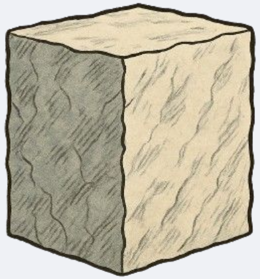
Large Elimination and Indexed Types in Refinement Types

Alessio Ferrarini and Niki Vazou

IMDEA Software Institute, Madrid, Spain
alessio.ferrarini@imdea.org

Refinement types and Liquid Haskell



`Int` `{ v:Int | v % 2 = 0 }`

Refinement type = Base type + predicate

In Liquid Haskell predicates are expressions **(no quantifiers)**

Programming and Proving in LH

```
data List a = Nil | Cons a (List a)

(++) :: List a → List a → List a
Nil           ++ ys = ys
(Cons x xs) ++ ys = Cons x (xs ++ ys)

{-@ appendAssoc :: xs:List a → ys:List a → zs:List a
                → { (xs ++ ys) ++ zs = xs ++ (ys ++ zs) } @-}
appendAssoc Nil          ys zs = trivial
appendAssoc (Cons x xs) ys zs = appendAssoc xs ys zs
```

Programming and **Proving** in LH

```haskell
data List a = Nil | Cons a (List a)

(++) :: List a → List a → List a
Nil         ++ ys = ys
(Cons x xs) ++ ys = Cons x (xs ++ ys)

{-@ appendAssoc :: xs:List a → ys:List a → zs:List a
                → { (xs ++ ys) ++ zs = xs ++ (ys ++ zs) } @-}
appendAssoc Nil         ys zs = trivial
appendAssoc (Cons x xs) ys zs = appendAssoc xs ys zs
```

Ok! Now let's do some proofs about the lambda calculus!

Simply Typed Lambda Calculus

$$(\textsc{App})\ \frac{\Gamma \vdash t_1 : \sigma \to \tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1\ t_2 : \tau}$$

$$(\textsc{Lam})\ \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$$

$$(\textsc{Var})\ \frac{(x, \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

Simply Typed Lambda Calculus in LH

```
data Term where
  {-@ App :: σ:Ty → τ:Ty → γ:Ctx → Prop (Term γ (TArrow σ τ))
          → Prop (Term γ σ) → Prop (Term γ τ) @-}
  {-@ Lam :: σ:Ty → τ:Ty → γ:Ctx → Prop (Term (Cons σ γ) τ)
          → Prop (Term γ (TArrow σ τ)) @-}
  {-@ Var :: σ:Ty → γ:Ctx → Prop (Ref σ γ) → Prop (Term γ σ) @-}
```

Simply Typed Lambda Calculus in LH

```
data Term where
  {-@ App :: σ:Ty → τ:Ty → γ:Ctx → Prop (Term γ (TArrow σ τ))
          → Prop (Term γ σ) → Prop (Term γ τ) @-}
  {-@ Lam :: σ:Ty → τ:Ty → γ:Ctx → Prop (Term (Cons σ γ) τ)
          → Prop (Term γ (TArrow σ τ)) @-}
  {-@ Var :: σ:Ty → γ:Ctx → Prop (Ref σ γ) → Prop (Term γ σ) @-}
```

