Simulating Large Eliminations in Cedille

Christa Jenkins
The University of Iowa, U.S.A.

Andrew Marmaduke
The University of Iowa, U.S.A.

Aaron Stump
The University of Iowa, U.S.A.

Abstract

Large eliminations provide an expressive mechanism for arity- and type-generic programming. However, as large eliminations are closely tied to a type theory's primitive notion of inductive type, this expressivity is not expected within polymorphic lambda calculi in which datatypes are encoded using impredicative quantification. We report progress on simulating large eliminations for datatype encodings in one such type theory, the *calculus of dependent lambda eliminations* (CDLE). Specifically, we show that the expected computation rules for large eliminations, expressed using a derived type of extensional equality of types, can be proven *within* CDLE. We present several case studies, demonstrating the adequacy of this simulation for a variety of generic programming tasks, and a generic formulation of the simulation allowing its use for a broad family of datatype encodings. All results have been mechanically checked by Cedille, an implementation of CDLE.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases large eliminations, generic programming, impredicative encodings, Cedille, Mendler algebra

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.7

Supplementary Material *Source Code*: `https://github.com/cedille/cedille-developments/`

1 Introduction

In dependently typed 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. For type theory semanticists, large eliminations rule out two-element models of types by providing a principle of proof discrimination (e.g., $0 \neq 1$)[26, 25]. For programmers, they give an expressive mechanism for arity- and type-generic programming with universe constructions [34]. As an example, the type $Nary\ n$ of n-ary functions (where n is a natural number) over type T can be defined as T when $n = 0$ and $T \to Nary\ n'$ when $n = succ\ n'$.

Large eliminations are closely tied to a type theory's primitive notion of inductive type. Thus, this expressivity is not expected within polymorphic pure typed lambda calculi in which datatypes are impredicatively encoded. The *calculus of dependent lambda eliminations* (CDLE) [27, 28] is one such theory that seeks to overcome historical difficulties of impredicative encodings, such as the lack of induction principles for datatypes [13].

Contributions In this paper, we report progress on overcoming another difficulty of impredicative encodings: the lack of large eliminations. We show that the expected definitional equalities of a large elimination can be simulated using a *derived type of extensional equality* for types (as CDLE is an extrinsic theory, we take the extent of a type to be the set of terms it classifies). In particular, we:

- describe our method for simulating large eliminations in CDLE (Section 3) using a concrete example, identifying the features of the theory that enable the development

(Section 2) and noting a limitation on the contexts in which it may be effectively used;
- formulate the method *generically* for all impredicative encodings of the form used by the datatype system of the Cedille tool (Section 5);
- demonstrate the adequacy of this simulation by applying it to several generic programming tasks: n-ary functions, a closed universe of datatypes, and an arity-generic map operation (Sections 3 and 4).

All results have been mechanically checked by Cedille, an implementation of CDLE, and are available in the supplementary material for this paper.

Outline Section 2 reviews background material on CDLE, focusing on the primitives which enable the simulation. In Section 3, we carefully explain the recipe for simulating large eliminations using as an example the type of n-ary functions over a given type. Section 4 shows two more case studies, a closed universe of strictly positive types and a generalized map operation for vectors, as evidence of the effectiveness of the simulation in tackling generic programming tasks. The recipe for concrete examples is then turned into a generic derivation (that is, parametric in a covariant datatype signature) of simulated large eliminations in Section 5. Finally, Section 6 discusses related work and Section 7 concludes with a discussion of future work.

2 Background on CDLE

In this section, we review CDLE, the kernel theory of Cedille. CDLE extends the impredicative extrinsically typed *calculus of constructions* (CC), overcoming historical difficulties of impredicative encodings (e.g., underivability of induction [14]) by adding three new type constructs: equality of untyped terms; the dependent intersections of Kopylov [20]; and the implicit products of Miquel [24]. The pure term language of CDLE is untyped lambda calculus, but to make type checking algorithmic terms t are presented with typing annotations which are removed during erasure (written $|t|$). Definitional equality of terms t_1 and t_2 is $\beta\eta$-equivalence modulo erasure of annotations, denoted $|t_1| =_{\beta\eta} |t_2|$.

The typing and erasure rules for the fragment of CDLE used in this paper are shown in Figure 1 and described in Section 2.1 (see also Stump and Jenkins [28]); the derived constructs we use are presented axiomatically in Section 2.2. We assume the reader is familiar with the type constructs inherited from CC: abstraction over types in terms is written $\Lambda\, X.\, t$ (erasing to $|t|$), application of terms to types (polymorphic type instantiation) is written $t \cdot T$ (erasing to $|t|$), and application of type constructors to type constructors is written $T_1 \cdot T_2$. In code listings, we sometimes omit type arguments to terms when Cedille can infer them.

2.1 Primitives

Below, we only discuss implicit products and the equality type. Though dependent intersections play a critical role in the derivation of induction for datatype encodings, they are otherwise not explicitly used in the coming developments.

The implicit product type $\forall\, x{:}T_1.\, T_2$ of Miquel [24] is the type for functions which accept an erased (computationally irrelevant) input of type T_1 and produce a result of type T_2. Implicit products are introduced with $\Lambda\, x.\, t$, and the type inference rule is the same as for ordinary function abstractions except for the side condition that x does not occur free in the erasure of the body t. Thus, the argument plays no computational role in the function and exists solely for the purposes of typing: the erasure of $\Lambda\, x.\, t$ is $|t|$. For application, if t has

$$\frac{\Gamma, x : T_1 \vdash t : T_2 \quad x \notin FV(|t|)}{\Gamma \vdash \Lambda\, x.\, t : \forall\, x{:}T_1.\, T_2} \qquad \frac{\Gamma \vdash t : \forall\, x{:}T_1.\, T_2 \quad \Gamma \vdash t' : T_1}{\Gamma \vdash t \text{ -}t' : [t'/x]T_2}$$

$$\frac{|t_1| =_{\beta\eta} |t_2|}{\Gamma \vdash \beta : \{t_1 \simeq t_2\}} \qquad \frac{\Gamma \vdash t : \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}}{\Gamma \vdash \delta \text{ - } t : T}$$

$$\frac{\Gamma \vdash t : \{t' \simeq t''\} \quad \Gamma \vdash t' : T \quad FV(t'') \subseteq dom(\Gamma)}{\Gamma \vdash \varphi\ t \text{ - } t'\ \{t''\} : T}$$

$$\begin{array}{lcllcl} |\Lambda\, x.\, t| & = & |t| & |t \text{ -}t'| & = & |t| \\ |\beta| & = & \lambda\, x.\, x & |\varphi\ t \text{ - } t'\ \{t''\}| & = & |t''| \\ |\delta \text{ - } t| & = & \lambda\, x.\, x & & & \end{array}$$

Figure 1 Typing and erasure for a fragment of CDLE

type $\forall\, x{:}T_1.\, T_2$ and t' has type T_1, then t -t' has type $[t'/x]T_2$ and erases to $|t|$. When x is not free in T_2, we write $T_1 \Rightarrow T_2$, similar to writing $T_1 \to T_2$ for $\Pi\, x{:}T_1.\, T_2$.

▶ Note. The notion of computational irrelevance here is not that of a different sort of classifier for types (e.g. *Prop* in Coq [31]) separating terms by whether they can be used for computation. Instead, it is similar to *quantitative type theory* [2]: relevance and irrelevance are properties of *binders*, indicating how functions may use arguments.

The equality type $\{t_1 \simeq t_2\}$ is the type of proofs that t_1 is propositionally equal to t_2. The introduction form β proves reflexive equations between $\beta\eta$-equivalence classes of terms: it can be checked against the type $\{t_1 \simeq t_2\}$ if $|t_1| =_{\beta\eta} |t_2|$. Note that this means equality is over *untyped* (post-erasure) terms. There is also a standard elimination form (substitution), but it is not used explicitly in the presentation of our results, so we omit its inference rule.

Equality types also come with two additional axioms.

- The φ axiom gives a strong form of the direct computation rule of NuPRL (see Allen et al. [1], Section 2.2). Though φ does not appear explicitly in the developments to come, it plays a central role by enabling the derivation of extensional type equality that enables zero-cost coercions between the equated types.
- The δ axiom provides a principle of proof discrimination. By enabling proofs that datatype constructors are disjoint, δ plays a vital role in our simulation of large eliminations.

