Simulating Large Eliminations in Cedille

Christopher 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 any datatype. 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

Supplementary Material `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., $0 \neq 1$)[25, 24]. For programmers, they give an expressive mechanism for arity- and type-generic programming with universe constructions [32]. As an example, the type *Nary* n of n-ary functions (where n is a natural number) over some type T can be defined as T when $n = 0$ and $T \to \mathit{Nary}\ n'$ when $n = 1 + n'$.

Large eliminations are closely tied to a type theory's primitive notion of inductive type, and 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) [26, 27] 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 (where the extent of a type is the set of terms it classifies). In particular, we:

- describe a method for simulating large eliminations in CDLE (Section 3), identifying the features of the theory that enable the development (Section 2);
- formulate the method *generically* (meaning *parametrically*) for all datatype signatures, using the framework of Firsov et al. [12] (Section 5);

- demonstrate the adequacy of this simulation by applying it to several generic programming tasks: n-ary functions, a closed universe of datatypes, a decision procedure for the inhabitation of simple types, 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 at the code repository associated with this paper.[1]

Outline The remainder of this paper is structured as follows. Section 2 reviews background material on CDLE, focusing on the primitives which enable the simulation and the derived extensional type equality. In Section 3, we carefully explain the recipe for simulating large eliminations using as an example the type n-ary functions over a given type. Section 4 shows three case studies, presented as evidence of the effectiveness of the simulation in tackling generic programming tasks. The recipe for concrete examples is then turned into a generic (that is, parametric) derivation 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 [19]; and the implicit products of Miquel [23]. The pure term language of CDLE is untyped lambda calculus, but to make type checking algorithmic terms are presented with typing annotations. 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 [27]); 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 [23] 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 the introduction form is $|t|$. For

[1] `https://github.com/cedille/cedille-developments/tree/master/large-elim-sim`

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

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

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

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

Figure 1 Typing and erasure for a fragment of CDLE

application, if t has 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, c.f. [29]) 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: a strong form of the direct computation rule of NuPRL (c.f. Allen et al. [1], Section 2.2) given by φ, and a principle of proof discrimination given by δ. The inference rule for an expression of the form $\varphi\ t$ - $t'\ \{t''\}$ 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'. Though φ does not appear explicitly in the developments to come, it plays a central role by enabling the derivation of extensional type equality.

An expression of the form δ - 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 [5] for discriminating between $\beta\eta$-inequivalent terms. By enabling proofs that datatype constructors are disjoint, δ plays a vital role in our simulation of large eliminations.

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

$$|\mathit{intrCast} \cdot S \cdot T\ \text{-}t_1\ \text{-}t_2| = \lambda\, x.\, x \qquad |\mathit{cast} \cdot S \cdot T\ \text{-}t| = \lambda\, x.\, x$$

Figure 2 Type inclusions

2.2 Derived Constructs

Type inclusions

The φ axiom of equality allows us to define a type constructor *Cast* which 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 [16] (also Diehl et al. [11]).

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 between the same. 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.

▶ Note. 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 such that t_1 is provably extensionally equal to the identity function on its domain, and *at the same time* refute $\{t_1 \simeq \lambda\, x.\, x\}$ using δ. Instead, taken together these rules should be read as saying that if t_1 is extensionally identity on its domain, then that justifies the *assignment of type* $S \to T$ *to* $\lambda\, x.\, x$.

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

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

$$|\mathit{intrTpEq} \cdot S \cdot T\ \text{-}t_1\ \text{-}t_2| = \lambda\, x.\, x \qquad |\mathit{tpEq1} \cdot S \cdot T\ \text{-}t| = \lambda\, x.\, x$$
$$|\mathit{tpEq2} \cdot S \cdot T\ \text{-}t| = \lambda\, x.\, x$$

Figure 3 Extensional type equality

Type equality

The extensional notion of type equality used to simulate large eliminations, *TpEq*, is merely the existence of a two-way type inclusion. Strictly speaking, a term of type $\mathit{TpEq} \cdot S \cdot T$ is evidence that $\lambda\, x.\, x$ may be assigned the intersection type $(S \to T) \cap (T \to S)$. However, a good intuition for understanding the introduction and elimination rules is to think of this

(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

as the types of pairs of casts t_1 and t_2 between S and T, with the elimination forms being compositions of the appropriate projection function with *cast* (the projections would not appear in the erasures of the elimination forms, as the argument to *cast* is erased).

