Under consideration for publication in Math. Struct. in Comp. Science

Monotone Recursive Types and Recursive Data Representations in Cedille

Christopher Jenkins and Aaron Stump

Computer Science, 14 MacLean Hall, The University of Iowa, Iowa City, Iowa, USA

email: `{firstname}-{lastname}@uiowa.edu`

Received 10 April 2021

Guided by Tarksi's fixpoint theorem in order theory, we show how to derive monotone recursive types with constant-time *roll* and *unroll* operations within Cedille, an impredicative, constructive, and logically consistent pure typed lambda calculus. As applications, we use monotone recursive types to generically derive two recursive representations of data in lambda calculus, the Parigot and Scott encoding, together with constant-time destructors, a recursion scheme, and expected induction principles.

1. Introduction

In type theory and programming languages, recursive types $\mu\, X.\, T$ are types where the variable X bound by μ in T stands for the entire type expression again. The relationship between a recursive type and its one-step unrolling $[\mu\, X.\, T/X]T$ is the basis for the important distinction of *iso-* and *equi-recursive* types (Crary et al., 1999) (see also Pierce, 2002, Section 20.2). With iso-recursive types, the two types are related by constant-time functions $\mathit{unroll} : \mu\, X.\, T \to [\mu\, X.\, T/X]T$ and $\mathit{roll} : [\mu\, X.\, T/X]T \to \mu\, X.\, T$ which are mutual inverses (composition of these two in any order produces a function that is extensionally the identity function). With equi-recursive types, the recursive type and its one-step unrolling are considered definitionally equal, and *unroll* and *roll* are not needed to pass between the two.

Without restrictions, adding recursive types as primitives to an otherwise terminating theory allows the typing of diverging terms. For example, let T be any type and let B abbreviate $\mu X. (X \to T)$. Then, we see that B is equivalent to $B \to T$, allowing us to assign type $B \to T$ to $\lambda x. x\ x$. From that type equivalence, we see that we may also assign type B to this term, allowing us to assign the type T to the diverging term $\Omega = (\lambda x. x\ x)\ \lambda x. x\ x$. This example shows that not only termination, but also soundness of the theory when interpreted as a logic under the Curry-Howard isomorphism (Sørensen and Urzyczyn, 2006), is lost when introducing unrestricted recursive types (T here is arbitrary, so this would imply all types are inhabited).

The usual restriction on recursive types is to require that to form (alternatively, to introduce or to eliminate) $\mu\ X.\ T$, the variable X must occur only positively in T, where the function-type operator $\to$ preserves polarity in its codomain part and switches polarity in its domain part. For example, X occurs only positively in $(X \to Y) \to Y$, while Y occurs both positively and negatively. Since positivity is a syntactic condition, it is not compositional: if X occurs positively in T_1 and in T_2, where T_2 contains also the free variable Y, this does not mean it will occur positively in $[T_1/Y]T_2$ (the substitution of T_1 for Y in T_2). For example, take T_1 to be X and T_2 to be $Y \to X$.

In search of a compositional restriction for ensuring termination in the presence of recursive types, Matthes (1999, 2002) investigated monotone iso-recursive types in a theory that requires evidence of monotonicity equivalent to the following property of a type scheme F (where the center dot indicates application to a type):

$$\forall Y. \forall Z. (Y \to Z) \to F \cdot Y \to F \cdot Z$$

In Matthes's work, monotone recursive types are an addition to an underlying type theory, and the resulting system must be analyzed anew for such properties as subject reduction, confluence, and normalization. In the present paper, we take a different approach by deriving monotone recursive types *within an existing type theory*, the Calculus of Dependent Lambda Eliminations (CDLE) (Stump, 2017; Stump and Jenkins, 2018). Given any type scheme F satisfying a form of monotonicity, we show how to define a type $Rec \cdot F$ together with constant-time rolling and unrolling functions that witness the isomorphism between $Rec \cdot F$ and $F \cdot (Rec \cdot F)$. The definitions are carried out in Cedille, an implementation of CDLE.[†] The main benefit to this approach is that the existing meta-theoretic results for CDLE – namely, confluence, logical soundness, and normalization for a class of types that includes ones defined here – apply, since they hold globally and hence perforce for the particular derivation of monotone recursive types.

Recursive representations of data in typed lambda calculi. One important application of recursive types is their use in forming inductive datatypes, especially within a pure typed lambda calculus where data must be encoded using lambda expressions. The most well-known method of lambda encoding is the *Church encoding*, or *iterative representation*, of data, which produces terms typable in System F unextended by recursive types. The main deficiency of Church-encoded data is that data destructors, such as predecessor for naturals, can take no better than linear time to compute (Parigot, 1989). Recursive types can be used to type lambda encodings with efficient data destructors (Splawski and Urzyczyn, 1999; Stump and Fu, 2016).

As practical applications of Cedille's derived recursive types, we generically derive two *recursive representations* of data described by Parigot (1989, 1992): the *Scott encoding* and the *Parigot encoding*. While both encodings support efficient data destructors,

[†] All code and proofs appearing in listings can be found in full at `https://github.com/cedille/cedille-developments/tree/master/recursive-representation-of-data`

the Parigot encoding readily supports the primitive recursion scheme but suffers from exponential-space complexity, whereas the Scott encoding has only linear-space complexity but only readily supports the case-distinction scheme. For both encodings, we derive a recursion scheme and induction principle. That this can be done for the Scott encoding in CDLE is itself a remarkable result that builds on the derivations by Lepigre and Raffalli (2019) and Parigot (1988) of a strongly normalizing recursor for Scott naturals in resp. a Curry style type theory with a sophisticated subtyping system and a logical framework. To the best of our knowledge, the generic derivation for Scott-encoded data is the second-ever demonstration of a lambda encoding with induction principles, efficient data destructors, and linear-space complexity (see Firsov et al. (2018) for the first).

Overview of this paper. We begin the remainder of this paper with an introduction to CDLE (Section 2), before proceeding to the derivation of monotone recursive types (Section 3). Our presentations of the application of recursive types in deriving lambda encodings with induction follows a common structure: Section 5 covers the Scott encoding by first giving a concrete derivation of natural numbers supporting a weak induction principle, then the fully generic derivation; Section 6 gives a concrete example for Parigot-encoded naturals with an induction principle, then the fully generic derivation, and some important properties of the generic encoding (proven within Cedille); and Section 7 revisits the Scott encoding, showing concretely how to derive the recursion principle for naturals, then generalizes to the derivation of the standard induction principle for generic Scott-encoded data, and finally shows that the same properties hold also for this derivation. Finally, Section 8 concludes by discussing related and future work.

2. Background: the Calculus of Dependent Lambda Eliminations

In this section, we review the Calculus of Dependent Lambda Eliminations (CDLE) and its implementation in the Cedille programming language (Stump, 2017; Stump and

Jenkins, 2018). CDLE is a logically consistent constructive type theory that contains as a subsystem the impredicative and extrinsically typed Calculus of Constructions (CC). It is designed to serve as a tiny kernel theory for interactive theorem provers, minimizing the trusted computing base. CDLE can be described concisely in 20 typing rules and is implemented by *Cedille Core* (Stump, 2018b), a type checker consisting of ~1K lines of Haskell. For the sake of exposition, we present the less dense version of CDLE implemented by Cedille, described by Stump and Jenkins (2018) (Stump (2017) describes an earlier version), and have slightly simplified the typing rule concerning the ρ term construct.

To achieve compactness, CDLE is a pure typed lambda calculus, and in particular has no primitive constructs for inductive datatypes. Instead, datatypes can be represented using lambda encodings. Geuvers (2001) showed that induction principles are not derivable for these in second-order dependent type theory. Stump (2018a) showed how CDLE overcomes this fundamental difficulty by extending CC with three new typing constructs: the dependent intersections of Kopylov (2003); equality over untyped terms; and the implicit products of Miquel (2001). The formation rules for these new constructs are first shown in Figure 1, and their introduction and elimination rules are explained in Section 2.2.

2.1. *Type and kind constructs*

There are two sorts of classifiers in CDLE. Kinds κ classify type constructors, and types (type constructors of kind $\star$) classify terms. Figure 1 gives the inference rules for the judgment $\Gamma \vdash \kappa$ that kind κ is well-formed under context Γ, and for judgment $\Gamma \vdash T \overset{\rightarrow}{\in} \kappa$ that type constructor T is well-formed and has kind κ under Γ. For brevity, we take these figures as implicitly specifying the grammar for types and kinds. In the inference rules, capture-avoiding substitution is written $[t/x]$ for terms and $[T/X]$ for types, and convertibility of classifiers is notated with $\cong$. The non-congruence rules for conversion for types are given in Figure 2 (the convertibility rules for kinds do not appear here as they consist entirely of congruence rules). In those rules, call-by-name reduction, written $\rightsquigarrow_{\mathbf{c}}$ and $\rightsquigarrow_{\mathbf{c}}^{*}$ for its reflexive transitive closure, is used to reduce types to weak head normal form before checking convertibility with the auxiliary relation $\cong^{\mathrm{t}}$, in which term subexpressions are checked for $\beta\eta$-equivalence modulo erasure (see Figure 4).

$$\boxed{\Gamma \vdash \kappa}$$

$$\frac{}{\Gamma \vdash \star} \qquad \frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma, x : T \vdash \kappa}{\Gamma \vdash \Pi\, x{:}T.\, \kappa} \qquad \frac{\Gamma \vdash \kappa' \quad \Gamma, X : \kappa' \vdash \kappa}{\Gamma \vdash \Pi\, X{:}\kappa'.\, \kappa}$$

$$\boxed{\Gamma \vdash T \overset{\rightarrow}{\in} \kappa}$$

$$\frac{(X : \kappa) \in \Gamma}{\Gamma \vdash X \overset{\rightarrow}{\in} \kappa} \qquad \frac{\Gamma \vdash \kappa \quad \Gamma, X : \kappa \vdash T \overset{\rightarrow}{\in} \star}{\Gamma \vdash \forall\, X{:}\kappa.\, T \overset{\rightarrow}{\in} \star}$$

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma, x : T \vdash T' \overset{\rightarrow}{\in} \star}{\Gamma \vdash \forall\, x{:}T.\, T' \overset{\rightarrow}{\in} \star} \qquad \frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma, x : T \vdash T' \overset{\rightarrow}{\in} \star}{\Gamma \vdash \Pi\, x{:}T.\, T' \overset{\rightarrow}{\in} \star}$$

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma, x : T \vdash T' \overset{\rightarrow}{\in} \kappa}{\Gamma \vdash \lambda\, x{:}T.\, T' \overset{\rightarrow}{\in} \Pi\, x{:}T.\, \kappa} \qquad \frac{\Gamma \vdash \kappa \quad \Gamma, X : \kappa \vdash T \overset{\rightarrow}{\in} \kappa'}{\Gamma \vdash \lambda\, X{:}\kappa.\, T \overset{\rightarrow}{\in} \Pi\, X{:}\kappa.\, \kappa'}$$

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \Pi\, x{:}T'.\, \kappa \quad \Gamma \vdash t \overset{\leftarrow}{\in} T'}{\Gamma \vdash T\ t \overset{\rightarrow}{\in} [t/x]\kappa} \qquad \frac{\begin{array}{c}\Gamma \vdash T_1 \overset{\rightarrow}{\in} \Pi\, X{:}\kappa_2.\, \kappa_1 \\ \Gamma \vdash T_2 \overset{\rightarrow}{\in} \kappa_2' \quad \kappa_2 \cong \kappa_2'\end{array}}{\Gamma \vdash T_1 \cdot T_2 \overset{\rightarrow}{\in} [T_2/X]\kappa_1}$$

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma, x : T \vdash T' \overset{\rightarrow}{\in} \star}{\Gamma \vdash \iota\, x{:}T.\, T' \overset{\rightarrow}{\in} \star} \qquad \frac{FV(t\ t') \subseteq dom(\Gamma)}{\Gamma \vdash \{t \simeq t'\} \overset{\rightarrow}{\in} \star}$$

Fig. 1. Types and kinds of CDLE

The kinds of CDLE are the same as those of CC: $\star$ classifies types, $\Pi\, X{:}\kappa_1.\, \kappa_2$ classifies type-level functions that abstract over type constructors, and $\Pi\, x{:}T.\, \kappa$ classifies type-level functions that abstract over terms. The type constructs that CDLE inherits from CC

$$\boxed{T_1 \cong T_2} \qquad \boxed{T_1 \cong^{\mathrm{t}} T_2}$$

$$\frac{T_1 \leadsto_{\mathbf{c}}^{*} T_1' \not\leadsto_{\mathbf{c}} \quad T_2 \leadsto_{\mathbf{c}}^{*} T_2' \not\leadsto_{\mathbf{c}} \quad T_1' \cong^{\mathrm{t}} T_2'}{T_1 \cong T_2}$$

$$\frac{}{X \cong^{\mathrm{t}} X} \qquad \frac{T_1 \cong^{\mathrm{t}} T_2 \quad |t_1| =_{\beta\eta} |t_2|}{T_1\ t_1 \cong^{\mathrm{t}} T_2\ t_2} \qquad \frac{|t_1| =_{\beta\eta} |t_1'| \quad |t_2| =_{\beta\eta} |t_2'|}{\{t_1 \simeq t_2\} \cong^{\mathrm{t}} \{t_1' \simeq t_2'\}}$$

Fig. 2. Non-congruence rules for classifier convertibility

are type variables, (impredicative) type constructor quantification $\forall\, X : \kappa.\, T$, dependent function (or *product*) types $\Pi\, x : T.\, T'$, abstractions over terms $\lambda\, x : T.\, T'$ and over type constructors $\lambda\, X : \kappa.\, T$, and application of type constructors to terms $T\ t$ and to type constructors $T_1 \cdot T_2$.

The additional type constructs are: types for dependent functions with erased arguments (or *implicit product types*) $\forall\, x : T.\, T'$; dependent intersection types $\iota\, x : T.\, T'$; and equality types $\{t \simeq t'\}$. Kinding for the first two of these follows the same format as kinding of dependent function types, e.g., $\forall\, x{:}T.\, T'$ has kind $\star$ if T has kind $\star$ and if T' has kind $\star$ under a typing context extended by the assumption $x{:}T$. For equality types, the only requirement for the type $\{t \simeq t'\}$ to be well-formed is that the free variables of both t and t' (written $FV(t\ t')$) are declared in the typing context. Thus, the equality is *untyped*, as neither t nor t' need be typable.

2.2. *Term constructs*

Figure 3 gives the type inference rules for annotated terms in Cedille. These rules are *bidirectional* (c.f. Pierce and Turner, 2000): judgment $\Gamma \vdash t \overset{\rightarrow}{\in} T$ indicates term t *synthesizes* type T under typing context Γ and judgment $\Gamma \vdash t \overset{\leftarrow}{\in} T$ indicates t can be *checked* against type T. These rules are to be read bottom-up as an algorithm for type inference, with Γ and t considered inputs in both judgments and the type T an output in the synthesis judgment and input in the checking judgment. As is common for a bidirectional

$$\boxed{\Gamma \vdash t \overset{\rightarrow}{\in} T} \quad \boxed{\Gamma \vdash t \overset{\leftarrow}{\in} T}$$

$$\Gamma \vdash t \overset{\leadsto_{\mathbf{c}}^{*}}{\in} T \;=\; \exists T'.\, \Gamma \vdash t \overset{\rightarrow}{\in} T' \land T' \leadsto_{\mathbf{c}}^{*} T$$

$$\dfrac{(x : T) \in \Gamma}{\Gamma \vdash x \overset{\rightarrow}{\in} T} \qquad \dfrac{\Gamma \vdash t \overset{\rightarrow}{\in} T' \quad T' \cong T}{\Gamma \vdash t \overset{\leftarrow}{\in} T}$$

$$\dfrac{T \leadsto_{\mathbf{c}}^{*} \Pi\, x{:}T_1.\, T_2 \quad \Gamma, x : T_1 \vdash t \overset{\leftarrow}{\in} T_2}{\Gamma \vdash \lambda\, x.\, t \overset{\leftarrow}{\in} T} \qquad \dfrac{\Gamma \vdash t \overset{\leadsto_{\mathbf{c}}^{*}}{\in} \Pi\, x{:}T'.\, T \quad \Gamma \vdash t' \overset{\leftarrow}{\in} T'}{\Gamma \vdash t\ t' \overset{\rightarrow}{\in} [t'/x]T}$$

$$\dfrac{T' \leadsto_{\mathbf{c}}^{*} \forall\, X{:}\kappa.\, T \quad \Gamma, X : \kappa \vdash t \overset{\leftarrow}{\in} T}{\Gamma \vdash \Lambda\, X.\, t \overset{\leftarrow}{\in} T'} \qquad \dfrac{\begin{array}{c}\Gamma \vdash t \overset{\leadsto_{\mathbf{c}}^{*}}{\in} \forall\, X{:}\kappa.\, T \\ \Gamma \vdash T' \overset{\rightarrow}{\in} \kappa' \quad \kappa' \cong \kappa\end{array}}{\Gamma \vdash t \cdot T' \overset{\rightarrow}{\in} [T'/X]T}$$

$$\dfrac{\begin{array}{c}T \leadsto_{\mathbf{c}}^{*} \forall\, x{:}T_1.\, T_2 \\ \Gamma, x : T_1 \vdash t \overset{\leftarrow}{\in} T_2 \quad x \notin FV(|t|)\end{array}}{\Gamma \vdash \Lambda\, x.\, t \overset{\leftarrow}{\in} T} \qquad \dfrac{\Gamma \vdash t \overset{\leadsto_{\mathbf{c}}^{*}}{\in} \forall\, x{:}T'.\, T \quad \Gamma \vdash t' \overset{\leftarrow}{\in} T'}{\Gamma \vdash t\ \text{-}t' \overset{\rightarrow}{\in} [t'/x]T}$$