The inference rule for an expression of the form $\varphi\ t \text{ - } t'\ \{t''\}$ says that the entire expression can be checked against type T if t' can be, if there are no undeclared free variables in t'' (so, t'' is a well-scoped but otherwise untyped term), and if t proves that t' and t'' are equal. The crucial feature of φ is its erasure: the expression erases to $|t''|$, effectively enabling us to cast t'' to the type of t'. An expression of the form $\delta \text{ - } t$ may be checked against any type if t synthesizes a type convertible with a particular false equation, $\{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}$. To broaden applicability of δ, the Cedille tool implements the *Böhm-out* semi-decision procedure [4] for discriminating between separable lambda terms.

$$\frac{\Gamma \vdash t_1 : S \to T \quad \Gamma \vdash t_2 : \Pi\, x{:}S.\, \{t_1\ x \simeq x\}}{\Gamma \vdash \mathit{intrCast} \cdot S \cdot T\ \text{-}t_1\ \text{-}t_2 : \mathit{Cast} \cdot S \cdot T}$$

$$\frac{\Gamma \vdash t : \mathit{Cast} \cdot S \cdot T}{\Gamma \vdash \mathit{cast} \cdot S \cdot T\ \text{-}t : S \to T} \qquad \frac{\Gamma \vdash t : \mathit{Cast} \cdot S \cdot T}{\Gamma \vdash \mathit{etaCast} \cdot S \cdot T\ \text{-}t : \{t \simeq \lambda\, x.\, x\}}$$

$$\begin{array}{lcllcl} |\mathit{intrCast} \cdot S \cdot T\ \text{-}t_1\ \text{-}t_2| & = & \lambda\, x.\, x & |\mathit{cast} \cdot S \cdot T\ \text{-}t| & = & \lambda\, x.\, x \\ |\mathit{etaCast} \cdot S \cdot T\ \text{-}t| & = & |\lambda\, x.\, x| & & & \end{array}$$

Figure 2 Type inclusions

$$\frac{\Gamma \vdash t_1 : \mathit{Cast} \cdot S \cdot T \quad \Gamma \vdash t_2 : \mathit{Cast} \cdot T \cdot S}{\Gamma \vdash \mathit{intrTpEq} \cdot S \cdot T\ \text{-}t_1\ \text{-}t_2 : \mathit{TpEq} \cdot S \cdot T}$$

$$\frac{\Gamma \vdash t : \mathit{TpEq} \cdot S \cdot T}{\Gamma \vdash \mathit{tpEq1} \cdot S \cdot T\ \text{-}t : S \to T} \qquad \frac{\Gamma \vdash t : \mathit{TpEq} \cdot S \cdot T}{\Gamma \vdash \mathit{tpEq2} \cdot S \cdot T\ \text{-}t : T \to S}$$

$$\begin{array}{lcllcl} |\mathit{intrTpEq} \cdot S \cdot T\ \text{-}t_1\ \text{-}t_2| & = & \lambda\, x.\, x & |\mathit{tpEq1} \cdot S \cdot T\ \text{-}t| & = & \lambda\, x.\, x \\ |\mathit{tpEq2} \cdot S \cdot T\ \text{-}t| & = & \lambda\, x.\, x & & & \end{array}$$

Figure 3 Extensional type equality

2.2 Derived Constructs

Type inclusions

The φ axiom of equality allows us to define a type constructor *Cast* that internalizes the notion that the set of all elements of some type S is contained within the set of all elements of type T (note that Curry-style typing makes this relation nontrivial). We describe its axiomatic summary, presented in Figure 2; for the full derivation, see Jenkins and Stump [17] (also Diehl et al. [11] for the related notion of Curry-style identity functions).

The introduction form *intrCast* takes two erased term arguments, a function $t_1 : S \to T$, and a proof that t_1 behaves extensionally as the identity function on its domain. The elimination form *cast* takes evidence that a type S is included into T and produces a function of type $S \to T$. The crucial property of *cast* is its erasure: $|\mathit{cast}\ \text{-}t| = \lambda\, x.\, x$. Thus, $\mathit{Cast} \cdot S \cdot T$ may also be considered the type of *zero-cost* type coercions from S to T — zero cost because the type coercion is performed in a constant number of β-reduction steps. The uniqueness principle *etaCast* tells us that every witness of a type inclusion is equal to $\lambda\, x.\, x$.

▶ Note. The significance to the results presented in this paper of the fact that any two witnesses of a type inclusion are equal is discussed in Remark 11.

▶ Remark. When inspecting the introduction and elimination forms, it may seem that *Cast* provides a form of function extensionality restricted to identity functions. This is not the case, however, as it is possible to choose S, T, and $t_1 : S \to T$ such that t_1 is provably extensionally equal to the identity function for terms of type S, and *at the same time* refute $\{t_1 \simeq \lambda\, x.\, x\}$ using δ. Instead, these rules should be read as saying that if t_1 is extensionally identity on its domain, then that fact justifies the *assignment of type* $S \to T$ *to* $\lambda\, x.\, x$.

Type equality

The extensional notion of type equality used to simulate large eliminations, *TpEq*, is the existence of a two-way type inclusion, as shown by the introduction form *intrTpEq* in Figure 3. Similar to *Cast*, the important feature of the summary of *TpEq* for the reader to keep in mind is the erasure rules for the elimination forms, *tpEq1* and *tpEq2*: both erase to $\lambda\, x.\, x$. In Section 3.2, we will see a proof that computes a type equality witness in linear time. However, in both elimination forms the type equality witness $t : \mathit{TpEq} \cdot S \cdot T$ is given as an *erased* argument. This means that the complexity of computing the witness is irrelevant to the functions that realize the two-way coercion.

▶ Remark. Strictly speaking, the type $\mathit{TpEq} \cdot S \cdot T$ is defined as the intersection of the types $\mathit{Cast} \cdot S \cdot T$ and $\mathit{Cast} \cdot T \cdot S$. In particular, this means *TpEq* enjoys the same uniqueness property as *Cast* that all witnesses are equal to $\lambda\, x.\, x$. However, the developments of this paper do not need to make explicit use of this property, so we omit this from the figure.

2.2.1 Substitution

Though we call *TpEq* extensional type *equality*, within CDLE it is only an isomorphism of types. To be considered a true notion of equality, *TpEq* would need a substitution principle. The type constructors for dependent function types (both implicit and explicit) can be proven to permit substitution if the domain and codomain parts do, as does quantification over types. However, a proof of general substitution principle would assume an arbitrary type constructor $X : \star \to \star$ and a term $t : X \cdot S$, and would need to produce a term of type $X \cdot T$, where S and T are types such that $\mathit{TpEq} \cdot S \cdot T$. To proceed, we appear to require additional assumption on X—otherwise, we cannot decompose or analyze the type $X \cdot S$ any further.

Nonetheless, the case studies presented in Sections 3 and 4 show that despite this limitation, our simulation of large eliminations using *TpEq* is adequate for dealing with common generic-programming tasks (see for example Note 9). Where we do use type constructors of higher order than $\star$ (such as in Section 4.2.1), we restrict ourselves to those which admit a substitution principle for *TpEq*.

3 n-ary Functions

In this section, we use a concrete example to detail the method of simulating large eliminations. Figure 4a shows the definition of *Nary*, the family of n-ary function types over some type T, as a large elimination of natural numbers. Our simulation of this begins by approximating this inductive definition of a *function* with an inductive *relation* between *Nat* and types, given as the generalized algebraic datatype [36] (GADT) *NaryR* in Figure 4b.

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*. Indeed, without a form of

(a) As a large elimination

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

(b) As a GADT

```
data NaryR : Nat → ⋆ → ⋆
= naryRZ : NaryR zero ·T
| naryRS : ∀ n: Nat. ∀ Ih: ⋆.
           NaryR n ·Ih → NaryR (succ n) ·(T → Ih)
```

Figure 4 n-ary functions over T [source]

proof discrimination we would not even be able to deduce that if a given type N satisfies *NaryR zero*, then from a term of type N we can extract a term of type T. Proceeding by induction, in the *naryRS* case the (impossible) goal is to show that T is the same as $T \to Ih$ for some arbitrary but fixed $Ih : \star$. We would need to derive a contradiction from the absurd equation that $\{succ\ n \simeq zero\}$ for some n. Fortunately, proof discrimination *is* available in CDLE in the form of δ, so we are able to define functions such as *extr0* below which require this form of reasoning.

```
extr0' : ∀ x: Nat. { x ≃ zero } ⇒ ∀ N: ⋆. NaryR x ·N → N → T
extr0' -zero -eqx ·T naryRZ x = x
extr0' -(succ n) -eqx ·(T → X) (naryRS n ·X r) x = δ - eqX

extr0 = extr0' -zero -β
```

▶ Note. In code listings such as the above, we present recursive Cedille functions using the syntax of (dependent) pattern matching to aid readability. This syntax is not currently supported by the Cedille tool. For the functions we present, those that compute terms are implemented in this paper's repository using the datatype system described by Jenkins et al. [16], and those that compute types use the simulation to be described next.