▶ Remark 1. 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, the presence of higher-order type constructors means that not all closed types allow substitution of types related by *TpEq* (consider $\forall\, X : \star \to \star.\, X \cdot S$, where one has S and T such that $\mathit{TpEq} \cdot S \cdot T$). 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.

▶ Note. In the developments in subsequent sections, *refl*, *sym*, and *trans* refer to the three proofs that together show *TpEq* is an equivalence relation. We have omitted their definitions; their types are as expected.

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 [34] (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 proof discrimination we would not even be able to deduce that if a given type N satisfies *NaryR zero*, then N must be T. Proceeding by induction, in the *naryRS* case the goal is to show that T is the same as $T \to \mathit{Ih}$ for some (arbitrary but fixed) $\mathit{Ih} : \star$. We would need to derive a contradiction from the absurd equation that $\{\mathit{zero} \simeq \mathit{succ}\ n\}$ 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* in Figure 5 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
```

Figure 5 Extracting a 0-ary term

▶ Note. In code listings such as Figure 5, we present recursive Cedille functions using the syntax of (dependent) pattern matching in order to aid readability. This syntax is not currently supported by the Cedille tool. For functions computing terms, the Cedille code in this paper's repository uses the datatype system described by Jenkins et al. [15]. For functions computing types, the repository code uses the simulation to be described next.

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 $|t|$). Similarly, every term of type T has type *Nary zero* by induction on the assumed proof of *NaryR zero* $\cdot X$ for arbitrary X, invoking δ in the *naryRS* case (see Section 3.2 for how certain properties may be proved using induction on erased arguments).

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! We therefore 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 Figure 6, 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
```

Figure 6 *NaryR* as a relation that is functional with respect to *TpEq*

3.1 Proof that *NaryR* is a Functional Relation

Respectfulness

We now detail the proof that *NaryR* is a functional relation. Having changed our notion of equality for types to extensional, we must now prove a third property (in addition to uniqueness and existence): that it respects extensional type equality, i.e., if *Nary* relates n to T_1 and T_1 is equal to T_2, then *Nary* relates n to T_2 also. This proof is shown as *naryRResp* in Figure 7. Proceeding by case analysis, in both cases we combine the assumed proof that T_1 and T_2 are equal types with the type equality proof given to the constructor.

Uniqueness

Figure 8 shows the proof *naryRUnique* that *Nary* n uniquely determines a type (up to type equality) for all n. To improve readability, this listing omits the two absurd clauses in which the given *Nary* proofs differ in their construction; in the code repository, these two cases are

```
naryRResp : ∀ n: Nat. ∀ T1: ⋆. NaryR n ·T1 → ∀ T2: ⋆. TpEq ·T1 ·T2 ⇒ NaryR n ·T2
naryRResp -zero ·T1 (naryRZ ·T1 -eqT1) ·T2 -eq =
  naryRZ ·T2 -(trans -(sym -eq) -eqT1)
naryRResp -(succ n) ·T1 (naryRS ·Ih -n r ·T1 -eqT1) ·T2 -eq =
  naryRS ·Ih -n r ·T2 -(trans -(sym -eq) -eqT1)
```