$$\dfrac{\begin{array}{c}T \leadsto_{\mathbf{c}}^{*} \iota\, x{:}T_1.\, T_1 \quad \Gamma \vdash t_1 \overset{\leftarrow}{\in} T_1 \\ \Gamma \vdash t_2 \overset{\leftarrow}{\in} [t_1/x]T_2 \quad |t_1| =_{\beta\eta} |t_2|\end{array}}{\Gamma \vdash [t_1, t_2] \overset{\leftarrow}{\in} T} \qquad \dfrac{\Gamma \vdash t \overset{\leadsto_{\mathbf{c}}^{*}}{\in} \iota\, x{:}T.\, T'}{\Gamma \vdash t.1 \overset{\rightarrow}{\in} T}$$

$$\dfrac{\Gamma \vdash t \overset{\leadsto_{\mathbf{c}}^{*}}{\in} \iota\, x{:}T.\, T'}{\Gamma \vdash t.2 \overset{\rightarrow}{\in} [t.1/x]T'} \qquad \dfrac{\begin{array}{c}T \leadsto_{\mathbf{c}}^{*} \{t_1 \simeq t_2\} \\ FV(t') \subseteq dom(\Gamma) \quad |t_1| =_{\beta\eta} |t_2|\end{array}}{\Gamma \vdash \beta\{t'\} \overset{\leftarrow}{\in} T}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash t \overset{\leadsto_{\mathbf{c}}^{*}}{\in} \{t_1 \simeq t_2\} \quad \Gamma \vdash [t_2/x]T' \overset{\rightarrow}{\in} \star \\ \Gamma \vdash t' \overset{\leftarrow}{\in} [t_2/x]T' \quad [t_1/x]T' \cong T\end{array}}{\Gamma \vdash \rho\ t\ @x.T'\ \text{-}\ t' \overset{\leftarrow}{\in} T} \qquad \dfrac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash t \overset{\leftarrow}{\in} T}{\Gamma \vdash \chi\ T\ \text{-}\ t\ \overset{\rightarrow}{\in} T}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash t \overset{\leftarrow}{\in} \{t' \simeq t''\} \\ \Gamma \vdash t' \overset{\leftarrow}{\in} T \quad FV(t'') \subseteq dom(\Gamma)\end{array}}{\Gamma \vdash \varphi\ t\ \text{-}\ t'\ \{t''\} \overset{\leftarrow}{\in} T} \qquad \dfrac{\Gamma \vdash t \overset{\leadsto_{\mathbf{c}}^{*}}{\in} \{t_1 \simeq t_2\}}{\Gamma \vdash \varsigma\ t \overset{\rightarrow}{\in} \{t_2 \simeq t_1\}}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash T_1 \overset{\rightarrow}{\in} \star \\ \Gamma \vdash t_1 \overset{\leftarrow}{\in} T_1 \quad \Gamma, x{:}T_1 \vdash t_2 \overset{\leftarrow}{\in} T_2\end{array}}{\Gamma \vdash [x \lhd T_1 = t_1]\ \text{-}\ t_2 \overset{\leftarrow}{\in} T_2} \qquad \dfrac{\Gamma \vdash t \overset{\rightarrow}{\in} T' \quad T' \cong \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}}{\Gamma \vdash \delta\ \text{-}\ t \overset{\leftarrow}{\in} T}$$

Fig. 3. Rules for synthesizing a type for a term and checking a term against a type

system, Cedille has a mechanism allowing the user to ascribe a type annotation to a term: $\chi\ T$ - t synthesizes type T if T is a well-formed type of kind $\star$ and t can be checked against this type. During type inference, types may be call-by-name reduced to weak head normal form in order to reveal type constructors. For brevity, we use the shorthand $\Gamma \vdash t \overset{\rightsquigarrow_c^*}{\in} T$ (defined formally near the top of Figure 3) in some premises to indicate that t synthesizes some type T' that reduces to T.

We assume the reader is familiar with the type constructs of CDLE inherited from CC. Abstraction over types in terms is written $\Lambda\, X.\, t$, and application of terms to types (polymorphic type instantiation) is written $t \cdot T$. In code listings, type arguments are sometimes omitted when Cedille can infer these from the types of term arguments (see Jenkins and Stump (2018) for details). Local term definitions are given with

$$[x \triangleleft T_1 = t_1]\ \text{-}\ t_2$$

to be read "let x of type T_1 be t_1 in t_2," and global definitions are given with $x \triangleleft T = t.$ (ended with a period), where t is checked against type T.

In describing the new type constructs of CDLE, we make reference to the erasures of the corresponding annotations for terms. The full definition of the erasure function $|-|$, which extracts an untyped lambda calculus term from a term with type annotations, is given in Figure 4. For the term constructs of CC, type abstractions $\Lambda\, X.\, t$ erase to $|t|$ and type applications $t \cdot T$ erase to $|t|$. As a Curry-style theory, the convertibility relation of Cedille is $\beta\eta$-conversion for untyped lambda calculus terms — there is no notion of reduction or conversion for the type-annotated language of terms.

The implicit product type $\forall\, x{:}T_1.\, T_2$ of Miquel (2001) is the type of function 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

$\lvert x \rvert$	=	x	$\lvert \lambda\, x.\, t \rvert$	=	$\lambda\, x.\, \lvert t \rvert$
$\lvert t\ t' \rvert$	=	$\lvert t \rvert\ \lvert t' \rvert$	$\lvert t \cdot T \rvert$	=	$\lvert t \rvert$
$\lvert \Lambda\, x.\, t \rvert$	=	$\lvert t \rvert$	$\lvert t\ \text{-}t' \rvert$	=	$\lvert t \rvert$
$\lvert [t, t'] \rvert$	=	$\lvert t \rvert$	$\lvert t.1 \rvert$	=	$\lvert t \rvert$
$\lvert t.2 \rvert$	=	$\lvert t \rvert$	$\lvert \beta\{t\} \rvert$	=	$\lvert t \rvert$
$\lvert \rho\ t\ @x.T'\ \text{-}\ t' \rvert$	=	$\lvert t' \rvert$	$\lvert \varphi\ t\ \text{-}\ t'\ \{t''\} \rvert$	=	$\lvert t'' \rvert$
$\lvert \chi\ T\ \text{-}\ t' \rvert$	=	$\lvert t \rvert$	$\lvert \varsigma\ t \rvert$	=	$\lvert t \rvert$
$\lvert [x \lhd T_1 = t_1]\ \text{-}\ t_2 \rvert$	=	$(\lambda\, x.\, \lvert t_2 \rvert)\ \lvert t_1 \rvert$	$\lvert \delta\ \text{-}\ t \rvert$	=	$\lambda\, x.\, x$

Fig. 4. Erasure for annotated terms

occur free in the erasure of the body t. Thus, the argument can play no computational role in the function but exists solely for the purposes of typing, and the erasure of the introduction form is $|t|$. For 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 $\forall\, x{:}T_1.\, T_2$, we may write $T_1 \Rightarrow T_2$, similarly to writing $T \to T'$ for $\Pi\, x{:}T.\, T'$.

Note that the notion of computational irrelevance here is not that of a different sort of classifier for types (e.g. *Prop* in Coq, c.f. The Coq development team, 2018) that separates terms in the language into those which can be used for computation and those which cannot. Instead, it is similar to *quantitative type theory* (Atkey, 2018): relevance and irrelevance are properties of binders, indicating how a function may *use* an argument.

The dependent intersection type $\iota\, x{:}T_1.\, T_2$ of Kopylov (2003) is the type for terms t which can be assigned both type T_1 and type $[t/x]T_2$. In Cedille's annotated language, the introduction form $[t_1, t_2]$ can be checked against type $\iota\, x{:}T_1.\, T_2$ if t_1 can be checked against type T_1, t_2 can be checked against $[t_1/x]T_2$, and the two terms are $\beta\eta$-equivalent modulo erasure (written $|t_1| =_{\beta\eta} |t_2|$). For the elimination forms, if t synthesizes type $\iota\, x : T_1.\, T_2$ then $t.1$ (which erases to $|t|$) synthesizes type T_1, and $t.2$ (erasing to the same) synthesizes type $[t.1/x]T_2$. Thus, dependent intersections can be thought of as a dependent pair type where the two components are equal, and so we may "forget" the second component: $[t_1, t_2]$ erases to $|t_1|$. Put another way, dependent intersections are a restricted form of computationally transparent subset types where the proof that some term t inhabits the subset must be definitionally equal to t — and, as a consequence of this restriction, the proof may be accessed for use in computation in the form of t itself.

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 $\beta\{t'\}$ proves reflexive equations between $\beta\eta$-equivalence classes of terms: it can be checked against the type $\{t_1 \simeq t_2\}$ if $|t_1| =_{\beta\eta} |t_2|$ and if the subexpression t' has no undeclared free variables. We discuss the significance of the fact that t' is unrelated to the terms being equated, dubbed the *Kleene trick*, below. In code listings, if t' is omitted from the introduction form, it defaults to $\lambda\, x.\, x$.

The elimination form $\rho\ t\ @x.T'\ \text{-}\ t'$ for the equality type $\{t_1 \simeq t_2\}$ replaces occurrences of t_1 in the checked type with t_2 before checking t'. The user indicates the occurrences of t_1 to replace with x in the annotation $@x.T'$, which binds x in T'. The rule requires that $[t_2/x]T'$ has kind $\star$, then checks t' against this type, and finally confirms that $[t_1/x]T'$ (which might not be well-kinded) is convertible with the expected type T. The entire expression erases to $|t'|$.

Example. Assume m and n have type Nat, suc and $pred$ have type $Nat \rightarrow Nat$, and furthermore that $|pred\ (suc\ t)| =_{\beta\eta} |t|$ for all t. If e has type $\{suc\ m \simeq suc\ n\}$, then $\rho\ e\ @x.\{pred\ x \simeq pred\ (suc\ n)\}\ -\ \beta$ can be checked with type $\{m \simeq n\}$ as follows: we check that $\{pred\ (suc\ n) \simeq pred\ (suc\ n)\}$, obtained from substituting x in the ρ annotation with the right-hand side of the equation of the type of e, has kind $\star$; we check β against this type; and we check that substituting x with $suc\ m$ is convertible with the expected type $\{m \simeq n\}$. By assumption $|pred\ (suc\ m)| =_{\beta\eta} |m|$ and $|pred\ (suc\ n)| =_{\beta\eta} |n|$, so $\{m \simeq n\} \cong \{pred\ (suc\ m) \simeq pred\ (suc\ n)\}$.

Equality types in CDLE come with two additional axioms, a strong form of the direct computation rule of NuPRL (see Allen et al., 2006, Section 2.2) given by φ and proof by contradiction 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'.

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 the class of false equations to which one may apply δ, the Cedille tool implements the *Böhm-out* semi-decision procedure (Böhm et al., 1979) for discriminating between $\beta\eta$-inequivalent terms. We use δ only once in this paper as part of a final comparison between the Scott and Parigot encoding (see Section 7.3).

Finally, Cedille provides a symmetry axiom ς for equality types, with $|t|$ the erasure of $\varsigma\ t$. This axiom is purely a convenience; without ς, symmetry for equality types can be proven with ρ.

2.3. *The Kleene trick*

As mentioned earlier, the introduction form $\beta\{t'\}$ for the equality type contains a subexpression t' that is unrelated to the equated terms. By allowing t' to be any closed (in context) term, we are able to define a type of all untyped lambda calculus terms.

Definition 1 (Top). Let *Top* be the type $\{\lambda\, x.\, x \simeq \lambda\, x.\, x\}$.

We dub this the *Kleene trick*, as one may find the idea in Kleene's later definitions of numeric realizability in which any number is allowed as a realizer for a true atomic formula (Kleene, 1965).

Combined with dependent intersections, the Kleene trick also allows us to derive computationally transparent equational subset types. For example, let *Nat* again be the type of naturals with *zero* the zero value, *Bool* the type of Booleans with *tt* the truth value, and $\mathit{isEven} : \mathit{Nat} \to \mathit{Bool}$ a function returning *tt* if and only if its argument is even. Then, the type *Even* of even naturals can be defined as $\iota\, x : \mathit{Nat}.\, \{\mathit{isEven}\ x \simeq \mathit{tt}\}$. Since $|\mathit{isEven}\ \mathit{zero}| =_{\beta\eta} |\mathit{tt}|$, we can check $[\mathit{zero}, \beta\{\mathit{zero}\}]$ against type *Even*, and the expression erases to $|\mathit{zero}|$. More generally, if n is a *Nat* and t a proof that $\{\mathit{isEven}\ n \simeq \mathit{tt}\}$, then $[n, \rho\ t\ @x.\{x \simeq \mathit{tt}\}\ -\ \beta\{n\}]$ can be checked against type *Even*: the erasure of the first and second components are equal, and within the second component ρ rewrites the expected type $\{\mathit{isEven}\ n \simeq \mathit{tt}\}$ to $\{\mathit{tt} \simeq \mathit{tt}\}$ then checks $\beta\{n\}$ against this.

2.4. *Meta-theory*

It may concern the reader that, with the Kleene trick, it is possible to type non-terminating terms, leading to a failure of normalization in general in CDLE. For example, the looping term Ω, $\beta\{(\lambda\, x.\, x\ x)\ \lambda\, x.\, x\ x\}$, can be checked against type *Top*. More subtly, the φ axiom allows non-termination in inconsistent contexts. Assume there is a typing context Γ and term t such that $\Gamma \vdash t \overrightarrow{\in} \forall\, X : \star.\, X$, and let ω be the term

$$\varphi\ (t \cdot \{\lambda\, x.\, x \simeq \lambda\, x.\, x\ x\})\ -\ (\Lambda\, X.\, \lambda\, x.\, x)\ \{\lambda\, x.\, x\ x\}$$

Under Γ, ω can be checked against the type $\forall\, X : \star.\, X \to X$, and by the erasure rules ω erases to $\lambda\, x.\, x\ x$. We can then type the looping term Ω:

$$\Gamma \vdash \omega \cdot (\forall\, X : \star.\, X \to X)\ \omega \overrightarrow{\in} \forall\, X : \star.\, X \to X$$

Unlike the situation for unrestricted recursive types discussed in Section 1, the existence of non-normalizing terms does not threaten the logical consistency of CDLE. For example, extensional Martin-Löf type theory is consistent but, due to a similar difficulty with inconsistent contexts, is non-normalizing (Dybjer and Palmgren, 2016).

Proposition 2 (Stump and Jenkins, 2018).

There is no term t such that $\vdash t \overset{\rightarrow}{\in} \forall\, X : \star.\, X$.

Neither does non-termination from the Kleene trick or φ with inconsistent contexts preclude the possibility of a qualified termination guarantee. In Cedille, closed terms of a function type are call-by-name normalizing.

Proposition 3 (Stump and Jenkins, 2018). Suppose that $\vdash t \overset{\rightarrow}{\in} T$, and that there exists t' such that $\vdash t' \overset{\rightarrow}{\in} T \rightarrow \Pi\, x : T_1.\, T_2$ and $|t'| = \lambda\, x.\, x$. Then $|t|$ is call-by-name normalizing.

Lack of normalization in general does, however, mean that type inference in Cedille is formally undecidable, as there are several inference rules in which full $\beta\eta$-equivalence of terms is checked. In practice, this is not a significant impediment: even in implementations of strongly normalizing dependent type theories, it is possible for type inference to trigger conversion checking between astronomically slow functions, effectively causing the implementation to hang. For the recursive representations of inductive types we derive in this paper, we show that closed lambda encodings do indeed satisfy the criterion required to guarantee call-by-name normalization.

3. Deriving Recursive Types in Cedille

To derive recursive types in Cedille, we implement a version of Tarski's fixpoint theorem for monotone functions over a complete lattice. We recall first recall the simple corollary of Tarski's more general result (c.f. Lassez et al., 1982).

Definition 4 (f-closed). Let f be a monotone function on a preorder $(S, \sqsubseteq)$. An element $x \in S$ is said to be *f-closed* if and only if $f(x) \sqsubseteq x$.

Theorem 5 (Tarski, 1955). Suppose f is a monotone function on complete lattice $(S, \sqsubseteq, \sqcap)$. Let R be the set of f-closed elements of S and $r = \sqcap R$. Then $f(r) = r$.

The version we implement is a strengthening of this corollary, in the sense that it has weaker assumptions than Theorem 5: rather than require S be a complete lattice, we only need that S is a preorder and R has a greatest lower bound.

3.1. *Tarski's Theorem*

Theorem 6. Suppose f is a monotone function on a preorder $(S, \sqsubseteq)$, and that the set R of all f-closed elements has a greatest lower bound r. Then $f(r) \sqsubseteq r$ and $r \sqsubseteq f(r)$.

Proof.

1 First prove $f(r) \sqsubseteq r$. Since r is the greatest lower bound of R, it suffices to prove $f(r) \sqsubseteq x$ for every $x \in R$. So, let x be an arbitrary element of R, and since r is a lower bound of x, $r \sqsubseteq x$. By monotonicity, we therefore obtain $f(r) \sqsubseteq f(x)$, and since $x \in R$ we have that $f(x) \sqsubseteq x$ By transitivity, we conclude that $f(r) \sqsubseteq x$.