In the digital version of this paper, figures with code listings are accompanied by hyperlinks to the Cedille implementation embedded in the text "[source]" in captions.

3.1 Sketch of the Idea

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* $n \cdot$ (*Nary* n) is inhabited. The candidate definition for this type family is:

```
Nary n = ∀ 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* $n \cdot X$ is inhabited. For example, every term t of type *Nary zero* has type T also, since T is in this family (specifically, we have that $t \cdot T$ *-naryRZ* has type T and erases to $|t|$). In the other direction, every term of type T also has type *Nary zero*, since the only type X satisfying *NaryR zero* $\cdot X$ is T itself.

However, at the moment we are stuck when attempting to prove *NaryR zero* $\cdot$ (*Nary zero*). Though we see from the preceding discussion that T and *Nary zero* are *extensionally* equal types (they classify the same terms), *naryRZ* requires that they be *definitionally* equal! Furthermore, and as noted in Section 2.2.1, derived extensional type equality does not admit a general substitution principle, which would allow us to rewrite the type *NaryR zero* $\cdot T$ to the desired type by proving *TpEq* $\cdot T \cdot$ (*Nary zero*). Therefore, we must modify the definition of *NaryR* so that it defines a relation that is functional with respect to extensional type equality. This is shown in below, with both constructors now quantifying over an additional type argument X together with evidence that it is extensionally equal to the type of interest.

```
data NaryR : Nat → ⋆ → ⋆
= naryRZ : ∀ X: ⋆. TpEq ·X ·T ⇒ NaryR zero ·X
| naryRS : ∀ Ih: ⋆. ∀ n: Nat. NaryR n ·Ih →
           ∀ X: ⋆. TpEq ·X ·(T → Ih) ⇒ NaryR (succ n) ·X
```

```
naryRResp : ∀ n: Nat. ∀ T1: ⋆. NaryR n ·T1 → ∀ T2: ⋆. TpEq ·T1 ·T2 ⇒ NaryR n ·T2

naryRUnique : ∀ n: Nat. ∀ T1: ⋆. NaryR n ·T1 → ∀ T2: ⋆. NaryR n ·T2 → TpEq ·T1 ·T2

naryZEq : TpEq ·(Nary zero) ·T
naryZ : NaryR zero ·(Nary zero)

narySEq : ∀ n: Nat. NaryR n ·(Nary n) → TpEq ·(Nary (succ n)) ·(T → Nary n)
naryS : ∀ n: Nat. NaryR n ·(Nary n) → NaryR (succ n) ·(Nary (succ n))

naryREx : Π n: Nat. NaryR n ·(Nary n)
naryREx zero = naryZ
naryREx (succ n) = naryS -n (naryREx n)
```

Figure 5 Respectfulness, uniqueness, and existence [source]

3.2 Proof that *NaryR* is a Functional Relation

We now overview the proof that *NaryR* is a functional relation, shown partially in Figure 5 and sketched below (the full Cedille proof can be found in the code repository). Though we omit many details of the machine-checked derivation from the code listing, we give proof sketches in natural language to convey the essence of the derivation.

Having made the operating notion of type equality extensional, we are required to prove another property (in addition to uniqueness and existence): it *respects* (or *is congruent with*) extensional type equality.

▶ **Proposition 1** (Respectfulness (`naryRResp`)). *For all n : Nat and $T_1, T_2 : \star$, if Nary relates n to T_1 and T_1 is equal to T_2, then Nary relates n to T_2 also.*

Proof idea. By case analysis on the assumed proof $x : NaryR\ n \cdot T_1$. In both cases, we have a type X which is equal to T_1, so use transitivity to conclude X is equal to T_2. ◀

▶ **Proposition 2** (Uniqueness (`naryRUnique`)). *For all n : Nat and $T_1, T_2 : \star$, if Nary relates n to T_1 and also to T_2, then T_1 and T_2 are equal.*

Proof idea. By induction on the assumed proofs $f_1 : NaryR\ n \cdot T_1$ and $f_2 : NaryR\ n \cdot T_2$. In the case for *naryRZ*, T_1 and T_2 are equal to T and thus to each other. In the case for *naryRS*, T_1 is equal to a type of the form $T \to Ih_1$ and T_2 is equal to a type of the form $T \to Ih_2$ for some $Ih_1, Ih_2 : \star$, both of which are assumed to satisfy $Nary\ n'$ (where $n = succ\ n'$). By the inductive hypothesis, Ih_1 and Ih_2 are equal. Since the type constructor $\to$ respects type equality in both domain and codomain, we have $T \to Ih_1$ is equal to $T \to Ih_2$, and thus T_1 is equal to T_2. ◀

Compared to the first two properties, the proof of existence, *naryEx*, is more involved. It proceeds by induction, using lemmas *naryZ* and *naryS*, which specialize the constructors *naryRZ* and *naryRS* to the corresponding members of the *Nary* family. We only sketch the idea of one of these two lemmas, *naryRS* (the idea for *naryRZ* appeared in Section 3.1).

▶ **Lemma 3** (`naryS`). *For all n : Nat, if NaryR relates n and Nary n, then it relates succ n and Nary (succ n).*

Proof idea. First, given the assumption that $NaryR\ n \cdot (Nary\ n)$ holds, we have $NaryR\ (succ\ n) \cdot (T \to Nary\ n)$ as an instance of the constructor *naryRS*. From this and the proof that

```
naryZC : Nary zero → T
naryZC = tpEq1 -naryZEq

narySC : ∀ n: Nat. Nary (succ n) → (T → Nary n)
narySC -n = tpEq1 -(narySEq -n (naryREx n))

naryZCId : { naryZC = λ x. x }
naryZCId = β

narySCId : ∀ n: Nat. { narySC -n ≃ λ x. x }
narySCId -n = β
```

Figure 6 Computation laws for *Nary* as zero-cost coercions [source]

NaryR respects type equality, it suffices to show that *Nary* (*succ n*) and $T \to$ *Nary n* are equal types. We proceed by proving a two-way type inclusion.

- In the first direction, we assume f : *Nary* (*succ n*). Since this type is the intersection of the family of types X such that *NaryR* (*succ n*) $\cdot X$ holds, we conclude by showing that $T \to$ *Nary n* is in this family; this allows us to assign this type to f.
- In the second direction, we assume $f : T \to$ *Nary n* and an arbitrary type X such that *Nary* (*succ n*) $\cdot X$ holds, and must show f can be assigned the type X. We appeal to uniqueness, as *NaryR* relates *succ n* to both X and $T \to$ *Nary n*. Since X and $T \to$ *Nary n* are equal, f can be assigned type X.

◀

▶ **Proposition 4** (Existence (`naryREx`)). *For all n* : *Nat, NaryR relates n and Nary n.*

Proof. By induction on n, using lemmas *naryZ* and *naryS*. ◀

3.3 Computation Laws as *Zero-cost* Type Coercions

The proof of existence, *naryREx*, takes time linear in its argument n to compute a proof of *NaryR* $n \cdot$ (*Nary* n). Therefore, at first glance it would seem that any type coercions using *naryEx* could not be constant time. However, thanks to erasure in CDLE this is *not* the situation: eliminators *tpEq1* and *tpEq2* (Figure 3) take the proof of type equality as an *erased* argument, meaning the runtime complexity of *naryREx* is irrelevant to the type coercion to which it entitles us!

Figure 6 demonstrates concretely the above discussion. Therein, we define the type coercions *naryZC* and *narySC*, corresponding to the two computation laws (left-to-right) for *NaryR*. In the definition of *narySC*, note that n is bound as an erased argument, and that our problematic linear-time proof *naryEx* occurs only as part of the *erased* argument to *tpEq1*. Furthermore, we are able to *prove* that these to coercions are equal to the identity function. The proofs, named *naryZCId* and *narySCId* in the figure, are given by β in both cases, meaning that this equality holds not just propositionally, but *definitionally*—as we would expect given the erasure rules for *tpEq1*.

▶ **Example 5.** We conclude with an example: applying an n-ary function to n arguments of type T, given as a length-indexed list (*Vec*). This is shown as *appN* below.

