  iMap imap ccs
iMap imap (Ren _ ccs)             =  iMap imap ccs
iMap imap (Rec nm ccs)            =  iMap imap ccs
iMap imap ccs                     =  imap

iSeqMap imap []          =  imap
iSeqMap imap (ccs:ccss)  =  let imap' = iMap imap ccs in iSeqMap imap' ccss

insMapping :: Name -> Index -> IxMap -> IxMap
insMapping nm i imap
  = M.insertWith S.union nm (S.singleton i) imap
```

2.2.3 Indexing with g^*

Based on [GEN] v17, Note 1, 23rd Sep. 2020.

Set of Indexes

Here we assume the following minor corrections to Def 2:

$$
\begin{aligned}
iXsucc(P_l) &\mathrel{\widehat{=}} allixof(P_l) \setminus ixof(P_l) \\
&\vdots \\
allixof(a_{l1}.P_{l2}) &\mathrel{\widehat{=}} \{l1\} \cup allixof(P_{l2}) \\
&\vdots \\
allixof(P_{l1}|_{ccsT}P_{l2}) &\mathrel{\widehat{=}} allixof(P_{l1}) \cup allixof(P_{l2})
\end{aligned}
$$

However, this would appear to be redundant - $ixSucc$ is easy to compute

Using g^* for Processes

```
gsp :: context -> CCS -> (CCS,context)
gsp ctxt ccs = error "g*(proc) not yet defined"
```

Using g^* for Actions

Note that this function is called in the context of a CCS parallel of the form:

$$P_1|P_2|\dots|P_k$$

where none of the P_i are themselves a parallel composition.

```
gsa :: context -> Prefix -> ([Prefix],context)
gsa ctxt T = ([T],ctxt)
gsa ctxt t'@(T' _) = ([t'],ctxt)
gsa ctxt (Evt e) =  error "g*(act) not yet defined for std event"
```

2.2.4 Translate toward CSP

This is based on whiteboard notes by Vasileios Koutavas, on MS Teams, on 24th Sep 2020.

We use $\Sigma_i a_i.P$ as shorthand for $\Sigma_i(a_i.p)$, and we consider a_{ij}, a_{ji} to be the same, with $i \neq j$. We also use α to range over $a, b, c, \ldots$ and $\bar{a}, \bar{b}, \bar{c}, \ldots$.

$$
\begin{array}{rcl}
pre{-}g_T(P) & = & namesOf(P) \subseteq dom(T) \\
g_T(0) & \widehat{=} & 0 \\
g_T(\alpha_i.P) & \widehat{=} & (\alpha_i + \Sigma_{j \in T(\bar{\alpha})}\alpha_{ij}).g_T(P) \\
g_T(P|Q) & \widehat{=} & (g_T(P)|g_T(Q)) \setminus \{\alpha_{ij} \mid \alpha_i \in P, \bar{\alpha}_j \in Q\} \\
g_T(P+Q) & \widehat{=} & (g_T(P) + g_T(Q)) \\
g_T(P \setminus L) & \widehat{=} & g_T(P) \setminus g'_T(L) \quad \text{can this be the identity?} \\
g_T(P[f]) & \widehat{=} & g_T(P)[f] \\
g_T(X) & \widehat{=} & X \\
g_T(\mu X \bullet P) & \widehat{=} & \mu X \bullet g_T(P)
\end{array}
$$

```
ccs2star :: CCS -> CCS
ccs2star ccs
 = c2star imap iccs
 where iccs = indexNames ccs
       imap = indexMap iccs

c2star :: IxMap -> CCS -> CCS

c2star imap (Pfx (Evt (alfa,(One i))) ccs)
  = sumPrefixes imap alfa i $ c2star imap ccs

c2star imap (Par ccss)
  = rstr (syncPre $ map (S.toList . prefixesOf) ccss)
         $ Par $ map (c2star imap) ccss

c2star imap (Sum ccss) = Sum $ map (c2star imap) ccss

c2star imap (Rstr es ccs) = Rstr es $ c2star imap ccs -- ? f es

c2star imap (Ren f ccs) = Ren f $ c2star imap ccs

c2star imap (Rec x ccs) = Rec x $ c2star imap ccs

c2star imap ccs = ccs -- 0, X
```

$$
g_T(\alpha_i.P) \quad \widehat{=} \quad (\alpha_i + \Sigma_{j \in T(\bar{\alpha})}\alpha_{ij}).g_T(P)
$$