Figure 7 *NaryR n* respects type equality

handled with δ. For the *naryRS* case, the inductive hypothesis gives us that Ih_1 and Ih_2 are equal types. We combine the lemma *arrowTpEqCod* (definition omitted) that the arrow type constructor respects type equality in its codomain with the proofs given to the constructors that $T \to Ih_1$ is equal to T_1 and $T \to Ih_2$ is equal to T_2, concluding the proof.

```
arrowTpEqCod : ∀ D: ⋆. ∀ C1: ⋆. ∀ C2: ⋆. TpEq ·C1 ·C2 ⇒ TpEq ·(D → C1) ·(D → C2)

naryRUnique : ∀ n: Nat. ∀ T1: ⋆. NaryR n ·T1 → ∀ T2: ⋆. NaryR n ·T2 → TpEq ·T1 ·T2
naryRUnique -zero ·T1 (naryRZ ·T1 -eqT1) ·T2 (naryRZ ·T2 -eqT2) =
  trans -eqT2 -(sym -eqT1)
naryRUnique -(succ n) ·T1 (naryRS ·Ih1 -n r1 ·T1 -eqT1) ·T2 (naryRS ·Ih2 -n r2 ·T2 -eqT2) =
  trans -eqT1 -(arrowTpEqCod -(naryRUnique -n r1 r2)) -(sym -eqT2)
```

Figure 8 *NaryR n* determines at most one type

Existence

Compared to the first two properties, the proof of existence, *naryREx* in Figure 9, is more involved. We take a top-down approach for its explanation to first impart the main idea. Proceeding by induction over the natural number argument, the proof relies on two lemmas: *naryZ*, which proves that *NaryR* relates *zero* to *Nary zero*, and *naryS*, which proves that *succ n* and *Nary* (*succ n*) are related if *n* and *Nary n* are. Put another way, to prove existence we need to specialize the *naryRZ* and *naryRS* constructors to the corresponding members of the *Nary* family.

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

Figure 9 *NaryR* relates *n* and *Nary n* for all *n*

The proofs of *naryZ* and *naryS* follow a similar three-part structure, so for the sake of brevity we detail the proof of the latter only (Figure 10). First, with $naryRS'$ we specialize the constructor *naryRS* to the reflexive type equality. Then, with *narySEq* we prove that the computation rule for *Nary* (*succ n*) (c.f. Figure 4a) holds for all *n* such that *NaryR n*·(*Nary n*) holds. This is proved as 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 S such *NaryR* (*succ n*) · S holds, we conclude by showing $T \to$ *Nary n* is in this family. By the erasure rules, the first argument to *intrCast* erases to $\lambda f.\, f$.
- In the second direction, we assume $f : T \to$ *Nary n* and an arbitrary type X such that *Nary* (*succ n*) · X holds, and must cast f to the type X. This is done by appealing to

```
naryRS' : ∀ Ih: ⋆. ∀ n: Nat. NaryR n ·Ih → NaryR (succ n) ·(T → Ih)
naryRS' ·Ih -n r = naryRS -n r -(refl ·(T → Ih))

narySEq : ∀ n: Nat. NaryR n ·(Nary n) → TpEq ·(Nary (succ n)) ·(T → Nary n)
narySEq -n rn =
  intrTpEq
    -(intrCast -(λ f. f ·(T → Nary n) -(naryRS' -n rn)) -(λ _. β))
    -(intrCast -(λ f. Λ X. Λ rs.
                  tpEq1 -(naryRUnique -(succ n) (naryRS' -n rn) ·X rs) f)
               -(λ _. β))

naryS : ∀ n: Nat. NaryR n ·(Nary n) → NaryR (succ n) ·(Nary (succ n))
naryS -n rn =
  naryRResp -(succ n) ·(T → Nary n) (naryRS' -n rn) -(sym -(narySEq -n rn))
```

Figure 10 *Nary*: *succ* case

```
tpEqIrrel : ∀ A: ⋆. ∀ B: ⋆. TpEq ·A ·B ⇒ TpEq ·A ·B
tpEqIrrel ·A ·B -eq =
  intrTpEq -(intrCast -(tpEq1 -eq) -(λ _. β)) -(intrCast -(tpEq2 -eq) -(λ _. β))

naryZC : TpEq ·(Nary zero) ·T
naryZC = naryZEq

narySC : ∀ n: Nat. TpEq ·(Nary (succ n)) ·(T → Nary n)
narySC -n = tpEqIrrel -(narySEq -n (naryREx n))
```

Figure 11 Computation laws of *Nary* as type equalities

uniqueness, as *NaryR* relates *succ* n to both X and $T \to Nary\ n$. Notice that while $rs : Nary\ (succ\ n) \cdot X$ must not occur within the erasure of the body of the Λ-expression which binds it, the elimination form *tpEq1* takes the type equality that *naryRUnique* computes from *rs* as an *erased* argument, ensuring that this condition holds.

