A Additional functions

Figure 3 showed the foundational monadic functions `parallel`, `congruently`, `comm`, `enclave`, `naked`, `fanOut`, and `fanIn`. Figure 9 shows type signatures for the entire library (as exported), in alphabetical order. Some functions are "core" in the sense that they can't be written without reference to `MultiChor`'s internal machinery. Other functions are "helper functions", which are implemented entirely using the exposed core functions.

B Additional Choreographies for the GMW Protocol

Oblivious transfer. Our implementation of oblivious transfer leverages a common strategy for building OT from public-key encryption. First, the receiver generates two public keys and one secret key (line 23); one of the public keys is real, and corresponds to the secret key, while the other is chosen at random from the space of public keys and has no corresponding secret key. The select bit determines the ordering of the public keys. The receiver sends the public keys to the sender (lines 24–25); the sender encrypts both b_1 and b_2 with the corresponding public keys and sends the ciphertexts back to the receiver (lines 26–27). Finally, the receiver decrypts the selected value (lines 28–30).

The sender treats both b_1 and b_2 in the same way, and cannot tell which public key is real and which one is fake—so the sender does not learn which value was selected. The receiver gets both encrypted values, but can only decrypt one of them, since only one of the public keys used has a corresponding secret key (the other public key is totally random). This version of OT is secure only against *honest but curious* or *passive* adversaries, who observe network communications but do not change the behavior of the parties, since an actively malicious receiver could create two actual key pairs in the first step of the protocol and thus be able to decrypt both b_1 and b_2 at the end. More complicated variants of the protocol can defend against this kind of attack.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

```
(-~>) :: forall a l ls' m ps. (Show a, Read a, KnownSymbol l, KnownSymbols ls')
      => (Member l ps, m a) -> Subset ls' ps -> Choreo ps m (Located ls' a)
infix 4 -~>

(~~>) :: forall a l ls' m ps. (Show a, Read a, KnownSymbol l, KnownSymbols ls')
      => (Member l ps, Unwrap l -> m a) -> Subset ls' ps -> Choreo ps m (Located ls' a)
infix 4 ~~>

(@@) :: Member x ys -> Subset xs ys -> Subset (x ': xs) ys
infixr 5 @@

(~>) :: (Show a, Read a, KnownSymbol l, KnownSymbols ls', CanSend s l a ls ps w)
     => s  -- ^ The message argument can take three forms:
           --     `(Member sender census, wrapped owners a)` where
           --        the sender is explicitly listed in owners,
           --     `(Member sender owners, Subset owners census, wrapped owners a)`, or
           --     `(Member sender census, (Member sender owners, wrapped owners a)`.
     -> Subset ls' ps -> Choreo ps m (Located ls' a)
infix 4 ~>

allOf :: forall ps. Subset ps ps

Backend :: Type -> Constraint  -- A phonebook for running Network monad expressions.

broadcast :: forall l a ps ls w m s.
             (Show a, Read a, KnownSymbol l, KnownSymbols ps, CanSend s l a ls ps w)
          => s -> Choreo ps m a

Choreo :: [LocTy] -> (Type -> Type) -> Type -> Type
type Choreo ps m = Freer (ChoreoSig ps m)

comm :: (Show a, Read a, KnownSymbol l, KnownSymbols ls', Wrapped w)
     => Member l ps -> (Member l ls, w ls a) -> Subset ls' ps
     -> Choreo ps m (Located ls' a)
infix 4 `comm`

cond :: (KnownSymbols ls)
     => (Subset ls ps, (Subset ls qs, Located qs a)) -> (a -> Choreo ls m b)
     -> Choreo ps m (Located ls b)

congruently :: (KnownSymbols ls)
             => Subset ls ps -> (Unwraps ls -> a) -> Choreo ps m (Located ls a)
infix 4 `congruently`

consSet :: Subset xs (x ': xs)

consSub :: Subset xs ys -> Member x ys -> Subset (x ': xs) ys

consSuper :: forall xs ys y. Subset xs ys -> Subset xs (y ': ys)
```

Figure 9. The `MultiChor` API, part 1/4.

```haskell
enclave :: (KnownSymbols ls)
        => Subset ls ps -> Choreo ls m a -> Choreo ps m (Located ls a)
infix 4 `enclave`

enclaveTo :: forall ls a rs ps m. (KnownSymbols ls)
          => Subset ls ps -> Subset rs ls -> Choreo ls m (Located rs a)
          -> Choreo ps m (Located rs a)
infix 4 `enclaveTo`

enclaveToAll :: forall ls a ps m. (KnownSymbols ls)
             => Subset ls ps -> Choreo ls m (Located ls a) -> Choreo ps m (Located ls a)
infix 4 `enclaveToAll`

epp :: (Monad m) => Choreo ps m a -> LocTm -> Network m a

ExplicitMember :: forall k. k -> [k] -> Constraint
explicitMember :: forall k (x :: k) (xs :: [k]). ExplicitMember x xs => Member x xs

ExplicitSubset :: forall {k}. [k] -> [k] -> Constraint
explicitSubset :: forall {k} (xs :: [k]) (ys :: [k]). ExplicitSubset xs ys => Subset xs ys

Faceted :: [LocTy] -> Type -> Type

fanIn :: (KnownSymbols qs, KnownSymbols rs)
       => Subset qs ps -> Subset rs ps
       -> (forall q. (KnownSymbol q)
           => Member q qs -> Choreo ps m (Located rs a))
       -> Choreo ps m (Located rs [a])

fanOut :: (KnownSymbols qs, Wrapped w)
       => Subset qs ps
       -> (forall q. (KnownSymbol q) => Member q qs -> Choreo ps m (w '[q] a))
       -> Choreo ps m (Faceted qs a)

flatten :: Subset ls ms -> Subset ls ns -> Located ms (Located ns a)
        -> Located ls a
infix 3 `flatten`

fracture :: forall ls a. (KnownSymbols ls) => Located ls a -> Faceted ls a

inSuper :: Subset xs ys -> Member x xs -> Member x ys

IsMember :: forall k. k -> [k] -> Type

IsSubset :: forall k. [k] -> [k] -> Type

KnownSymbols :: [Symbol] -> Constraint  -- lift KnownSymbol to type-level lists

listedFifth :: forall p5 p4 p3 p2 p1 ps. Member p5 (p1 ': p2 ': p3 ': p4 ': p5 ': ps)

listedFirst :: forall p1 ps. Member p1 (p1 ': ps)

listedForth :: forall p4 p3 p2 p1 ps. Member p4 (p1 ': p2 ': p3 ': p4 ': ps)

listedSecond :: forall p2 p1 ps. Member p2 (p1 ': p2 ': ps)
```

Figure 9. The `MultiChor` API, part 2/4.

```
listedSixth :: forall p6 p5 p4 p3 p2 p1 ps. Member p6 (p1 ': p2 ': p3 ': p4 ': p5 ': p6 ': ps)

listedThird :: forall p3 p2 p1 ps. Member p3 (p1 ': p2 ': p3 ': ps)

localize :: (KnownSymbol l) => Member l ls -> Faceted ls a -> Located '[l] a

locally :: (KnownSymbol (l :: LocTy))
        => Member l ps -> (Unwrap l -> m a) -> Choreo ps m (Located '[l] a)
infix 4 `locally`

locally_ :: (KnownSymbol l) => Member l ps -> (Unwrap l -> m ()) -> Choreo ps m ()
infix 4 `locally_`

_locally :: (KnownSymbol l) => Member l ps -> m a -> Choreo ps m (Located '[l] a)
infix 4 `_locally`

_locally_ :: (KnownSymbol l) => Member l ps -> m () -> Choreo ps m ()
infix 4 `_locally_`

Located :: [LocTy] -> Type -> Type

LocTm :: Type
type LocTm = String

LocTy :: Type
type LocTy = Symbol

Member :: forall {k}. k -> [k] -> Type
type Member x xs = Proof (IsMember x xs)

mkLoc :: String -> Q [Dec]  -- Template Haskell

naked :: Subset ps qs -> Located qs a -> Choreo ps m a
infix 4 `naked`

Network :: (Type -> Type) -> Type -> Type

NetworkSig :: (Type -> Type) -> Type -> Type

nobody :: Subset '[] ys

parallel :: (KnownSymbols ls)
         => Subset ls ps -> (forall l. (KnownSymbol l) => Member l ls -> Unwrap l -> m a)
         -> Choreo ps m (Faceted ls a)

parallel_ :: forall ls ps m. (KnownSymbols ls)
          => Subset ls ps -> (forall l. (KnownSymbol l) => Member l ls -> Unwrap l ->m ())
          -> Choreo ps m ()

_parallel :: forall ls a ps m. (KnownSymbols ls)
          => Subset ls ps -> m a -> Choreo ps m (Faceted ls a)

recv :: forall a (m :: Type -> Type). Read a => LocTm -> Network m a

run :: forall (m :: Type -> Type) a. m a -> Network m a
```

Figure 9. The `MultiChor` API, part 3/4.

```
runChoreo :: forall ps b m. Monad m => Choreo ps m b -> m b

runNetwork :: (Backend c, MonadIO m) => c -> LocTm -> Network m a -> m a

send :: forall a (m :: Type -> Type). Show a => a -> [LocTm] -> Network m ()

singleton :: forall p. Member p (p ': '[])

Subset :: forall {k}. [k] -> [k] -> Type
type Subset xs ys = Proof (IsSubset xs ys)

toLocs :: forall (ls :: [LocTy]) (ps :: [LocTy]). KnownSymbols ls => Subset ls ps -> [LocTm]

toLocTm :: forall (l :: LocTy) (ps :: [LocTy]). KnownSymbol l => Member l ps -> LocTm

Unwrap :: LocTy -> Type
type Unwrap (l :: LocTy) = forall ls a w. (Wrapped w) => Member l ls -> w ls a -> a

Unwraps :: [LocTy] -> Type
type Unwraps (qs :: [LocTy]) = forall ls a. Subset qs ls -> Located ls a -> a

Wrapped :: ([Symbol] -> Type -> Type) -> Constraint
```

Figure 9. The `MultiChor` API, part 4/4.

```
genKeys :: (CRT.MonadRandom m) => Bool -> m (RSA.PublicKey, RSA.PublicKey, RSA.PrivateKey)
genKeys s = do -- Generate keys for OT. One key is real, and one is fake - select bit decides
  (pk, sk) <- genKeyPair
  fakePk <- generateFakePK
  return $ if s then (pk, fakePk, sk) else (fakePk, pk, sk)

encryptS :: (CRT.MonadRandom m) => -- Encryption based on select bit
            (RSA.PublicKey, RSA.PublicKey) -> Bool -> Bool -> m (ByteString, ByteString)
encryptS (pk1, pk2) b1 b2 = do c1 <- encryptRSA pk1 b1; c2 <- encryptRSA pk2 b2; return (c1, c2)

decryptS :: (CRT.MonadRandom m) => -- Decryption based on select bit
       (RSA.PublicKey, RSA.PublicKey, RSA.PrivateKey) -> Bool -> (ByteString, ByteString) -> m Bool
decryptS (_, _, sk) s (c1, c2) = if s then decryptRSA sk c1 else decryptRSA sk c2

-- One out of two OT
ot2 :: (KnownSymbol sender, KnownSymbol receiver, MonadIO m, CRT.MonadRandom m) =>
  Located '[sender] (Bool, Bool) -> Located '[receiver] Bool
  -> Choreo '[sender, receiver] (CLI m) (Located '[receiver] Bool)
ot2 bb s = do
  let sender = listedFirst :: Member sender '[sender, receiver]
  let receiver = listedSecond :: Member receiver '[sender, receiver]

  keys <- receiver `locally` \un -> liftIO $ genKeys $ un singleton s
  pks <- (receiver, \un -> let (pk1, pk2, _) = un singleton keys
                           in return (pk1, pk2)) ~~> sender @@ nobody
  encrypted <- (sender, \un -> let (b1, b2) = un singleton bb
                               in liftIO $ encryptS (un singleton pks) b1 b2) ~~> receiver @@ nobody
  receiver `locally` \un -> liftIO $ decryptS (un singleton keys)
                                              (un singleton s)
                                              (un singleton encrypted)
```

Figure 10. A choreography for performing 1 out of 2 oblivious transfer (OT) using RSA public-key encryption. The choreography involves exactly two parties, `sender` and `receiver`.