```
appN : ∀ n: Nat. Nary n → Vec ·T n → T
appN -zero     f vnil            = naryZC f
appN -(succ n) f (vcons -n x xs) = appN -n ((narySC -n f) x) xs
```

The definition proceeds by induction on the list of arguments of type $Vec \cdot T\ n$. In the *vcons* case, the given natural number is revealed to have the form $succ\ n$, so we may coerce the type of $f : Nary\ (succ\ n)$ to the type $T \to Nary\ n$ to may apply f to the head of the list, then recursively call *appN* on the tail.

4 Generic Programming Case Studies

In the previous section, we outlined the recipe simulating large eliminations, and in particular we showed explicitly the use of type coercions for the example of applying an n-ary function. For the case studies we consider next, all code listings are presented in a syntax that omits the uses of type coercions to improve readability. In our implementation, we must explicitly use these coercions as well as several substitution lemmas for *TpEq* over type constructors. As CDLE is a kernel theory (and thus not intended to be ergonomic to program in), the purpose of these examples is to show that this simulation is indeed capable of expressing common generic programming tasks, and we leave the implementation of a high-level surface language for its utilization as future work. We do, however, remark on any new difficulties that are obscured by this presentation (such as Remark 9). Full details of all examples of this section can be found in the repository associated with this paper.

4.1 A Closed Universe of Strictly Positive Datatypes

```
data Descr : ⋆
= idD    : Descr
| constD : Descr
| pairD  : Descr → Descr → Descr
| sumD   : Π c: C. (I c → Descr) → Descr
| sigD   : Π n: Nat. (Fin n → Descr) → Descr

Decode : ⋆ → Descr → ⋆
Decode ·T idD           = T
Decode ·T constD        = Unit
Decode ·T (pairD d1 d2) = Pair ·(Decode ·T d1) ·(Decode ·T d2)
Decode ·T (sumD c f)    = Sigma ·(I c) ·(λ i: I c. Decode ·T (f i))
Decode ·T (sigD n f)    = Sigma ·(Fin n) ·(λ i: Fin n. Decode ·T (f i))

U : Descr → ⋆
U d = μ (λ T: ⋆. Decode ·T d)

inSig : ∀ n: Nat. ∀ cs: Fin n → Descr. Π i: Fin n. U (cs i) → U (sigD n cs)
inSig -n -cs i d = in (i , d)
```

Figure 7 A closed universe of strictly positive types [Descr source] [Decode source]

In the preceding section, we saw an example of arity-generic programming. We consider now a *type-generic* task: proving the *no confusion* property [5] of datatype constructors for a closed universe of strictly positive types. For the datatype universe, the idea (describe in more detail by Dagand and McBride [9]) is to define a type whose elements are interpreted as codes for datatype signatures and combine this with a type-level least fixedpoint operator.

This universe is shown in Figure 7, where *Descr* is the type of codes for signatures, *Decode* the large elimination interpreting them, and $C : \star$ and $I : C \to \star$ are parameters to

the derivation. Signatures comprise the identity functor (*idD*), a constant functor returning the unitary type *Unit* (*constD*), a product of signatures (*pairD*), and two forms of sums. The latter of these, *sigD*, takes an argument $n : Nat$ for the number of constructors and a family of n descriptions of the constructor argument types ($Fin\ n$ is the type of natural numbers less than n). The former, *sumD*, is a more generalized form that takes a code $c : C$ for a constructor argument type, and a mapping of values of type $I\ c$ (where I interprets these codes) to descriptions. Both are interpreted by *Decode* as dependent pairs which pack together an element of the indexing type ($I\ c$ or $Fin\ n$) with the decoding of the description associated with that index.

▶ Remark 6. In order to express a variety of datatypes, our universe is parameterized by codes C and interpretations $I : C \to \star$ for constructor argument types, such as used in Example 8 below. Unlike much of the literature describing the definition of a closed universe of strictly positive types [6, 9, 8] wherein the host language is a variation of intrinsically typed Martin-Löf type theory, CDLE is *extrinsically typed*—type arguments to constructors can play no role in computation, *even* in the (simulated) computation of other types. This appears to be essential for avoiding paradoxes of the form described by Coquand and Paulin [7], as CDLE is an impredicative theory in which datatype signatures need not be strictly positive.

Finally, the family of datatypes within this universe is given as U, defined using a type-level least fixedpoint operator μ which we discuss in more detail in Section 5. We define a constructor *inSig* for datatypes whose signatures are described by codes of the form $sigD\ n\ cs$ (for $n : Nat$ and $cs : Fin\ n \to Descr$) using the generic constructor $in : F\ \mu F \to \mu F$.

▶ **Example 7** (Natural numbers)**.** The type of natural numbers can be defined as:

```
unatSig : Descr
unatSig = sigD 2 (fvcons constD (fvcons idD fvnil))

UNat = U unatSig
```

where *fvcons* and *fvnil* are utilities for expressing functions out of $Fin\ n$ in a list-like notation.

The constructors of *UNat* are:

```
uzero : UNat
uzero = inSig fin0 unit

usucc : UNat → UNat
usucc n = inSig fin1 n
```

We do not need the parameters C and I for these definitions.

▶ **Example 8** (Lists)**.** Let $T : \star$ be an arbitrary type, and let parameters C and I be resp. *Unit* and $\lambda\_.T$. The type of lists containing elements of type T is defined as:

```
ulistSig : Descr
ulistSig = sigD 2 (fvcons constD (fvcons (sumD unit (λ _. idD)) fvnil))

UList = U ulistSig
```

with constructors defined similarly to those of *UNat* in the preceding example.

Proving *No Confusion*

Figure 8 shows the definition of the *no confusion* property, *NoConfusion*, as well as the type of the proof *noConfusion* which states that the property holds for all equal datatype values.

```
NoConfusion : Π n: Nat. Π cs: Fin n → Descr. U (sigD n cs) → U (sigD n cs) → ⋆
NoConfusion n cs (in (i1 , d1)) (in (i2 , d2)) | i1 =? i2
NoConfusion n cs (in (i1 , d1)) (in (i1 , d2)) | yes _ = { d1 ≃ d2 }
NoConfusion n cs (in (i1 , d1)) (in (i2 , d2)) | no _  = False

noConfusion : ∀ n: Nat. ∀ cs: Fin n → Descr.
              Π d1: U (sigD n cs). Π d2: U (sigD n cs).
              { d1 ≃ d2 } → NoConfusion d1 d2
```

Figure 8 Statement and proof of *no confusion* [source]

NoConfusion is defined by case analysis over the two datatype values, and additionally abstracts over a test of equality between the constructor labels *i1* and *i2*. The clause in which they are equal corresponds to the statement of constructor injectivity (the two terms are equal only if equal arguments were given to the constructor); the clause where $i1 \neq i2$ gives the statement of disjointness (datatype expressions cannot be equal and also be in the image of distinct constructors). The proof *noConfusion* (definition omitted) proceeds by abstracting over the same equality test, and in both cases relies on injectivity of *inSig*.

▶ Note. Though our definition of *NoConfusion* follows that of Dagand and McBride [9], it has a subtle difference: the primitive equality type in CDLE is *untyped*. Specifically, in the case where we have that i_1 and i_2 are equal, we do not need the evidence of this fact to make $\{d_1 \simeq d_2\}$, the type of equalities between $d_1 : U\ (cs\ i_1)$ and $d_2 : U\ (cs\ i_2)$, well-formed.

4.2 Arity-generic Map Operation

The last case study we consider is an arity-generic vector operation that generalizes *map*. We summarize the goal (Weirich and Casinghino [34] give a more detailed explanation): define a function which, for all n and families of types $(A_i)_{i \in \{1 \cdots n+1\}}$, takes an n-ary function of type $A_1 \to \ldots \to A_n \to A_{n+1}$ and n vectors of type $Vec \cdot A_i\ m$ (for arbitrary m and $i \in \{1, \ldots, n\}$), and produces a result vector of type $Vec \cdot A_{n+1}\ m$. Note that when $n = 1$, this is the usual map operation, and when $n = 2$ it is *zipWith* (when $n = 0$, we have $repeat : \Pi\, m : Nat.\, A_1 \to Vec \cdot A_1\ m$).

4.2.1 Vectors of Types

Our first task is to represent *Nat*-indexed families—i.e., length-indexed lists, or *vectors*—of types. As discussed in Remark 6, it is not possible to define vectors of types which support lookup as a Cedille datatype. We instead use simulated large eliminations to define them directly as lookup functions. This definition, along with some operations, is shown in Figure 9.

The kind of length n vectors of types, $\kappa\, TpVec\ n$, is defined as a function from $Fin\ n$ to $\star$. For the empty type vector *TVNil*, it does not matter what type we give for the right-hand side of the equation as $Fin\ zero$ is uninhabited. For *TVCons*, we use a (non-recursive) large elimination of the given index, returning the head type H if it is zero and performing a lookup in the tail vector L otherwise. The destructors *TVHead* and *TVTail* and the mapping function *TVMap* are defined as expected. The fold operation, *TVFold*, is given as a large elimination of the *Nat* argument; in the successor case, the recursive call is made on the tail of the given type vector L.

▶ Remark 9. Somewhat hidden by our use of high-level pseudocode is the fact that, since type equality does not admit a general substitution principle, effective use of *TVFold* requires

```
κTpVec (n : Nat) = Fin n → ⋆

TVNil : κTpVec zero
TVNil _ = ∀ X: ⋆. X.

