Functional Extensionality as a Refinement Type

Niki Vazou

IMDEA Software Institute
niki.vazou@imdea.org

Refinement types have been used to encode equational reasoning and prove equalities of functions. For example, in [VBK+18] we used Liquid Haskell to prove that two functions, say `slow` and `fast` behave the same for all inputs, as captured by the following refinement type.

```
lemma :: x:Int → { fast x == slow x }
```

The type of `lemma` states the theorem that for all inputs `x` the result of `fast x` and `slow x` are equal. To prove that lemma we use equational reasoning to define an inhabitant of the function.

To derive function equality from the above lemma, we need functional extensionality, which can naturally be encoded with the following refinement type.

```
extensionality :: f:(a → b) → g:(a → b) → (x:a → {f x == g x}) → {f == g}
```

That is, for each functions `f` and `g`, given a proof that forall `x`, `f x` equal `g x`, `f` is equal to `g`. Extensionality cannot have an inhabitant, since there is no available value of type `a` to "unlock" the argument lemma. Still, we can assume the above type and use it to prove equality of functions. For example, below we call `exteniolality` to prove equality of `fast` and `slow`.

```
theoremEq :: { fast == slow }
theoremEq = extensionality fast slow lemma
```

To our surprise, the combination of funcional extensionality and liquid types is incompatible. Next we explain the incompatibility and propose a sound, yet imprecise solution.[1]

From Type to Implication Checking: Refinement typing rules reduce type to implication checking. We explain this reduction when checking the `theoremEq` above. For generality, we abstract the types of the involved functions as follows:

```
slow  :: {x:Int | d_sl x } → {v:Int | r_sl x v }
fast  :: {x:Int | d_fs x } → {v:Int | r_fs x v }
lemma :: {x:Int | d_lm x } → {v:() | prop x }
```

That is, `slow` and `fast` are respectively defined over the domains $\texttt{d}_{sl}$ and $\texttt{d}_{fs}$ and ranges $\texttt{r}_{sl}$ and $\texttt{r}_{fs}$. Lemma is defined on the domain $\texttt{d}_{lm}$ and proves a property `prop` on `x` that should imply function equality. For simplicity we assume both functions are on `Int`s.

Since `extensionality` is polymorphic, its call instantiates the parametric types. Instantiation happens with refined types, so the domain type variable `a` gets instantiated with `{x:Int |` $\texttt{k}_a$ `}` and the range type variable `b` with `{y:Int |` $\texttt{k}_b$ `}`, where $\texttt{k}_a$ and $\texttt{k}_b$ are refinement variables.

```
theoremEq = extensionality @{x:Int | k_a } @{y:Int | k_b } fast slow lemma
```

After the implicit type instantiation, extensionality has the type below.

```
extensionality @{v:Int | k_a } @{v:Int | k_b }
  :: f:({v:Int | k_a } → {v:Int | k_b }) → g:({v:Int | k_a } → {v:Int | k_b })
  → (x:{v:Int | k_a } → { f x == g x }) → {f == g}
```

[1] Since functional extensionality is not supported by SMTs, its axiomatization is required by any SMT-based verifier. Both Dafny and Fstar have axiomatization of extensionality, while the later also used special treatment to address initial unsoundness https://github.com/FStarLang/FStar/issues/1542.

Application of `fast` checks subtyping of the actual and expected types, as follows:

$$\frac{\dfrac{\forall x. k_a \Rightarrow d_{fs}\ x}{\{x:\texttt{Int} \mid k_a\} \preceq \{x::\texttt{Int} \mid d_{fs}\ x\}} \qquad \dfrac{\forall x\ v. k_a \Rightarrow r_{fs}\ x\ v \Rightarrow k_b}{\{x:\texttt{Int} \mid k_a\} \vdash \{v:\texttt{Int} \mid r_{fs}\ x\ v\} \preceq \{v:\texttt{Int} \mid k_b\}}}{\{x:\texttt{Int} \mid d_{fs}\ x\} \rightarrow \{v:\texttt{Int} \mid r_{fs}\ x\ v\} \preceq \{x:\texttt{Int} \mid k_a\} \rightarrow \{v:\texttt{Int} \mid k_b\}}$$

That is, application of the function `fast` imposes two logical constraints: 1) the variable $\texttt{k}_a$ should imply the domain of `fast` and 2) assuming $\texttt{k}_a$, the range of `fast` should imply $\texttt{k}_b$. Similarly, checking the application of the functions `slow` and `lemma` reduces two logical constraints. In all, checking the application of the extensionality axiom reduces to the checking the validity of the set of the 6 implications below with the two unknowns k_a and k_b.

$$\begin{array}{lll}
\forall x. & k_a & \Rightarrow d_{fs}\ x & (1)\\
\forall x. & k_a & \Rightarrow d_{sl}\ x & (2)\\
\forall x. & k_a & \Rightarrow d_{lm}\ x & (3)\\
\forall x\ v. & k_a & \Rightarrow r_{fs}\ x\ v \Rightarrow k_b & (4)\\
\forall x\ v. & k_a & \Rightarrow r_{sl}\ x\ v \Rightarrow k_b & (5)\\
\forall x. & k_a & \Rightarrow \texttt{proved}\ x \Rightarrow \texttt{fast}\ x == \texttt{slow}\ x & (6)
\end{array}$$

The first three implications encode unification of the functions domains. The next two encode unification of the functions ranges. The last states that `proved` should imply function equality.

The last step on type checking the application of `extensionality` is solving this set of implications with respect to the two unknowns $\texttt{k}_a$ and $\texttt{k}_b$.

Unsound Implication Solving: Liquid types [RKJ08] presents a fixpoint algorithm to solve exactly such (in general recursive) set of constraints. The algorithm generates the strongest solutions for all refinement variables, which, for our system of implication is $k_a, k_b := false$.

This solution makes the set of implications valid, for any property `proved` x. That is, the liquid types inference algorithm will unsoundly accept any calls to `extensionality` even when the proof argument `lemma` is too weak to imply function equality.

This observation does not imply unsoundness of liquid types. Instead it means that axiomatization of extentionality is incompatible with the liquid types framework.

Solution: Inspecting the type of extensionality we observe that the parametric type variable `a` only appears in positive positions. When `a` gets instantiated with a concrete type `{v:b | k`$_a$`}`, $\texttt{k}_a$ will only appear on the left hand side of the reduced implications – because of the positivity of `a` – thus can be soundly solved to true.

This trivial solution is currently used by Liquid Haskell. Concretely, in the above implication system, Liquid Haskell solves, $k_a := true, , k_b := r_{fs}\ x\ v \wedge r_{sl}\ x\ v$. That is, the validity of the implication system now actually depends on implication (6) and can be valid *only if* $\forall x.\texttt{proved}\ x \Rightarrow \texttt{fast}\ x == \texttt{slow}\ x$.

Imprecision: This solution is sound and compatible with extentionality, but it is imprecise. The implications (1-3) are only valid *iff* the domains of the functions `fast` and `slow` are true. This restriction complies with the semantics of function extentionality that requires the domains of functions to be equal. Yet, it is too restrictive, since it does not allow extentionality to be applied to functions with refined domains.

Conclusion: We presented that functional extensionality is incompatible with liquid type inference. To allow compatibility we solve to true refinement variables derived from positive, parametric type variables. This solution is imprecise, since extensionality can only be used on functions with unrefined domains and we are investigating on a both precise and sound solution.

References

[RKJ08] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

[VBK+18] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: Equational reasoning in liquid haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, pages 132–144, New York, NY, USA, 2018. ACM.