Finally, we prove *naryS* by appealing to respectfulness.

3.2 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, it would seem that any type coercions using this proof could not be zero-cost (that is, constant time). We now show that this issue is neatly dealt with by using the fact that type equality is *proof-irrelevant*.

Proof irrelevance for a type P is often understood to mean that any two proofs of P are equal. While type equalities do satisfy this notion of proof irrelevance, in CDLE (and other theories with usage restrictions on binders), one can formulate an alternative notion of proof irrelevance: that one can construct a proof of P from an *erased* proof of P, i.e., that the type $P \Rightarrow P$ is inhabited.

The proof of proof-irrelevance for type equality, and the type equalities for the computation laws of *Nary*, are shown in Figure 11. In *narySC*, we invoke the existence proof within an expression given as an *erased* argument to *tpEqIrrel*. Thus, no computation involving n is performed in the operational semantics of CDLE when using these type coercions.

```
appN : ∀ n: Nat. Nary n → Vec ·T n → T
appN -zero f vnil = tpEq1 -naryZC f
appN -(succ n) f (vcons -n x xs) = appN -n (tpEq1 -(narySC -n) f x) xs
```

Figure 12 Application of an n-ary function to n arguments

We conclude this section with an example: applying an n-ary function to n arguments of type T, given as a length-indexed list (*Vec*). In Figure 12, *appN* is defined by induction on the list of arguments. 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 $T \to$ *Nary* n to apply it to the head of the list, then recursively call *appN* on the tail.

4 Generic Programming Case Studies

In the previous section, we gave a detailed outline of the recipe for simulating a large elimination, in particular showing explicitly the use of type coercions. 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 lemmas showing that CDLE's primitive type constructors respect *TpEq* (except for quantification over higher-kinded type constructors, as mentioned in Remark 1). As CDLE is a kernel theory (and thus not 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 task of implementing 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 5). 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

The preceding section described an example of arity-generic programming. We consider now a *type-generic* task: proving the *no confusion* property [4] of datatype constructors (that is, they are injective and disjoint) for a closed universe of strictly positive types. For defining a universe of datatypes, 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 13, 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*, is to be used to choose one of the datatype constructors. It takes an argument n : *Nat* (the number of datatype 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 2. 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 4.

```
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 13 A closed universe of strictly positive types

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 $in : F\ \mu F \to \mu F$.

▶ Note. For comparison with *Decode*, the corresponding relational encoding used in our implementation is given in Figure 14 as *DecodeR*. Since the type argument T does not

