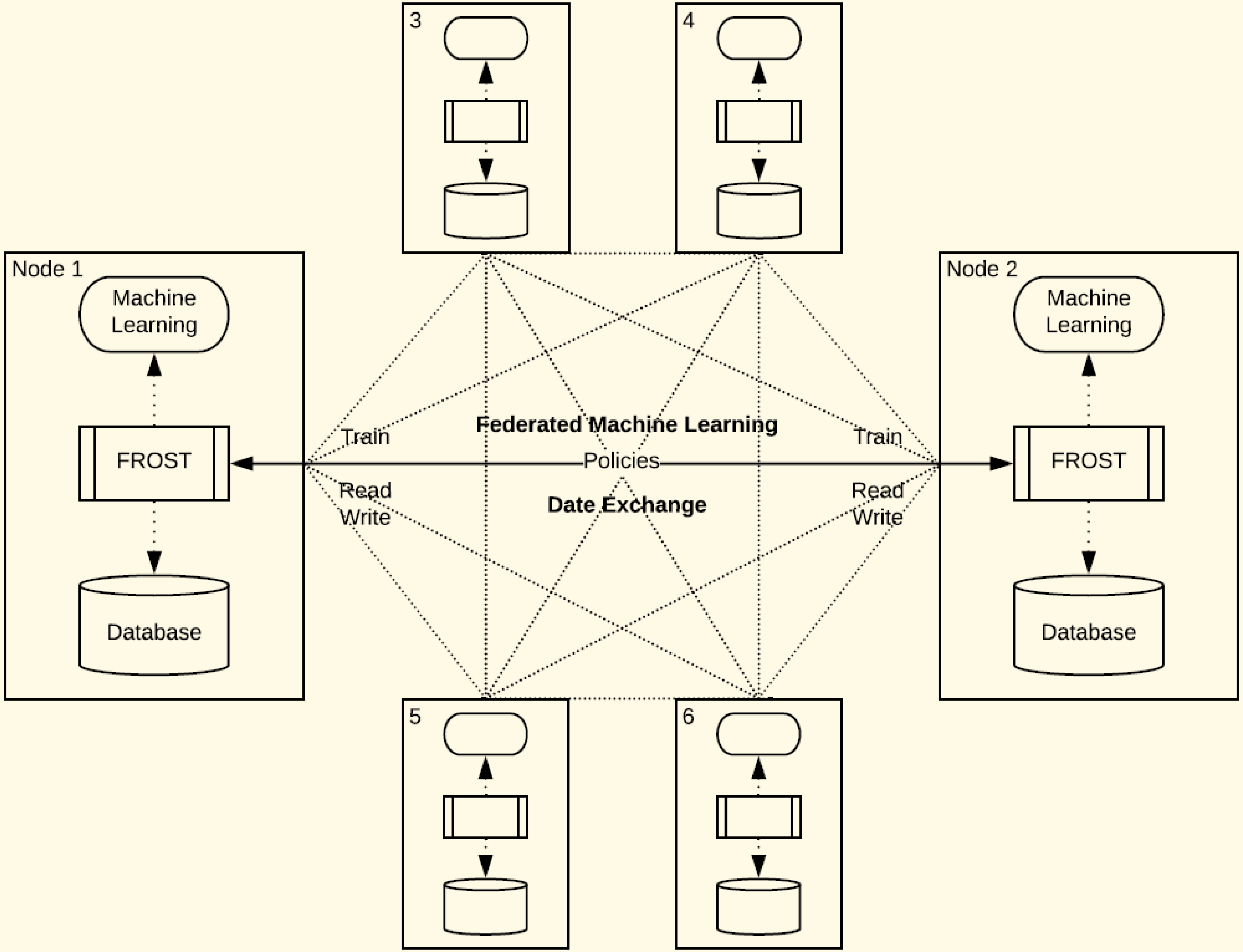
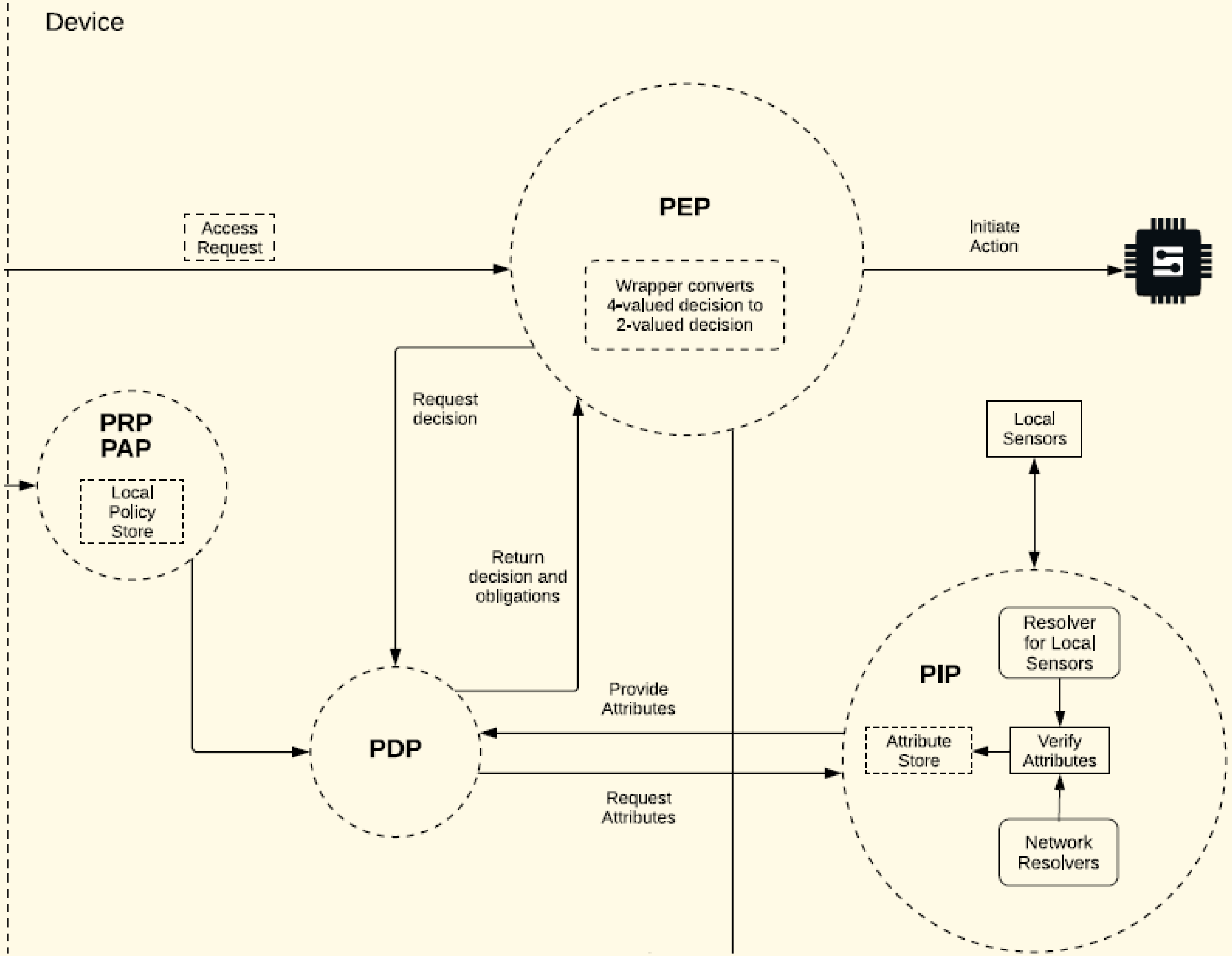