We are not adding indexes to the system

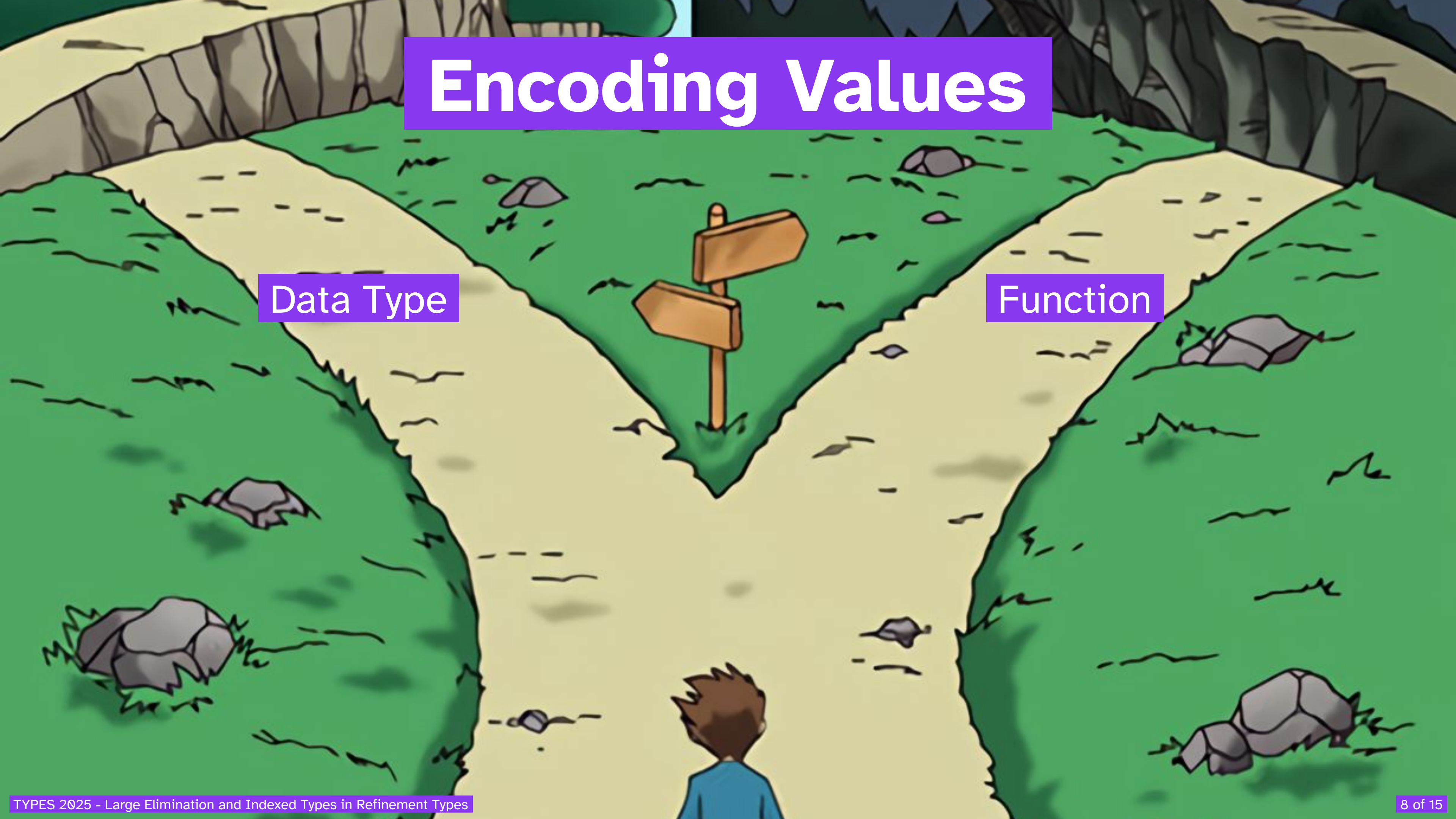
```
{-@ type Prop E = { v:_ | prop v = E } @-}
```

We represent index trough refinements

Representation of Values

$$\mathit{val}(\mathit{TInt}) = \mathbb{Z}$$
$$\mathit{val}(\mathit{TArrow}\ \sigma\ \tau) = \mathit{val}(\sigma) \to \mathit{val}(\tau)$$

Encoding Values



The Path of Functions

$$val = El_{Ty}(Int.\ \mathbb{Z},\ (\sigma, \tau).\ \sigma \to \tau) : \Pi(Ty, Type)$$

The Path of Functions

$$val = El_{Ty}(Int.\ \mathbb{Z},\ (\sigma, \tau).\ \sigma \rightarrow \tau) : \Pi(Ty, \underline{Type})$$

We are producing a type

The Path of Functions

$$val = El_{Ty}(Int.\ \mathbb{Z},\ (\sigma, \tau).\ \sigma \to \tau) : \Pi(Ty, \underline{Type})$$

We are producing a type, does it make sense in RT?

```
val :: Ty → ⁇
```

The Path of Functions

$$val = El_{Ty}(Int.\ \mathbb{Z},\ (\sigma,\tau).\ \sigma \to \tau) : \Pi(Ty, Type)$$

We are producing a type, does it make sense in RT?

```
val :: Ty → ⁇
```

The difference between DT and RT lies in what is “dependent”

```
{-@ val :: τ:Ty → { v:⁇ | … } @-}
```

The Path of Functions

$$val = El_{Ty}(Int.\ \mathbb{Z},\ (\sigma, \tau).\ \sigma \to \tau) : \Pi(Ty, \underline{Type})$$

NOT POSSIBLE WITH REFINEMENT TYPES!

We are producing a type, does it make se… RT?

```
val :: Ty → ??
```

The difference betw… and RT lies in what is “dependent”

```
{-@ val :: τ:Ty → { v:?? | … } @-}
```

Ok, let's try with Data Types!

The Path of Data Types

```
data Value where
  {-@ VInt :: Int → Prop (Value TInt) @-}
  {-@ VFun :: σ:Ty → τ:Ty → (Prop (Value σ) → Prop (Value τ))
           → Prop (Value (TArrow σ τ)) @-}
```

The Path of Data Types

```
data Value where
  {-@ VInt  :: Int → Prop (Value TInt) @-}
  {-@ VFun  :: σ:Ty → τ:Ty → (Prop (Value σ) → Prop (Value τ))
            → Prop (Value (TArrow σ τ)) @-}
```

Negative occurrence! 🚨

The Path of Data Types

```
data Value where
  {-@ VInt :: Int → Prop (Value TInt) @-}
  {-@ VFun :: σ:Ty → τ:Ty → (Prop (Value σ) → Prop (Value τ))
          → Prop (Value (TArrow σ τ)) @-}
```

Negative occurrence! 🚨

Unfortunately, Coq rejects this definition because it violates the *strict positivity requirement* for inductive definitions, which says that the type being defined must not occur to the left of an arrow in the type of a constructor argument. Here, it is the third argument to `R_arrow`, namely `(∀ s, R T1 s → R TS (app t s))`, and specifically the `R T1 s` part, that violates this rule. (The outermost arrows separating the constructor arguments don't count when applying this rule; otherwise we could never have genuinely inductive properties at all!) The reason for the rule is that types defined with non-positive recursion can be used to build non-terminating functions, which as we know would be a disaster for Coq's logical soundness. Even though the relation we want in this case might be perfectly innocent, Coq still rejects it because it fails the positivity test.

Fortunately, it turns out that we *can* define `R` using a `Fixpoint`:

Software Foundations Vol. 2 - Programming Language Foundations - Pierce et al

The Path of Data Types

```
data Value where
  {-@ VInt :: Int → Prop (Value TInt) @-}
  {-@ VFun :: σ:Ty → τ:Ty → (Prop (Value σ) → Prop (Value τ))
           → Prop (Value (TArrow σ τ)) @-}
```

Negative occurrence! 🚨

Dependently typed PL rejects this definition because it violates the *strict positivity requirement* for inductive definitions, which says that the type being defined must not occur to the left of an arrow in the type of a constructor argument. Here, it is the third argument to `R_arrow`, namely `(∀ s, R T1 s → R TS (app t s))`, and specifically the `R T1 s` part, that violates this rule. (The outermost arrows separating the constructor arguments don't count when applying this rule; otherwise we could never have genuinely inductive properties at all!) The reason for the rule is that types defined with non-positive recursion can be used to build non-terminating functions, which as we know would be a disaster for Coq's logical soundness. Even though the relation we want in this case might be perfectly innocent, Coq still rejects it because it fails the positivity test.

Fortunately, it turns out that we *can* define R using a **Function**

Software Foundations Vol. 2 - Programming Language Foundations - Pierce et al.

Why are Negative Occurrences Bad?

```
data Bad where
  MkBad :: (Bad → Void) → Bad
```

Why are Negative Occurrences Bad?

```
data Bad where
  MkBad :: (Bad → Void) → Bad
```

We are not guaranteed that there exists an initial algebra

If there is no initial algebra, like in this case then, we can use Bad to construct non terminating terms

Why are Negative Occurrences Bad?

```
data Bad where
  MkBad :: (Bad → Void) → Bad
```

We are not guaranteed that there exists an initial algebra

If there is no initial algebra, like in this case then, we can use Bad to construct non terminating terms

The strict positivity condition implies that the initial algebra exists

An Unequal Treatment

Data Types

Functions

An Unequal Treatment

Data Types

“Well definedness” enforced by the positivity checker

Functions

“Well definedness” enforced by the termination checker

An Unequal Treatment

Data Types

“Well definedness” enforced by the positivity checker

Very restrictive

Functions

“Well definedness” enforced by the termination checker

Spot on

The positivity check almost feels like disallowing recursive functions

“Smaller” Negativity is Ok

```
data Value where
  {-@ VInt :: Int → Prop (Value TInt) @-}
  {-@ VFun :: σ:Ty → τ:Ty → (Prop (Value σ) → Prop (Value τ))
              → Prop (Value (TArrow σ τ)) @-}
```

Not a single type, but an infinite family of types

“Smaller” Negativity is Ok

```
data Value where
  {-@ VInt :: Int → Prop (Value TInt) @-}
  {-@ VFun :: σ:Ty → τ:Ty → (Prop (Value σ) → Prop (Value τ))
           → Prop (Value (TArrow σ τ)) @-}
```

Not a single type, but an infinite family of types

If i < j then values at index j don’t influence values at i, the type is technically a constant from the point of view of the current constructor

"Smaller" Negativity is Ok

```
data Value where
  {-@ VInt :: Int → Prop (Value TInt) @-}
  {-@ VFun :: σ:Ty → τ:Ty → (Prop (Value σ) → Prop (Value τ))
              → Prop (Value (TArrow σ τ)) @-}
```

Not a single type, but an infinite family of types

If i < j then values at index j don't influence values at i, the type is technically a constant from the point of view of the current constructor

In this case we pick the partial order induced structurally on types

Independent of refinement typed (May also work in DTT)

Problem

Refinement types aren’t expressive enough for some proofs

Solution

Inductive data types with “smaller” negative occurrences

Challenges

Is the system consistent? Model?

Problem

Refinement types aren’t expressive enough for some proofs

Solution

Inductive data types with “smaller” negative occurrences

Challenges

Is the system consistent? Model?

THANKS FOR YOUR ATTENTION!

alessio.ferrarini@imdea.org

Try Liquid Haskell online! https://liquidhaskell.goto.ucsd.edu/index.html

ACKNOWLEDGMENTS

Partially funded by the European Union (GA 101039196). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the European Research Council can be held responsible for them.

Extra slides

What are we missing out on?

$$\lambda\ x.\ ElList(\mathbb{N}, (h, acc).\ \Pi(h, acc)) : \Pi(List\ Type, Type)$$

Issue down to the usual issue that we can’t manipulate types

We can’t reason about types generally we need to give a grammar to construct them, ex. lambda calculus values

Arity polymorphism

```
data Env where
  {-@ Empty :: Prop (Env Nil) @-}
  Empty :: Env
  {-@ With :: σ:Ty → γ:Ctx → Prop (Value σ) → Prop (Env γ)
           → Prop (Env (Cons σ γ)) @-}
  With  :: Ty → Ctx → Value → Env → Env

{-@ eval :: σ:Ty → γ:Ctx → t:Prop (Term γ σ) → Prop (Env γ)
         → Prop (Value σ) @-}
eval :: Ty → Ctx → Term → Env → Value
```

Arity polymorphism can be obtained through indexed lists

Is some sort of uncurried representation