```
sumPrefixes :: IxMap -> Name -> Int -> CCS -> CCS
sumPrefixes imap alfa i ccs
```

```
  = case M.lookup (bar alfa) imap of
      Nothing  ->  Pfx (Evt (alfa,One i)) ccs
      Just evts
        -> let alpha2s = map (mkSyncEvt alfa i) $ S.toList evts
              in Sum $ map (affix ccs) ((alfa,One i):alpha2s)

mkSyncEvt alfa i (One j) = i2event alfa i j
affix ccs e = Pfx (Evt e) ccs
```

$$
\begin{array}{rcl}
g_T(P|Q) & \widehat{=} & (g_T(P)|g_T(Q)) \setminus \{\alpha_{ij} \mid \alpha_i \in P, \bar{\alpha}_j \in Q\} \\
g_T(\Pi_p P_p) & \widehat{=} & (\Pi_p g_T(P_p)) \setminus \{\alpha_{ij} \mid \alpha_i \in P_m, \bar{\alpha}_j \in P_n, m \neq n\}
\end{array}
$$

```
syncPre :: [[Prefix]] -> [Event]
syncPre = concat . findSync syncPre1

--
syncPre1 :: [Prefix] -> [Prefix] -> [[Event]]
syncPre1 ps1 ps2 = map (f ps1) ps2
 where f ps1 p2 = concat $ map (syncPre2 p2) ps1
```

```
syncPre2 :: Prefix -> Prefix -> [Event]
syncPre2 (Evt (Std m,One i)) (Evt (Bar n,One j)) | m == n  =  syncE m i j
syncPre2 (Evt (Bar m,One i)) (Evt (Std n,One j)) | m == n  =  syncE m i j
syncPre2 _                   _                             =  []

syncE s i j = [i2event (Std s) i j]
```

We need the following

```
findSync :: (a -> a -> [b]) -> [a] -> [b]
findSync op [] = []
findSync op [_] = []
findSync op (as:ass) = concat (map (op as) ass) ++ findSync op ass
```

2.3 Semantics

Copyright Andrew Buttefield (c) 2017

LICENSE: BSD3, see file LICENSE at reasonEq root

```
module Semantics where

import Control.Monad
import Syntax
--import Debug.Trace
--dbg msg x = trace (msg++show x) x
```

2.3.1 Operational Semantics

From [CC],p46

$$\frac{}{\alpha.E \xrightarrow{\alpha} E}\,Act \qquad \frac{j \in I \qquad E_j \xrightarrow{\alpha} E'_j}{\Sigma_{i\in I} E_i \xrightarrow{\alpha} E'_j}\,Sum_j \qquad \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F}\,Com_1 \qquad \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}\,Com_2$$

$$\frac{E \xrightarrow{\ell} E' \qquad F \xrightarrow{\bar{\ell}} F'}{E|F \xrightarrow{\tau} E'|F'}\,Com_3 \qquad \frac{\alpha, \bar{\alpha} \notin L \qquad E \xrightarrow{\alpha} E'}{E \setminus L \xrightarrow{\alpha} E' \setminus L}\,Res \qquad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}\,Rel$$

$$\frac{A \mathrel{\widehat{=}} P \qquad P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\,Con$$

From [GEN], in the CCS-τ variant, we replace Com_3 with

$$\frac{E \xrightarrow{\ell} E' \qquad F \xrightarrow{\bar{\ell}} F'}{E|F \xrightarrow{\tau[\ell|\bar{\ell}]} E'|F'}$$

2.3.2 Equational Laws

From [CC],pp62–80.
Law $E_1 = E_2$ means that, for all α, that $E_1 \stackrel{\alpha}{\Longrightarrow} E'$ iff $E_2 \stackrel{\alpha}{\Longrightarrow} E'$.