2 Now prove $r \sqsubseteq f(r)$. Using 1 above and monotonicity of f, we have that $f(f(r)) \sqsubseteq f(r)$. This means that $f(r) \in R$, and since r is a lower bound of R, we have $r \sqsubseteq f(r)$.

□

Notice in this proof *prima facie* impredicativity: we pick a fixpoint r of f by reference to a collection R which (by 1) contains r. We will see that this impredicativity carries over to Cedille. We will instantiate the underlying set S of the preorder in Theorem 6 to the set of Cedille types — this is why we need to relax the assumption of Theorem 5 that S is a complete lattice. However, we must still answer several questions:

— how should the ordering $\sqsubseteq$ be implemented;

— ho do we express the idea of a monotone function; and

— how do we obtain the greatest lower bound of R?

One possibility that is available in System F is to choose functions $A \to B$ as the ordering $A \sqsubseteq B$, positive type schemes T (having a free variable X, and such that $A \to B$ implies $[A/X]T \to [B/X]T$) as monotonic functions, and use universal quantification to define the desired greatest lower bound as $\forall\, X.\, (T \to X) \to X$. This approach, described by Wadler (1990), is essentially a generalization of order theory to category theory, and the terms inhabiting recursive types so derived are Church encodings. However, the resulting recursive types lack property that *roll* and *unroll* are constant-time operations. In Cedille, another possibility is available: we can interpret the ordering relation as *type inclusions*, in the sense that T_1 is included into T_2 if and only if every term t of type T_1 is definitionally equal to some term of type T_2. To show how type inclusions can be expressed as a type within Cedille (*Cast*, Section 3.3), we first demonstrate how to internalize the property that some untyped term t can be viewed has having type T (*View*, Section 3.2): type inclusions are thus a special case of internalized typing where we view $\lambda\, x.\, x$ has having type $T_1 \to T_2$.

3.2. *Views*

Figure 5 gives the implementation of the *View* type family in Cedille, and Figure 6 summarizes this derivation with an axiomatic presentation. Type $\mathit{View} \cdot T\ t$ is the subset of type T consisting of terms provably equal to the untyped (more precisely *Top*-typed, see Definition 1) term t, and is defined as the dependent intersection of terms which have type T and prove themselves equal to t.

The introduction from *intrView* takes an untyped term t_1 and two computationally irrelevant arguments: a term t_2 of type T and a proof t that t_2 is equal to t_1. The definition uses the φ axiom (Figure 3) and the Kleene trick (Section 2.3) so that the resulting $\mathit{View} \cdot T$ erases to $|t_1|$ (see Figure 4 for erasure rules). Because of the Kleene

```
module view .

import utils/top .

View ◁ Π T: ⋆. Top → ⋆ = λ T: ⋆. λ t: Top. ι x: T. { x ≃ t } .

intrView ◁ ∀ T: ⋆. Π t1: Top. ∀ t2: T. { t2 ≃ t1 } ⇒ View ·T t1
= Λ T. λ t1. Λ t2. Λ t. [ φ t - t2 { t1 } , β{ t1 } ] .

elimView ◁ ∀ T: ⋆. Π t: Top. View ·T t ⇒ T
= Λ T. λ t. Λ v. φ v.2 - v.1 { t } .

eqView ◁ ∀ T: ⋆. ∀ t: Top. ∀ v: View ·T t. { t ≃ v }
= Λ T. Λ t. Λ v. ρ v.2 @x.{ t ≃ x } - β .

selfView ◁ ∀ T: ⋆. Π t: T. View ·T β{ t }
= Λ T. λ t. intrView β{ t } -t -β .

extView
◁ ∀ S: ⋆. ∀ T: ⋆. Π t: Top. (Π x: S. View ·T β{ t x }) ⇒ View ·(S → T) t
= Λ S. Λ T. λ t. Λ v.
  intrView ·(S → T) β{ t } -(λ x. elimView β{ t x } -(v x)) -β .
```

Fig. 5. Internalized typing (`view.ced`)

$$\frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t \overleftarrow{\in} \mathit{Top}}{\Gamma \vdash \mathit{View} \cdot T\ t : \star} \qquad \frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t_1 \overleftarrow{\in} \mathit{Top} \quad \Gamma \vdash t_2 \overleftarrow{\in} \mathit{View} \cdot T\ t_1}{\Gamma \vdash \mathit{elimView} \cdot T\ t_1\ \text{-}t_2 \overrightarrow{\in} T}$$

$$\frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t_1 \overleftarrow{\in} \mathit{Top} \quad \Gamma \vdash t_2 \overleftarrow{\in} T \quad \Gamma \vdash t \overleftarrow{\in} \{t_2 \simeq t_1\}}{\Gamma \vdash \mathit{intrView} \cdot T\ t_1\ \text{-}t_2\ \text{-}t \overrightarrow{\in} \mathit{View} \cdot X\ t_1}$$

$$\frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t_1 \overleftarrow{\in} \mathit{Top} \quad \Gamma \vdash t_2 \overleftarrow{\in} \mathit{View} \cdot T\ t_1}{\Gamma \vdash \mathit{eqView} \cdot T\ \text{-}t_1\ \text{-}t_2 \overrightarrow{\in} \{t_1 \simeq t_2\}}$$

$$\begin{array}{lcl} |\mathit{elimView} \cdot T\ t_1\ \text{-}t_2| & = & (\lambda\, x.\, x)\ |t_1| \\ |\mathit{intrView} \cdot T\ t_1\ \text{-}t_2\ \text{-}t| & = & (\lambda\, x.\, x)\ |t_1| \\ |\mathit{eqView}| & = & \lambda\, x.\, x \end{array}$$

Fig. 6. Internalized typing, axiomatically

trick, the requirement that a term has type T and also proves itself equal to t_1 does not restrict the terms and types over which we may form a View. The elimination form $\mathit{elimView}$ takes an untyped term t and an erased argument v proving that t may be viewed as having type T, and returns a term of type T. The crucial property of $\mathit{elimView}$ is that this returned term is definitionally equal to t itself!

Definition eqView provides a reasoning principle for views. It states that every proof v that t may be viewed as having type T is in fact propositionally equal to t. In the body, we use ρ to rewrite the expected type with the equational component of the dependent intersection defining $\mathit{View} \cdot T\ t$.

The last two definitions of Figure 5, $\mathit{selfView}$ and $\mathit{extView}$, are auxiliary. Since they can be derived solely from the introduction and elimination forms, they are not included in the axiomatization given in Figure 6. Definition $\mathit{selfView}$ reflects the typing judgment that t has type T into the proposition that $\beta\{t\}$ can be viewed as having type T. Definition $\mathit{extView}$ provides an extensional typing principle for functions: if we can show of an untyped term t that for all inputs x of type S, $t\ x$ can be viewed as having type T, then in fact t can be viewed as having type $S \to T$.

We give a few of examples.

— Let List be the type family of lists with constructor $\mathit{nil} : \forall\, A : \star.\ \mathit{List} \cdot A$, and Bool the type of Booleans. We can construct a proof of $\mathit{View} \cdot (\mathit{List} \cdot \mathit{Bool})\ \beta\{\mathit{nil}\}$ as follows: provide for the second argument of $\mathit{intrView}$ the term $\mathit{nil} \cdot \mathit{Bool}$, and the third argument β. This is accepted because $|\beta\{\mathit{nil}\}| = |\mathit{nil} \cdot \mathit{Bool}| = |\mathit{nil}|$.

— Let S and T be types and t a term of type $\iota\, x{:}S.\, T$. We can construct a view of t as having type S using $t.1$ and β, since $|t.1| = |t|$.

— Let $\mathit{Bool} = \forall\, X : \star.\ X \to X \to X$ be the type of encoded Booleans, with $\mathit{tt} : \mathit{Bool}$ and $\mathit{ff} : \mathit{Bool}$ its constructors. Using $\mathit{extView}$, we can view the untyped term $\lambda\, x.\, x\ \mathit{tt}\ \mathit{ff}$ as having type $\mathit{Bool} \to \mathit{Bool}$. Let $t : \mathit{Bool}$ be arbitrary, then by $\mathit{selfView}$ we can construct

```
module cast.

import view .

Cast ◁ ⋆ → ⋆ → ⋆ = λ S: ⋆. λ T: ⋆. View ·(S → T) β{ λ x. x } .

intrCast ◁ ∀ S: ⋆. ∀ T: ⋆. ∀ t: S → T. (Π x: S. { t x ≃ x }) ⇒ Cast ·S ·T
= Λ S. Λ T. Λ t. Λ t'.
  extView ·S ·T β{ λ x. x } -(λ x. intrView β{ x } -(t x) -(t' x)) .

elimCast ◁ ∀ S: ⋆. ∀ T: ⋆. Cast ·S ·T ⇒ S → T
= Λ S. Λ T. Λ c. elimView β{ λ x. x } -c .

eqCast ◁ ∀ S: ⋆. ∀ T: ⋆. ∀ c: Cast ·S ·T. { λ x. x ≃ c }
= Λ S. Λ T. Λ c. eqView -β{ λ x. x } -c .
```

Fig. 7. Casts (`cast.ced`)

a proof of $\mathit{View} \cdot \mathit{Bool}\ \beta\{t \cdot \mathit{Bool}\ \mathit{tt}\ \mathit{ff}\}$. When checking that this is convertible (see Figure 2) with the expected type $\mathit{View} \cdot \mathit{Bool}\ \beta\{(\lambda\, x.\, x\ \mathit{tt}\ \mathit{ff})\ t\}$, we erase annotations in terms and find that $|t|\ |\mathit{tt}|\ |\mathit{ff}| =_{\beta\eta} (\lambda\, x.\, x\ |\mathit{tt}|\ |\mathit{ff}|)\ |t|$, as desired.

3.3. *Casts*

Type inclusions, or *casts*, are represented by functions from S to T that are provably equal to $\lambda\, x.\, x$ (see Breitner et al., 2016, and also Firsov et al., 2018 for the related notion of Curry-style "identity functions"). With types playing the role of elements of the preorder, existence of a cast from types S to T will play the role of the ordering $S \sqsubseteq T$ in the proof of Theorem 6. We give the derivation of casts in Cedille in Figure 7, and summarize our results axiomatically in Figure 8.

We define $\mathit{Cast} \cdot S \cdot T$ as a view of $\lambda\, x.\, x$ as having type $S \to T$. In intrinsic type theory, there would not be much more to say: identity functions cannot map from S to T unless S and T are convertible types. But in an extrinsic type theory like CDLE, there are many nontrivial casts, and this is especially true in the presence of the φ axiom. Indeed, by enabling the definition of *extView*, φ plays a crucial role in the definition of

$$\frac{\Gamma \vdash S \overset{\rightarrow}{\in} \star \quad \Gamma \vdash T \overset{\rightarrow}{\in} \star}{\Gamma \vdash \mathit{Cast} \cdot S \cdot T \overset{\rightarrow}{\in} \star}$$

$$\frac{\Gamma \vdash S \overset{\rightarrow}{\in} \star \quad \Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash t \overset{\leftarrow}{\in} S \to T \quad \Gamma \vdash t' \overset{\leftarrow}{\in} \Pi\, x{:}S.\, \{t\ x \simeq x\}}{\Gamma \vdash \mathit{intrCast} \cdot S \cdot T \text{ -}t \text{ -}t' \overset{\rightarrow}{\in} \mathit{Cast} \cdot S \cdot T}$$

$$\frac{\Gamma \vdash S \overset{\rightarrow}{\in} \star \quad \Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash c \overset{\leftarrow}{\in} \mathit{Cast} \cdot S \cdot T}{\Gamma \vdash \mathit{elimCast} \cdot S \cdot T \text{ -}c \overset{\rightarrow}{\in} S \to T}$$

$$\frac{\Gamma \vdash S \overset{\rightarrow}{\in} \star \quad \Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash c \overset{\leftarrow}{\in} \mathit{Cast} \cdot S \cdot T}{\Gamma \vdash \mathit{eqCast} \cdot S \cdot T \text{ -}c \overset{\rightarrow}{\in} \{\lambda\, x.\, x \simeq c\}}$$

$$\begin{array}{lcl}
|\mathit{intrCast} \cdot S \cdot T \text{ -}t \text{ -}t'| & = & (\lambda\, x.\, (\lambda\, x.\, x)\ x)\ \lambda\, x.\, x \\
|\mathit{elimCast} \cdot S \cdot T \text{ -}c| & = & (\lambda\, x.\, x)\ \lambda\, x.\, x \\
|\mathit{eqCast}| & = & \lambda\, x.\, x
\end{array}$$

Fig. 8. Casts, axiomatically

the introduction form *intrCast*, which takes as erased arguments a function $t : S \to T$ and a proof that t is *extensionally* the identity function for all terms of type S.

Without *extView*, we might expect that the body of *intrCast* should look like

$$\mathit{intrView} \cdot (S \to T)\ \beta\{\lambda\, x.\, x\} \text{ -}t \text{ -}\bullet$$

where $\bullet$ holds the place of a proof of $\{t \simeq \lambda\, x.\, x\}$. However, Cedille's type theory is intensional, so from the assumption that t behaves like the identity function on terms of type S we cannot conclude that t is equal to $\lambda\, x.\, x$. Since Cedille's operational semantics and definitional equality is over untyped terms, our assumption regarding the behavior of t on terms of type S gives us no guarantees about the behavior of t on terms of other types, and thus no guarantee about the *intensional* structure of t.

The trick used in the definition of *intrCast* is rather to give an *extensional typing* to the identity function, using the typing and property of t. In the body, we use *extView* with an erased proof that assumes an arbitrary $x : S$ and constructs a view of x having

```
castRefl ◁ ∀ S: ⋆. Cast ·S ·S
= Λ S. intrCast -(λ x. x) -(λ _. β) .

castTrans ◁ ∀ S: ⋆. ∀ T: ⋆. ∀ U: ⋆. Cast ·S ·T ⇒ Cast ·T ·U ⇒ Cast ·S ·U
= Λ S. Λ T. Λ U. Λ c1. Λ c2.
intrCast -(λ x. elimCast -c2 (elimCast -c1 x)) -(λ _. β) .

castUnique ◁ ∀ S: ⋆. ∀ T: ⋆. ∀ c1: Cast ·S ·T. ∀ c2: Cast ·S ·T. { c1 ≃ c2 }
= Λ S. Λ T. Λ c1. Λ c2. ρ ς (eqCast -c1) @x.{ x ≃ c2 } - eqCast -c2 .
```

Fig. 9. Casts form a preorder (`cast.ced`)

type T using the typing of $t\ x$ and the proof $t'\ x$ that $\{t\ x \simeq x\}$. So rather than showing t is $\lambda\, x.\, x$, we are showing that t justifies giving $\lambda\, x.\, x$ the type $S \to T$.

The elimination form, *elimCast*, takes an erased argument c of type $Cast \cdot S \cdot T$ and produced a function of type $S \to T$. *Cast* is ultimately defined using a dependent intersection, and we can see that $c.1$ has the desired type. However, c is an erased argument to *elimCast* and can only be used for the purposes of typing, not computation — so we cannot use $c.1$. Fortunately, *elimView* allows us to give $\lambda\, x.\, x$ the type $S \to T$ using c as an erased argument. The entire definition, then, is definitionally equal to $\lambda\, x.\, x$.

3.3.1. *Casts form a preorder on types.* Recall that a preorder $(S, \sqsubseteq)$ consists of a set S and a reflexive and transitive binary relation $\sqsubseteq$ over S. In the proof-relevant setting of type theory, establishing that a relation on types induces a preorder requires that we also show, for all types T_1 and T_2, proofs of $T_1 \sqsubseteq T_2$ are unique — otherwise, we might only be working in a category. We now show that *Cast* satisfies all three of these properties.

Theorem 7. *Cast induces a preorder (or thin category) on Cedille types.*

Proof. Figure 9 gives the proofs in Cedille of reflexivity (*castRefl*), transitivity (*castTrans*), and uniqueness (*castUnique*) for *Cast*. □

3.3.2. *Monotonicity.* Monotonicity of a type scheme $F : \star \to \star$ in this preorder is defined as *Mono* in Figure 10 as a lifting, for all types S and T, of any cast from S to T to a

```
module mono .

import cast .

Mono ◁ (⋆ → ⋆) → ⋆
= λ F: ⋆ → ⋆. ∀ X: ⋆. ∀ Y: ⋆. Cast ·X ·Y → Cast ·(F ·X) ·(F ·Y) .
```

Fig. 10. Monotonicity (`mono.ced`)

cast from $F \cdot S$ to $F \cdot T$. In the subsequent derivations of Scott and Parigot encodings, we shall omit the details of monotonicity proofs; once the general principle behind them is understood, these proofs are mechanical and do not provide further insight into the encoding. We give an example in Figure 11 to highlight the method.

```
NatF ◁ ⋆ → ⋆ = λ N: ⋆. ∀ X: ⋆. X → (N → X) → X.

monoNatF ◁ Mono ·NatF
= Λ X. Λ Y. λ c.
  intrCast
    -(λ n. Λ Z. λ z. λ s. n z (λ r. s (elimCast -c r)))
    -(λ n. β).
```

Fig. 11. Monotonicity for *NatF*

Type scheme *NatF* is the impredicative encoding of the signature for natural numbers. To prove that it is monotonic, we assume arbitrary types X and Y such that there is a cast c from the former to the latter, and must exhibit a cast from $\mathit{NatF} \cdot X$ to $\mathit{NatF} \cdot Y$. We do this using *intrCast* on a function that is *definitionally* equal ($\beta\eta$-convertible modulo erasure) to the identity function.