```
data DecodeR (T: ⋆) : Descr → ⋆ → ⋆
= decIdR    : ∀ X: ⋆. TpEq ·X T ⇒ DecodeR idD ·X
| decConstR : ∀ X: ⋆. TpEq ·X ·Unit ⇒ DecodeR constD ·X
| decPairR  : ∀ d1: Descr. ∀ Ih1: ⋆. DecodeR d1 ·Ih1 →
              ∀ d2: Descr. ∀ Ih2: ⋆. DecodeR d2 ·Ih2 →
              ∀ X: ⋆. TpEq ·X ·(Pair ·Ih1 ·Ih2) ⇒
              DecodeR (pairD d1 d2) ·X
| decSumR   : ∀ c: C. ∀ f: I c → Descr. ∀ Ih: I c → ⋆.
              (Π i: I c. DecodeR (f i) ·(Ih i)) →
              ∀ X: ⋆. TpEq ·X ·(Sigma ·(I c) ·Ih) ⇒ DecodeR (sumD c f) ·X
| decSigR   : ∀ n: Nat. ∀ f: Fin n → Descr. ∀ Ih: Fin n → ⋆.
              (Π i : Fin n. DecodeR (f i) ·(Ih i)) →
              ∀ X: ⋆. TpEq ·X ·(Sigma ·(Fin n) ·Ih) ⇒ DecodeR (sigD n f) ·X
```

Figure 14 Relational encoding of *Decode*

vary through recursive calls in *Decode*, we have made it a parameter rather than index to *DecodeR* (in Cedille, recursive occurrences of the datatype being defined are not written applied to parameters; one writes e.g. $DecodeR\ id \cdot X$ rather than $DecodeR \cdot T\ id \cdot X$ in the type declarations of constructors). Aside from this, the method of translation from the syntax of pattern matching and recursion for large eliminations to GADTs with type equality constraints is the same as used in Section 3.

▶ **Example 3** (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

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

▶ **Example 4** (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 15 shows the statement and proof of the *no confusion* property of datatype constructors. *NoConfusion* is defined by case analysis over the two datatype values, and additionally abstracts over a test of equality between *i1* and *i2* to determine whether those values were both formed by the same constructor. The clause for 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* proceeds by abstracting over the same equality test, using the lemma *inSigInj* that *inSig* is injective (definition omitted; it follows from injectivity of both *in* and the constructor for pairs) to finish the two cases.

4.2 Type Inhabitation of Simple Types

In this section we describe a decision procedure for type inhabitation for a universe of simple types. Figure 16 lists the datatype of codes (*SimpleTp*) and its decoding by the large elimination (*Simple*). Like the previous section, the code is parametric in a type C of codes for base types and an interpretation $I : C \to \star$ mapping them to Cedille types.

```
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 eq = { d1 ≃ d2 }
NoConfusion n cs (in (i1 , d1)) (in (i2 , d2)) | no neq = False

inSigInj
: ∀ i1: Fin n. ∀ d1: Decode ·U (cs i1). ∀ i2: Fin n. ∀ d2: Decode ·U (cs i2).
  { inSig i1 d1 ≃ inSig i2 d2 } ⇒ Pair ·{ i1 ≃ i2 } ·{ d1 ≃ d2 }

noConfusion : ∀ n: Nat. ∀ cs: Fin n → Descr.
              Π d1: U (sigD n cs). Π d2: U (sigD n cs).
              { d1 ≃ d2 } → NoConfusion d1 d2
noConfusion -n -cs (in (i1 , d1)) (in (i2 , d2)) eq | i1 =? i2
noConfusion -n -cs (in (i1 , d1)) (in (i2 , d2)) eq | yes eq' =
  snd (inSigInj -i1 -d1 -i2 -d2 eq)
noConfusion -n -cs (in (i1 , d1)) (in (i2 , d2)) eq | no neq' =
  neq' (fst (inSigInj -i1 -d1 -i2 -d2 eq))
```

Figure 15 Statement and proof of *no confusion*

```
data SimpleTp : ⋆
= baseTp  : C → SimpleTp
| arrowTp : SimpleTp → SimpleTp → SimpleTp

Simple : SimpleTp → ⋆
Simple (baseTp c)    = I c
Simple (arrowTp a b) = Simple a → Simple b

explode : ∀ X: ⋆. False → X

Not : ⋆ → ⋆
Not A = A → False

decide : Π t: SimpleTp. (Π c: C. Sum ·(I c) ·(Not ·(I c))) →
         Sum ·(Simple t) ·(Not ·(Simple t))
decide (baseTp c) ctx = ctx c
decide (arrowTp a b) ctx | decide b ctx | decide a ctx
decide (arrowTp a b) ctx | in1 witB     | _        = in1 λ _. witB
decide (arrowTp a b) ctx | in2 notB     | in1 witA = in2 λ f. notB (f witA)
decide (arrowTp a b) ctx | in2 notB     | in2 notA = in1 λ x. explode (notA x)
```

Figure 16 Decision procedure for type inhabitation of a universe of simple types

```
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 17 Vectors of types

When inhabitation of the base types is decidable, to decide inhabitation of a simple type it suffices to case split on the subdata. This decision procedure is *decide* in Figure 16. In the base case (*baseTp*) the decidability of the base type is by assumption. The proof for the arrow case (*arrowTp*) is in the style of classical logic, using the equivalence $A \to B \equiv \neg A \vee B$. Thus, we decide inhabitation of an arrow type by performing case analysis on inhabitation of its domain and codomain types.