TVCons : Π n: Nat. Π H: ⋆. Π L: κTpVec n → κTpVec (succ n)
TVCons n ·H ·L zeroFin = H
TVCons n ·H ·L (succFin i) = L i

TVHead : Π n: Nat. κTpVec (succ n) → ⋆
TVHead n ·L = L zeroFin

TVTail : Π n: Nat. κTpVec (succ n) → κTpVec n
TVTail n ·L i = L (succFin i)

TVMap : Π F: ⋆ → ⋆. Π n: Nat. κTpVec n → κTpVec n
TVMap ·F n ·L i = F ·(L i)

TVFold : Π F: ⋆ → ⋆ → ⋆. Π n: Nat. κTyVec (succ n) → ⋆
TVFold ·F zero ·L     = TVHead zero ·L
TVFold ·F (succ n) ·L = F ·(TVHead n ·L) ·(TVFold n ·(TVTail (succ n) ·L))
```

Figure 9 Vectors of types [source]

restricting its first argument to type constructors $F : \star \to \star \to \star$ which support substitution with type equality. In particular, if we do not make this assumption, then for types of the form $\mathit{TVFold} \cdot F\ (\mathit{succ}\ (\mathit{succ}\ n)) \cdot L$ we can in general simulate only *one* computation step.

Additionally, under the assumption type constructor F respects type equality in both its type arguments, we are able to give an alternative, more familiar characterization of $\mathit{TVFold} \cdot F$ by expressing its action over type vectors constructed from TVNil and TVCons. We show this characterization in Figure 10, where $\mathit{RespTpEq2} \cdot F$ formally expresses that F respects type equality, and *tvFoldZEq* and *tvFoldSEq* respectively express the action of TVFold on a singleton list and a list with two or more arguments.

```
RespTpEq2 : Π F: ⋆ → ⋆ → ⋆. ⋆
RespTpEq2 ·F = ∀ A1: ⋆. ∀ A2: ⋆. TpEq ·A1 ·A2 ⇒
               ∀ B1: ⋆. ∀ B2: ⋆. TpEq ·B1 ·B2 ⇒
               TpEq ·(F ·A1 ·B1) ·(F ·A2 ·B2)

tvFoldZEq : ∀ F: ⋆ → ⋆ → ⋆. RespTpEq2 ·F ⇒
            ∀ X: ⋆. TpEq ·(TVFold ·F zero ·(TVCons zero ·X ·TVNil)) ·X

tvFoldSEq : ∀ F: ⋆ → ⋆ → ⋆. RespTpEq2 ·F ⇒
            ∀ n: Nat. ∀ X: ⋆. ∀ L: κTyVec (succ n).
            TpEq ·(TVFold ·F (succ n) ·(TVCons (succ n) ·X ·L)) ·(F ·X ·(TVFold ·F n ·L))
```

Figure 10 Variant computation laws for TVFold [source]

```
ArrTp : Π n: Nat. κTpVec (succ n) → ⋆
ArrTp = TVFold ·(λ X: ⋆. λ Y: ⋆. X → Y)

ArrTpVec m n ·L = ArrTp n ·(TVMap ·(λ A: ⋆. Vec ·A m) (succ n) ·L)

vrepeat : ∀ A: ⋆. Π m: Nat. A → Vec ·A m
vapp    : ∀ A: ⋆. ∀ B: ⋆. ∀ m: Nat. Vec ·(A → B) m → Vec ·A m → Vec ·B m