After assuming $n : \mathit{NatF} \cdot X$, we introduce a term of type $\mathit{NatF} \cdot Y$ by abstracting over a type Z and terms $z : Z$ and $s : Y \to Z$. Instantiating the type argument of n with Z, the second term argument we need to provide must have type $X \to Z$. This is done by η-expanding s and inserting the assumed cast c from X to Y.

```
module recType (F : ⋆ ➔ ⋆).

import cast .
import mono .

Rec ◁ ⋆ = ∀ X: ⋆. Cast ·(F ·X) ·X ⇒ X.

recLB ◁ ∀ X: ⋆. Cast ·(F ·X) ·X ⇒ Cast ·Rec ·X
= Λ X. Λ c. intrCast -(λ x. x -c) -(λ _. β) .

recGLB ◁ ∀ Y: ⋆. (∀ X: ⋆. Cast ·(F ·X) ·X ⇒ Cast ·Y ·X) ⇒ Cast ·Y ·Rec
= Λ Y. Λ u. intrCast -(λ y. Λ X. Λ c. elimCast -(u -c) y) -(λ _. β) .

recRoll ◁ Mono ·F ⇒ Cast ·(F ·Rec) ·Rec
= Λ mono.
  recGLB ·(F ·Rec)
    -(Λ X. Λ c. castTrans ·(F ·Rec) ·(F ·X) ·X -(mono (recLB -c)) -c) .

recUnroll ◁ Mono ·F ⇒ Cast ·Rec ·(F ·Rec)
= Λ mono. recLB -(mono (recRoll -mono)).
```

Fig. 12. Monotone recursive types derived in Cedille (`recType.ced`)

3.4. *Translating the proof of Theorem 6 to Cedille*

Figure 12 shows the translation of the proof of Theorem 6 to Cedille, deriving monotone recursive types. Cedille's module system allows us to parametrize the module shown in Figure 12 by the type scheme F, and all definitions implicitly take F as an additional type argument. In prose (in the axiomatic presentation in Figure 14), we give F explicitly.

As noted in Section 3.1, it is enough to require that the set of f-closed elements (here, F-closed types) has a greatest lower bound. In Cedille's meta-theory (Stump and Jenkins, 2018), types are interpreted as closed sets of ($\beta\eta$-equivalence classes of) terms, and in particular the meaning of an impredicative quantification $\forall\, X\!:\!\star.\, T$ is the intersection of the meanings (under different assignments of meanings to the variable X) of the body. Such an intersection functions as the greatest lower bound, as we will see.

The definition of *Rec* in Figure 12 expresses the intersection of the set of all F-closed types. This *Rec* corresponds to r in the proof of Theorem 6. Semantically, we are taking the intersection of all those sets X which are F-closed. So the greatest lower bound of the set of all f-closed elements in the context of a partial order is translated to the intersection of all F-closed types, where X being F-closed means there is a cast from $F \cdot X$ to X. We require just an erased argument of type $\mathit{Cast} \cdot (F \cdot X) \cdot X$. By making the argument erased, we express the idea that we are taking the intersection of sets of terms satisfying a property, and not a set of functions that take the property as an argument.

Theorem 8. *Rec is the greatest lower bound of the set of all types X for which there exists a cast from $F \cdot X$ to X.*

Proof. In Figure 12, definition *recLB* establishes that *Rec* is a lower bound of this set and *recGLB* establishes that for any other lower bound Y, there is a cast from Y to *Rec*. For the first, assume we have an F-closed type X and some $x : \mathit{Rec}$. It suffices to give a term of type X that is intensionally equal to x. Instantiate the type argument of x to X and use the proof that X is F-closed as an erased argument.

For *recGLB*, assume we have some Y which is a lower bound of the set of all F-closed types, witnessed by u, and a term $y : Y$. It suffices to give a term of type *Rec* intensionally equal to y. Unfolding the type of *Rec*, we assume an arbitrary type X that is F-closed, witnessed by c, and must produce a term of type X. Use the assumption u and c to cast y to the type X, noting that abstraction over X and c is erased. □

In Figure 12, *recRoll* implements part 1 of the proof of Theorem 6, and *recUnroll* implements part 2. In *recRoll*, we invoke the property that *Rec* contains any other lower bound of the set of F-closed types in order to show *Rec* contains $F \cdot \mathit{Rec}$, and must show that $F \cdot \mathit{Rec}$ is included into any arbitrary F-closed type X. We do so using the fact that *Rec* is also a lower bound of this set and so is contained in X, monotonicity of F, and transitivity of *Cast* with the assumption that $F \cdot X$ is contained in X. In *recUnroll*, we use *recRoll* and monotonicity of F to obtain that $F \cdot \mathit{Rec}$ is F-closed, then use *recLB* to

```
roll ◁ Mono ·F ⇒ F ·Rec → Rec
= Λ m. elimCast -(recRoll -m) .

unroll ◁ Mono ·F ⇒ Rec → F ·Rec
= Λ m. elimCast -(recUnroll -m) .

_ ◁ { roll   ≃ λ x. x } = β.
_ ◁ { unroll ≃ λ x. x } = β.

recIso1 ◁ { λ x. roll (unroll x) ≃ λ x. x} = β.
recIso2 ◁ { λ x. unroll (roll x) ≃ λ x. x} = β.
```

Fig. 13. Operators *roll* and *unroll* (`recType.ced`)

conclude. It is here we see the impredicativity noted earlier: in *recLB*, we instantiate the type argument of the $Rec \cdot F$ argument to the given type X; in *recUnroll*, the given type is $F \cdot (Rec \cdot F)$. This would not be possible in a predicative type theory.

3.5. *Operational semantics for Rec*

We conclude this section by giving the definitions of the constant-time *roll* and *unroll* operators for recursive types in Figure 13. The derivation of recursive types with these operators is summarized axiomatically in Figure 14.

Operations *roll* and *unroll* are implemented simply by using the elimination form for casts on resp. *recRoll* and *recUnroll*, assuming a proof m that F is monotonic. By erasure, this means both operations erase to $(\lambda\, x.\, x)\ \lambda\, x.\, x$, and thus they are both definitionally equal to $\lambda\, x.\, x$. This is show in Figure 13 with two anonymous proofs (indicated by _) of equality types that hold by β alone. This fact makes trivial the proof that these operators for recursive types satisfy the desired computational laws.

Theorem 9. For all $F : \star \to \star$ and monotonicity witnesses $m : Mono \cdot F$, function $roll {\cdot} F\ \text{-}m : F {\cdot} (Rec {\cdot} F) \to Rec {\cdot} F$ has a two-sided inverse $unroll {\cdot} F\ \text{-}m : Rec {\cdot} F \to F {\cdot} (Rec {\cdot} F)$.

Proof. By definitional equality; see *recIso1* and *recIso2* in Figure 13. □

$$\frac{\Gamma \vdash F \overset{\rightarrow}{\in} \star \to \star}{\Gamma \vdash \mathit{Rec} \cdot F \overset{\rightarrow}{\in} \star}$$

$$\frac{\Gamma \vdash F \overset{\rightarrow}{\in} \star \to \star \quad \Gamma \vdash t \overset{\leftarrow}{\in} \mathit{Mono} \cdot F}{\Gamma \vdash \mathit{roll} \cdot F \text{ -}t \overset{\rightarrow}{\in} \mathit{Rec} \to F \cdot \mathit{Rec} \cdot F} \quad \frac{\Gamma \vdash F \overset{\rightarrow}{\in} \star \to \star \quad \Gamma \vdash t \overset{\leftarrow}{\in} \mathit{Mono} \cdot F}{\Gamma \vdash \mathit{unroll} \cdot F \text{ -}t \overset{\rightarrow}{\in} \mathit{Rec} \cdot F \to F \cdot (\mathit{Rec} \cdot F)}$$

$$\begin{array}{lcl} |\mathit{roll}| & = & (\lambda\, x.\, x)\ \lambda\, x.\, x \\ |\mathit{unroll}| & = & (\lambda\, x.\, x)\ \lambda\, x.\, x \end{array}$$

Fig. 14. Monotone recursive types, axiomatically

We remark that, given the erasures of *roll* and *unroll*, the classification of *Rec* as either being iso-recursive or equi-recursive is unclear. On the one hand, $\mathit{Rec} \cdot F$ and its one-step unrolling are not definitionally equal types and require explicit operations to pass between the two. On the other hand, their *denotations* as sets of $\beta\eta$-equivalence classes of untyped lambda terms are equal, and in intensional type theories with iso-recursive types it is not usual that the η-law $\mathit{roll}\ (\mathit{unroll}\ t) = t$ holds by the operational semantics (however, the β-law $\mathit{unroll}\ (\mathit{roll}\ t) = t$ should, unless one is satisfied with Church encodings). *Rec* is, instead, a synthesis of these two formulations of recursive types.

4. Datatypes and recursion schemes

Before we proceed with the application of derived recursive types to encodings of datatypes with induction in Cedille, we first elaborate on the close connection between datatypes, structured recursion schemes, and impredicative encodings. An inductive datatype D can be understood semantically as the least fixpoint of a signature functor F. Together with D comes a generic constructor $\mathit{inD} : F \cdot D \to D$, which we can understand as constructing a new value of D from an "F-collection" of predecessors. For example, the datatype *Nat* of natural numbers has the signature $\lambda\, X : \star.\, 1 + X$, where $+$ is the binary coproduct type constructor and 1 is the unitary type. The more familiar constructors $\mathit{zero} : \mathit{Nat}$ and $\mathit{suc} : \mathit{Nat} \to \mathit{Nat}$ can be merged together into a single constructor $\mathit{inNat} : (1 + \mathit{Nat}) \to \mathit{Nat}$.

What separates our derived monotone recursive types (which also constructs a fixpoint of F) and inductive datatypes is, essentially, the difference between preorder theory and category theory: *proof relevance.* In moving from the first setting to the second, we observe the following correspondences.

— The ordering corresponds to *morphisms*. Here, this means working with functions $S \to T$, not type inclusions $\mathit{Cast} \cdot S \cdot T$, and while there is at most one witness of an inclusion from one type to another there of course may be multiple functions.

— Monotonicity corresponds to *functorality.* Here, this means that a type scheme F comes with an operation *fmap* that lifts, for all types S and T, functions $S \to T$ to functions $F \cdot S \to F \cdot T$, and this lifting respects identity and composition of functions. We give a formal definition in Cedille of functors later in Section 6.2.

— Where we had F-closed sets, we now have F-algebras. Here, this means a type T together with a function $t : F \cdot T \to T$.

Carrying the correspondence further, in Section 3.4 we proved that $\mathit{Rec} \cdot F$ is a lower bound of the set of F-closed types. The related property for a datatype D with signature functor F is the existence of a *iteration operator*, *foldD*, satisfying both a typing and (because of the proof-relevant setting) a computation law. This is shown in Figure 15.

$$\frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t \overleftarrow{\in} F \cdot T \to T}{\Gamma \vdash \mathit{foldD} \cdot T\ t \overrightarrow{\in} D \to T}$$

$$|\mathit{foldD}\ t\ (\mathit{in}_F\ t')| \rightsquigarrow |t\ (\mathit{fmap}\ (\mathit{foldD}\ t)\ t')|$$

Fig. 15. Generic iteration scheme

For the typing law, we read T as the type of results we wish to iteratively compute and t as a function that constructs a result from an F-collection of previous results recursively computed from predecessors. This reading is confirmed by the computation law, which states that for all T, t, and t', the function *foldD* t acts on *inD* t' by first making recursive

$$\frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t_1 \overleftarrow{\in} T \quad \Gamma \vdash t_2 \overleftarrow{\in} T \to T}{\Gamma \vdash \mathit{foldNat} \cdot T\ t_1\ t_2 \overrightarrow{\in} \mathit{Nat} \to T}$$

$$\begin{array}{lcl} |\mathit{foldNat}\ t_1\ t_2\ \mathit{zero}| & \rightsquigarrow & |t_1| \\ |\mathit{foldNat}\ t_1\ t_2\ (\mathit{suc}\ t)| & \rightsquigarrow & |t_2\ (\mathit{foldNat} \cdot T\ t_1\ t_2\ t)| \end{array}$$

Fig. 16. Iteration scheme for *Nat*

calls on the subdata of t' (accessed using *fmap*) then using t to compute a result from this. 495 Instantiating F with the signature for *Nat*, and working through standard isomorphisms, we can specialize the typing and computation laws of the generic iteration scheme to the usual laws for iteration over *Nat*, shown in Figure 16.

Following the approach of Geuvers (2014), we present lambda encodings as solutions to structured recursion schemes over datatypes and evaluate them by how well they simulate 500 the computation laws for these schemes. Read Figure 15 as a collection of constraints with unknowns D, *foldD*, and *inD*. From these constraints, we can calculate an encoding of D in System F that is a variant of the Church encoding. For comparison, the figure also shows the familiar Church encoding of naturals, which we may similarly read from the iteration scheme for *Nat*.

$$\begin{array}{lcl} D & = & \forall\, X.\ \star\, (F \cdot X \to X) \to X \\ \mathit{foldD} & = & \Lambda\, X.\, \lambda\, a.\, \lambda\, x.\, x \cdot X\ a \\ \mathit{inD} & = & \lambda\, x.\, \Lambda\, X.\, \lambda\, a.\, a\ (\mathit{fmap} \cdot D \cdot X\ (\mathit{foldD}\ \cdot X\ a)\ x) \\ & & \\ \mathit{Nat} & = & \forall\, X{:}\star.\, X \to (X \to X) \to X \\ \mathit{foldNat} & = & \Lambda\, X.\, \lambda\, z.\, \lambda\, s.\, \lambda\, n.\, n \cdot X\ z\ s \\ \mathit{zero} & = & \Lambda\, X.\, \lambda\, z.\, \lambda\, s.\, z \\ \mathit{suc} & = & \lambda\, n.\, \Lambda\, X.\, \lambda\, z.\, \lambda\, s.\, s\ (\mathit{foldNat} \cdot X\ z\ s\ n) \end{array}$$

Fig. 17. Church encoding of D and *Nat*

505 For the typing law to hold, we let D be the type of functions, polymorphic in X, that take functions of type $F \cdot X \to X$ to a result of type X. For the iterator *foldD*, we simply provide its argument $t : F \cdot X \to X$ to the given encoding. Finally, we use the right-hand

side of the computation law to give a definition for constructor inD, and we can confirm that inD has type $F \cdot D \to D$. Thus, the Church encoding arises as a direct solution to the iteration scheme in (impredicative) polymorphic lambda calculi.

Notice that with the definitions of inD and $foldD$, we simulate the computation law for iteration in a constant number of β-reduction steps. For call-by-name operational semantics, we see that

$$\begin{array}{lllll} & |foldD\ t\ (inD\ t')| & \rightsquigarrow & (\lambda\, y.\, y\ |t|)\ |inD\ t'| & \rightsquigarrow\ |inD\ t'\ t| \\ \rightsquigarrow & (\lambda\, x.\, x\ |(fmap\ (foldD\ x)\ t')|)\ |t| & \rightsquigarrow & |t\ (fmap\ (foldD\ t)\ t')| & \end{array}$$

For call-by-value semantics, we would assume that t and t' are values value and first reduce $inD\ t'$.

The issue of inefficiency in computing predecessor for Church naturals has an analogue for an arbitrary datatype D supporting only the iteration scheme. The destructor for datatype D is a function $outD : D \to F \cdot D$ which satisfies the following computation law for all $t : F \cdot D$.

$$|outD\ (inD\ t)| \rightsquigarrow |t|$$

With $foldD$, we can define a candidate for the destructor that satisfies the desired typing.

$$outD = foldD\ (fmap\ inD)$$

However, this definition of $outD$ does not efficiently simulate the computation law. By definitional equality alone, we have only

$$|outD\ (inD\ t)| \rightsquigarrow^* fmap\ inD\ (fmap\ outD\ t)$$

which means we recursively destruct predecessors of t only to reconstruct them with inD. In particular, if t is a variable the recursive call becomes stuck, and we cannot reduce further to obtain a right-hand side of t.

4.1. *Characterizing datatype encodings*

Throughout the remainder of this paper, we will give thorough characterization of both the computational and extensional properties of the datatype encodings we present. We now detail the criteria we shall use, and the corresponding Cedille definitions, for the iteration scheme. This begins with Figure 18, which takes as a module parameter a type scheme $F : \star \to \star$ and gives type definitions for the typing law of the iteration scheme. Type family *Alg* gives the shape of the types of functions used in iteration, and family *Iter* gives the shape of the type of the combinator *foldD* itself.

```
module data-char/iter-typing (F: ⋆ → ⋆) .

Alg ◁ ⋆ → ⋆ = λ X: ⋆. F ·X → X .

Iter ◁ ⋆ → ⋆ = λ D: ⋆. ∀ X: ⋆. Alg ·X → D → X .
```

Fig. 18. Iteration typing (`data-char/iter-typing.ced`)

```
module data-char/iter
  (F: ⋆ → ⋆) (fmap: ∀ X: ⋆. ∀ Y: ⋆. (X → Y) → F ·X → F ·Y)
  (D: ⋆) (inD: F ·D → D).

import data-char/iter-typing ·F .

AlgHom ◁ Π X: ⋆. Alg ·X → (D → X) → ⋆
= λ X: ⋆. λ a: Alg ·X. λ h: D → X.
  ∀ xs: F ·D. { h (inD xs) ≃ a (fmap h xs) } .

IterBeta ◁ Iter ·D → ⋆
= λ iter: Iter ·D.
  ∀ X: ⋆. ∀ a: Alg ·X. AlgHom ·X a (iter a) .

IterEta ◁ Iter ·D → ⋆
= λ iter: Iter ·D.
  ∀ X: ⋆. ∀ a: Alg ·X. ∀ h: D → X. AlgHom ·X a h →
  Π x: D. { h x ≃ iter a x } .
```

Fig. 19. Iteration characterization (`data-char/iter.ced`)

Iteration scheme. Figure 19 lists the computation and extensionality laws for the iteration scheme. For the module parameters, read *fmap* as the functorial operation lifting functions over type scheme F, D as the datatype, and *inD* as the constructor. *IterBeta* expresses the computation law using the auxiliary definition *AlgHom* (the category-theoretic notion of an F-algebra homomorphism from *inD*): for all $xs : F \cdot D$, X, and $a : Alg \cdot X$, $foldD\ a\ (inD\ x)$ is propositionally equal to $a\ (fmap\ (foldD\ a)\ xs)$. For all datatype encodings and recursion schemes, we will be careful to note whether the computation law is efficiently simulated by the encoding under call-by-name and call-by-value operational semantics.