4.3 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 [32] 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 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 $repeat : \Pi\, m : Nat.\, A_1 \to Vec \cdot A_1\ m$).

4.3.1 Vectors of Types

Our first task is to represent *Nat*-indexed families — i.e., length-indexed vectors — of types. For reasons discussed in Remark 2, it is not possible to define vectors of types which support lookup as a Cedille datatype. Instead, we use simulated large eliminations to define them as lookup functions directly. This definition, along with some operations, is shown in Figure 17.

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

```
data TVFoldR (F: ⋆ → ⋆ → ⋆): Π n: Nat. κTyVec (succ n) → ⋆ → ⋆
= tvFoldRZ : ∀ L: κTyVec num1. ∀ X: ⋆. TpEq ·X ·(TVHead zero ·L) ⇒
             TVFoldR zero ·L ·X
| tvFoldRS : ∀ n: Nat. ∀ L: κTyVec (add num2 n).
             ∀ Ih: ⋆. TVFoldR n ·(TVTail (succ n) ·L) ·Ih →
             ∀ X: ⋆. TpEq ·X ·(F ·(TVHead (succ n) ·L) ·Ih) ⇒
             TVFoldR (succ n) ·L ·X
```

Figure 18 Relational encoding of *TVFold*

```
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 n: Nat. A → Vec ·A n
vapp    : ∀ A: ⋆. ∀ B: ⋆. ∀ n: Nat. Vec ·(A → B) n → Vec ·A n → Vec ·B n

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 19 Arity-generic map

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.

Unlike the previous examples of large eliminations we have considered, *TVFold* computes a type *constructor* (of kind $\kappa\, TyVec\ (succ\ n) \to \star$). We therefore show its relational encoding as *TVFoldR* in Figure 18. As the figure suggests, little of the method of simulation described in Section 3 needs changing to accommodate this higher-kinded type constructor.

▶ Note. As noted in Remark 1, *TpEq* does not admit a general substitution principle. Concerning *TVFold*, this means if F is not congruent with respect to type equality in its second argument, then for types of the form $TVFold \cdot F\ (succ\ (succ\ n)) \cdot L$ we can simulate only one computation step.

4.3.2 *ArrTp* and *nvecMap*

We are now ready to define the arity-generic vector operation *nvecMap*, shown in Figure 19. 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*; we can prove that *Vec* respects type equality). In the successor case, *fs* is a

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

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

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

Figure 20 Variant computation laws for *TVFold*

vector of functions whose type is equal to

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

and the expected type is

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

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.

▶ Remark 5. The high-level syntax we use to present *TVCons* and *TVFold* obscures the fact that additional lemmas are needed to effectively use the latter on type vectors built with the former. These lemmas are *tvFoldZC′* and *tvFoldSC′* in Figure 20, and they may be understood as providing alternative computation laws for *TVFold* when it is applied to type vectors of the form $TVCons\ n \cdot X \cdot L$ and to type constructors F that respect extensional type equality in both arguments.

5 Generic Simulation

We now generalize the approach outlined in Section 3 and derive simulated large eliminations *generically* (here meaning *parametrically*) for datatypes. For this, we first review Mendler-style iteration and the generic framework of Firsov et al. [12] for inductive Mendler-style lambda encodings of datatypes in Cedille. Then, for an arbitrary positive datatype signature we define a notion of a Mendler-style algebra at the level of types, overcoming a technical difficulty arising from Cedille's truncated sort hierarchy, and show a sufficient condition for type-level algebras to yield a simulated large elimination.

5.1 Mendler-style Iteration and Encodings

We briefly review the datatype iteration scheme *à la* Mendler. Originally proposed by Mendler [22] 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 [30], and the same have advocated the Mendler style of coding recursion as more idiomatic than the classical formulation of structured recursion schemes [31].

▶ **Definition 6** (Mendler-style iteration). *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 iteration scheme for μF is described by the typing and computation law given for* fold *below:*

$$\frac{\Gamma \vdash T : \star \quad \Gamma \vdash a : \forall\, R{:}\star.\, (R \to T) \to F \cdot R \to T}{\Gamma \vdash \mathit{fold} \cdot T\ a : \mu F \to T} \qquad \mathit{fold} \cdot T\ a\ (\mathit{in}\ d) \rightsquigarrow a \cdot \mu F\ (\mathit{fold} \cdot T\ a)\ d$$

In Definition 6, 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. 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 $R \to T$, is the handle for making recursive calls; in the computation law, it is instantiated to $\mathit{fold} \cdot T\ a$. Finally, the last argument is an "F-collections" 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 $\mathit{in}\ d$.

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

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

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

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

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

Figure 21 Axiomatic summary of the generic framework of Firsov et al. [12]

Generic framework for Mendler-style datatypes Figure 21 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*.
- *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 informal intuition that R is a subtype of the datatype μF is made explicit by requiring a type inclusion of the former into the latter; and
  - 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 Cedille, however, which 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 $R \to \star$.