```
type LawFun m = CCS -> m CCS
```

Monoid Laws

Proposition 1 ([CC],p62).

$$P + Q = Q + P \tag{2.1}$$
$$P + (Q + R) = (P + Q) + R \tag{2.2}$$
$$P + P = P \tag{2.3}$$
$$P + 0 = P \tag{2.4}$$

```
sumComm, sumAssoc, sumIdem, sumId :: MonadPlus m => LawFun m
sumComm (Sum [p,q]) = return $ Sum [q,p]
sumComm _ = fail "not P+Q"
```

```haskell
sumAssoc (Sum [p,Sum [q,r]]) = return $ Sum [Sum [p,q],r]
sumAssoc _ = fail "not P+(Q+R)"
sumIdem (Sum [p,q]) | p==q  = return p
sumIdem _ = fail "not P+P"
sumId (Sum [p,Zero]) = return p
sumId _ = fail "not P+0"
```

We can normalise sums by flattening and sorting:

```haskell
sumNorm :: MonadPlus m => LawFun m
sumNorm p = return p -- to be implemented
```

τ Laws

Proposition 2 ([CC],p62).

$$\begin{aligned}\alpha.\tau.P &= a.P &(2.5)\\ P+\tau.P &= \tau.P &(2.6)\\ \alpha.(P+\tau.Q)+\alpha.Q &= \alpha.(P+\tau.Q) &(2.7)\end{aligned}$$

```haskell
tauAbsorb, sumTau, sumTau2 :: MonadPlus m => LawFun m
tauAbsorb _ = fail "not a.t.P"
sumTau _ = fail "not P+t.P"
sumTau2 = fail "not a.(P+t.Q)+a.Q"
```

Corrollary 3 ([CC],p63)

$$P+\tau.(P+Q) = \tau.(P+Q) \tag{2.8}$$

2.3.3 Trace Semantics

We can define traces for CCS terms, in a number of ways. One is simply the full set of complete traces, some of which may be infinite. Another has partial traces, all finite, with a prefix-closure healthiness condition.

In the sequel, we play fast and loose, using recursion to define functions over potentially infinite lists. These can all be cast into an appropriate co-recursive form, or simply interpreted as such (which is what Haskell's laziness does by default). We also use cons-notation for lists ($x : \sigma$ is same as $\langle x \rangle \frown \sigma$).

Here is the full trace set version[1] :

$$\begin{array}{rcl} trc & : & CCS \to \mathcal{P}(Event^\omega) \\ trc(0) & \widehat{=} & \{\langle\rangle\} \\ trc(\alpha.P) & \widehat{=} & \{\alpha : \sigma \mid \sigma \in trc(P)\} \\ trc(P+Q) & \widehat{=} & trc(P) \cup trc(Q) \\ trc(P|Q) & \widehat{=} & \{t \mid t \in t_P|t_Q, t_P \in trc(P), t_Q \in trc(Q)\} \\ trc(P \setminus L) & \widehat{=} & \{\sigma \mid \sigma \in trc(P), \sigma \cap L = \emptyset\} \\ trc(P[f]) & \widehat{=} & \{f(\sigma) \mid \sigma \in trc(P)\} \\ trc(\mu X \bullet P) & \widehat{=} & trc(P(X \mapsto \mu X \bullet P) \end{array}$$

Here the interesting function is one on traces: $s|t$ returns all valid interleavings of s and t.

$$\begin{array}{rcl} \_|\_ & : & (Event^\omega)^2 \to \mathcal{P}(Event^\omega) \\ \langle\rangle|t & \widehat{=} & \{t\} \\ t|nil & \widehat{=} & \{t\} \\ (\alpha : t_1)|(\bar{\alpha} : t_2) & \widehat{=} & \{\alpha : (t_1|\bar{\alpha} : t_2), \quad \tau : (t_1|t_2), \quad \bar{\alpha} : (\alpha : t_1|t_2)\} \\ (\alpha : t_1)|(\beta : t_2) & \widehat{=} & \{\alpha : (t_1|\beta : t_2), \quad \beta : (\alpha : t_1|t_2)\} \end{array}$$

Hypothesis The definition given here gives the same results as the usual derivation of trc from the operational semantics.

