Simulating large eliminations in Cedille

Christopher Jenkins, Andrew Marmaduke, and Aaron Stump

The University of Iowa, Iowa City, Iowa, U.S.A.
{firstname-lastname}@uiowa.edu

1 Introduction

In dependently typed programming languages, large eliminations allow programmers to define types by induction over datatypes — that is, as an elimination of a datatype into the large universe of types. This provides an expressive mechanism for arity- and data-generic programming [7]. However, as large eliminations are closely tied to a type theory's primitive notion of inductive type, this expressivity is not expected within polymorphic pure typed lambda calculi in which datatypes are encoded using impredicative quantification.

Seeking to overcome historical difficulties of impredicative encodings, the *calculus of dependent lambda eliminations* (CDLE) [5, 6] extends the Curry-style (i.e., extrinsically typed) *calculus of constructions* (CC) [1] with three type constructs that together enable the derivation of induction for impredicative encodings of datatypes (Geuvers [3] showed this was not possible for CC). In this paper, we report progress on overcoming another difficulty: the lack of large eliminations for these encodings. We show that the expected computation rules for a large elimination, expressed using a *derivable* notion of extensional equality for types, can be proven within CDLE. We outline our method with a definition of n-ary functions in the remainder of this paper; omitted are many other examples and a generic formulation of the method for the Mendler-style encodings of the framework of Firsov et al. [2]. These results have been mechanically checked by Cedille, an implementation of CDLE.

2 Simulating large eliminations: n-ary functions

Figure 1a shows the definition of *Nary*, the family of n-ary function types over some type T, as a large elimination of natural numbers *Nat*. Our method begins by approximating this inductive definition of a *function* as an inductive *relation* between *Nat* and types, given as *NaryR* in Figure 1b. This approximation is inadequate: we lack a canonical name for the type *Nary* n because n does not *a priori* determine the type argument of *NaryR* n. In fact, without a method of proof discrimination we are unable to define a function of type $\forall N.\, \mathit{NaryR}\ \mathit{zero}\ N \to N \to T$ to extract a 0-ary term of type T. One would need to handle the impossible *naryRS* case (reaching this case implies $\{\mathit{zero} \simeq \mathit{suc}\ n\}$ for some n). CDLE provides such a discriminator with the δ axiom [6] for its primitive equality type, allowing one to abort impossible cases.

(a) As a large elimination

```
Nary : Nat → ⋆
Nary zero = T
Nary (suc n) = T → Nary n
```

(b) As a GADT

```
data NaryR : Nat → ⋆ → ⋆
= naryRZ : NaryR zero T
| naryRS : ∀ n,Y. NaryR n Y → NaryR (suc n) (T → Y)
```

Figure 1: n-ary functions over T

Our task is to show that *NaryR* defines a *functional* relation, i.e., for all $n : Nat$ there exists a unique type *Nary* n such that *NaryR* n (*Nary* n) is inhabited. Using implicit products (c.f. Miquel [4]), a candidate for *Nary* can be defined in CDLE as:

```
Nary = λ n: Nat. ∀ X: ⋆. NaryR n X ⇒ X
```

For all n, read *Nary* n as the type of terms contained in the intersection of the family of types X such that *NaryR* n X is inhabited. For example, every term of type *Nary zero* has type T (since T is in this family), and every term of type T has type *Nary zero* (by induction on the assumed proof of *NaryR zero* X for arbitrary X). However, at the moment we are stuck when attempting to prove *NaryR zero* (*Nary zero*). Though we see that T and *Nary zero* are *extensionally* equal types (they classify the same terms), using *naryRZ* requires that they be *definitionally* equal!

$$\frac{\Gamma \vdash \lambda\, x.\, x : S \rightarrow T \quad \Gamma \vdash \lambda\, x.\, x : T \rightarrow S}{\Gamma \vdash \lambda\, x.\, x : \{S \cong T\}} \qquad \frac{\Gamma \vdash t : T_j \quad \Gamma \vdash t' : \{T_1 \cong T_2\} \quad i, j \in \{1,2\}, i \neq j}{\Gamma \vdash t : T_i}$$

Figure 2: Derived extensional equality of types

Figure 2 gives an axiomatic presentation of a derived type family expressing extensional type equality in CDLE. The introduction rule states that S and T are equal if the identity function can be assigned both the types $S \rightarrow T$ and $T \rightarrow S$, i.e., we can exhibit a two-way inclusion between the set of terms of type S and terms of type T. The elimination rule allows us to coerce the type of a term when that type is provably equal to another type. We change the definition of *Nary* so that its type index respects extensional type equality:

```
data NaryR : Nat → ⋆ → ⋆
= naryRZ : ∀ X. { X ≅ T } → NaryR zero X
| naryRS : ∀ n,Y,X. NaryR n Y → { X ≅ T → Y } → NaryR (suc n) X
```

With the move to an extensional notion of type equality, to show that *NaryR* is functional requires showing that it is *well-defined* with respect to this notion. These three properties — well-definedness, uniqueness, and existence — can be proven in CDLE. We show the types of these proofs below.

```
naryRWd : ∀ n,X1,X2. NaryR n X1 → { X1 ≅ X2 } → NaryR n X2
naryREq : ∀ n,X1,X2. NaryR n X1 → NaryR n X2 → { X1 ≅ X2 }
naryREx : Π n. NaryR n (Nary n)
```

From this, we prove that the computation laws of Figure 1a hold as extensional type equalities:

```
naryZC : { Nary zero ≅ T }
narySC : ∀ n. { Nary (succ n) ≅ T → Nary n }
```

The upshot is we can simulate large eliminations with two-way type inclusions between the left- and right-hand sides of such a definition. For example, the function *app* that applies an n-ary function to a length-indexed list of n elements of type T, written in Agda-like pseudocode as:

```
app : ∀ n. Nary n → Vec T n → T
app .zero f vnil = f
app .(succ n) f (vcons hd tl) = app n (f hd) tl
```

is typeable in CDLE using *naryZC* on f in the case for *vnil* and *narySC* in the case for *vcons*.

References

[1] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[2] Denis Firsov, Richard Blair, and Aaron Stump. Efficient Mendler-style lambda-encodings in Cedille. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 235–252, Cham, 2018. Springer International Publishing.

[3] Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *International Conference on Typed Lambda Calculi and Applications*, pages 166–181, Berlin, Heidelberg, 2001. Springer.

[4] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*, TLCA'01, page 344–359, Berlin, Heidelberg, 2001. Springer-Verlag.

[5] Aaron Stump. The calculus of dependent lambda eliminations. *J. Funct. Program.*, 27:e14, 2017.

[6] Aaron Stump and Christopher Jenkins. Syntax and semantics of Cedille. Manuscript, 2018.

[7] Stephanie Weirich and Chris Casinghino. Generic programming with dependent types. In Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 217–258. Springer, 2010.