nvecMap : Π m: Nat. Π n: Nat. ∀ L: κTpVec (succ n). ArrTp n ·L → ArrTpVec m n ·L
nvecMap m n ·L f = go n ·L (vrepeat m f)
  where
  go : Π n: Nat. ∀ L: κTpVec (succ n) → Vec ·(ArrTp n ·L) → ArrTpVec m n ·L
  go zero     ·L fs = fs
  go (succ n) ·L fs = λ xs. go n ·(TVTail (succ n) ·L) (vapp -m fs xs)
```

Figure 11 Arity-generic map [source]

4.2.2 *ArrTp* and *nvecMap*

We are now ready to define the arity-generic vector operation *nvecMap*, shown in Figure 11. We begin with *ArrTp*, the large elimination that computes the type $A_1 \to \cdots \to A_n \to A_{n+1}$ as a fold over a vector of types $L = (A_i)_{i \in \{1 \cdots n+1\}}$. The type $Vec \cdot A_1\ m \to \ldots \to Vec \cdot A_n\ m \to Vec \cdot A_{n+1}\ m$ is then constructed simply by composing $ArrTp\ n$ with a map over L taking each entry A_i to the type $Vec \cdot A_i\ m$, shown in $ArrTp\,Vec$.

For *nvecMap*, we use *vrepeat* to create m replicas of the given n-ary function argument f, then invoke the helper function *go* which is defined by recursion over n. In the zero case, *fs* has type $Vec \cdot (TVHead\ zero \cdot L)\ m$, which is equal to the expected type (by the computation laws for *ArrTp* and a proof that *Vec* respects type equality). In the successor case, *fs* is a vector of functions where the type of each element is equal to

$$TVHead\ (succ\ n) \cdot L \to ArrTp\ n\ \cdot (TVTail\ (succ\ n) \cdot L)$$

and the expected type is

$$Vec \cdot (TVHead\ (succ\ n) \cdot L)\ m \to ArrTp\,Vec\ m\ n \cdot (TVTail\ (succ\ n) \cdot L)$$

so we assume such a vector *xs*, use *vapp* to apply each function of *fs* point-wise to the elements of *xs*, then recurse to consume the remaining arguments.

5 Generic Simulation

We now generalize the approach outlined in Section 3 and simulate large eliminations *generically* for datatypes. In the Cedille tool, datatype declarations are elaborated [16] to impredicative encodings provided by the generic framework of Firsov et al. [12]. This framework is based on the categorical semantics of datatypes as initial algebras [15], specifically *Mendler-style* algebras [32], and it supports a broad class of datatypes including those that are nonstrictly positive. To enjoy this same generality and to establish a foundation for surface-language syntax of large eliminations in the Cedille tool, the developments in this section also uses the framework of op. cit. Specifically, we simulate large eliminations for all datatypes of the form $\mu F : \star$, where $F : \star \to \star$ is a covariant (but otherwise arbitrary) type scheme and μ is the operator for type-level least fixedpoints provided by Firsov et. al. [12].

We first review Mendler-style recursion, and the framework of op. cit. for inductive Mendler-style lambda encodings of datatypes in CDLE. Then, we define the notion of a Mendler-style F-algebra at the level of types, overcoming a technical difficulty for classical F-algebras arising from CDLE's truncated sort hierarchy. Finally, we show that if a type-level Mendler F-algebra A satisfies a certain condition with respect to derived type equality, then A can be used for a simulated large elimination.

5.1 Mendler-style Recursion and Encodings

We briefly review the datatype recursion scheme *à la* Mendler. Originally proposed by Mendler [23] as a method of impredicatively encoding datatypes, Uustalu and Vene have shown that it forms the basis of an alternative categorical semantics of inductive datatypes [32], and the same have advocated for the Mendler style of coding recursion, arguing that it is more idiomatic than the classical formulation of structured recursion schemes [33].

▶ **Definition 10** (Mendler-style primitive recursion)**.** *Let $F : \star \to \star$ be a positive type scheme. The datatype with signature F is μF with constructor $in : F \cdot \mu F \to \mu F$. The Mendler-style primitive recursion scheme for μF is described by the typing and computation law given for* rec *below:*

$$\frac{\Gamma \vdash T : \star \quad \Gamma \vdash a : \forall\, R{:}\star.\, (R \to \mu F) \to (R \to T) \to F \cdot R \to T}{\Gamma \vdash \mathit{rec} \cdot T\ a : \mu F \to T}$$

$$\mathit{rec} \cdot T\ a\ (\mathit{in}\ d) \rightsquigarrow a \cdot \mu F\ (\mathit{id} \cdot \mu F)\ (\mathit{rec} \cdot T\ a)\ d$$

In Definition 10, the type T (the *carrier*) is the type of results we wish to compute, and the term a (the *action*) gives a single step of a recursive function, and we call the two of them together a Mendler-style F-algebra for recursion. We understand the type argument R of the action as a kind of subtype of the datatype μF — specifically, a subtype containing only predecessors on which we are allowed to make recursive calls.

The first term argument of the action, a function of type $R \to \mu F$, we can view as the coercion that realizes the subtyping relation; in the computation law, type argument R is instantiated to μF, and the coercion is just $\mathit{id} \cdot \mu F$, the identity. The next argument, a function of type $R \to T$, is the handle for making recursive calls; in the computation law, it is instantiated to $\mathit{rec} \cdot T\ a$. Finally, the last argument is an "F-collection" of predecessors of the type R; in the computation law, it is instantiated to the collection of predecessors $d : F\ \mu F$ of the datatype value $\mathit{in}\ d$.

▶ Note. We can use the fact that the "coercion" function for Mendler recursion is always instantiated to the identity. In such cases, in CDLE it is idiomatic to have, instead of a computationally relevant argument of type $R \to \mu F$, a computationally *irrelevant* argument of type $\mathit{Cast} \cdot R \cdot \mu F$ (Figure 2). Since a *Cast* term is a proof of a type inclusion, this makes explicit the previously informal intuition that the quantified type R is a subtype of μF.

Generic framework for Mendler-style datatypes Figure 12 gives an axiomatic summary of the generic framework of Firsov et al. [12] for deriving efficient Mendler-style lambda encodings of datatypes with induction. In all inference rules save the type formation rule of μ, the datatype signature F is required to be *Monotonic* (that is, positive).

- *in* is the datatype constructor. For the developments in this section, we find the Mendler-style presentation given in the figure more convenient than the classical type of *in*.

$$
\begin{array}{lcl}
\mathit{Monotonic} \cdot F & = & \forall\, X : \star.\, \forall\, Y : \star.\, \mathit{Cast} \cdot X \cdot Y \Rightarrow \mathit{Cast} \cdot (F \cdot X) \cdot (F \cdot Y) \\
\mathit{PrfAlg} \cdot F\ m \cdot P & = & \forall\, R : \star.\, \forall\, c : \mathit{Cast} \cdot R \cdot \mu F. \\
 & & \quad (\Pi\, x : R.\, P\ (\mathit{cast}\ \text{-}c\ x)) \to \Pi\, \mathit{xs} : F \cdot R.\, P\ (\mathit{in}\ \text{-}m\ \text{-}c\ \mathit{xs})
\end{array}
$$

$$
\frac{\Gamma \vdash F : \star \to \star}{\Gamma \vdash \mu F : \star} \qquad \frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash m : \mathit{Monotonic} \cdot F}{\Gamma \vdash \mathit{in}\ \text{-}m : \forall\, R : \star.\, \mathit{Cast} \cdot R \cdot \mu F \Rightarrow F \cdot R \to \mu F}
$$

$$
\frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash m : \mathit{Monotonic} \cdot F}{\Gamma \vdash \mathit{out}\ \text{-}m : \mu F \to F \cdot \mu F}
$$

$$
\frac{\Gamma \vdash F : \star \to \star \quad \Gamma \vdash m : \mathit{Monotonic} \cdot F}{\Gamma \vdash \mathit{ind}\ \text{-}m : \forall\, P : \mu F \to \star.\, \mathit{PrfAlg} \cdot F\ m \cdot P \to \Pi\, x : \mu F.\, P\ x}
$$

$$
\begin{array}{lcl}
|\mathit{ind}\ \text{-}m \cdot P\ a\ (\mathit{in}\ \text{-}m \cdot R\ \text{-}c\ \mathit{xs})| & =_{\beta\eta} & |a \cdot R\ \text{-}c\ (\lambda\, x.\, \mathit{ind}\ \text{-}m \cdot P\ a\ (\mathit{cast}\ \text{-}c\ x))\ \mathit{xs}| \\
|\mathit{out}\ \text{-}m\ (\mathit{in}\ \text{-}m \cdot R\ \text{-}c\ \mathit{xs})| & =_{\beta\eta} & |\mathit{xs}|
\end{array}
$$

Figure 12 Axiomatic summary of the generic framework of Firsov et al. [12]

- *out* is the datatype destructor, revealing the F-collection of predecessors used to construct the given value.
- *PrfAlg* is a generalization of Mendler-style algebras to dependent types. Compared to the earlier discussion:
  - the carrier is a predicate $P : \mu F \to \star$ instead of a type;
  - the coercion function $R \to \mu F$ from Mendler recursion becomes an erased witness of type $\mathit{Cast} \cdot R \cdot \mu F$;
  - given a handle for invoking the inductive hypothesis on predecessors of type R and an F-collection of such predecessors, a P-proof F-algebra action must show that P holds for the value constructed from these predecessors using *in*.
- *ind* gives the induction principle: to prove a property P for an arbitrary term of type μF, it suffices to give a P-proof F-algebra.

5.2 Mendler-style Type Algebras

Like other (well-founded) recursive definitions, a large elimination can be expressed as a fold of an algebra. In theories with a universe hierarchy, expressing this algebra is no difficult task: the signature F can be universe polymorphic so that its application to either a type or kind is well-formed. This is *not* the case for CDLE, however, as it has a truncated hierarchy of sorts and no sort polymorphism. More specifically, there is no way to express a classical F-algebra on the level of types, e.g., a kind $(F\ \star) \to \star$, as it is not possible to define a function on the level of kinds (which F would need to be).

Thankfully, this difficulty *disappears* when the type algebra is expressed in the Mendler style! This is because F does not need to be applied to the kind $(\star)$ of previously computed types, only to the universally quantified type R. Instead, types are computed from predecessors using an assumption of kind $R \to \star$.

Figure 13 shows the definition $\kappa \mathit{AlgTy}$ of the kind of Mendler-style algebras for primitive recursion having carrier $\star$ (henceforth we will refer to the action of type algebras simply

```
κAlgTy = Π R: ⋆. Cast ·R ·μF → (R → ⋆) → F ·R → ⋆ .

AlgTyResp : κAlgTy → ⋆
= λ A: κAlgTy.
  ∀ R1: ⋆. ∀ R2: ⋆. ∀ c1: Cast ·R1 ·μF. ∀ c2: Cast ·R2 ·μF.
  ∀ Ih1: R1 → ⋆. ∀ Ih2: R2 → ⋆.
  (Π r1: R1. Π r2: R2. { r1 ≃ r2 } → TpEq ·(Ih1 r1) ·(Ih2 r2)) →
  Π xs1: F ·R1. Π xs2: F ·R2. { xs1 ≃ xs2 } →
  TpEq ·(A ·R1 c1 ·Ih1 xs1) ·(A ·R2 c2 ·Ih2 xs2) .
```

Figure 13 Mendler-style type algebras [source]

as *algebra*). Just as in the concrete derivation of Section 3, we require that algebras must respect type equality. This condition is codified in the figure as *AlgTyResp*, which says:

- given two subtypes R_1 and R_2 of μF (which need *not* be equal),
- and two inductive hypotheses Ih_1 and Ih_2 for computing types from values of type R_1 and R_2, resp.,
- that return equal types on equal terms, then
- we have that the algebra A returns equal types on equal F-collections of predecessors (where the types of predecessors are resp. R_1 and R_2).

▶ Remark 11. A careful reader may have noticed that, in Figure 13, we place no constraints on the witnesses $c_1 : \mathit{Cast} \cdot R_1\ \mu F$ and $c_2 : \mathit{Cast} \cdot R_2 \cdot \mu F$, even though they both appear in the final equality of *AlgTyResp*. This is because none are needed: recall from Figure 2 that *etaCast* tells us *all* witnesses of a type coercion are provably equal to $\lambda\, x.\, x$, so in particular c_1 and c_2 are provably equal to each other.

▶ **Example 12.** Let $F \cdot R = 1 + R$ be the signature of natural numbers with $\mathit{zeroF} : \forall\, R{:}\star.\, F \cdot R$ and $\mathit{succF} : \forall\, R{:}\star.\, R \to F \cdot R$ the signature's injections. For a given property $P : \mu F \to \star$, we can express as a fold over a Mendler type algebra the property that P holds for a given value *and* all its predecessors, as might be used for a hypothesis for strong induction. That algebra is given below as *StrongIndAlg*:

```
StrongIndAlg : κAlgTy
StrongIndAlg ·R c Ih (zeroF ·R) = P (in -c (zeroF ·R))
StrongIndAlg ·R c Ih (succF ·R n) =
  Pair ·(P (in -c (succF ·R n))) ·(StrongIndAlg ·R c Ih n)
```

Note that unlike previous examples, this algebra is *recursive* rather than being only *iterative*, as the cast c is used (by *in*) to access predecessors at type μF.

Inspecting this definition, we see it indeed satisfies the condition *AlgTyResp*, with the proof sketch as follows. We assume R_1, R_2, $c_1 : \mathit{Cast} \cdot R_1 \cdot \mu F$, $c_2 : \mathit{Cast} \cdot R_2 \cdot \mu F$, and $xs_1 : F \cdot R_1$ and $xs_2 : F \cdot R_2$ such that $\{xs_1 \simeq xs_2\}$. We may proceed by considering the cases where both are formed by the same injection.

- In the *zeroF* case, the algebra returns $P\ (\mathit{in}\ \text{-}c_1\ (\mathit{zeroF} \cdot R_1))$ and $P\ (\mathit{in}\ \text{-}c_2\ (\mathit{zeroF} \cdot R_2))$, which are convertible by erasure (we do not need *etaCast* for this).
- In the *succF* case, *Pair* respects type equality, so it suffices to prove that the types of the components are equal. From the assumption that $\mathit{succF} \cdot R_1\ n_1$ is equal to $\mathit{succF} \cdot R_2\ n_2$ (for some $n_1 : R_1$, $n_2 : R_2$), we obtain $\{n_1 \simeq n_2\}$, allowing us to conclude by using term substitution in the first component type and the inductive hypothesis for the second.

