Transition System (a Coalgebra)

```
type TS f s     = s → f s
```

Labelled Transition System

```
type LTS a f s = Map a (TS f s)
```

$\boxed{\text{map is a finite function}}$

```
alphabet = M.keys
```

```
infixl 9 \$
(\$)   :: Map a b → (a → b)
f \$ k = guardFromJust "action not defined" (M.lookup k f)
guardFromJust _   (Just x) = x
guardFromJust err Nothing = error err
```

$$\begin{aligned}
&\mathit{runForgetful} :: (..., \mathit{Crush}\ f, \mathit{Functor}\ f) \Rightarrow \mathit{LTS}\ a\ f\ s \rightarrow [a] \rightarrow s \rightarrow \mathit{Set}\ s\\
&\mathit{runForgetful}\ \delta\ [\,] \qquad\quad = \mathit{singleton}\\
&\mathit{runForgetful}\ \delta\ (a : \mathit{as}) = \mathit{setjoin} \cdot \mathit{toSet} \cdot \mathit{fmap}\ (\mathit{runForgetful}\ \delta\ \mathit{as}) \cdot (\delta \backslash\$\ a)
\end{aligned}$$

$$\begin{aligned}
&\mathit{toSet} :: (..., \mathit{Crush}\ f) \Rightarrow f\ a \rightarrow \mathit{Set}\ a\\
&\mathit{toSet} = \mathit{crush}\ S.\mathit{insert}\ \emptyset
\end{aligned}$$

$$\mathit{setjoin} :: \mathit{Set}\ (\mathit{Set}\ a) \rightarrow \mathit{Set}\ a$$

```
runForgetful :: (..., Crush f, Functor f) ⇒ LTS a f s → [a] → s → Set s
runForgetful δ []       = singleton
runForgetful δ (a : as) = setjoin · toSet · fmap (runForgetful δ as) · (δ \$ a)
```

```
runPreserving :: (..., Functor f) ⇒ LTS a f s → [a] → s → Star f s
runPreserving δ []       = End
runPreserving δ (a : as) = Step · fmap (runPreserving δ as) · (δ \$ a)
```

```
data Star f s = End s | Step (f (Star f s))
```

```
runForgetful :: (..., Crush f, Functor f) ⇒ LTS a f s → [a] → s → Set s
runForgetful δ [ ]      = singleton
runForgetful δ (a : as) = setjoin · toSet · fmap (runForgetful δ as) · (δ \$ a)
```

```
runPreserving :: (..., Functor f) ⇒ LTS a f s → [a] → s → Star f s
runPreserving δ [ ]      = End
runPreserving δ (a : as) = Step · fmap (runPreserving δ as) · (δ \$ a)
```

```
runInMonad :: (..., Functor f, Monad f) ⇒ LTS a f s → [a] → s → f s
runInMonad δ [ ]      = return
runInMonad δ (a : as) = join · fmap (runInMonad δ as) · (δ \$ a)
```

```
bisimilar :: (Eq s, Ord a) ⇒ LTS a f s → s → s → Bool
bisimilar δ p q = runReader (bisim δ p q) [ ]


bisim :: (Eq s, Ord a) ⇒ LTS a f s → s → s → Reader [(s, s)] Bool
bisim δ p q = do stack ← ask
                 if p ≡ q ∨ (p, q) ∈ stack ∨ (q, p) ∈ stack
                   then return True
                   else liftM and $ mapM (bisimBy δ p q) (alphabet δ)
bisimBy :: (Eq s, Ord a) ⇒ LTS a f s → s → s → a → Reader [(s, s)] Bool
bisimBy δ p q a = let p′ = (δ \$ a) p
                      q′ = (δ \$ a) q
                  in local ((p, q):) $
                    liftM (maybe False and) $ fSafeZipWithM (bisim δ) p′ q′


fSafeZipWithM :: (a → b → m c) → f a → f b → m (Maybe (f c))
```

$$T \cong F\ T$$
$$T = \mu T$$

$$T\ A \cong B\ A\ (T A)$$
$$T\ A = \mu(B A)$$

```
class BiFunctor (f :: ∗ → ∗ → ∗) where
  bifmap :: (a → b) → (c → d) → f a c → f b d
```

```
data Unit a r        = Unit
data (f :+: g) a r = L (f a r) | R (g a r)
data (f :∗: g) a r  = f a r :∗: g a r
data Id a r = Id r
data P a r  = P a
```

```
class BiFunctor (PBF t) ⇒ BiRegular t where
  type PBF t :: ∗ → ∗ → ∗
  from :: t a → PBF t a (t a)
  to :: PBF t a (t a) → t a
```

```
PBT List = 1 :+: P :∗: Id
```