```
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 22 Mendler-style type algebras

Figure 22 shows the definition *κAlgTy* of the kind of Mendler-style type algebras with carrier $\star$ (henceforth we will refer to the actions of type algebras simply as *algebra*). Just as in the concrete derivation of Section 3, we require that type 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 type algebra A returns equal types on equal F-collections of predecessors (where the types of predecessors are resp. R_1 and R_2).

▶ **Example 7.** Let $F \cdot R = 1 + R$ be the signature of natural numbers with $zeroF : \forall\, R{:}\star.\, F \cdot R$ and $succF : \forall\, R{:}\star.\, R \to F \cdot R$ the signature's injections. The Mendler-style type algebra for n-ary functions over type T is:

```
NaryAlg : κAlgTy
NaryAlg ·R c Ih (zeroF ·R) = T
NaryAlg ·R c Ih (succF ·R n) = T → Ih n
```

Inspecting this definition, we see it indeed satisfies the above condition on type algebras, with the proof sketch as follows. Assuming xs_1 and xs_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

```
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)
```

Figure 23 Generic large elimination

zeroF case, the algebra returns T, which is equal to itself; in the *succF* case, we have $\{succF\ x_1 \simeq succF\ x_2\}$ for some $x_1 : R_1$ and $x_2 : R_2$, so by injectivity of *succF* we have $\{x_1 \simeq x_2\}$. We conclude by using our assumption to obtain that $TpEq \cdot (Ih_1\ x_1) \cdot (Ih_2\ x_2)$ and a lemma that the arrow type constructor respects type equality.

▶ Remark 8. 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 23 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). In the proof of uniqueness, shown in Figure 24, we use the condition on type algebras.

