ϕ	::=		*Formulas:*
		$\{v : \texttt{Proof} \mid e\}$	*first order terms, with* True, False $\in e$
	$\mid$	$\phi_1 \to \phi_2$	*implication:* $\phi_1 \Rightarrow \phi_2$
	$\mid$	$\phi \to \{v : \texttt{Proof} \mid \texttt{False}\}$	*negation:* $\neg\phi$
	$\mid$	$\texttt{PAnd}\ \phi_1\ \phi_2$	*conjunction:* $\phi_1 \wedge \phi_2$
	$\mid$	$\texttt{POr}\ \phi_1\ \phi_2$	*disjunction:* $\phi_1 \vee \phi_2$
	$\mid$	$x : a \to \phi$	*forall:* $\forall x.\phi$
	$\mid$	$(x :: a, \phi)$	*exists:* $\exists x.\phi$

Fig. 1. Encoding of Higher Order Logic in `Liquid Haskell` types. Function binders are not represented in negation and implication where they are not relevant.

1 ENCODING OF HIGHER ORDER LOGICS IN LIQUID HASKELL

`Liquid Haskell` can express arbitrary higher order properties, *i.e.,* has the same expressive power as Isabelle/HOL or Agda with a single universe type. For decidable type checking, refinements are first order, non-quantified expressions. We quantify refinements by encoding

- $\forall$ as a lambda abstraction and
- $\exists$ as a dependent pair

getting the HOL of Figure 1.

1.1 First Order Terms

The logical terms in `Liquid Haskell` are non-qualified Haskell expressions e as presented in Figure 1 of [?] (and defunctionalized in Figure 2 to the SMT logic). These expressions include constants, boolean operations, lambda abstractions, applications and in practice are extended to include decidable SMT theories, including non-qualified linear arithmetic and set theory. In the absence of reflected functions, reasoning over first order terms is automatically performed by the SMT-solver on decidable theories including linear arithmetic and congruence. When first order terms include reflected functions reasoning is performed via reflection of type level computations.

1.2 Implication

Implication $\phi_1 \Rightarrow \phi_2$ is encoded as a function from the proof of ϕ_1 to the proof of ϕ_2.

Implication Elimination. This encoding let us eliminate implication proofs by function application, thus safely encoding the natural deduction rule of modus ponens:

```
implElim :: p:Bool → q:Bool → {v:Proof | p} → ({v:Proof | p} → {v:Proof | q})
         → {v:Proof | q}
implElim _ _ p f = f p
```

Implication Refinement & Reification. If the formulas ϕ_1 and ϕ_2 are over basic expressions (non-qualified), that is $\phi_i \equiv \{v : \texttt{Proof} | e_i\}$, then implication can be directly encoded in the refinements as $\{v : \texttt{Proof} | e_1 \Rightarrow e_2\}$. We call this process refinement of the implication and the dual reification:

```
implRefine :: b1:Bool → b2:Bool
           → ({v:Proof | b1} → {v:Proof | b2})
           → {v:Proof | b1 ⇒ b2}
implRefine b1 _ fb
  | b1       = fb trivial
```

```
  | otherwise = trivial

implReify :: b1:Bool → b2:Bool
          → {v:Proof | b1 ⇒ b2}
          → ({v:Proof | b1} → {v:Proof | b2})
implReify _ _ b1b2 b1 = b1b2
```

1.3 Negation

Negation is encoded as an implication to the proof of false.

Negation Refinement & Reification. We reify negation by trivially proving using SMT that for each property `b` both `b` and its negation imply false.

```
type False = {v:Proof | false }

notReify :: b:Bool → {v:Proof | not b} → ({v:Proof | b} → False)
notReify _ notb b = trivial
```

To refine the negation of a property `b`, if `b` holds, then we apply its negation to get `false`., otherwise, the negation of `b` is trivially true.

```
notRefine :: b:Bool → ({v:Proof | b} → False) → {v:Proof | not b}
notRefine b f
  | b         = f trivial
  | otherwise = trivial
```

1.4 Conjunction

HERE HERE

```
data PAnd a b = PAnd a b

andRefine :: b1:Bool → b2:Bool → PAnd {v:Proof | b1} {v:Proof | b2}
          → {v:Proof | b1 ∧ b2 }
andRefine _ _ (PAnd b1 b2) = b1 ∧& b2


andReify :: b1:Bool → b2:Bool → {v:Proof | b1 ∧ b2 }
         → PAnd {v:Proof | b1} {v:Proof | b2}
andReify _ _ b = PAnd b b


andIntro :: b1:{Bool | b1} → b2:{Bool | b2} → PAnd {v:Proof | b1} {v:Proof | b2}
andIntro b1 b2 = PAnd trivial trivial

andElimLeft :: b1:Bool → b2:Bool → PAnd {v:Proof | b1} {v:Proof | b2}
            → {v:Proof | b1 }
andElimLeft _ _ (PAnd b1 b2) = b1

andElimRight :: b1:Bool → b2:Bool → PAnd {v:Proof | b1} {v:Proof | b2}
             → {v:Proof | b2 }
```

```
andElimRight _ _ (PAnd b1 b2) = b2
```