3
4
Node 1
Machine Learning
FROST
Database
Node 2
Machine Learning
FROST
Database
Train
Federated Machine Learning
Train
Policies
Read Write
Date Exchange
Read Write
5
6

FROST: an access control policy language

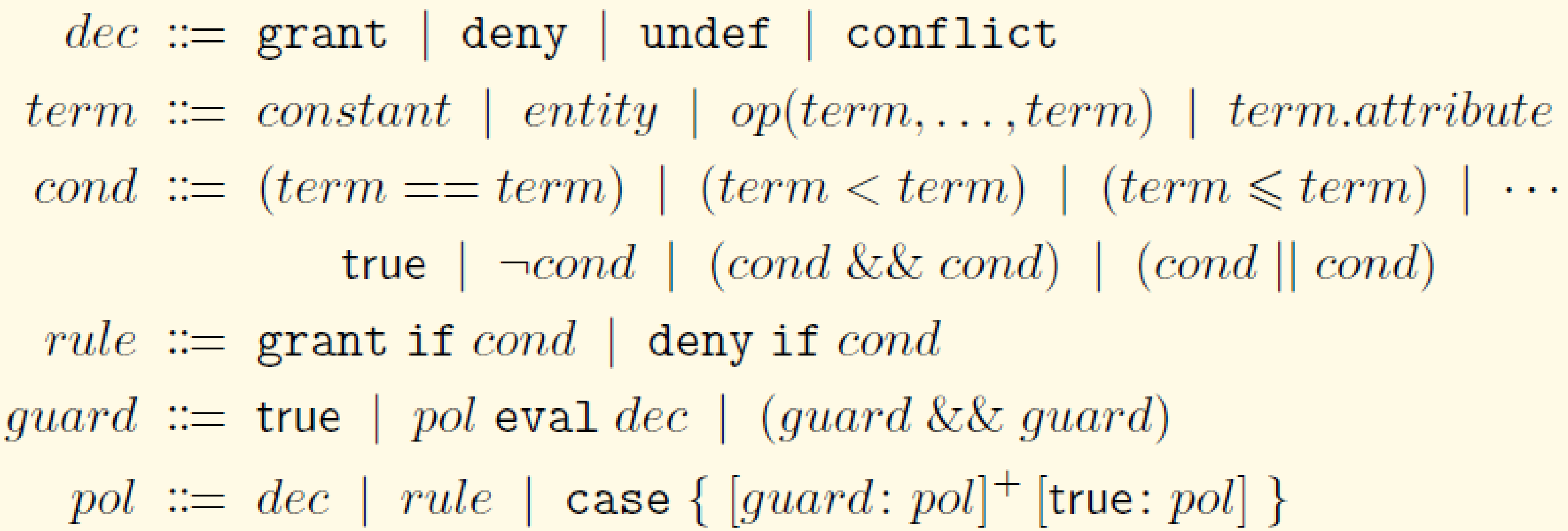
F = Flexible
R = Robust
O = Open
S = Service-Enabling
T = Trusted


Device
Access Request
PEP
Wrapper converts 4-valued decision to 2-valued decision
Initiate Action
Request decision
PRP
PAP
Local Policy Store
Return decision and obligations
Local Sensors
PIP
Resolver for Local Sensors
Provide Attributes
PDP
Attribute Store
Verify Attributes
Request Attributes
Network Resolvers