The extensionality law, given by *IterEta*, gives the property that the combinator *foldD* for iteration is a *unique* solution to the computation law. More precisely, if there is any other function $h : D \to X$ that satisfies the computation law with respect to some $a : Alg \cdot X$, then $foldD\ a$ and h are equal up to function extensionality. Extensionality laws are proved with induction.

```
module data-char/destruct (F: ⋆ → ⋆) (D: ⋆) (inD : F ·D → D) .

Destructor ◁ ⋆ = D → F ·D .

Lambek1 ◁ Destructor → ⋆
= λ outD: Destructor. Π xs: F ·D. { outD (inD xs) ≃ xs } .

Lambek2 ◁ Destructor → ⋆
= λ outD: Destructor. Π x: D. { inD (outD x) ≃ x } .
```

Fig. 20. Laws for the datatype destructor (`data-char/destruct.ced`)

Destructor. In Figure 20, *Destructor* gives the type of the generic data destructor, and *Lambek1* and *Lambek2* together state that the property that the destructor *outD* is a two-sided inverse of the constructor *inD*. The names for these properties come from *Lambek's lemma* Lambek (1968) which states that the action of the initial algebra is an isomorphism. For all encodings of datatypes, we will be careful to note whether *Lambek1*

$$\frac{\Gamma \vdash T \mathrel{\overrightarrow{\in}} \star \quad \Gamma \vdash t_1 \mathrel{\overleftarrow{\in}} T \quad \Gamma \vdash t_2 \mathrel{\overleftarrow{\in}} \mathit{Nat} \to T}{\Gamma \vdash \mathit{caseNat} \cdot T\ t_1\ t_2 \mathrel{\overrightarrow{\in}} \mathit{Nat} \to T}$$

$$\begin{array}{lcl} |\mathit{caseNat} \cdot T\ t_1\ t_2\ \mathit{zero}| & \rightsquigarrow & |t_1| \\ |\mathit{caseNat} \cdot T\ t_1\ t_2\ (\mathit{suc}\ n)| & \rightsquigarrow & |t_2\ n| \end{array}$$

Fig. 21. Typing and computation laws for case distinction on *Nat*

(the computation law) is efficiently simulated by the encoding under call-by-name and call-by-value operational semantics. As we noted earlier, for the generic Church encoding the solution for *outD* is *not* an efficient one.

5. Scott encoding

The Scott encoding was first described in unpublished lecture notes by Scott (1962), and appears also in the work of Parigot (1989, 1992). Unlike Church naturals, an efficient predecessor function is definable for Scott naturals (Parigot, 1989), but it is not known how to express the type of Scott-encoded data in System F (Splawski and Urzyczyn, 1999, point towards a negative result). Furthermore, it is not obvious how to define recursive functions over Scott naturals without a general fixpoint mechanism for terms.

As a first application of monotone recursive types in Cedille – and as a warm-up for the generic derivations to come – we show how to derive Scott-encoded natural numbers with a weak form of induction. By *weak*, we mean that this form of induction does not provide an inductive hypothesis, only a mechanism for proof by case analysis. In Section 7, we will derive both primitive recursion and standard induction for Scott encodings.

Scott encodings can be seen as a solution to the case-distinction scheme in polymorphic lambda calculi with recursive types. For the datatype *Nat*, the typing and computation laws for this scheme are given in Figure 21. Unlike the iteration scheme, in the successor case the case-distinction scheme provides direct access to the predecessor itself but no apparent form of recursion.

$$\begin{array}{lcl} \mathit{Nat} & = & \mu N.\, \forall X.\, X \to (N \to X) \to X \\ \mathit{caseNat} & = & \Lambda X.\, \lambda z.\, \lambda s.\, \lambda x.\, x \cdot X\ z\ s \\ \mathit{zero} & = & \Lambda X.\, \lambda z.\, \lambda s.\, z \\ \mathit{suc} & = & \lambda x.\, \Lambda X.\, \lambda z.\, \lambda s.\, s\ x \end{array}$$

Fig. 22. Scott naturals

Using *caseNat*, we can define the predecessor function *pred* for naturals.

$$\mathit{pred} = \mathit{caseNat}\ \mathit{zero}\ \lambda x.\, x$$

Function *pred* then computes as follows over the constructors of *Nat*.

$$\begin{array}{lclcl} |\mathit{pred}\ \mathit{zero}| & \rightsquigarrow & |\mathit{zero}| & & \\ |\mathit{pred}\ (\mathit{suc}\ t)| & \rightsquigarrow & |(\lambda x.\, x)\ t| & \rightsquigarrow & |t| \end{array}$$

Thus we see that with an efficient simulation of *caseNat*, we have an efficient implementation of the predecessor function.

Using the same method as discussed in Section 4.1, from the typing and computation laws we obtain the solutions for *Nat*, *caseNat*, *zero*, and *suc* given in Figure 22. It is then a mechanical exercise to confirm that the computation laws are efficiently simulated by these term definitions. For the definition of *Nat*, the premises of the typing law mention *Nat* itself, so a direct solution requires monotone recursive types.

5.1. *Scott-encoded naturals, concretely*

Our construction of Scott-encoded naturals supporting weak induction consists of three stages. In Figure 23 we give the definition of the non-inductive datatype signature *NatF* with its constructors. In Figure 24 we define a predicate *WkIndNatF* over types N and terms n of type $\mathit{NatF} \cdot N$ that says a certain form of weak induction suffices to prove properties about n. Finally, in Figure 25 the type *Nat* is given using recursive types and weak induction is derived.

```
module scott/concrete/nat .

import view .
import cast .
import mono .
import recType .

NatF ◁ ⋆ ➔ ⋆ = λ N: ⋆. ∀ X: ⋆. X ➔ (N ➔ X) ➔ X.

zeroF ◁ ∀ N: ⋆. NatF ·N
= Λ N. Λ X. λ z. λ s. z.

sucF ◁ ∀ N: ⋆. N ➔ NatF ·N
= Λ N. λ n. Λ X. λ z. λ s. s n.

monoNatF ◁ Mono ·NatF
= <..>
```

Fig. 23. Scott naturals (part 1) (`scott/concrete/nat.ced`)

Signature *NatF***.** In Figure 23, type scheme *NatF* is the usual impredicative encoding of the signature functor for natural numbers. Terms *zeroF* and *sucF* are its constructors, quantifying over the parameter N; using the erasure rules (Figure 4) we can confirm that these have the same erasure as would the solutions for *zero* and *suc* given in Figure 22. The proof that *NatF* is monotone is omitted, indicated by `<..>` in the figure (we detailed the proof in Section 3.3.2).

```
WkIndNatF ◁ Π N: ⋆. NatF ·N ➔ ⋆
= λ N: ⋆. λ n: NatF ·N.
  ∀ P: NatF ·N ➔ ⋆. P (zeroF ·N) ➔ (Π m: N. P (sucF m)) ➔ P n .

zeroWkIndNatF ◁ ∀ N: ⋆. WkIndNatF ·N (zeroF ·N)
= Λ N. Λ P. λ z. λ s. z .

sucWkIndNatF ◁ ∀ N: ⋆. Π n: N. WkIndNatF ·N (sucF n)
= Λ N. λ n. Λ P. λ z. λ s. s n .
```

Fig. 24. Scott naturals (part 2) (`scott/concrete/nat.ced`)

Predicate *WkIndNatF***.** We next define a predicate, parametrized by a type N, over terms of type $NatF \cdot N$. For such a term n, $WkIndNat \cdot N\ n$ is the property that, to prove $P\ n$ for arbitrary $P : NatF \cdot N \to \star$, it suffices to show certain cases for *zeroF* and *sucF*.

— In the base case, we must show that P holds for *zeroF*.

— In the step case, we must show that for arbitrary $m : N$ that P holds for $sucF\ m$.

Next in Figure 24 are proofs *zeroWkIndNatF* and *sucWkIndNatF*, which show resp. that *zeroF* satisfies the predicate *WkIndNatF* and *sucF* n satisfies this predicate for all n. Notice that *zeroWkIndNatF* is definitionally equal to *zeroF* and *sucWkIndNatF* is definitionally equal to *sucF*. We can confirm this fact by having Cedille check that β proves they are propositionally equal (_ denotes an anonymous proof):

```
_ ◁ { zeroF ≃ zeroWkIndNatF } = β .
_ ◁ { sucF ≃ sucWkIndNatF } = β .
```

This correspondence, first observed for Church encodings by Leivant (1983), between lambda-encoded data and the proofs these satisfy the datatype's typing laws, is an essential part of the recipe described by Stump (2018a) for deriving inductive types in CDLE.

*Nat***, the type of Scott naturals.** Figure 25 gives the third and final phase of the derivation of Scott naturals. The datatype signature *NatFI* is defined using a dependent intersection, producing the subset of those terms of type $NatF \cdot N$ that are definitionally equal to some proof that they satisfy the predicate $WkIndNatF \cdot N$. Monotonicity of *NatFI* is given by *monoNatFI* (proof omitted).

Using the recursive type former *Rec* derived in Section 3, we define *Nat* as the least fixpoint of *NatFI*, and specialize the rolling and unrolling operators to *NatFI*, obtaining *rollNat* and *unrollNat*. The operators, along with the facts that $|zeroF| =_{\beta\eta}$

```
NatFI ◁ ⋆ ➔ ⋆ = λ N: ⋆. ι x: NatF ·N. WkIndNatF ·N x .

monoNatFI ◁ Mono ·NatFI
= <..>

Nat ◁ ⋆ = Rec ·NatFI .
rollNat   ◁ NatFI ·Nat ➔ Nat = roll -monoNatFI .
unrollNat ◁ Nat ➔ NatFI ·Nat = unroll -monoNatFI .

zero ◁ Nat
= rollNat [ zeroF ·Nat , zeroWkIndNatF ·Nat ] .

suc ◁ Nat ➔ Nat
= λ m. rollNat [ sucF m , sucWkIndNatF m ] .

LiftNat ◁ (Nat ➔ ⋆) ➔ NatF ·Nat ➔ ⋆
= λ P: Nat ➔ ⋆. λ x: NatF ·Nat.
  ∀ v: View ·Nat β{ x }. P (elimView β{ x } -v) .

wkIndNat ◁ ∀ P: Nat ➔ ⋆. P zero ➔ (Π x: Nat. P (suc x)) ➔ Π n: Nat. P n
= Λ P. λ z. λ s. λ n.
  (unrollNat n).2 ·(LiftNat ·P) (Λ v. z) (λ m. Λ v. s m) -(selfView n) .
```

Fig. 25. Scott naturals (part 3) (`scott/concrete/nat.ced`)

$|\mathit{zeroWkIndNatF}|$ and $|\mathit{sucF}| =_{\beta\eta} |\mathit{sucWkIndNatF}|$, are then used to define the constructors *zero* and *suc*.

From the fact that $|\mathit{rollNat}| =_{\beta\eta} \lambda\, x.\, x$, and by the erasure of dependent intersection introductions, we see that the constructors for *Nat* are in fact definitionally equal to the corresponding constructors for *NatF*. We can again confirm by using Cedille to check that β proves they are propositionally equal.

```
_ ◁ { zero ≃ zeroF } = β .
_ ◁ { suc ≃ sucF } = β .
```

Weak induction for *Nat*. Weak induction for *Nat*, given by *wkIndNat* in Figure 25, allows us to prove $P\ n$ for arbitrary $n : \mathit{Nat}$ and $P : \mathit{Nat} \to \star$ if we can show that P holds of *zero* and that for arbitrary m we can construct a proof of $P\ (\mathit{suc}\ m)$. However, there is a gap between this proof principle and the proof principle $\mathit{WkIndNatF} \cdot \mathit{Nat}$ associated to n — the latter allows us to prove properties over terms of type $\mathit{NatF} \cdot \mathit{Nat}$, not terms of type Nat! To bridge this gap, we introduce a predicate transformer $\mathit{LiftNat}$ that takes properties of kind $\mathit{Nat} \to \star$ to properties of kind $\mathit{NatF} \cdot \mathit{Nat} \to \star$. For any $t : \mathit{NatF} \cdot \mathit{Nat}$, the new property $\mathit{LiftNat} \cdot P\ t$ states that P holds for t *if* we have a way of viewing t at type Nat (View, Figure 6).

The key to the proof of weak induction for Nat is that by using View, the retyping operation of terms of type $\mathit{NatF} \cdot \mathit{Nat}$ is definitionally equal to$\lambda\, x.\, x$, and the fact that the constructors for NatF are definitionally equal to the constructors for Nat. We elaborate further on this point. Let z and s be resp. the assumed proofs of the base and inductive cases. From the second projection of the unrolling of n we have a proof of $\mathit{WkIndNatF} \cdot \mathit{Nat}\ (\mathit{unroll}\ n).1$. Instantiate the predicate argument of this with $\mathit{LiftNat} \cdot P$. This gives us three subgoals:

— $\mathit{LiftNat} \cdot P\ (\mathit{zeroF} \cdot \mathit{Nat})$

Assuming $v : \mathit{View} {\cdot} \mathit{Nat}\ \beta\{\mathit{zeroF}\}$, we wish to give a proof of $P\ (\mathit{elimView}\ \beta\{\mathit{zeroF}\}\ \text{-}v)$. This is convertible with the type $P\ \mathit{zero}$ of z, since $|\mathit{elimView}\ \beta\{\mathit{zeroF}\}\ \text{-}v| =_{\beta\eta} |\mathit{zero}|$.

— $\Pi\, m{:}\mathit{Nat}.\, \mathit{LiftNat} \cdot P\ (\mathit{sucF}\ m)$

Assume we have such an m, and that v is a proof of $\mathit{View} \cdot \mathit{Nat}\ \beta\{\mathit{sucF}\ m\}$. We are expected to give a proof of $P\ (\mathit{elimView}\ \beta\{\mathit{sucF}\ m.1\}\ \text{-}v)$. The expression $s\ m$ has type $P\ (\mathit{suc}\ m)$, which is convertible with that expected type.

— $\mathit{View} \cdot \mathit{Nat}\ \beta\{(\mathit{unrollNat}\ n).1\}$

This holds by $\mathit{selfView}\ n$ of type $\mathit{View} {\cdot} \mathit{Nat}\ \beta\{n\}$, since $|\beta\{n\}| =_{\beta\eta} |\beta\{(\mathit{unrollNat}\ n).1\}|$.

5.1.1. *Computational and extensional character.* As mentioned at the outset of this section, one of the crucial characteristics of Scott-encoded naturals is that they may be used to efficiently simulate the computation laws for the case-distinction scheme. We now demonstrate this is the case for the Scott naturals we have derived. Additionally, we prove using weak induction that the solution we give for the combinator for case dis-

```
caseNat ◁ ∀ X: ⋆. X → (Nat → X) → Nat → X
= Λ X. λ z. λ s. λ n. (unrollNat n).1 z s .

caseNatBeta1 ◁ ∀ X: ⋆. ∀ z: X. ∀ s: Nat → X. { caseNat z s zero ≃ z }
= Λ X. Λ z. Λ s. β .

caseNatBeta2
◁ ∀ X: ⋆. ∀ z: X. ∀ s: Nat → X. ∀ n: Nat. { caseNat z s (suc n) ≃ s n }
= Λ X. Λ z. Λ s. Λ n. β .

pred ◁ Nat → Nat = caseNat zero λ p. p .

predBeta1 ◁ { pred zero ≃ zero } = β .

predBeta2 ◁ ∀ n: Nat. { pred (suc n) ≃ n }
= Λ n. β .

wkIndNatComp ◁ { caseNat ≃ wkIndNat } = β .
```

Fig. 26. Computation laws for case distinction and predecessor (`scott/concrete/nat.ced`)

tinction satisfies the corresponding *extensionality* law, i.e., it is the unique such solution up to function extensionality.

Computational laws. The definition of the operator *caseNat* for case distinction is given in Figure 26, along with predecessor *pred* (defined using *caseNat*) and proofs for both that they satisfy the desired computation laws (or β-laws) by definition. Compared to the solutions given in Figure 22, we have introduced an additional application *unrollNat* in *caseNat* and additional applications of *rollNat* in the constructors. However, both *rollNat* and *unrollNat* erase to $(\lambda\, x.\, x)\ (\lambda\, x.\, x)$ (Figure 14), so this introduces only a constant number of reductions in the simulation of case distinction. With an efficient operation for case distinction, we obtain an efficient predecessor *pred*.