```
semantics :: String
semantics = "Semantics"
```

[1]This may omit any deadlock traces (see defn. of $P \setminus L$).

Appendix A

Tests

A.1 Examples

```
Copyright  Andrew Buttefield (c) 2017

LICENSE: BSD3, see file LICENSE at reasonEq root
```

```
module Examples where

import Control.Monad
import Syntax
import Translate
import Semantics

--import Debug.Trace
--dbg msg x = trace (msg++show x) x
```

Milners "Comms and Conc" book.

```
  -- p44  R+a.P|b.Q\L = R+((a.P)|(b.(Q\L)))
na = Std "a" ; ea = (na,None);  a = Evt ea
nb = Std "b" ; b = Evt (nb,None)
r = PVar "R"
p = PVar "P"
ell = (Std "L",None)
q = PVar "Q"
cc44 = Sum [ r
           , Par [ Pfx a p
                 , Pfx b (Rstr [ell] q)
                 ]
           ]
```

Examples from Gerard's document, v17.

```
-- v17, 4.1.2, p18
s = PVar "S"
abar = pfxbar a
x18 = Rstr [ea] $ Par [Pfx a p, Pfx abar q, Pfx abar r, Pfx abar s]

--v17, 4.1.2., p19
xl19 = Par [Pfx a Zero, Pfx abar Zero]
ta = T' "a"
a0 = Pfx a Zero; abar0 = Pfx abar Zero
xr19 = Sum [Pfx a $ abar0, Pfx abar $ a0, Pfx ta Zero]

--v17, 4.1.2, p19 bottom
xb19 = Par [ Pfx a (Par [a0,a0,a0,a0])
           , abar0
           , Pfx abar (Par [a0,a0])
           ]
```

Examples from Vasileios MS Team whiteboard, 24th Sep.

```
-- a.b.0 | b-bar.a-bar.0
bbar = pfxbar b
xms1 = Par [ Pfx a (Pfx b Zero), Pfx bbar (Pfx abar Zero)]

-- a.b.(abar.0|b.0) | bbar.abar.0
xms2 = Par [ Pfx a (Pfx b (Par [ Pfx abar Zero, Pfx b Zero]))
           , Pfx bbar (Pfx abar Zero)
           ]
-- manually laid out below -- need better pretty-printing
-- ( (   a0.(   b1.((a-bar2 + a-bar0;2) | (b3 + b3;4))
--             + b1;4.((a-bar2 + a-bar0;2) | (b3 + b3;4))
--             )
--    + a0;2.(   b1.((a-bar2 + a-bar0;2) | (b3 + b3;4))
--             + b1;4.((a-bar2 + a-bar0;2) | (b3 + b3;4))
--             )
--    + a0;5.(   b1.((a-bar2 + a-bar0;2) | (b3 + b3;4))
--             + b1;4.((a-bar2 + a-bar0;2) | (b3 + b3;4))
--             )
--    )
--    |
--    (   b-bar4.(a-bar5 + a-bar0;5)
--    + b-bar1;4.(a-bar5 + a-bar0;5)
--    + b-bar3;4.(a-bar5 + a-bar0;5)
--    )
-- )
-- \{a0;5,b1;4,b3;4}
```

Bibliography