$\texttt{grant if}\ cond$ $\qquad\qquad$ $\texttt{deny if}\ cond$

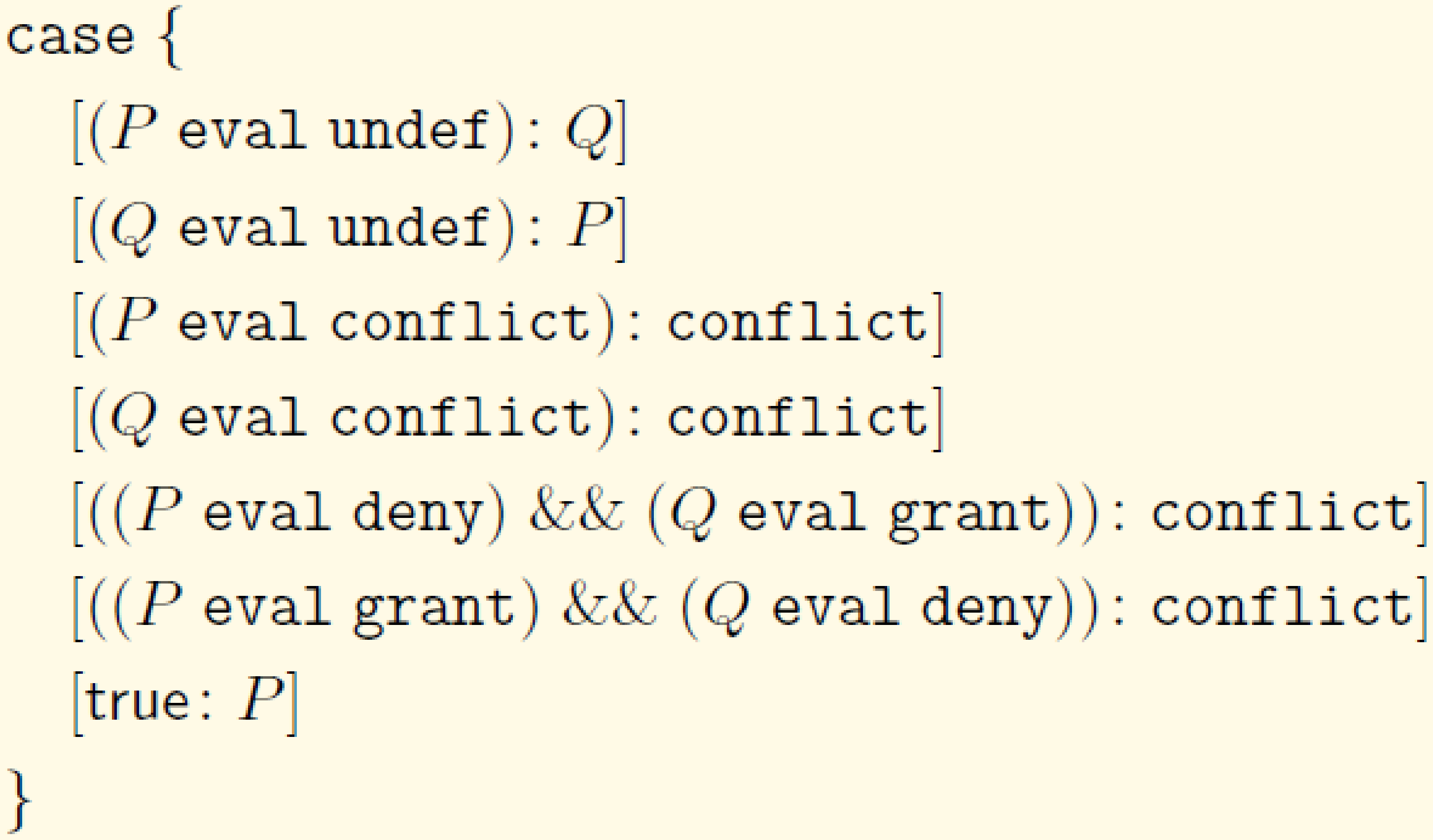
$$
\begin{aligned}
\texttt{grant if}\quad & (\mathbf{object} == vehicle)\ \&\&\ (\mathbf{subject} == vehicle.owner.daughter)\ \&\& \\
& (\mathbf{action} == driveVehicle)\ \&\& \\
& (owner.daughter.isInsured == true)\ \&\& \\
& (0900 \leqslant localTime)\ \&\&\ (localTime \leqslant 2000)
\end{aligned}
$$

Grammars for FROST



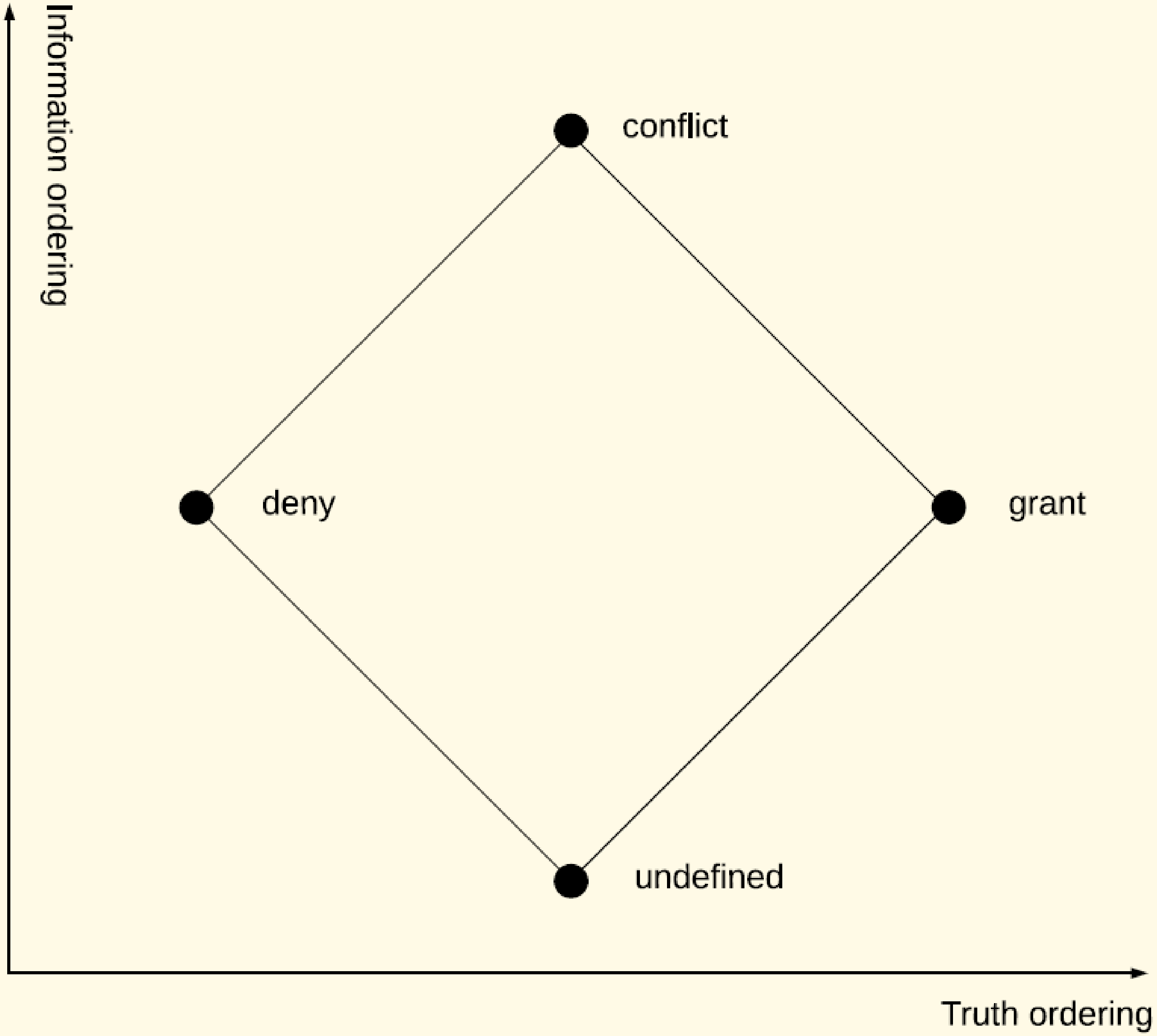
$$
\begin{aligned}
dec &::= \texttt{grant} \mid \texttt{deny} \mid \texttt{undef} \mid \texttt{conflict} \\
term &::= constant \mid entity \mid op(term, \ldots, term) \mid term.attribute \\
cond &::= (term == term) \mid (term < term) \mid (term \leqslant term) \mid \cdots \\
&\qquad\quad \mathsf{true} \mid \neg cond \mid (cond \;\&\&\; cond) \mid (cond \mid\mid cond) \\
rule &::= \texttt{grant if}\ cond \mid \texttt{deny if}\ cond \\
guard &::= \mathsf{true} \mid pol\ \texttt{eval}\ dec \mid (guard \;\&\&\; guard) \\
pol &::= dec \mid rule \mid \texttt{case}\ \{\ [guard\colon pol]^{+}\ [\mathsf{true}\colon pol]\ \}
\end{aligned}
$$