To confirm the efficiency of this simulation, we consider the erasure of the right-hand side of the computation law for case distinction for the successor case (*caseNatBeta2*).

$$\underbrace{(\lambda z.\, \lambda s.\, \lambda n.\, \underbrace{(\lambda x.\, x)\ (\lambda x.\, x)}_{unrollNat}\ n\ z\ s)}_{caseNat}\ z\ s\ \underbrace{((\lambda n.\, \underbrace{(\lambda x.\, x)\ (\lambda x.\, x)}_{rollNat}\ (\underbrace{(\lambda n.\, \lambda z.\, \lambda s.\, s\ n)}_{sucF}\ n))}_{suc}\ n)$$

This both call-by-name and (under the assumption that n is a value) call-by-value reduces to $s\ n$ in a constant number of steps.

Additionally, it is satisfying to note that the computational content underlying the weak induction principle is precisely the same as that underlying the case-distinction scheme *caseNat*. This is proven by *wkIndComp*, which shows not just that they satisfy the same computation laws, but in fact that the two terms are definitionally equal.

```
caseNatEta
◁ ∀ X: ⋆. ∀ z: X. ∀ s: Nat → X.
  ∀ h: Nat → X. { h zero ≃ z } ⇒ (Π n: Nat. { h (suc n) ≃ s n }) ⇒
  Π n: Nat. { caseNat z s n ≃ h n }
= Λ X. Λ z. Λ s. Λ h. Λ hBeta1. Λ hBeta2.
  wkIndNat ·(λ x: Nat. { caseNat z s x ≃ h x })
    (ρ hBeta1 @x.{ z ≃ x } - β)
    (λ m. ρ (hBeta2 m) @x.{ s m ≃ x } - β) .

reflectNat ◁ Π n: Nat. { caseNat zero suc n ≃ n }
= caseNatEta ·Nat -zero -suc -(λ x. x) -β -(λ m. β) .
```

Fig. 27. Extensional laws for case distinction (`scott/concrete/nat.ced`)

Extensional laws. Using weak induction, we can prove the extensionality law (or η-law) of the case-distinction scheme. This is *caseNatEta* in Figure 27. The precise statement of uniqueness is that, for every type X and terms $z : X$ and $s : Nat \to X$, if there exists a function $h : Nat \to X$ satisfying the computation laws of the case-distinction scheme with respect to z and s, then h is extensionally equal to $caseNat\ z\ s$.

From uniqueness, we can obtain the proof *reflectNat* that using case distinction with the constructors themselves reconstructs the given number. The name for this is taken from the *reflection law* (Uustalu and Vene, 1999) of the iteration scheme for datatypes.

The standard formulation of the reflection law, and the variation given by *reflectNat*, both express that the only elements of a datatype are those generated by its constructors. This idea plays a crucial role in the future derivations of (full) induction for both the Parigot and Scott encodings.

5.2. *Scott-encoded data, generically*

For the generic Scott encoding, we begin our discussion by phrasing the case-distinction scheme *generically* (meaning *parametrically*) in a signature $F : \star \to \star$. Let D be the datatype whose signature is F and whose constructor is $inDF \cdot D \to D$. Datatype D satisfies the case-distinction scheme if there exists a definition of *caseD* satisfying the typing and computation laws listed in Figure 28.

$$\frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t \overleftarrow{\in} F \cdot D \to T}{\Gamma \vdash caseD \cdot T\ t \overrightarrow{\in} T}$$

$$|caseD \cdot T\ t\ (inD\ t')| \rightsquigarrow |t\ t'|$$

Fig. 28. Generic case-distinction scheme

We understand the computation law as saying that when acting on data constructed with *inD*, the case-distinction scheme gives to its function argument t the revealed subdata $t' : F \cdot D$ directly. Notice that unlike the iteration scheme, we do not require that F comes together with an operation *fmap* as there is no recursive call to be made on the predecessors.

From these typing and computation laws, we can define the generic destructor *outD* :

$$\begin{array}{lcl} D & = & \mu D. \forall X. (F \cdot D \to X) \to X \\ caseD & = & \Lambda X. \lambda a. \lambda x. x \cdot X\ a \\ inD & = & \lambda x. \Lambda X. \lambda a. a\ x \end{array}$$

Fig. 29. Generic Scott encoding of D

```
module data-char/case-typing (F: ⋆ → ⋆) .

AlgCase ◁ ⋆ → ⋆ → ⋆
= λ D: ⋆. λ X: ⋆. F ·D → X .

Case ◁ ⋆ → ⋆
= λ D: ⋆. ∀ X: ⋆. AlgCase ·D ·X → D → X .
```

Fig. 30. Case distinction typing (`data-char/case-typing.ced`)

$D \to F \cdot D$ that reveals the F-collection of predecessors from which a term of type D was constructed.

$$outD = caseD\ (\lambda\, x.\, x)$$

This satisfies the expected computation law for the destructor in a number of steps that is constant with respect to t.

Reading an encoding directly from these laws results in the solutions for D, *caseD*, and *inD* given in Figure 29. This is the generic Scott encoding, and is the basis for the developments in Section 5.2.2. Notice once again that the premises make reference to the datatype D itself, so in order to read the impredicative encoding of D directly from the typing law we require some form of recursive types.

5.2.1. *Characterization criteria.* We formalize in Cedille the above description of the generic case-distinction scheme in Figures 30 and 31. Definitions for the typing law of case distinction for datatype D are given in Figure 30, where the module is parametrized by a type scheme F which gives the datatype signature. The type family *AlgCase* gives the shape of the types of functions used for case distinction, and *Case* gives the shape of the type of operator *caseD* itself.

In Figure 31, we now take the signature F as well as the datatype D and its constructor *inD* as parameters. For a candidate $case : Case \cdot D$ for the operator for case distinction, the property *CaseBeta case* states that it satisfies the desired computation law, where the shape of the computation law is given by *AlgCaseHom*. *CaseEta case* is the property

```
module data-char/case
  (F: ⋆ ➔ ⋆) (D: ⋆) (inD: F ·D ➔ D).

import data-char/case-typing ·F .

AlgCaseHom ◁ Π X: ⋆. AlgCase ·D ·X ➔ (D ➔ X) ➔ ⋆
= λ X: ⋆. λ a: AlgCase ·D ·X. λ h: D ➔ X.
  ∀ xs: F ·D. { h (inD xs) ≃ a xs } .

CaseBeta ◁ Case ·D ➔ ⋆
= λ case: Case ·D.
  ∀ X: ⋆. ∀ a: AlgCase ·D ·X. AlgCaseHom ·X a (case a) .

CaseEta ◁ Case ·D ➔ ⋆
= λ case: Case ·D.
  ∀ X: ⋆. ∀ a: AlgCase ·D ·X. ∀ h: D ➔ X. AlgCaseHom ·X a h ➔
  Π x: D. { h x ≃ case a x } .
```

Fig. 31. Case distinction characterization (`data-char/case.ced`)

that *case* satisfies the extensionality law, i.e., any other function $h : D \to X$ that satisfies the computation law with respect to $a : AlgCase \cdot D \cdot X$ is extensionally equal to *case a*.

5.2.2. *Generic Scott encoding.* We now detail the generic derivation of Scott-encoded data supporting a weak induction principle. The developments in this section are parametrized by a type scheme $F : \star \to \star$ that is monotonic (the curly braces around *mono* indicate that it is an erased module parameter). As we did for the concrete derivation of naturals, the construction is separated into several phases. In Figure 32, we give the unrefined signature *DF* for Scott-encoded data and its constructor. In Figure 33, we define the predicate *WkIndDF* expressing that *DF* terms satisfy a weak induction principle, and show that the constructor satisfies this predicate. In Figure 34, we take the fixpoint of the refined signature, defining the datatype *D* with its constructor, and prove the weak induction principle for *D*.

Signature *DF* In Figure 32, *DF* is the type whose fixpoint is solution to the equation for *D* in Figure 29. Definition *inDF* is the polymorphic constructor for signature *DF*, i.e.,

```
import mono .

module scott/encoding (F: ★ → ★) {mono: Mono ·F} .

import view .
import cast .
import recType .
import utils .

import data-char/typing ·F .

DF ◁ ★ → ★ = λ D: ★. ∀ X: ★. AlgCase ·D ·X → X .

inDF ◁ ∀ D: ★. AlgCase ·D ·(DF ·D)
= Λ D. λ xs. Λ X. λ a. a xs .

monoDF ◁ Mono ·DF
= <..>
```

Fig. 32. Generic Scott encoding (part 1) (`scott/generic/encoding.ced`)

the generic grouping together of the collection of constructors for the datatype signature (e.g., *zeroF* and *sucF*, Figure 23). Notice that the erasure of *inDF* is α-equivalent to what would be the erasure of *inD* in Figure 29. Finally, *monoDF* is a proof that type scheme *DF* is monotonic (definition omitted, indicated by `<..>`).

```
WkPrfAlg ◁ Π D: ★. (DF ·D → ★) → ★
= λ D: ★. λ P: DF ·D → ★. Π xs: F ·D. P (inDF xs) .

WkIndDF ◁ Π D: ★. DF ·D → ★
= λ D: ★. λ x: DF ·D.
  ∀ P: DF ·D → ★. WkPrfAlg ·D ·P → P x .

inWkIndDF ◁ ∀ D: ★. WkPrfAlg ·D ·(WkIndDF ·D)
= Λ D. λ xs. Λ P. λ a. a xs .
```

Fig. 33. Generic Scott encoding (part 2) (`scott/generic/encoding.ced`)

Predicate *WkIndDF***.** In Figure 33, we give the definition of *WkIndDF*, the property (parametrized by type *D*) that terms of type $DF \cdot D$ satisfy a certain weak induction principle. More precisely, $WkIndDF \cdot D\ t$ is the type of proofs that, for all properties

$P : DF \cdot D \to \star$, P holds for t if a weak inductive proof can be given for P. The type of weak inductive proofs is $\mathit{WkPrfAlg} \cdot D \cdot P$, to be read "weak (F, D)-proof-algebras for P". A term a of this type takes an F-collection of D values and produces a proof that P holds for the value constructed from this using inDF.

In the concrete derivation of Scott naturals, the predicate $\mathit{WkIndNatF}$ (Figure 24) required terms of the following types be given for proofs by weak induction:

$$P\ (\mathit{zeroF} \cdot N)$$
$$\Pi\, m{:}N.\, P\ (\mathit{sucF}\ m)$$

We can understand $\mathit{WkPrfAlg}$ as combing these types together into a single type, parametrized by the signature F:

$$\Pi\, xs{:}F \cdot D.\, P\ (\mathit{inDF}\ xs)$$

Next in the figure is $\mathit{inWkIndDF}$, which is for all types D a weak (F, D)-proof-algebra for $\mathit{WkIndDF} \cdot D$. Put more concretely, it is a proof that every term of type $DF \cdot D$ that was constructed by inDF admits the weak induction principle given by $\mathit{WkIndDF} \cdot D$. The corresponding definitions from Section 5.2 are $\mathit{zeroWkIndNatF}$ and $\mathit{sucWkIndNatF}$. Observe that the proof $\mathit{inWkIndDF}$ is definitionally equal to inDF.

```
_ ◁ { inDF ≃ inWkIndDF } = β .
```

The Scott-encoded datatype D. With the inductivity predicate $\mathit{WkIndDF}$ and weak proof algebra $\mathit{inWkIndDF}$ for it, we are now able to form a refinement of signature DF whose fixpoint supports weak induction (proof by cases). This is DFI in Figure 34, which uses dependent intersections to map types D to the subset of $DF \cdot D$ which are also proofs that they satisfy $\mathit{WkIndDF} \cdot D$. Since DFI is monotonic (proof omitted), we can form the datatype D as the fixpoint of DFI using Rec, with rolling and unrolling operations rollD and $\mathit{unrollD}$ that are definitionally equal to $\lambda\, x.\, x$.

The constructor inD for D takes an F-collection of D predecessors xs, and constructs

```
DFI ◁ ⋆ ➔ ⋆ = λ D: ⋆. ι x: DF ·D. WkIndDF ·D x .

monoDFI ◁ Mono ·DFI = <..>

D ◁ ⋆ = Rec ·DFI .
rollD   ◁ DFI ·D ➔ D = roll -monoDFI .
unrollD ◁ D ➔ DFI ·D = unroll -monoDFI .

inD ◁ AlgCase ·D ·D
= λ xs. rollD [ inDF xs , inWkIndDF xs ] .

LiftD ◁ (D ➔ ⋆) ➔ DF ·D ➔ ⋆
= λ P: D ➔ ⋆. λ x: DF ·D.
  ∀ v: View ·D β{ x }. P (elimView β{ x } -v) .

wkIndD ◁ ∀ P: D ➔ ⋆. (Π xs: F ·D. P (inD xs)) ➔ Π x: D. P x
= Λ P. λ a. λ x.
  (unrollD x).2 ·(LiftD ·P) (λ xs. Λ v. a xs) -(selfView x) .
```

Fig. 34. Generic Scott encoding (part 3) (`scott/generic/encoding.ced`)

a value of type D using the fixpoint rolling operator, the constructor *inDF*, and the proof *inWkIndDF*. Note again that, by the erasure of dependent intersections, we have that *inD* and *inDF* are definitionally equal.

Weak induction for D. As was the case for the concrete encoding of Scott naturals, we must now bridge the gap between the desired weak induction principle, where we wish to prove properties of kind $D \to \star$, and what we are given by *WkIndDF* (the ability to prove properties of kind $DF \cdot D \to \star$). This is achieved using the predicate transformer *LiftD* that maps predicates over D to predicates over $DF \cdot D$ by requiring an additional assumption that the given $x : DF \cdot D$ can be viewed as having type D.

The weak induction principle *wkIndD* for D states that the property P holds for term $t : D$ if we can provide a function a which, when given an arbitrary F-collection of D predecessors, produces a proof that P holds for the successor of this collection constructed from *inD*. In the body of *wkIndD*, we invoke the proof principle $\mathit{WkIndDF} \cdot D\ (\mathit{unroll}\ x).1$, given by $(\mathit{unroll}\ x).2$, on the lifting of P. For the weak proof algebra, we apply the assumption a to the revealed predecessors xs. This expression has type $P\ (inD\ xs)$, and the expected type is $P\ (elimView\ \beta\{inDF\ xs\}\ \text{-}v)$. These two types are convertible, since the two terms in question are definitionally equal:

$$|elimView\ \beta\{inDF\ xs\}\ \text{-}v| =_{\beta\eta} |inD\ xs|$$

since in particular $|inDF| =_{\beta\eta} |inD|$.

5.2.3. *Computational and extensional character.* We now analyze the properties of our generic Scott encoding. In particular, we give the normalization guarantee for terms of type D and confirm that we can give definitions for the case-distinction scheme and destructor that both efficiently simulate their expected computation laws and provably satisfy their expected extensionality laws.

Normalization guarantee. Recall that Proposition 3 guarantees call-by-name normalization for closed terms whose type can be included into some function type. The proof *normD* of Figure 35 establishes the existence of a cast from D to $AlgCase \cdot D \cdot D \to D$, meaning that closed terms of type D satisfy this criterion.

Case-distinction scheme. We next bring into scope the definitions for characterizing the case-distinction scheme (Figure 31). For our solution *caseD* in Figure 35, *caseDBeta* proves it satisfies the computation law and *caseDEta* proves it satisfies extensional law. As we saw for the concrete example of Scott naturals in Section 5.1.1, the proof *caseDBeta* of the computation law holds by *definitional equality*, not just propositional equality, since the propositional equality is proved by β. By inspecting the definitions of *inD* and *caseD*, and the erasures of *roll* and *unroll* (Figure 14), we can confirm that in fact $caseD\ t\ (inD\ t')$ reduces to t in a number of steps that is constant with respect to t' under both call-by-name and call-by-value operational semantics (for call-by-value semantics, we would first assume t' is a value).

```
import cast .
import mono .
import recType .
import utils .

module scott/generic/props
  (F: ⋆ ➔ ⋆) {mono: Mono ·F} .

import data-char/case-typing ·F .
import scott/generic/encoding ·F -mono .

normD ◁ Cast ·D ·(AlgCase ·D ·D ➔ D)
= intrCast -(λ x. (unrollD x).1 ·D) -(λ x. β) .

import data-char/case ·F ·D inD .

caseD ◁ Case ·D
= Λ X. λ a. λ x. (unrollD x).1 a .

caseDBeta ◁ CaseBeta caseD
= Λ X. Λ a. Λ xs. β .

caseDEta ◁ CaseEta caseD
= Λ X. Λ a. Λ h. λ hBeta.
  wkIndD ·(λ x: D. { h x ≃ caseD a x })
    (λ xs. ρ (hBeta -xs) @x.{ x ≃ a xs } - β) .

reflectD ◁ Π x: D. { caseD inD x ≃ x }
= λ x. ρ ς (caseDEta ·D -inD -(id ·D) (Λ xs. β) x) @y.{ y ≃ x } - β .
```

Fig. 35. Characterization of *caseD* (`scott/generic/props.ced`)

For *caseDEta*, We proceed by weak induction where we must show that $h\ (inD\ xs)$ is propositionally equal to $caseD\ a\ (inD\ xs)$. This follows from the assumption that h satisfies the computation law with respect to $a : AlgCase \cdot D \cdot X$. As an expected result of uniqueness, *reflectD* shows that applying *caseD* to the constructor produces a function extensionally equal to the identity function.