Proof idea (uniqueness). By induction on the two *FoldR* arguments, we know that the argument $x : \mu F$ has the form $in \cdot R_1\ \text{-}c_1\ xs_1$ and also the form $in \cdot R_2\ \text{-}c_2\ xs_2$ (this is what the $[-,-]$ notation means), where $xs_1 : F \cdot R_1$ and $xs_2 : F \cdot R_2$. We therefore have $|xs_1| =_{\beta\eta} |xs_2|$ (but not that R_1 and R_2 are equal).

```
foldRUnique : ∀ x: μ F. ∀ T1: ⋆. FoldR x ·T1 → ∀ T2: ⋆. FoldR x ·T2 → TpEq ·T1 ·T2
foldRUnique -([in ·R1 -c1 xs1 , in ·R2 -c2 xs2])
            ·T1 (foldR ·R1 -c1 xs1 ·Ih1 rec1 ·T1 -eqT1)
            ·T2 (foldR ·R2 -c2 xs2 ·Ih2 rec2 ·T2 -eqT2)
  = trans -eqT1 -(trans -eqA -(sym -eqT2))
  where
  ih : Π r1: R1. Π r2: R2. { r1 ≃ r2 } → TpEq ·(X1 r1) ·(X2 r2)
  ih [r1 , r2] r2 β = foldRUnique -(cast -c1 r1) ·(Ih1 r1) (rec1 r1) ·(Ih2 r2) (rec2 r2)

  eqA : TpEq ·(A ·R1 c1 ·Ih1 xs1) ·(A ·R2 c2 ·Ih2 xs2)
  eqA = AC ·R1 ·R2 -c1 -c2 ·Ih1 ·Ih2 ih xs1 xs2 β
```

Figure 24 Uniqueness proof for *FoldR*

To make use of the assumption $AC : \kappa AlgTyResp \cdot A$ (a parameter to the derivation), we must show that all equal terms of the two subtypes R_1 and R_2 are mapped by Ih_1 and Ih_2 to equal types. This is obtained by invoking the inductive hypothesis on values returned by each $rec_i : \Pi\, x : R_i.\, FoldR\ (cast\ \text{-}c_i) \cdot (Ih_i\ x)$ (for $1 \leq i \leq 2$) revealed by pattern matching (in *ih*, we again use the notation $[r_1, r_2]$ to indicate that pattern matching on the proof of equality gives us that r_1 and r_2 are equal, but *not* that R_1 and R_2 are). ◀

▶ Remark 9. At present, we are unable to express in a *single definition* folds over type-constructor algebras with arbitrarily kinded carriers. Thus, while this result 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.

6 Related Work

CDLE In an earlier formulation of CDLE [26], 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 [27]. Lifting was subsequently removed from the theory, replaced with the simpler δ axiom for proof discrimination.

Marmaduke et al. [21] 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.

MLTT and CC Smith [25] 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 [33] 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 [18] recently proposed a semantics which makes them functorial, but in which the above-given intuition fails to hold. In subsequent work, Johann et al. [17] 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 8, 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 [20, 28].

Semantics As discussed in Remark 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 [27], 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 Rod M Burstall and Joseph A Goguen. Algebras, theories and freeness: An introduction for computer scientists. In *Theoretical Foundations of Programming Methodology*, pages 329–349. Springer, 1982.

5 C. Böhm, M. Dezani-Ciancaglini, P. Peretti, and S.Ronchi Della Rocca. A discrimination algorithm inside λ-β-calculus. *Theoretical Computer Science*, 8(3):271 – 291, 1979. URL: http://www.sciencedirect.com/science/article/pii/0304397579900148, doi:https://doi.org/10.1016/0304-3975(79)90014-8.

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-Evariste Dagand and Conor McBride. Elaborating inductive definitions, 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. URL: http://doi.acm.org/10.1145/3236799, 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. URL: https://doi-org.proxy.lib.uiowa.edu/10.1145/3167087, doi:10.1145/3167087.

14 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.

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

16 Christopher Jenkins and Aaron Stump. Monotone recursive types and recursive data representations in Cedille. *To appear in Mathematical Structures for Computer Science*, 2021.

17 Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. Gadts, functoriality, parametricity: Pick two. *CoRR*, abs/2105.03389, 2021. URL: https://arxiv.org/abs/2105.03389, arXiv:2105.03389.

18 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.

19 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.

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

21 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.

22 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.

23 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.

24 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.

25 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.

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

27 Aaron Stump and Christopher Jenkins. Syntax and semantics of Cedille. Manuscript, 2018. URL: http://arxiv.org/abs/1806.04709, arXiv:1806.04709.

28 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.

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

30 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.

31 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.

32 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.

33 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.

34 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.