```
data FoldR : μF → ⋆ → ⋆
= foldRIn
  : ∀ R: ⋆. ∀ c: Cast ·R ·μF. ∀ xs: F ·R.
    ∀ Ih: R → ⋆. (Π x: R. FoldR (cast -c x) ·(Ih x)) →
    ∀ X: ⋆. TpEq ·X ·(A ·R c ·Ih xs) ⇒ FoldR (in -c xs) ·X

Fold : μF → ⋆
Fold x = ∀ X: ⋆. FoldR x ·X ⇒ X .

foldRResp   : ∀ x: μF. ∀ X1: ⋆. FoldR x ·X1 → ∀ X2: ⋆. TpEq ·X1 ·X2 ⇒ FoldR x ·X2
foldRUnique : ∀ x: μF. ∀ X1: ⋆. FoldR x ·X1 → ∀ X2: ⋆. FoldR x ·X2 → TpEq ·X1 ·X2
foldREx     : Π x: μF. FoldR x ·(Fold x)

foldBeta : ∀ R: ⋆. ∀ c: Cast ·R ·μF. ∀ xs: F ·R.
           TpEq ·(Fold (in -c xs)) ·(A ·R c ·(λ x: R. Fold (cast -c x)) xs)

foldEta : ∀ H: μF → ⋆.
          Π homH: ∀ R: ⋆. ∀ c: Cast ·R ·μF. Π xs: F ·μF.
                  TpEq ·(H (in -c xs)) ·(A ·R c ·(λ x: R. H (cast -c x)) xs).
          Π x: μF. TpEq ·(H x) ·(Fold x)
```

Figure 14 Generic large elimination [source]

▶ Remark 13. We again note that, in the definition of *AlgTyResp*, the two assumed subtypes R_1 and R_2 need not be equal. As a consequence, in order to satisfy this condition the type produced by the algebra should *not* depend on its type argument R. A high-level surface language implementation for large eliminations in Cedille could require that the bound type variable R only occurs in type arguments of term subexpressions. As definitional equality of types is modulo erasure of typing annotations in term subexpressions, this would ensure that the meaning (extent) of the type does not depend on R.

5.3 Relational Folds of Type Algebras

Figure 14 gives the definition of *FoldR*, a GADT expressing the fold of a type level algebra $A : \kappa AlgTy$ over μF as a functional relation (A and F are parameters to the definition). It has a single constructor, *foldRIn*, corresponding to the single generic constructor *in* of the datatype, whose type we read as follows:

- given a subtype R of μF and a collection of predecessors $xs : F \cdot R$, and
- a function $Ih : R \to \star$ that, for every element x in its domain, produces a type related (by *FoldR*) to that element, then
- the datatype value constructed from xs is related to all types that are equal to $A \cdot R\, c \cdot Ih\, xs$.

Just as in Section 3, to show that the inductive relation given by *FoldR* determines a function (from μF to equivalence classes of types), we define a canonical name (*Fold*) for the types determined by the datatype elements and prove that the relation satisfies three properties: it respects type equality, and every datatype element uniquely determines a type. The proofs of respectfulness and existence properties proceed similarly to the concrete proofs given for n-ary functions (see the code repository for full details). We use the condition on type algebras in the proof of uniqueness, so we give a proof sketch below.

▶ **Proposition 14** (Uniqueness (`foldRUnique`)). *For all $x : \mu F$ and $X_1, X_2 : \star$, if FoldR relates x to both X_1 and X_2, then X_1 and X_2 are equal.*

Proof idea. By induction on the proofs $f_1 : \mathit{FoldR}\ x \cdot X_1$ and $f_2 : \mathit{FoldR}\ x \cdot X_2$. In the case for *foldRIn* for f_1, we know that x is of the form $\mathit{in} \cdot R_1\ \text{-}c_1\ \mathit{xs}_1$ for some $R_1 : \star$, $c_1 : \mathit{Cast} \cdot R_1 \cdot \mu F$, and $\mathit{xs}_1 : F \cdot R$. Similarly, from f_2 we know that x is *also* of the form $\mathit{in} \cdot R_2\ \text{-}c_2\ \mathit{xs}_2$. By injectivity of *in*, we know that $\{\mathit{xs}_1 \simeq \mathit{xs}_2\}$. Call this result (1).

In the case for *foldRIn*, we also are given: from f_1, a type family $\mathit{Ih}_1 : R_1 \to \star$ such that *FoldR* relates every $x : R_1$ (R_1 is a subtype of μF) to $\mathit{Ih}_1\ x$, and a type X_1 extensionally equal to $A \cdot R_1\ c_1 \cdot \mathit{Ih}_1\ \mathit{xs}_1$; from f_2, a type family Ih_2 and type X_2 satisfying similar conditions. By the inductive hypothesis, we can obtain the fact that for all $r_1 : R_1$ and $r_2 : R_2$ such that $\{r_1 \simeq r_2\}$, $\mathit{Ih}_1\ r_1$ is equal to $\mathit{Ih}_2\ r_2$. Call this result (2).

We may now use results (1) and (2) to invoke the assumed condition on A that it respects type equality, obtaining a proof that $A \cdot R_1\ c_1 \cdot \mathit{Ih}_1\ \mathit{xs}_1$ is equal to $A \cdot R_2\ c_2 \cdot \mathit{Ih}_2\ \mathit{xs}_2$. From this and some equational reasoning it follows that X_1 is equal to X_2, concluding the proof. ◀

▶ Remark 15. At present, we are unable to express in a *single definition* type constructor algebras with arbitrarily kinded carriers. Thus, while our derivation is parametric in a datatype signature, it must be repeated once for each type constructor kind. This process is however entirely mechanical, so an implementation of a higher-level surface language for large eliminations in Cedille could elaborate each variant of the derivation as needed, removing the burden of writing boilerplate code.

5.3.1 Characterization

The last two definitions of Figure 14 characterize *Fold* as a recursion scheme. The computation law, given by *foldBeta*, follows a similar pattern as for the Mendler-style recursion shown in Definition 10. *Fold* acts over a datatype value constructed with predecessors $\mathit{xs} : F \cdot R$ by calling the type level algebra A with *Fold* as the handle for recursive calls. Since *in* is a Mendler-style datatype constructor, we instantiate the type argument of A to R so that A may be applied directly to *xs*. The requirement that A satisfies *AlgTyResp* means that this is equivalent (up to type equality) to instantiating A with μF and applying this to *xs* after casting *xs* to the type $F \cdot \mu F$.

In the case studies of Sections 3 and 4, we discussed only the computation laws of our simulated large eliminations. For the generic result, we go further: *Fold* satisfies (up to type equality and function extensionality) the *extensionality law* for Mendler-style recursion, which says that *Fold* is uniquely defined by its action on the values generated by the constructor *in*. This is shown as *foldEta* in the figure, whose type says that any other function $H : \mu F \to \star$ that satisfies the same computation law as *Fold* is in fact equal to *Fold*.

6 Related Work

CDLE In an earlier formulation of CDLE [27], Stump proposed a mechanism called *lifting* which allowed simply typed terms to be lifted to the level of types. While adequate for both proving constructor disjointness for natural numbers and enabling some type-generic programming (such as formatted printing in the style of `printf`), its presence significantly complicated the meta-theory of CDLE and its expressive ability was found to be incomplete [28]. Lifting was subsequently removed from the theory, replaced with the simpler δ axiom for proof discrimination.

Marmaduke et al. [22] described a method of encoding datatype signatures that enables constructor subtyping (*à la* Barthe and Frade [3]) with zero-cost type coercions. A key technique for this result was the use of intersection types and equational constraints to

simulate (again with type coercions) the computation of types by case analysis on terms — that is, non-recursive large eliminations. Their method of simulation is therefore suitable for expressing type algebras, but not their folds.

System F_C The intermediate language used by the Haskell compiler GHC, System F_C [29], is an extension of System F with type coercions and equalities. In particular, within System F_C one can express nonparametric type-level functions by adding type equality axioms, such as $f\ \mathit{Int} \sim \mathit{Bool}$ (where $\sim$ is the type equality operator for F_C). In our approach, clauses of type-level functions are encoded using datatype constructors, and incoherent or partial functions cannot be used because the relation defined by the underlying datatype must be *proven* to be functional.

MLTT and CC Smith [26] showed that disjointness of datatype constructors was not provable in Martin-Löf type theory without large eliminations by exhibiting a model of types with only two elements — a singleton set and the empty set. In the calculus of constructions, Werner [35] showed that disjointness of constructors would be contradictory by using an erasure procedure to extract System F^ω terms and types, showing that a proof of $1 \neq 0$ in CC would imply a proof of $(\forall\, X : \star.\, X \to X) \to \forall\, X : \star.\, X$ in F^ω. *Proof irrelevance* is central to both results. Since in CDLE proof relevance is axiomatized with δ, this paper can be viewed as a kind of converse to these results: large eliminations enable proof discrimination, and proof discrimination together with extensional type equality enable the simulation of large eliminations.