Destructor. In Figure 36, we give the definition proposed earlier in this section for the datatype destructor *outD*. The proof *lambek1D* establishes that $|outD\ (inD\ t)|$ is definitionally equal to $|t|$ for all terms $t : F \cdot D$. As *caseD* is an efficient simulation

```
import data-char/destruct ·F ·D inD .

outD ◁ Destructor
= caseD (λ xs. xs) .

lambek1D ◁ Lambek1 outD
= λ xs. β .

lambek2D ◁ Lambek2 outD
= wkIndD ·(λ x: D. { inD (outD x) ≃ x }) (λ xs. β) .
```

Fig. 36. Characterization of *outD* (`scott/generic/props.ced`)

of the case-distinction scheme, we know that *outD* is an efficient destructor. The proof *lambek2D* establishes the other side of the isomorphism between D and $F \cdot D$, and follows by weak induction on D.

6. Parigot encoding

In this section we derive inductive Parigot-encoded data, illustrating with a concrete example in Section 6.1 the main techniques we use before proceeding with the generic derivation in Section 6.3. The Parigot encoding was first described by Parigot (1988, 1992) for natural numbers and later for a more general class of datatypes by Geuvers (2014) (wherein it is called the *Church-Scott* encoding). This encoding is a combination of the Church and Scott encoding, directly supporting access to previously computed results as well as to predecessors. For the inductive versions we derive in this section, this means that unlike the inductive Scott encoding of Section 5 we have access to an inductive hypothesis. However, for the Parigot encoding this additional power comes at a cost: the space complexity of the encoding of natural number n is exponential in n.

The Parigot encoding can be seen as a solution to the *primitive recursion scheme* in polymorphic lambda calculi with recursive types. For natural numbers, the typing and computation laws for this scheme are given in Figure 37. The significant feature of

$$\frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t_1 \overleftarrow{\in} T \quad \Gamma \vdash t_2 \overleftarrow{\in} Nat \to T \to T}{\Gamma \vdash recNat \cdot T\ t_1\ t_2 \overrightarrow{\in} Nat \to T}$$

$$\begin{array}{lcl} |recNat \cdot T\ t_1\ t_2\ zero| & \rightsquigarrow & |t_1| \\ |recNat \cdot T\ t_1\ t_2\ (suc\ n)| & \rightsquigarrow & |t_2\ n\ (recNat \cdot T\ t_1\ t_2\ n)| \end{array}$$

Fig. 37. Typing and computation laws for primitive recursion on *Nat*

$$\begin{array}{lcl} Nat & = & \mu\, N. \forall\, X. X \to (N \to X \to X) \to X \\ recNat & = & \Lambda\, X. \lambda\, z. \lambda\, s. \lambda\, x. x \cdot X\ z\ s \\ zero & = & \lambda\, X. \lambda\, z. \lambda\, s. z \\ suc & = & \lambda\, n. \Lambda\, X. \lambda\, z. \lambda\, s. s\ n\ (recNat \cdot X\ z\ s\ n) \end{array}$$

Fig. 38. Parigot naturals

primitive recursion is that in the successor case, the user-supplied function t_2 has access both to the predecessor n *and* the result recursively computed from n.

825 With primitive recursion, we can give the following implementation of the predecessor function *pred*.

$$pred = recNat \cdot Nat\ zero\ (\lambda\, x. \lambda\, y. x)$$

The efficiency of this definition of *pred* depends on the operational semantics of the language. Under call-by-name semantics, we have that $|pred\ (suc\ n)|$ reduces to n in a constant number of steps since the recursively computed result (the predecessor of 830 n) is discarded before it can be further evaluated. This is not the case for call-by-value semantics: for closed n we would compute *all* predecessors of n, then discard these results.

We can obtain Parigot naturals from the typing and computation laws for primitive recursion over naturals. The solutions for *Nat*, *recNat*, *zero*, and *suc* we acquire in this way are shown in Figure 38. In addition to being yet another demonstration of the 835 application of derived recursive types in Cedille, the derivations of this section serve two pedagogical purposes. First, the Parigot encoding more readily supports primitive recursion scheme than does the Scott encoding, for which the construction is rather complex (see Section 7). Second, the derivation of induction for Parigot-encoded data involves a very different approach than that used in Section 5, taking full advantage of the embedding of the untyped lambda calculus in Cedille.

We elaborate on this second point further: as observed by Geuvers (2014), there is a deficiency in the definition of the type of Parigot-encoded data in polymorphic type theory with recursive types. For example, the type *Nat* is not precise enough: it admits the definition of the following bogus constructor.

$$suc' = \lambda\, n.\, \lambda\, z.\, \lambda\, s.\, s\ zero\ (recNat\ z\ s\ n)$$

The difficulty is that the type does not enforce that the first argument to the bound s is the same number that we use to compute the second argument. Put another way, this represents a failure to secure the extensional law (uniqueness) for the primitive recursion scheme with this encoding.

To address this, we observe that there is a purely computational characterization of the subset of *Nat* that contains all and only the canonical Parigot naturals. This characterization is the *reflection law*: the set of closed canonical Parigot naturals is precisely the set of closed terms n of type *Nat* satisfying the following definitional equality:

$$|recNat \cdot Nat\ zero\ (\lambda\, m.\, suc)\ n| =_{\beta\eta} |n|$$

As an example, the non-canonical Parigot natural $suc'\ (suc\ zero)$ does *not* satisfy this criterion: rebuilding it with the constructors *zero* and *suc* produces $suc\ (suc\ zero)$.

With *Top* and the Kleene trick (Section 2.3), we can express the property that a term satisfies the reflection law *before* we give a type for Parigot naturals. This is good, because we wish to use the reflection law *in the definition* of the type of Parigot naturals!

```
import cast .
import mono .
import recType .
import view .
import utils/top .

module parigot/concrete/nat .

recNatU ◁ Top
= β{ λ z. λ s. λ n. n z s } .

zeroU ◁ Top
= β{ λ z. λ s. z } .

sucU ◁ Top → Top
= λ n. β{ λ z. λ s. s n (recNatU z s n) } .

reflectNatU ◁ Top
= β{ recNatU zeroU (λ m. sucU) } .

NatC ◁ Top → ★ = λ n: Top. { reflectNatU n ≃ n } .

zeroC ◁ NatC zeroU = β{ zeroU } .

sucC ◁ Π n: Top. NatC n ⇒ NatC (sucU n)
= λ n. Λ nc. ρ nc @x.{ sucU x ≃ sucU n } - β{ sucU n } .
```

Fig. 39. Parigot naturals (part 1) (`parigot/concrete/nat.ced`)

6.1. *Parigot-encoded naturals, concretely*

We split the derivation of inductive Parigot naturals into three parts. In Figure 39, we define untyped operations for Parigot naturals and prove that its untyped constructors preserve the reflection law. In Figure 40, we define the type *Nat* of canonical Parigot naturals and its constructors. Finally, in Figure 41 we define the subset of Parigot naturals supporting induction, then show that the type *Nat* is included in this subset.

Reflection law. The first definitions in Figure 39 are untyped operations for Parigot naturals. Definition *recNatU* is the combinator for primitive recursion, and *zeroU* and *sucU* are the constructors (compare these to the corresponding definitions in Figure 38).

The term *reflectU* is the function which rebuilds Parigot naturals with their constructors, and the predicate *NatC* expresses the reflection law for untyped Parigot naturals.

The proofs *zeroC* and *sucC* show respectively that *zeroU* satisfies the reflection law, and that if n satisfies the reflection law then so does *sucU* n. In the proof for *sucU*, the expected type reduces to an equality type whose right-hand side is convertible with:

$$sucU\ (reflectNatU\ n)$$

We finish the proof by rewriting with the assumption that n satisfies the reflection law.

Note that in addition to using the Kleene trick (Section 2.3) to define a type of untyped terms with *Top*, we are also using it so that the proofs *zeroC* and *sucC* are definitionally equal to the untyped constructors *zeroU* and *sucU* (see Figure 4 for the erasure of ρ).

```
_ ◁ { zeroC ≃ zeroU } = β .
_ ◁ { sucC  ≃ sucU  } = β .
```

(where _ indicates an anonymous proof). This is so that we may define Parigot naturals as an equational subset type with dependent intersection, which we will see next.

***Nat*, the type of Parigot naturals.** In Figure 40, we first define the type scheme $NatF'$ whose fixpoint over-approximates the type of Parigot naturals. Using dependent intersection, we then define the type scheme *NatF* as mapping types N to the subset of terms of type $NatF' \cdot N$ which satisfy the reflection law. This type scheme is monotonic (*monoNatF*, definition omitted), so we may use the recursive type former *Rec* to define the type *Nat* with rolling and unrolling operators *rollNat* and *unrollNat* that are definitionally equal to $\lambda\, x.\, x$ (see Figure 14).

Constructors of *Nat*. Definitions *recNat*, *zero*, and *suc* are the typed versions of the primitive recursion combinator and constructors for Parigot naturals. The definitions of the constructors are split into two parts, with $zero'$ and suc' constructing terms of

```
NatF' ◁ ⋆ → ⋆
= λ N: ⋆. ∀ X: ⋆. X → (N → X → X) → X .

NatF ◁ ⋆ → ⋆
= λ N: ⋆. ι n: NatF' ·N. NatC β{ n } .

monoNatF ◁ Mono ·NatF = <..>

Nat ◁ ⋆ = Rec ·NatF .
rollNat   ◁ NatF ·Nat → Nat = roll -monoNatF .
unrollNat ◁ Nat → NatF ·Nat = unroll -monoNatF .

recNat ◁ ∀ X: ⋆. X → (Nat → X → X) → Nat → X
= Λ X. λ z. λ s. λ n. (unrollNat n).1 z s .

zero' ◁ NatF' ·Nat = Λ X. λ z. λ s. z .

zero ◁ Nat = rollNat [ zero' , zeroC ] .

suc' ◁ Nat → NatF' ·Nat
= λ n. Λ X. λ z. λ s. s n (recNat z s n) .

suc ◁ Nat → Nat
= λ n. rollNat [ suc' n , sucC β{ n } -(unrollNat n).2 ] .
```

Fig. 40. Parigot naturals (part 2) (`parigot/concrete/nat.ced`)

type $\mathit{NatF}' \cdot \mathit{Nat}$ and the unprimed constructors combining their primed counterparts with the respective proofs that they satisfy the reflection law. For example, in *suc* the second component of the dependent intersection is a proof of $\{\mathit{reflectNatU}\ (\mathit{sucU}\ \beta\{n\}) \simeq \mathit{sucU}\ \beta\{n\}\}$ obtained from invoking the proof *sucC* with

$$(\mathit{unrollNat}\ n).2 : \{\mathit{reflectNatU}\ \ \beta\{(\mathit{unrollNat}\ n).1\} \simeq \beta\{(\mathit{unrollNat}\ n).1\}\}$$

This is accepted by Cedille by virtue of the following definition equalities:

$$\begin{array}{lllll} |\mathit{sucU}| & =_{\beta\eta} & |\mathit{suc}'| & =_{\beta\eta} & |\mathit{sucC}| \\ |\beta\{(\mathit{unrollNat}\ n).1\}| & =_{\beta\eta} & |n| & =_{\beta\eta} & |\beta\{n\}| \end{array}$$

Finally, as expected the typed and untyped versions of each of these three operations are definitionally equal.

```
IndNat ◁ Nat → ★
= λ n: Nat. ∀ P: Nat → ★. P zero → (Π m: Nat. P m → P (suc m)) → P n .

NatI ◁ ★ = ι n: Nat. IndNat n .

recNatI
◁ ∀ P: Nat → ★. P zero → (Π m: Nat. P m → P (suc m)) → Π n: NatI. P n.1
= Λ P. λ z. λ s. λ n. n.2 z s .

indZero ◁ IndNat zero
= Λ P. λ z. λ s. z .

zeroI ◁ NatI = [ zero , indZero ] .

indSuc ◁ Π n: NatI. IndNat (suc n.1)
= λ n. Λ P. λ z. λ s. s n.1 (recNatI z s n) .

sucI ◁ NatI → NatI
= λ n. [ suc n.1 , indSuc n ] .

reflectNatI ◁ Nat → NatI
= recNat zeroI (λ _. sucI) .

toNatI ◁ Cast ·Nat ·NatI
= intrCast -reflectNatI -(λ n. (unrollNat n).2) .

indNat ◁ ∀ P: Nat → ★. P zero → (Π m: Nat. P m → P (suc m)) → Π n: Nat. P n
= Λ P. λ z. λ s. λ n. recNatI z s (elimCast -toNatI n) .
```

Fig. 41. Parigot naturals (part 3) (`parigot/concrete/nat.ced`)

```
_ ◁ { recNat ≃ recNatU } = β .

_ ◁ { zero   ≃ zeroU }   = β .

_ ◁ { suc    ≃ sucU }    = β .
```

900 ***NatI*, the type of inductive Parigot naturals.** The derivation of inductive subset of Parigot naturals begins in Figure 41 with the predicate *IndNat* over *Nat*, with *IndNat n* being the property that in order to prove an arbitrary predicate P holds for n, it suffices to give corresponding proofs for the constructors *zero* and *suc*. We again note that in the successor case, we have access to both the predecessor m and a proof of $P\ m$. The

type *NatI* is then defined with dependent intersection as the subset of *Nat* for which the predicate *IndNat* holds.

Definition *recNatI* brings us close to the derivation of an induction principle for *Nat*, but does not quite achieve it. With an inductive proof for predicate P, we have only that P holds for the Parigot naturals in the inductive subset. It remains to show that *every* Parigot natural is in this subset. We begin this proof by defining the proofs *indZero* and *indSuc* stating resp. that *zero* satisfies *IndNat* and for every n in the inductive subset *NatI*, *suc* $n.1$ satisfies *IndNat*. As $|\mathit{indZero}| =_{\beta\eta} |\mathit{zero}|$ and $|\mathit{indSuc}| =_{\beta\eta} |\mathit{suc}|$, the constructors *zeroI* and *sucI* for *NatI* can be formed with dependent intersection introduction.

Reflection and induction. We can now show that every term of type *Nat* also has type *NatI*, i.e., every (canonical) Parigot natural is in the inductive subset. We do this by leveraging the fact that satisfaction of the reflection law is baked into the type Parigot naturals. First, we define *reflectNatI* which uses *recNat* to recursively rebuild a Parigot natural with the constructors *zeroI* and *sucI* of the inductive subset. Next, we observe that $|\mathit{reflectNatI}| =_{\beta\eta} |\mathit{reflectNatU}|$, so we define a cast *toNatI* where the given proof

$$(\mathit{unrollNat}\ n).2 : \{\mathit{reflectNatU}\ \ \beta\{(\mathit{unrollNat}\ n).1\} \simeq \beta\{(\mathit{unrollNat}\ n).1\}\}$$

has a type convertible with the expected type $\{\mathit{reflectNatI}\ n \simeq n\}$.

From here, the proof *indNat* of the induction principle for Parigot naturals follows from *recNatI* and the use of *toNatI* to convert the given $n : \mathit{Nat}$ to the type *NatI*.

6.1.1. *Computational and extensional character.* We now give a characterization of *Nat*. From the code listing in Figure 40, it is clear *recNat* satisfies the typing law. Figure 42 shows the proofs of the computation laws (*recNatBeta1* and *recNatBeta2*), which hold be definitional equality. By inspecting the definitions of *recNat*, *zero*, and *suc*, and from the

```
recNatBeta1
◁ ∀ X: ⋆. ∀ z: X. ∀ s: Nat → X → X.
  { recNat z s zero ≃ z }
= Λ X. Λ z. Λ s. β .

recNatBeta2
◁ ∀ X: ⋆. ∀ z: X. ∀ s: Nat → X → X. ∀ n: Nat.
  { recNat z s (suc n) ≃ s n (recNat z s n) }
= Λ X. Λ z. Λ s. Λ n. β .

indNatComp ◁ { indNat ≃ recNat } = β .

pred ◁ Nat → Nat
= recNat zero (λ n. λ r. n) .

predBeta1 ◁ { pred zero ≃ zero } = β .

predBeta2 ◁ ∀ n: Nat. { pred (suc n) ≃ n }
= Λ n. β .
```

Fig. 42. Computation laws for primitive recursion and predecessor (`parigot/concrete/nat.ced`)

erasures of *roll* and *unroll*, we can confirm that these computation laws are simulated in a constant number of reduction steps under both call-by-name and call-by-value semantics.

