Large Elimination and Indexed Types in Refinement Types

Alessio Ferrarini[1] and Niki Vazou[2]

[1] IMDEA Software Institute, Madrid, Spain alessio.ferrarini@imdea.org
[2] IMDEA Software Institute, Madrid, Spain niki.vazou@imdea.org

Abstract

We explore how to approximate large elimination and indexed datatypes in refinement type systems with a non-dependent base type system.

Introduction. At the core of modern dependent type theories [2], indexed inductive types and large elimination are the main means of how new datatypes are introduced. They also play a vital role in the kind of theorems that are provable in our theory. As an example, removing large elimination from MLTT [4] causes the loss of the ability to prove the disjointness of constructors [5]. This makes the theory unable to prove trivial theorems; for example, we cannot show that $\Pi(Eq(Bool, true, false), Void)$ is inhabited.

Refinement types as implemented in Liquid Haskell [8, 9] extend the type system of Haskell, SystemFC, with logical predicates. These predicates are Haskell (terminating) boolean expressions and operators that belong to the decidable fragment of SMT-Lib.

As refinement type systems lack direct support for large elimination and indexed datatypes, their absence makes it challenging to encode certain proofs and datatype definitions. In this work, we explore how to approximate these features within the refinement type system. As a running example, we translate the correct compiler of STLC to the SKI calculus presented in [6] from Agda [7] to Liquid Haskell.[1]

```
data Term : Ctx → Ty → Set where
  app : ∀ {Γ σ τ} → Term Γ (σ ⇒ τ)
      → Term Γ σ → Term Γ τ
  lam : ∀ {Γ σ τ} → Term (σ :: Γ) τ
      → Term Γ (σ ⇒ τ)
  var : ∀ {Γ σ} → Ref σ Γ → Term Γ σ
```

(a) Agda definition

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

(b) Liquid Haskell definition

Figure 1: Well-typed STLC terms definition

[1] The full translation can be found at https://github.com/ucsd-progsys/liquidhaskell/blob/develop/tests/ple/pos/SKILam.hs.

```
data Ty : Set where
    ι    : Ty
    _⇒_  : Ty → Ty → Ty

Value : Ty → Set
Value ι        = ℤ
Value (σ ⇒ τ)  = Value σ → Value τ
```

(a) Agda definition

```
data Value where
{-@ VIota :: Int → Prop (Value Iota) @-}
  VIota :: Int → Value
{-@ VFun :: σ:Ty → τ:Ty
         → (Prop (Value σ) → Prop (Value τ))
         → Prop (Value (Arrow σ τ)) @-}
  VFun :: Ty → Ty → (Value → Value)
       → Value
data VALUE = Value Ty
```

(b) Liquid Haskell definition

Figure 2: HOAS representation of values

Indexed Inductive Datatypes. A first solution for the absence of indexed inductive datatypes was proposed in [1] under the name of *data proposition*. The idea is to encode the information and constraints that in the dependently typed world is carried through indexes by refining the constructors of the datatype. In Liquid Haskell, this role is taken by the Prop type, which is a type alias for the refined type: `type Prop e = {v:a | e=` prop v. `Here, prop is treated as an uninterpreted function in the`logic and `e` is substituted by the index information. Since the logic has no knowledge about `prop` other than its type, two types can match only if the arguments passed to `Prop` are the same. In Figure 1, we see side by side the datatypes that encode well-typed terms of the simply typed lambda calculus.[2]

Large elimination. Large elimination instead is trickier as the natural solution would be to encode it through functions, but unfortunately this is not feasible as functions need to remain valid Haskell code and types are not part of valid Haskell expressions. A solution could be to use codes to encode types through codes mimicking the universe level encoding in dependent types but at the Haskell level there is no way to have a Haskell type depend on a value. The only possible solution then is to use a datatype declaration indexed by the argument of the dependent elimination where we have a constructor for each case and in each constructor we wrap a value of the type returned by the elimination. However this leads to problem when we construct an arrow type, as the same type will appear in a negative position, and it will fail to type check. But are these type declaration actually problematic? The type definition is actually well-founded as the index is structurally decreasing, otherwise the same declaration would be rejected by Agda. Thus, at least, the usual example showing that negative types lead to inconsistency does not apply. Another feature obtained via large elimination is *non-uniform dependencies*, i.e., functions are able to inspect their argument and change their type accordingly. The usual example is the type safe `sprintf`, but this kind of pattern can still be emulated through indexed families even if in a less elegant manner through sized lists.

In Figure 2 we compare the HOAS representation of values. In the Liquid Haskell translation, the `VFun` constructor has the negative occurrences issue.

Completing the Connection. Naturally, we can ask if this transformation always works and what is the relationship between the original type and its refined encoding? If these questions have satisfactory answers, we would then ask if any statement encodable with dependent types

[2] In the (Liquid) Haskell definition we use γ instead of Γ to denote contexts as as variable names cannot begin with capital letters.

can be translated into refinement types. The work in [3] shows that $ATTT$, a variant of System F with refinements, from an expressiveness perspective corresponds to second-order logic. However, it is unclear whether the notion of refinement in $ATTT$ aligns with that of Liquid Haskell.

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.

References

[1] M. H. Borkowski, N. Vazou, and R. Jhala. Mechanizing Refinement Types. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 2024.

[2] T. Coquand and C. Paulin. Inductively defined types. *Lecture Notes in Computer Science*, 1990.

[3] S. Hayashi. Logic of refinement types. In *Types for Proofs and Programs*, 1994.

[4] P. Martin-Löf. An intuitionistic theory of types: Predicative part. *Logic Colloquium*, 1975.

[5] J. M. Smith. The independence of peano's fourth axiom from martin-löf's type theory without universes. *The Journal of Symbolic Logic*, 1988.

[6] W. Swierstra. A correct-by-construction conversion from lambda calculus to combinatory logic. *Journal of Functional Programming*, 2023.

[7] The Agda Development Team. *Agda wiki*, 2024.

[8] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2014.

[9] N. Vazou, A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala. Refinement Reflection: Complete Verification with SMT. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.