GADT Semantics Our simulation of large eliminations rests upon a semantics of GADTs which (intuitively) interprets them as the least set generated by their constructors. However, the semantics of GADTs is a subject which remains under investigation. Johann and Polonsky [19] recently proposed a semantics which makes them functorial, but in which the above-given intuition fails to hold. In subsequent work, Johann et al. [18] explain that GADTs whose semantics are instead based on impredicative encodings (in which case they are not in general functorial) may be equivalently expressed using explicit type equalities. Though they exclude functorial semantics for GADTs in CDLE, the presence of type equalities (both implicit in the semantics and the explicit uses of derived extensional type equality) are essential for defining a relational simulation of large eliminations.

7 Conclusion and Future Work

We have shown that large eliminations may be simulated in CDLE using a derived extensional type equality, zero-cost type coercions, and GADTs to inductively define functional relations. This result overcomes seemingly significant technical obstacles, chiefly CDLE's lack of primitive inductive types and universe polymorphism, and is made possible by an axiom for proof discrimination. To demonstrate the effectiveness of the simulation, we examine several case studies involving type- and arity-generic programming. Additionally, we have shown that the simulation may be derived generically (that is, parametric in a datatype signature) with Mendler-style type algebras satisfying a certain condition with respect to type equality.

Syntax In this paper, we have chosen to present code examples using a high-level syntax to improve readability. While the current version of Cedille [10] supports surface language syntax for datatype declarations and recursion, syntax for large eliminations remains future

work. Support for this requires addressing (at least) two issues. First, it requires a sound criterion for determining when the type algebra denoted by the surface syntax satisfies the condition *AlgTyResp* (Section 5.2). We conjecture that a simple syntactic occurrence check, along the lines outlined in Remark 13, for erased arguments will suffice. Second, it is desirable that the type coercions that simulate the computation laws of a large elimination be automatically inferred using a subtyping system based on coercions [21, 30].

Semantics As discussed in Section 2.2.1, the derived form of extensional type equality used in our simulation lacks a substitution principle. However, we claim that such a principle is validated by CDLE's semantics [28], wherein types are interpreted as sets of ($\beta\eta$-equivalence classes of) terms of untyped lambda calculus. Under this semantics, a proof of extensional type equality in the syntax implies equality of the semantic objects. We are therefore optimistic that CDLE may be soundly extended with a kind-indexed family of type constructor equalities with an extensional introduction form and substitution for its elimination form, removing all limitations of the simulation of large eliminations.

References

1 Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. Innovations in computational type theory using NuPRL. *J. Applied Logic*, 4(4):428–469, 2006. doi:10.1016/j.jal.2005.10.005.

2 Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi:10.1145/3209108.3209189.

3 Gilles Barthe and Maria João Frade. Constructor subtyping. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1999. doi:10.1007/3-540-49099-X_8.

4 Corrado Böhm, Mariangiola Dezani-Ciancaglini, P. Peretti, and Simona Ronchi Della Rocca. A discrimination algorithm inside lambda-beta-calculus. *Theor. Comput. Sci.*, 8:265–292, 1979. doi:10.1016/0304-3975(79)90014-8.

5 R. M. Burstall and J. A. Goguen. *Algebras, Theories and Freeness: An Introduction for Computer Scientists*, pages 329–349. Springer Netherlands, Dordrecht, 1982. doi:10.1007/978-94-009-7893-5_11.

6 James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14, 2010. doi:10.1145/1863543.1863547.

7 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. doi:10.1007/3-540-52335-9_47.

8 Pierre-Évariste Dagand. *A cosmology of datatypes: reusability and dependent types*. PhD thesis, University of Strathclyde, Glasgow, UK, 2013. URL: http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22713.

9 Pierre-Évariste Dagand and Conor McBride. Elaborating inductive definitions. *CoRR*, abs/1210.6390, 2012. arXiv:1210.6390.

10 Cedille development team. Cedille v1.2.1. https://github.com/cedille/cedille.

11 Larry Diehl, Denis Firsov, and Aaron Stump. Generic zero-cost reuse for dependent types. *Proc. ACM Program. Lang.*, 2(ICFP):104:1–104:30, Jul 2018. doi:10.1145/3236799.

12 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. doi:10.1007/978-3-319-94821-8_14.

13 Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in Cedille. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 215–227, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167087.

14 Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 166–181, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-45413-6_16.

15 Tatsuya Hagino. *A categorical programming language*. PhD thesis, 1987.

16 Christopher Jenkins, Colin McDonald, and Aaron Stump. Elaborating inductive definitions and course-of-values induction in Cedille, 2019. arXiv:1903.08233.

17 Christopher Jenkins and Aaron Stump. Monotone recursive types and recursive data representations in Cedille. *Math. Struct. Comput. Sci.*, 31(6):682–745, 2021. doi:10.1017/S0960129521000402.

18 Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. GADTs, functoriality, parametricity: Pick two. *CoRR*, 2021. arXiv:2105.03389.

19 Patricia Johann and Andrew Polonsky. Higher-kinded data types: Syntax and semantics. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785657.

20 Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada*, LICS '03, pages 86–95. IEEE Computer Society, 2003. doi:10.1109/LICS.2003.1210048.

21 Zhaohui Luo. Coercive subtyping. *J. Logic and Computation*, 9(1):105–130, 1999. doi:10.1093/logcom/9.1.105.

22 Andrew Marmaduke, Christopher Jenkins, and Aaron Stump. Zero-cost constructor subtyping. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, IFL 2020, page 93–103, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3462172.3462194.

23 N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of the Symposium on Logic in Computer Science*, (LICS '87), pages 30–36, Los Alamitos, CA, June 1987. IEEE Computer Society.

24 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. doi:10.1007/3-540-45413-6_27.

25 Jan M. Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *J. Symb. Log.*, 49(3):730–753, 1984. doi:10.2307/2274128.

26 Jan M. Smith. The independence of Peano's fourth axiom from Martin-Lof's type theory without universes. *J. Symb. Log.*, 53(3):840–845, 1988. doi:10.2307/2274575.

27 Aaron Stump. The calculus of dependent lambda eliminations. *J. Funct. Program.*, 27:e14, 2017. doi:10.1017/S0956796817000053.

28 Aaron Stump and Christopher Jenkins. Syntax and semantics of Cedille. 2018. arXiv:1806.04709.

29 Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pages 53–66. ACM, 2007. doi:10.1145/1190315.1190324.

30 Nikhil Swamy, Michael W. Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 329–340. ACM, 2009. doi:10.1145/1596550.1596598.

31 The Coq Development Team. *The Coq Reference Manual, version 8.13*, 2021. Available electronically at https://coq.github.io/doc/v8.13/refman/.

32 Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361, Sep 1999. URL: http://dl.acm.org/citation.cfm?id=774455.774462.

33 Tarmo Uustalu and Varmo Vene. Coding recursion a la Mendler (extended abstract). In *Proc. of the 2nd Workshop on Generic Programming, WGP 2000, Technical Report UU-CS-2000-19*, pages 69–85. Dept. of Computer Science, Utrecht University, 2000.

34 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. doi:10.1007/978-3-642-32202-0_5.

35 Benjamin Werner. A normalization proof for an impredicative type system with large elimination over integers. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Proc. of the 1992 Workshop on Types for Proofs and Programs*, pages 341–357, June 1992.

36 Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 224–235. ACM, 2003. doi:10.1145/604131.604150.