1.5 Disjunction

```
data POr a b = POrLeft a | POrLeft b

orRefine :: b1:Bool → b2:Bool
         → POr {v:Proof | b1} {v:Proof | b2}  ]
         → {v:Proof | b1 || b2 }
orRefine _ _ (POrLeft  p1) = p1
orRefine _ _ (POrRight p2) = p2

orReify :: b1:Bool → b2:Bool
        → {v:Proof | b1 || b2 }
        → POr {v:Proof | b1} {v:Proof | b2}
orReify b1 b2 p
  | b1 = POrLeft  p
  | b2 = POrRight p


orIntroLeft :: b1:Bool → b2:Bool → {v:Proof | b1}  → POr {v:Proof | b1} {v:Proof |
     b2}
orIntroLeft _ _ p = POrLeft p


orIntroRight :: b1:Bool → b2:Bool → {v:Proof | b2}  → POr {v:Proof | b1} {v:Proof
     | b2}
orIntroRight _ _ p = POrRight p


orElim :: p:Bool → q:Bool → r:Bool
       → POr {v:Proof | p} {v:Proof | q}
       → ({v:Proof | p} → {v:Proof | r})
       → ({v:Proof | q} → {v:Proof | r})
       → {v:Proof | r} @-}
orElim _ _ _ (POrLeft  p) fp _ = fp p
orElim _ _ _ (POrLeft q) _ fq = fq q
```

1.6 Forall

Forall $\forall x.\phi$ is encoded as a lambda abstraction $x : a \to \phi$.

```
forallElim :: p:(a → Bool) → (x:a → {v:Proof | p x} ) → y:a → {v:Proof | p y}
forallElim _ f y = f y

forallIntro :: p:(a → Bool) → (t:a → {v:Proof | p t}) → (x:a → {v:Proof | p x})
```

```
forallIntro _ f = f
```

1.7 Exists

Existentials $\exists x.\phi$ is encoded as a dependent pair: a pair that contains x and a proof of a formula that depends on the first element x. In `Liquid Haskell` we name the first element of the pair as `(x::a, ϕ)`. Internally dependent pairs are implemented via Abstract Refinement Types, while preserving decidable type checking.

```
existsIntro :: p:(a → Bool)
            → x:a → {v:Proof | p x}
            → (y::a,{v:Proof | p y})
existsIntro p x prop = (x, prop)

existsElim :: x:Bool → p:(a → Bool) → (t::a,{v:Proof | p t})
           → (s:a → {v:Proof | p s}
           → {v:Proof | x})
           → {v:Proof | x } @-}

existsElim x p (t, pt) f = f t pt
```

2 EXAMPLES

We present some proofs of higher order propertied and present how such properties can extend specific theories (like lists). These and more examples can be found in https://github.com/nikivazou/LiquidHOL.

2.1 Existentials over disjunction

We prove distribution of existentials over disjunction:

$$(\exists x.(f\ x \vee g\ x)) \Rightarrow ((\exists x.f\ x) \vee (\exists x.g\ x)))$$

The proof proceeds by existential case splitting and introduction:

```
existsOrDistr :: f:(a → Bool) → g:(a → Bool)
              → (x::a, POr {v:Proof | f x} {v:Proof | g x})
              → POr (x::a, {v:Proof | f x}) (x::a, {v:Proof | g x})
existsOrDistr f g (x,POrLeft fx)  = POrLeft  (x,fx)
existsOrDistr f g (x,POrRight fx) = POrRight (x,fx)
```

2.2 Foralls over conjunction

We prove distribution of foralls over conjunction:

$$(\forall x.(f\ x \wedge g\ x)) \Rightarrow ((\forall x.f\ x) \wedge (\forall x.g\ x)))$$

The proof proceeds by forall introduction and elimination:

```
forallAndDistr :: f:(a → Bool) → g:(a → Bool)
               → (x:a → PAnd {v:Proof | f x} {v:Proof | g x})
               → PAnd (x:a → {v:Proof | f x}) (x:a → {v:Proof | g x})
forallAndDistr f g andx
  = PAnd (\x → case andx x of PAnd fx _ → fx)
         (\x → case andx x of PAnd _ gx → gx)
```

2.3 Forall - exists over implication

We prove distribution of foralls over conjunction:

$$(\forall x.\exists y.(p\ x \Rightarrow q\ x\ y)) \Rightarrow (\forall x.(p\ x \Rightarrow (\exists y.q\ x\ y)))$$

The proof proceeds by forall elimination and existential introduction:

```
forallExistsImpl :: p:(a → Bool) → q:(a → a → Bool)
  → (x:a → (y::a, {v:Proof | p x} → {v:Proof | q x y} ))
  → (x:a → ({v:Proof | p x} → (y::a, {v:Proof | q x y})))
forallExistsImpl p q f x px
  = case f x of
      (y, pxToqxy) → (y,pxToqxy px)
```

2.4 Even lists

As a last example we see how quantifiers interact with the reflected functions by proving that forall lists xs if there exists a ys so that xs == ys ++ ys then xs has even length.

$$(\forall xs.\exists ys.xs\ = ys{+}{+}ys) \Rightarrow (\exists n.\mathtt{length} xs = n + n))$$

The proof proceeds by existential elimination and introduction, and by invocation of the `lenAppend` lemma.

```
even_lists :: xs:List a → (ys::List a, {v:Proof | xs == ys ++ ys })
           → (n::Int, {v:Proof | length xs == n + n})
even_lists xs (ys,pf) = (length ys, lenAppend ys ys ∧& pf)

lenAppend :: xs:List a → ys:List a → {length (xs ++ ys) == length xs + length ys}
lenAppend N _           = trivial
lenAppend (Cons x xs) ys = lenAppend xs ys
```