Derived Policy Composition: Information Join



```
case {
    [($P$ eval undef): $Q$]
    [($Q$ eval undef): $P$]
    [($P$ eval conflict): conflict]
    [($Q$ eval conflict): conflict]
    [(($P$ eval deny) && ($Q$ eval grant)): conflict]
    [(($P$ eval grant) && ($Q$ eval deny)): conflict]
    [true: $P$]
}
```

Belnap Bilattice



“Grant-or-Conflict” Circuit Compilation

$$\begin{aligned}
\mathsf{GoC}(\texttt{grant}) &\equiv \mathsf{true} & \mathsf{GoC}(\texttt{deny}) &\equiv \mathsf{false}\\
\mathsf{GoC}(\texttt{conflict}) &\equiv \mathsf{true} & \mathsf{GoC}(\texttt{undef}) &\equiv \mathsf{false}\\
\mathsf{GoC}(\texttt{grant if } cond) &\equiv cond & \mathsf{GoC}(\texttt{deny if } cond) &\equiv \mathsf{false}
\end{aligned}$$

$$\begin{aligned}
\mathsf{GoC}(\mathsf{case}\,\{\,[g_1 : p_1]\ \ldots\ [g_{n-1} : p_{n-1}]\,[\mathsf{true}\!: p_n]\,\}) \;\equiv\; &(\mathsf{R}(g_1)\ \&\&\ \mathsf{GoC}(p_1))\ ||\ \cdots\\
&\cdots\ ||\ (\mathsf{R}(g_{n-1})\ \&\&\ \mathsf{GoC}(p_{n-1}))\ ||\ (\mathsf{R}(\mathsf{true})\ \&\&\ \mathsf{GoC}(p_n))
\end{aligned}$$

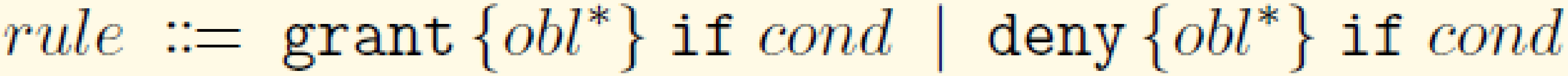
$$
\begin{aligned}
\mathsf{R}(g_1) &\equiv \quad \mathsf{T}(g_1) \\
\mathsf{R}(g_i) &\equiv \neg\mathsf{T}(g_1)\ \&\&\ \ldots\ \&\&\ \neg\mathsf{T}(g_{i-1})\ \&\&\ \mathsf{T}(g_i)\,, \quad 1 < i < n \\
\mathsf{R}(\mathsf{true}) &\equiv \neg\mathsf{T}(g_1)\ \&\&\ \ldots\ \&\&\ \neg\mathsf{T}(g_{n-1})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{T}(\mathsf{true}) &\equiv \mathsf{true} \\
\mathsf{T}(g_1\ \&\&\ g_2) &\equiv \mathsf{T}(g_1)\ \&\&\ \mathsf{T}(g_2) \\
\mathsf{T}(pol\ \mathtt{eval}\ dec) &\equiv \begin{cases}
\mathsf{GoC}(pol)\ \&\&\ \mathsf{DoC}(pol) & \text{if } dec \text{ equals } \mathtt{conflict} \\
\neg\mathsf{GoC}(pol)\ \&\&\ \mathsf{DoC}(pol) & \text{if } dec \text{ equals } \mathtt{deny} \\
\mathsf{GoC}(pol)\ \&\&\ \neg\mathsf{DoC}(pol) & \text{if } dec \text{ equals } \mathtt{grant} \\
\neg\mathsf{GoC}(pol)\ \&\&\ \neg\mathsf{DoC}(pol) & \text{if } dec \text{ equals } \mathtt{undef}
\end{cases}
\end{aligned}
$$

Join normal form

$$pol \;\equiv\; (\texttt{grant if } \mathsf{GoC}(pol))\ \texttt{join}\ (\texttt{deny if } \mathsf{DoC}(pol))$$