930 Figure 42 also includes *indNatComp*, which shows that the computational content underlying the induction principle is precisely the recursion scheme, and the predecessor function *pred* with its expected computation laws. As mentioned earlier, we must qualify that with an efficient simulation of *recNat* we only obtain an efficient solution for the predecessor function under call-by-name operational semantics.

```
recNatEta
◁ ∀ X: ⋆. ∀ z: X. ∀ s: Nat → X → X.
  ∀ h: Nat → X. { h zero ≃ z } ⇒ (Π n: Nat. { h (suc n) ≃ s n (h n) }) ⇒
  Π n: Nat. { h n ≃ recNat z s n }
= Λ X. Λ z. Λ s. Λ h. Λ hBeta1. Λ hBeta2.
  indNat ·(λ x: Nat. { h x ≃ recNat z s x })
    (ρ hBeta1 @x.{ x ≃ z } - β)
    (λ m. λ ih.
       ρ (hBeta2 m) @x.{ x ≃ s m (recNat z s m) }
     - ρ ih @x.{ s m x ≃ s m (recNat z s m) } - β) .
```

Fig. 43. Extensional law for primitive recursion (`parigot/concrete/nat.ced`)

```
module functor (F : ★ ➔ ★).

Fmap ◁ ★ = ∀ X: ★. ∀ Y: ★. (X ➔ Y) ➔ (F ·X ➔ F ·Y).

FmapId ◁ Fmap ➔ ★ = λ fmap: Fmap.
  ∀ X: ★. ∀ Y: ★. Π c: X ➔ Y. (Π x: X. {c x ≃ x}) ➔ Π x: F ·X . {fmap c x ≃ x}.

FmapCompose ◁ Fmap ➔ ★ = λ fmap: Fmap.
  ∀ X: ★. ∀ Y: ★. ∀ Z: ★. Π f: Y ➔ Z. Π g: X ➔ Y. Π x: F ·X.
  {fmap f (fmap g x) ≃ fmap (λ x. f (g x)) x}.
```

Fig. 44. Functors (`functor.ced`)

Finally, in Figure 43 we use induction to prove that for all $z : X$ and $s : Nat \to X \to X$, $recNat\ z\ s$ is the unique solution satisfying the computation laws for primitive recursion with respect to z and s. Unlike the analogous proof for *caseNat* in Section 5.1.1, in the successor case we reach a subgoal where we must prove $s\ m\ (h\ m)$ is equal to $s\ m\ (recNat\ z\ s\ m)$ for an arbitrary $m : Nat$ and function $h : Nat \to X$ which satisfies the computation laws with respect to z and s. At that point, we must use the inductive hypothesis, which is unavailable using weak induction, to conclude the proof.

6.2. *Functor and Sigma*

The statement of the generic primitive recursion scheme requires functors and pair types, and the induction principle additionally requires dependent pair types. As we will use this scheme to define the generic Parigot encoding, in this section we first show the definition of functors and the functor laws and give an axiomatic presentation of the derivation of dependent pair types with induction in Cedille.

Functors. Functors and the associated identity and composition laws for them are given in Figure 44. Analogous to monotonicity, functorality of a type scheme $F : \star \to \star$ means that F comes together with an operation $fmap : Fmap \cdot F$ lifting functions $S \to T$ to functions $F \cdot S \to F \cdot T$, for all types S and T. Unlike monotonicity, in working with functions

```
import functor.

module functorThms (F: ⋆ → ⋆) (fmap: Fmap ·F)
  {fmapId: FmapId ·F fmap} {fmapCompose: FmapCompose ·F fmap}.

import cast .
import mono .

monoFunctor ◁ Mono ·F
= Λ X. Λ Y. λ c.
  intrCast
    -(λ d. fmap (elimCast -c) d)
    -(λ d. fmapId (elimCast -c) (λ x. β) d).
```

Fig. 45. Functors and monotonicity (`functorThms`)

instead of type inclusions we find ourselves in a proof-relevant setting, so we will require that this lifting respects identity (*FmapId fmap*) and composition (*FmapCompose fmap*).

Notice also that our definition of the identity law has an extrinsic twist: the domain and codomain of the lifted function *c* need not be convertible types for us to satisfy the constraint that *c* acts extensionally like the identity function. Phrasing the identity law in this way allows us to derive a useful lemma, *monoFunctor* in Figure 45, that establishes that every functor is a monotone type scheme.

Dependent pair types. Figure 46 gives an axiomatic presentation of the dependent pair type *Sigma* (see the code repository for the full derivation). The constructor is *mksigma*, the first and second projections are *proj1* and *proj2*, and the induction principle is *indsigma*. Below the type inference rules, we confirm that the projection functions and induction principle compute as expected over pairs formed from the constructor. The type *Pair* is defined in terms of *Sigma* for the case that the type of the second component of a pair does not depend upon the first component. Additionally, we define a utility function *fork* for constructing non-dependent pairs to help express the computation law of the primitive recursion scheme.

$$\frac{\Gamma \vdash S \overset{\rightarrow}{\in} \star \quad \Gamma \vdash T \overset{\rightarrow}{\in} S \to \star}{\Gamma \vdash \mathit{Sigma} \cdot S \cdot T \overset{\rightarrow}{\in} \star}$$

$$\frac{\Gamma \vdash \mathit{Sigma} \cdot S \cdot T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash s \overset{\leftarrow}{\in} S \quad \Gamma \vdash t \overset{\leftarrow}{\in} T\ s}{\Gamma \vdash \mathit{mksigma} \cdot S \cdot T\ s\ t \overset{\rightarrow}{\in} \mathit{Sigma} \cdot S \cdot T}$$

$$\frac{\Gamma \vdash p \overset{\rightarrow}{\in} \mathit{Sigma} \cdot S \cdot T}{\Gamma \vdash \mathit{proj1}\ p \overset{\rightarrow}{\in} S} \qquad \frac{\Gamma \vdash p \overset{\rightarrow}{\in} \mathit{Sigma} \cdot S \cdot T}{\Gamma \vdash \mathit{proj2}\ p \overset{\rightarrow}{\in} T\ (\mathit{proj1}\ p)}$$

$$\frac{\Gamma \vdash p \overset{\rightarrow}{\in} \mathit{Sigma} \cdot S \cdot T \quad \Gamma \vdash P \overset{\rightarrow}{\in} \mathit{Sigma} \cdot S \cdot T \to \star \quad f : \Pi\, x{:}A.\, \Pi\, y{:}B\ a.\, P\ (\mathit{mksigma}\ x\ y)}{\Gamma \vdash \mathit{indsigma}\ p \cdot P\ f :\overset{\rightarrow}{\in} P\ p}$$

$$\begin{array}{lll}
|\mathit{proj1}\ (\mathit{mksigma}\ s\ t)| & =_{\beta\eta} & |s| \\
|\mathit{proj2}\ (\mathit{mksigma}\ s\ t)| & =_{\beta\eta} & |t| \\
|\mathit{indsigma}\ (\mathit{mksigma}\ s\ t)\ f| & =_{\beta\eta} & |f\ s\ t|
\end{array}$$

Fig. 46. *Sigma*, axiomatically (`utils/sigma.ced`)

```
Pair ◁ ★ → ★ → ★
= λ A: ★. λ B: ★. Sigma ·A ·(λ _: A. B).

fork ◁ ∀ X: ★. ∀ A: ★. ∀ B: ★. (X → A) → (X → B) → X → Pair ·A ·B
= Λ X. Λ A. Λ B. λ f. λ g. λ x. mksigma (f x) (g x) .
```

Fig. 47. *Pair* (`utils/sigma.ced`)

6.3. *Parigot-encoded data, generically*

In this section we derive inductive Parigot-encoded datatypes generically. The derivation is parametric in a signature functor F for the datatype with an operation $\mathit{fmap} : \mathit{Fmap} \cdot F$ that satisfies the functor identity and composition law.

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash t \overset{\leftarrow}{\in} F \cdot (\mathit{Pair} \cdot D \cdot T) \to T}{\Gamma \vdash \mathit{recD} \cdot T\ t \overset{\rightarrow}{\in} D \to T}$$

$$|\mathit{recD}\ t\ (\mathit{inD}\ t')| \quad =_{\beta\eta} \quad |t\ (\mathit{fmap}\ (\mathit{fork}\ \mathit{id}\ (\mathit{recD}\ t))\ t')|$$

Fig. 48. Generic primitive recursion scheme

$$\begin{array}{lcl}
D & = & \forall\, X.\, (F \cdot (\mathit{Pair} \cdot D \cdot X) \to X) \to X \\
\mathit{rec}_F & = & \Lambda\, X.\, \lambda\, t.\, \lambda\, d.\, d \cdot X\ t \\
\mathit{in}_F & = & \lambda\, d.\, \Lambda\, X.\, \lambda\, t.\, t\ (\mathit{fmap}_F\ (\mathit{fork}\ (\mathit{id} \cdot D)\ (\mathit{rec}_F\ t))\ d)
\end{array}$$

Fig. 49. Generic Parigot encoding of D

The typing and computation laws for the generic primitive recursion scheme for datatype D with signature functor F are given in Figure 48. For the typing rule, we see that the primitive recursion scheme allows recursive functions to be defined in terms an F-collection of tuples containing both direct predecessors *and* the recursive results computed from those predecessors. This reading is further affirmed by the computation law, which states that the action of $\mathit{recD}\ t$ over values built from $\mathit{inD}\ t'$ (for some $t' : F \cdot D$) is to apply t to the result of tupling each predecessor (accessed with *fmap*) with the result of $\mathit{recD}\ t$ (here *id* is the polymorphic identity function).

With primitive recursion, we can give the following implementation of the datatype destructor *outD*.

$$\mathit{outD} = \mathit{recD}\ (\mathit{fmap}\ \mathit{proj1})$$

This simulates the desired computation law $|\mathit{outD}\ (\mathit{inD}\ t)| =_{\beta\eta} |t|$ only up to the functor identity and composition laws. With definitional equality alone, we obtain a right-hand side of:

$$|\mathit{fmap}\ \mathit{proj1}\ (\mathit{fmap}\ (\mathit{fork}\ \mathit{id}\ \mathit{outD})\ t)|$$

Additionally, and as we saw for Parigot naturals, this is not an efficient implementation of the destructor under call-by-value operational semantics since the predecessors of t are recursively destructed.

Using the typing and computation laws for primitive recursion to read an encoding for D, we obtain a generic supertype of Parigot-encoded data (Figure 49). Similar to the case of the Scott encoding, we find that to give the definition of D we need monotone

```
import utils .

module primrec-typing (F: ⋆ → ⋆) .

AlgRec ◁ ⋆ → ⋆ → ⋆
= λ D: ⋆. λ X: ⋆. F ·(Pair ·D ·X) → X .

PrimRec ◁ ⋆ → ⋆
= λ D: ⋆. ∀ X: ⋆. AlgRec ·D ·X → D → X .
```

Fig. 50. Primitive recursion typing (`data-char/primrec-typing.ced`)

recursive types. For our derivation, we must further refine the type of D so that we only include canonical Parigot encodings (i.e., those build only from *inD*). We use the same approach that we took for Parigot naturals: we use *Top* and the Kleene trick to express satisfaction of the reflection law for untyped terms, then use this to give a refined definition of D.

6.3.1. *Characterization criteria.* We formalize in Cedille the above description of the generic primitive recursion scheme in Figures 50 and 51. Definitions for the typing law of the primitive recursion scheme are given in Figure 50, where parameter F gives the datatype signature. Type family *AlgRec* gives the shape of the type functions used for primitive recursion, and *PrimRec* gives the shape of the type of operator *recD* itself.

In Figure 51, we now assume that F is a functor and take additional module parameters D for the datatype and *inD* for its constructor. *AlgRecHom* gives the shape of the computation law for primitive recursion with respect to a particular $a : \mathit{AlgRec} \cdot D \cdot X$, *PrimRecBeta* is a predicate on candidates for the combinator for primitive recursion stating that it satisfies the computation law with respect to *all* such functions a, and *PrimRecEta* is predicate stating that a candidate is the unique such solution up to function extensionality.

The figure also lists *PrfAlgRec*, a dependent version of *AlgRec*. Read the type *PrfAlgRec*· P as the type of "(F, D)-proof algebras for P"; it is the type of proofs that take an F-

```
import functor .
import utils .

module data-char/primrec
  (F: ⋆ → ⋆) (fmap: Fmap ·F)
  {fmapId: FmapId ·F fmap} {fmapCompose: FmapCompose ·F fmap}
  (D: ⋆) (inD: F ·D → D).

import data-char/primrec-typing ·F .

AlgRecHom ◁ Π X: ⋆. AlgRec ·D ·X → (D → X) → ⋆
= λ X: ⋆. λ a: AlgRec ·D ·X. λ h: D → X.
  ∀ xs: F ·D. { h (inD xs) ≃ a (fmap (fork id h) xs) } .

PrimRecBeta ◁ PrimRec ·D → ⋆
= λ rec: PrimRec ·D.
  ∀ X: ⋆. ∀ a: AlgRec ·D ·X. AlgRecHom ·X a (rec a) .

PrimRecEta ◁ PrimRec ·D → ⋆
= λ rec: PrimRec ·D.
  ∀ X: ⋆. ∀ a: AlgRec ·D ·X. ∀ h: D → X. AlgRecHom ·X a h →
  Π x: D. { h x ≃ rec a x } .

PrfAlgRec ◁ (D → ⋆) → ⋆
= λ P: D → ⋆. Π xs: F ·(Sigma ·D ·P). P (inD (fmap (proj1 ·D ·P) xs)) .

import data-char/iter-typing ·F .
import data-char/case-typing ·F .

fromAlgCase ◁ ∀ X: ⋆. AlgCase ·D ·X → AlgRec ·D ·X
= Λ X. λ a. λ xs. a (fmap ·(Pair ·D ·X) ·D (λ x. proj1 x) xs) .

fromAlg ◁ ∀ X: ⋆. Alg ·X → AlgRec ·D ·X
= Λ X. λ a. λ xs. a (fmap ·(Pair ·D ·X) ·X (λ x. proj2 x) xs) .
```

Fig. 51. Primitive recursion characterization (`data-char/primrec.ced`)

collection of D predecessors tupled with proofs that P holds for them and produces a proof that P holds for the value constructed from these predecessors with *inD*. *PrfAlgRec* will be used in the derivations of full induction for both the generic Parigot and generic Scott encoding.

Finally, as the primitive recursion scheme can be used to subsume both the iteration and case-distinction scheme, the figure lists the helper functions *fromAlgCase* and *fromAlg*. Definition *fromAlgCase* converts a function for use in case distinction to by ignoring previously computed results, and *fromAlg* converts a function for use in iteration by ignoring predecessors.

6.3.2. *Generic Parigot encoding.* We now detail the generic derivation of inductive Parigot-encoded data. The developments of this section are parametrized by a functor F, with *fmap* giving the lifting of functions and *fmapId* and *fmapCompose* the proofs that this lifting respects identity and composition. The construction is separated into several phases: in Figure 52 we give the computational characterization of canonical Parigot encoding as a predicate on untyped terms satisfying the reflection law, then prove that the untyped constructor preserves this property; in Figure 53, we define the type of Parigot encodings, its primitive recursion combinator, and its constructors; in Figure 54 we define the inductive subset of Parigot encodings and its constructor; finally, in Figure 55 we show that every Parigot encoding is already in the inductive subset and prove induction.

Reflection law. The definitions *recU*, *inU*, and *reflectU* of Figure 52 are untyped versions of resp. the combinator for primitive recursion, the generic constructor, and the operation that builds canonical Parigot encodings by recursively rebuilding the encoding with the constructor *inU* (compare to Figure 39 of Section 6.1). Predicate DC gives the characterization of canonical Parigot encodings that *reflectU* behaves extensionally like the identity function for them.

Even without having a type for Parigot-encoded data, we can still effectively reason about the behaviors of these untyped programs. This is shown in the proof of *inC*, which states *inU xs* satisfies the predicate DC if *xs* is an F-collection of untyped terms that satisfy DC. In the body, the expected type is convertible with the type

$$\{\mathit{inU}\ (\mathit{fmap}\ \mathit{proj2}\ (\mathit{fmap}\ (\mathit{fork}\ \mathit{id}\ \mathit{reflectU})\ \mathit{xs})) \simeq \mathit{inU}\ \mathit{xs}\}$$

```
import functor .
import utils .

import cast .
import mono .
import recType .

module parigot/generic/encoding
  (F: ⋆ → ⋆) (fmap: Fmap ·F)
  {fmapId: FmapId ·F fmap} {fmapCompose: FmapCompose ·F fmap } .

import functorThms ·F fmap -fmapId -fmapCompose .

recU ◁ Top
= β{ λ a. λ x. x a } .

inU ◁ Top
= β{ λ xs. λ a. a (fmap (fork id (recU a)) xs) } .

reflectU ◁ Top
= β{ recU (λ xs. inU (fmap proj2 xs)) } .

DC ◁ Top → ⋆ = λ x: Top. { reflectU x ≃ x } .

inC ◁ Π xs: F ·(ι x: Top. DC x). DC β{ inU xs }
= λ xs.
  ρ (fmapCompose ·(ι x: Top. DC x) ·(Pair ·Top ·Top) ·Top
       (λ x. proj2 x) (fork (λ x. x.1) (λ x. β{ reflectU x })) xs)
    @x.{ inU x ≃ inU xs }
- ρ (fmapId ·(ι x: Top. DC x) ·Top (λ x. β{| reflectU x.1 |}) (λ x. x.2) xs)
    @x.{ inU x ≃ inU xs }
- β{ inU xs } .
```

Fig. 52. Generic Parigot encoding (part 1) (`parigot/generic/encoding.ced`)

We rewrite by the functor composition law to fuse the mapping of *proj2* with that of *fork id reflectU*, and now the right-hand side of the equation is convertible (by the computation law for *proj2*) with

$$inU\ (fmap\ reflectU\ xs)$$

Here we can rewrite by the functor identity law, using the assumption that on the predecessors contained in *xs*, *reflectU* behaves as the identity function. Note that we use

```
import data-char/primrec-typing ·F .

DF' ◁ ⋆ ➔ ⋆ = λ D: ⋆. ∀ X: ⋆. AlgRec ·D ·X ➔ X .
DF  ◁ ⋆ ➔ ⋆ = λ D: ⋆. ι x: DF' ·D. DC β{ x } .

monoDF ◁ Mono ·DF = <..>

D ◁ ⋆ = Rec ·DF .
rollD   ◁ DF ·D ➔ D = roll -monoDF .
unrollD ◁ D ➔ DF ·D = unroll -monoDF .

recD ◁ PrimRec ·D
= Λ X. λ a. λ x. (unrollD x).1 a .

inD' ