Note VIII

by Chenchao Ding

Dec. 2, 2024

Expression and Value (Syntax)


```haskell
type Name = String
data Expr where
    Var :: Name -> Expr
    Const :: Bool -> Expr
    -- Rand :: Int -> Expr
    NotE :: Expr -> Expr
    OrE :: Expr -> Expr -> Expr
    AndE :: Expr -> Expr -> Expr
    PairE :: Expr -> Expr -> Expr
    Fst :: Expr -> Expr
    Snd :: Expr -> Expr
    LetE :: Name -> Expr -> Expr -> Expr
    deriving (Show)

data Value where
    Error :: Value
    BoolV :: Bool -> Value
    PairV :: Value -> Value -> Value
    deriving (Show)

type Outcome = Bool
type Env = [(Name, Value)]
type M = State Bool
```

Interpreter

```haskell
lookupM :: Name -> Env -> M Value
lookupM s [] = return Error
lookupM s ((x,v):rest) = if (x == s) then (return v) else (lookupM s rest)

notM :: Value -> M Value
notM (BoolV b) = return (BoolV (not b))
notM _ = return Error

orM, andM :: Value -> Value -> M Value
orM (BoolV b1) (BoolV b2) = return (BoolV (or [b1, b2]))
orM _ _ = return Error

andM (BoolV b1) (BoolV b2) = return (BoolV (and [b1, b2]))
andM _ _ = return Error

interpM :: Env -> Expr -> M Value
interpM env (Var s) = lookupM s env
interpM env (Const b) = return (BoolV b)
interpM env (NotE e) = do
    v <- interpM env e
    notM v
```

Single Observable (Measurement)


```haskell
pureContingent :: Expr -> M Outcome
pureContingent expr = get

contingentOnInterp :: Expr -> M Outcome
contingentOnInterp expr = do
    v <- interpM [] expr
    case v of
        BoolV b -> return b
        _ -> get
```

```haskell
causeTrue :: Expr -> M Outcome
causeTrue expr = do
    put True
    contingentOnInterp expr
```

```haskell
causeFalse :: Expr -> M Outcome
causeFalse expr = do
    put False
    contingentOnInterp expr
```

```haskell
hasError :: Expr -> M Outcome
hasError expr = do
    v <- interpM [] expr
    case v of
        Error -> return True
        _ -> return False
```

```haskell
isBoolV :: Expr -> M Outcome
isBoolV expr = do
    v <- interpM [] expr
    case v of
        BoolV b -> return True
        _ -> return False
```

Simulating Experiment

```
-- experiment apparatus

-- expression as quantum system
type Qsystem = Expr
-- test as single observable
type Observable = Qsystem -> M Outcome
-- test suite as measurement context
type Context = [Observable]
-- joint outcomes given a context
type Experiment = Qsystem -> Context -> M [Outcome]
```

Simulating Experiment


```haskell
exp1 :: Qsystem -> Observable -> M Outcome
exp1 expr f = f expr

exp2 :: Qsystem -> (Observable, Observable) -> M (Outcome, Outcome)
exp2 expr (f1, f2) = do
    o1 <- f1 expr
    o2 <- f2 expr
    return (o1, o2)
```

```haskell
expn :: Qsystem -> Context -> M [Outcome]
expn expr [] = return []
expn expr (f:fs) = do
    o <- f expr
    os <- expn expr fs
    return (o:os)
```

```haskell
type Seed = Int
-- joint outcomes given a context
runExperiment :: Qsystem -> Context -> Seed -> [Outcome]
runExperiment expr ctx seed =
    let m = expn expr ctx in
        evalState m (fst (randomBool seed))
```

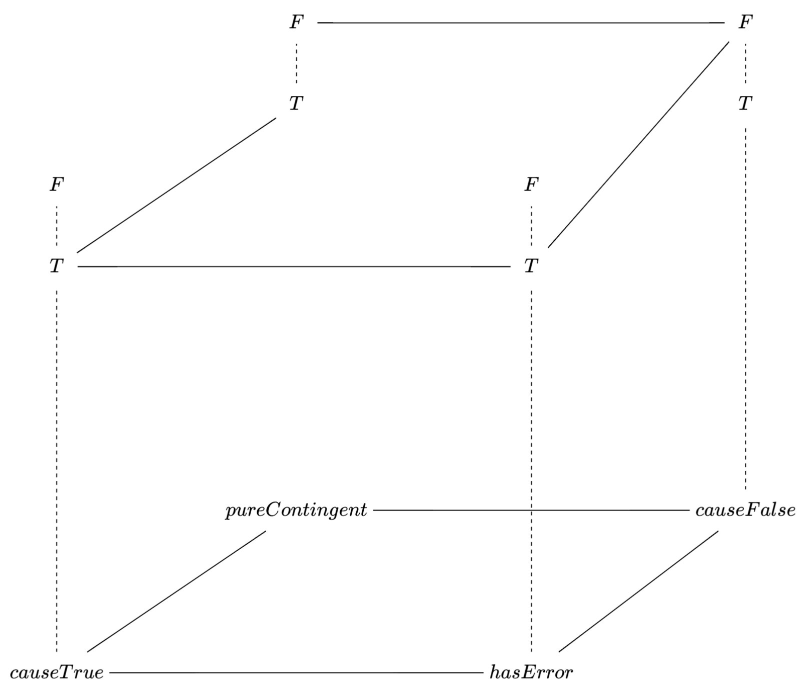
Examples


```
-- Qsystem independent contextuality...
ctx1 :: [Expr -> M Outcome]
ctx1 = [causeTrue, pureContingent]
ctx2 :: [Expr -> M Outcome]
ctx2 = [causeFalse, pureContingent]
ctx3 :: [Expr -> M Outcome]
ctx3 = [causeTrue, hasError]
ctx4 :: [Expr -> M Outcome]
ctx4 = [causeFalse, hasError]

-- two by two (Bell configuration)
suite1 :: [[Expr -> M Outcome]]
suite1 = [ctx1,ctx2,ctx3,ctx4]
-- one by one
suite2 :: [[Expr -> M Outcome]]
suite2 = [[pureContingent], [causeFalse], [hasError], [causeTrue]]
```

```
e1, e2, e3 :: Expr
e1 = NotE (Var "s")
e2 = LetE "x" (AndE (NotE (Const True)) (Const False)) (NotE (Var "x"))
e3 = PairE (PairE (Const True) e2) (Const True)
```

Examples




```haskell
printResult :: Qsystem -> [Context] -> Seed -> IO ()
printResult expr [] seed = pure ()
printResult expr (c:cs) seed = do
    print (runExperiment expr c seed)
    printResult expr cs (seed + 1)
```

```haskell
printResult e1 suite1 111
```

COMBINING CONTEXTUALITY AND CAUSALITY: A GAME SEMANTICS APPROACH

SAMSON ABRAMSKY, RUI SOARES BARBOSA, AND AMY SEARLE

ABSTRACT. We develop an approach to combining contextuality with causality, which is general enough to cover causal background structure, adaptive measurement-based quantum computation, and causal networks. The key idea is to view contextuality as arising from a game played between Experimenter and Nature, allowing for causal dependencies in the actions of both the Experimenter (choice of measurements) and Nature (choice of outcomes).

Definition 3 We say that a separated presheaf $A : C^{\mathrm{op}} \to \mathbf{Sets}$ is no-signalling, or locally consistent, if every $A_{U\subseteq V} : A_V \to A_U :: s \mapsto s|_U$ is a surjection, i.e., if $A : C^{\mathrm{op}} \to \mathbf{Surj}$ for the category **Surj** of sets and surjections.