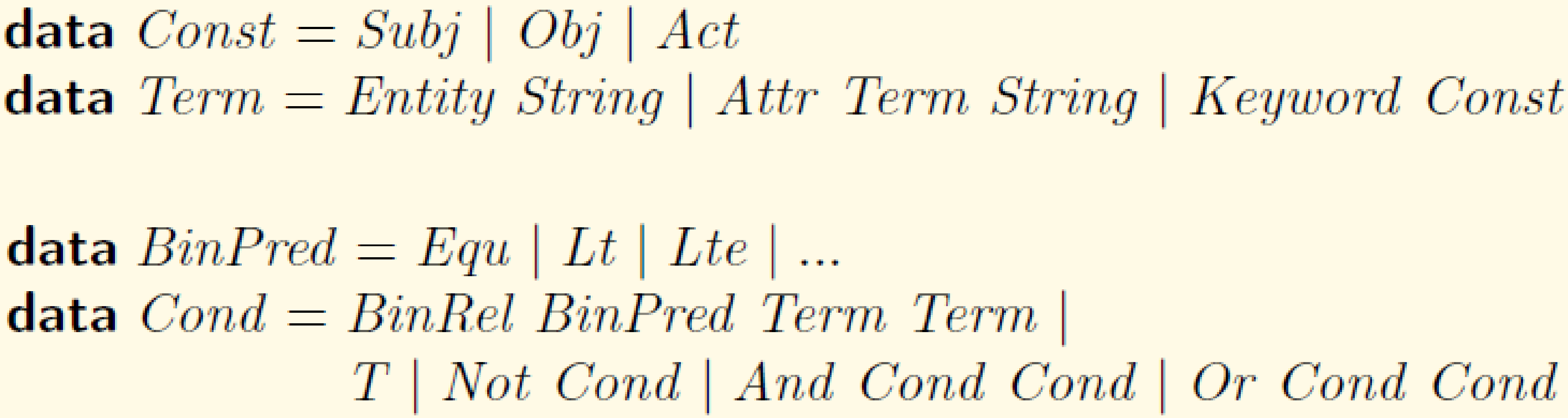
Obligations: annotations to rules



$$rule \ ::= \ \texttt{grant}\,\{obl^*\}\ \texttt{if}\ cond \ \mid \ \texttt{deny}\,\{obl^*\}\ \texttt{if}\ cond$$

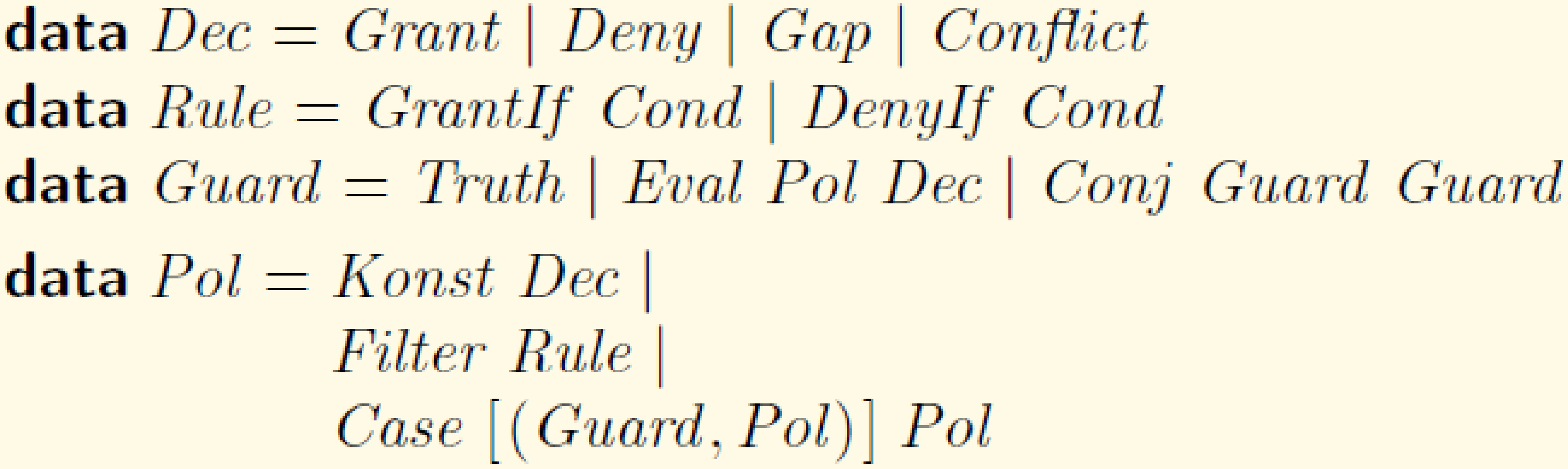
$$
\begin{aligned}
\mathsf{oblg}(dec,\, dec',\, \rho) \;&\equiv\; \{\,\} \\
\mathsf{oblg}(dec,\, \mathtt{grant}\,\{obl^*\}\;\mathtt{if}\;cond,\, \rho) \;&\equiv\; \begin{cases} \{obl^*\} & \text{if } dec = \mathtt{grant} \text{ and } \rho \models cond \\ \{\,\} & \text{otherwise} \end{cases} \\
\mathsf{oblg}(dec,\, \mathtt{deny}\,\{obl^*\}\;\mathtt{if}\;cond,\, \rho) \;&\equiv\; \begin{cases} \{obl^*\} & \text{if } dec = \mathtt{deny} \text{ and } \rho \models cond \\ \{\,\} & \text{otherwise} \end{cases} \\
\mathsf{oblg}(dec,\, \mathsf{case}\,\{\,[g_1 : p_1]\;\ldots\;[g_{n-1} : p_{n-1}]\,[\mathsf{true}\colon p_n]\,\},\, \rho) \;&\equiv\; \mathsf{oblg}'(dec,\, g_i,\, \rho) \cup \mathsf{oblg}(dec,\, p_i,\, \rho) \\
&\qquad \text{where } \rho \models \mathsf{R}(g_i) \\
\mathsf{oblg}'(dec,\, \mathsf{true},\, \rho) \;&\equiv\; \{\,\} \\
\mathsf{oblg}'(dec,\, pol\;\mathtt{eval}\;dec',\, \rho) \;&\equiv\; \begin{cases} \mathsf{oblg}(dec,\, pol,\, \rho) & \text{if } dec = dec' \\ \{\,\} & \text{otherwise} \end{cases} \\
\mathsf{oblg}'(dec,\, g_1 \;\&\&\; g_2,\, \rho) \;&\equiv\; \mathsf{oblg}'(dec,\, g_1,\, \rho) \cup \mathsf{oblg}'(dec,\, g_2,\, \rho)
\end{aligned}
$$

Embedded DSL: terms & conditions



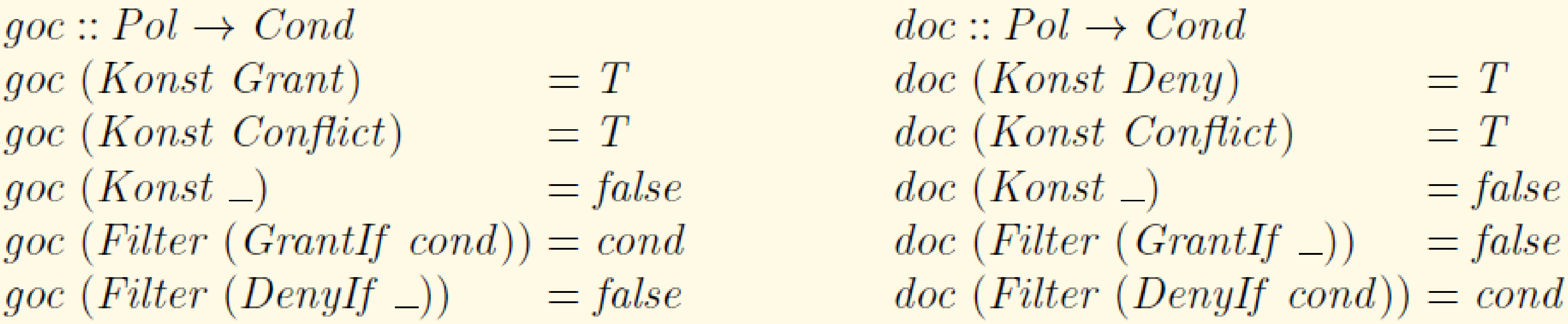
```
data Const = Subj | Obj | Act
data Term = Entity String | Attr Term String | Keyword Const

data BinPred = Equ | Lt | Lte | ...
data Cond = BinRel BinPred Term Term |
            T | Not Cond | And Cond Cond | Or Cond Cond
```

Policies



```
data Dec = Grant | Deny | Gap | Conflict
data Rule = GrantIf Cond | DenyIf Cond
data Guard = Truth | Eval Pol Dec | Conj Guard Guard
data Pol = Konst Dec |
           Filter Rule |
           Case [(Guard, Pol)] Pol
```

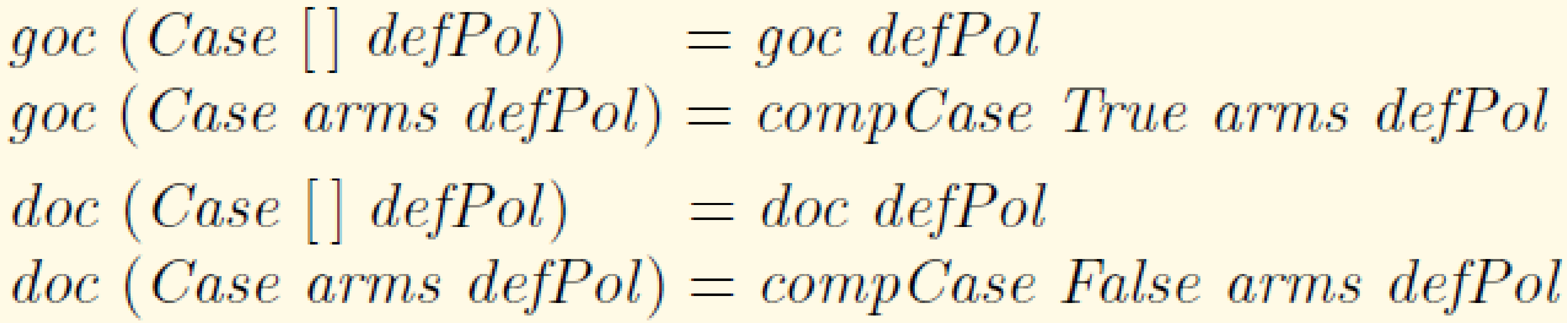
Compiling to Circuits



$$
\begin{array}{ll}
\mathit{goc} :: \mathit{Pol} \rightarrow \mathit{Cond} & \\
\mathit{goc}\ (\mathit{Konst}\ \mathit{Grant}) & = T \\
\mathit{goc}\ (\mathit{Konst}\ \mathit{Conflict}) & = T \\
\mathit{goc}\ (\mathit{Konst}\ \_) & = \mathit{false} \\
\mathit{goc}\ (\mathit{Filter}\ (\mathit{GrantIf}\ \mathit{cond})) & = \mathit{cond} \\
\mathit{goc}\ (\mathit{Filter}\ (\mathit{DenyIf}\ \_)) & = \mathit{false}
\end{array}
$$

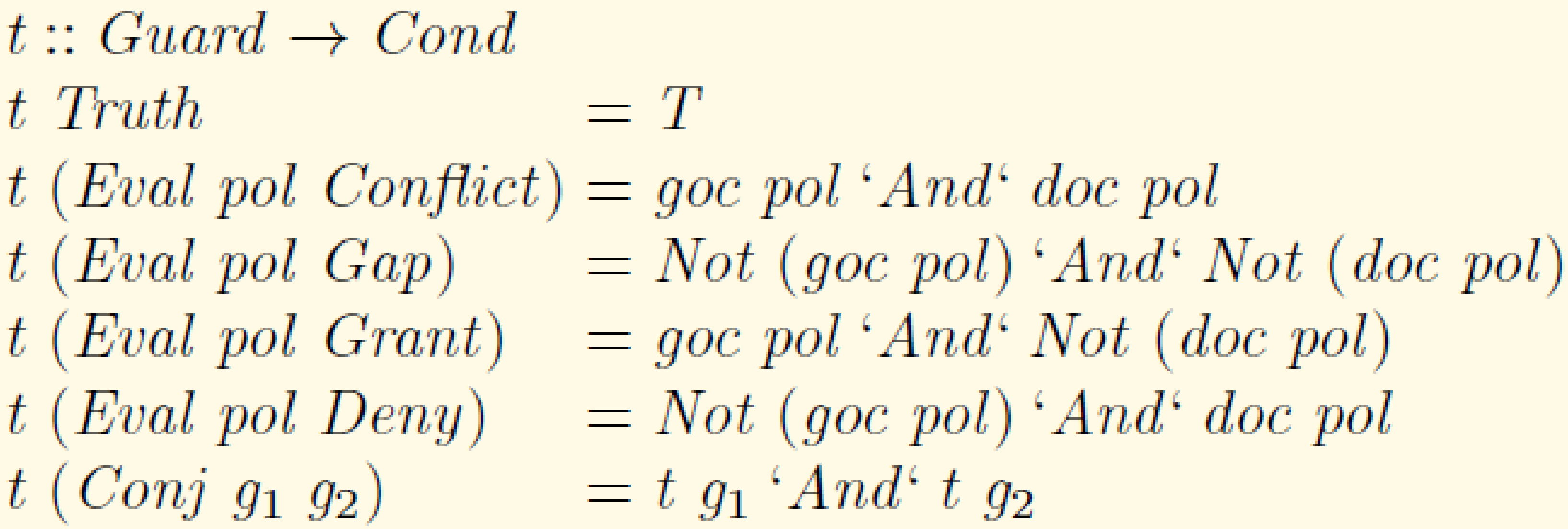
$$
\begin{array}{ll}
\mathit{doc} :: \mathit{Pol} \rightarrow \mathit{Cond} & \\
\mathit{doc}\ (\mathit{Konst}\ \mathit{Deny}) & = T \\
\mathit{doc}\ (\mathit{Konst}\ \mathit{Conflict}) & = T \\
\mathit{doc}\ (\mathit{Konst}\ \_) & = \mathit{false} \\
\mathit{doc}\ (\mathit{Filter}\ (\mathit{GrantIf}\ \_)) & = \mathit{false} \\
\mathit{doc}\ (\mathit{Filter}\ (\mathit{DenyIf}\ \mathit{cond})) & = \mathit{cond}
\end{array}
$$

Case Policies



$$\begin{aligned}
goc\ (Case\ []\ defPol) &= goc\ defPol \\
goc\ (Case\ arms\ defPol) &= compCase\ True\ arms\ defPol \\
doc\ (Case\ []\ defPol) &= doc\ defPol \\
doc\ (Case\ arms\ defPol) &= compCase\ False\ arms\ defPol
\end{aligned}$$

Guards -> Conditions



$$
\begin{aligned}
&t :: \mathit{Guard} \rightarrow \mathit{Cond} \\
&t\ \mathit{Truth} &&= T \\
&t\ (\mathit{Eval}\ \mathit{pol}\ \mathit{Conflict}) &&= \mathit{goc}\ \mathit{pol}\ \text{‘}\mathit{And}\text{‘}\ \mathit{doc}\ \mathit{pol} \\
&t\ (\mathit{Eval}\ \mathit{pol}\ \mathit{Gap}) &&= \mathit{Not}\ (\mathit{goc}\ \mathit{pol})\ \text{‘}\mathit{And}\text{‘}\ \mathit{Not}\ (\mathit{doc}\ \mathit{pol}) \\
&t\ (\mathit{Eval}\ \mathit{pol}\ \mathit{Grant}) &&= \mathit{goc}\ \mathit{pol}\ \text{‘}\mathit{And}\text{‘}\ \mathit{Not}\ (\mathit{doc}\ \mathit{pol}) \\
&t\ (\mathit{Eval}\ \mathit{pol}\ \mathit{Deny}) &&= \mathit{Not}\ (\mathit{goc}\ \mathit{pol})\ \text{‘}\mathit{And}\text{‘}\ \mathit{doc}\ \mathit{pol} \\
&t\ (\mathit{Conj}\ g_1\ g_2) &&= t\ g_1\ \text{‘}\mathit{And}\text{‘}\ t\ g_2
\end{aligned}
$$

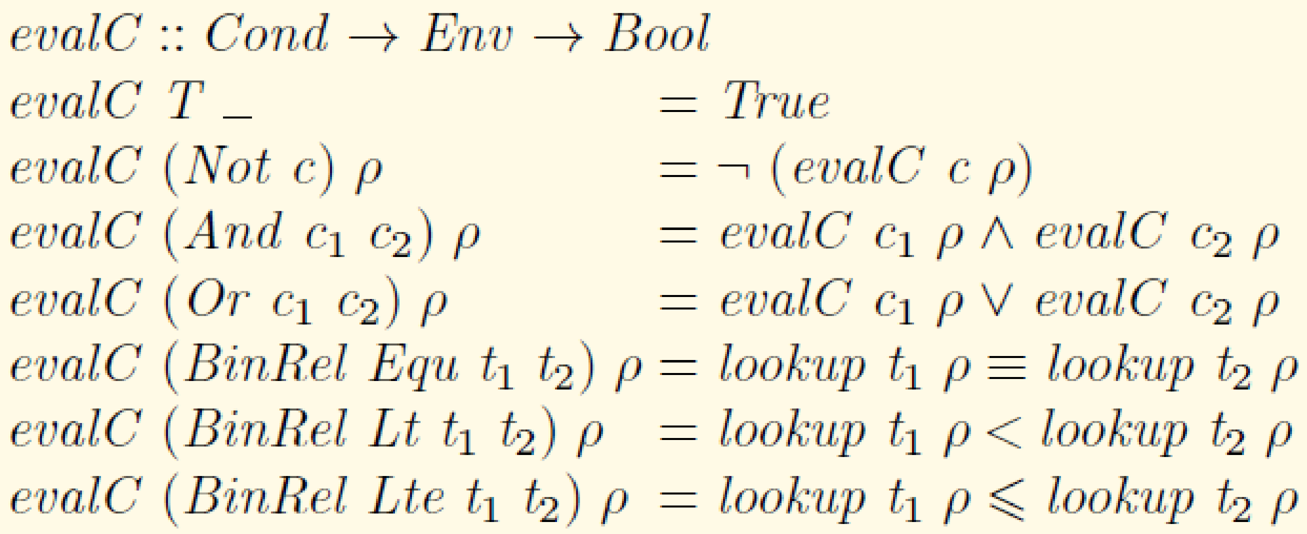
```
compCase :: Bool → [(Guard, Pol)] → Pol → Cond
compCase isGoc arms defPol =
  foldr (Or ∘ disjunct isGoc) (lastDisjunct isGoc guards defPol) armInits
  where
    armInits = tail (inits arms)
    guards   = map fst arms
```

```
disjunct :: Bool → [(Guard, Pol)] → Cond
disjunct b arms = foldr (And ∘ Not ∘ t ∘ fst) (t trueGuard) pairs
  ‘And‘ compPol pol
  where
    (pairs, [(trueGuard, pol)]) = splitAt (length arms − 1) arms
    compPol = if b then goc else doc
```

```
lastDisjunct :: Bool → [Guard] → Pol → Cond
lastDisjunct b gs pol = foldr (And ∘ Not ∘ t) T gs
  ‘And‘ compPol pol
  where
    compPol = if b then goc else doc
```

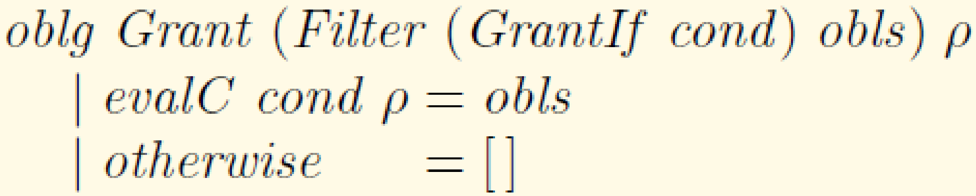
```
data Oblg = ...

data Pol = Konst Dec |
           Filter Rule [Oblg] |
           Case [(Guard, Pol)] Pol


data Env = ...

lookup :: Term → Env → Integer
```

Evaluating Conditions



$$
\begin{aligned}
&\mathit{evalC} :: \mathit{Cond} \to \mathit{Env} \to \mathit{Bool} \\
&\mathit{evalC}\ T\ \_ &&= \mathit{True} \\
&\mathit{evalC}\ (\mathit{Not}\ c)\ \rho &&= \neg\,(\mathit{evalC}\ c\ \rho) \\
&\mathit{evalC}\ (\mathit{And}\ c_1\ c_2)\ \rho &&= \mathit{evalC}\ c_1\ \rho \wedge \mathit{evalC}\ c_2\ \rho \\
&\mathit{evalC}\ (\mathit{Or}\ c_1\ c_2)\ \rho &&= \mathit{evalC}\ c_1\ \rho \vee \mathit{evalC}\ c_2\ \rho \\
&\mathit{evalC}\ (\mathit{BinRel}\ \mathit{Equ}\ t_1\ t_2)\ \rho &&= \mathit{lookup}\ t_1\ \rho \equiv \mathit{lookup}\ t_2\ \rho \\
&\mathit{evalC}\ (\mathit{BinRel}\ \mathit{Lt}\ t_1\ t_2)\ \rho &&= \mathit{lookup}\ t_1\ \rho < \mathit{lookup}\ t_2\ \rho \\
&\mathit{evalC}\ (\mathit{BinRel}\ \mathit{Lte}\ t_1\ t_2)\ \rho &&= \mathit{lookup}\ t_1\ \rho \leqslant \mathit{lookup}\ t_2\ \rho
\end{aligned}
$$

e.g. Grant-obligations for a rule



```
oblg Grant (Filter (GrantIf cond) obls) ρ
    | evalC cond ρ = obls
    | otherwise    = []
```

Writer Monad

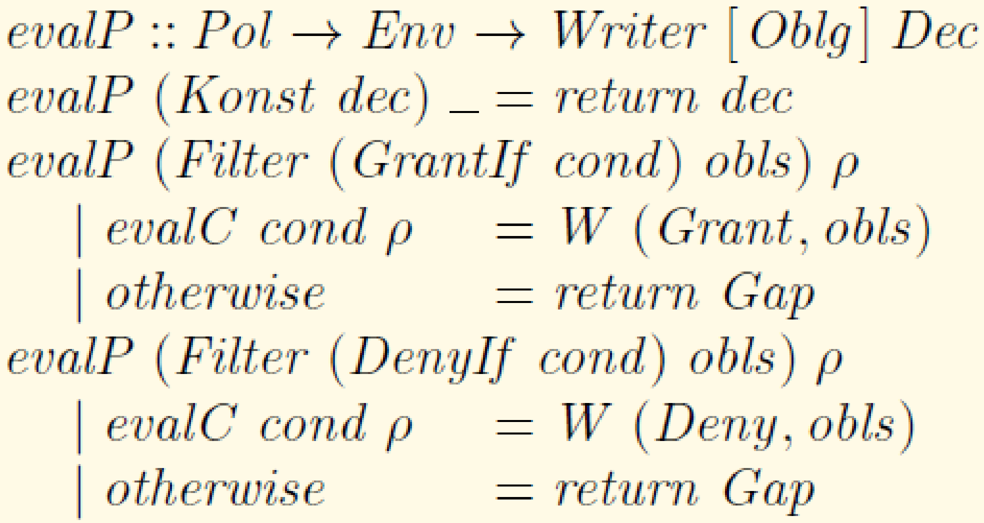
```
data Writer o a = W (a, o)
```

```
class Monoid o where
  ∅   :: o
  (⊕) :: o → o → o
```

```
instance Monoid [a] where
  ∅   = []
  (⊕) = (++)
```

```
instance Monoid o ⇒ Monad (Writer o) where
  return x         = W (x, ∅)
  W (x, v) ≫= f = let W (y, v′) = f x in W (y, v ⊕ v′)
```

Evaluating Policies



```
evalP :: Pol → Env → Writer [Oblg] Dec
evalP (Konst dec) _ = return dec
evalP (Filter (GrantIf cond) obls) ρ
    | evalC cond ρ     = W (Grant, obls)
    | otherwise        = return Gap
evalP (Filter (DenyIf cond) obls) ρ
    | evalC cond ρ     = W (Deny, obls)
    | otherwise        = return Gap
```

```
clearIf :: MonadWriter o m ⇒ m a → (a → Bool) → m a
clearIf xm pred = pass (do
  x ← xm
  return (x, if pred x then const ∅ else id))
```

```
evalP (Case [] defPol) ρ = evalP defPol ρ
evalP (Case ((g, p) : as) defPol) ρ = do
  b ← evalG g ρ `clearIf` not
  if b then evalP p ρ
       else evalP (Case as defPol) ρ
```

Evaluating Guards

```
$evalG :: Guard \rightarrow Env \rightarrow Writer\ [Oblg]\ Bool$
$evalG\ Truth\ \_ = return\ True$
$evalG\ (Eval\ pol\ dec)\ \rho = \mathbf{do}$
    $d \leftarrow clearIf\ (evalP\ pol\ \rho)\ (\lambda d \rightarrow d \not\equiv dec \vee d \in [Gap, Conflict])$
    $return\ (d \equiv dec)$
$evalG\ (Conj\ g_1\ g_2)\ \rho = \mathbf{do}$
    $b_1 \leftarrow evalG\ g_1\ \rho$
    $b_2 \leftarrow evalG\ g_2\ \rho$
    $return\ (b_1 \wedge b_2)$
```

Thanks + Q&A

www.xain.io