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 program-
ming [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 deriva-
tion 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 ∀N. NaryR zero N → N → T
to extract a 0-ary term of type T. One would need to handle the impossible naryRS case
(reaching this case implies {zero ≃ 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 \( n : \text{Nat} \) 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:

\[
Nary = \lambda n : \text{Nat}. \forall X : \ast. NaryR n X \Rightarrow X
\]

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 \( Nary \text{zero} \) has type \( \text{T} \) (since \( \text{T} \) is in this family), and every term of type \( T \) has type \( Nary \text{zero} \) (by induction on the assumed proof of \( NaryR \text{zero} X \) for arbitrary \( X \)). However, at the moment we are stuck when attempting to prove \( NaryR \text{zero} (Nary \text{zero}) \). Though we see that \( \text{T} \) and \( \text{Nary zero} \) are extensionally equal types (they classify the same terms), using \( naryRZ \) requires that they be definitionally equal!

![Figure 2: Derived extensional equality of types](image-url)

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:

\[
\begin{align*}
\text{data NaryR : Nat} & \rightarrow \ast \rightarrow \ast \\
\text{= naryRZ : } & \forall X. \{ X \cong T \} \rightarrow NaryR \text{zero} X \\
\text{| naryRS : } & \forall n,Y,X. NaryR n Y \rightarrow \{ X \cong T \rightarrow Y \} \rightarrow NaryR (\text{suc} n) X
\end{align*}
\]

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.

\[
\begin{align*}
naryRD : & \forall n,X1,X2. NaryR n X1 \rightarrow \{ X1 \cong X2 \} \rightarrow NaryR n X2 \\
naryRE : & \forall n,Y,X. NaryR n X1 \rightarrow NaryR n X2 \rightarrow \{ X1 \cong X2 \} \\
naryREx : & \Pi n. NaryR n (Nary n)
\end{align*}
\]

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

\[
\begin{align*}
naryZC : & \{ Nary \text{zero} \cong T \} \\
narySC : & \forall n. \{ Nary (\text{suc} n) \cong T \rightarrow Nary n \}
\end{align*}
\]

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 \( \text{app} \) that applies an \( n \)-ary function to a length-indexed list of \( n \) elements of type \( T \), written in Agda-like pseudocode as:

\[
\begin{align*}
\text{app : } & \forall n. \text{Nary} n \rightarrow \text{Vec} T n \rightarrow T \\
\text{app .zero f vnil = f} \\
\text{app .(suc n) f (vcons hd tl) = app n (f hd) tl}
\end{align*}
\]

is typeable in CDLE using \( naryZC \) on \( f \) in the case for \( \text{vnil} \) and \( narySC \) in the case for \( \text{vcons} \).
Simulating large eliminations

Jenkins, Marmaduke, and Stump

References


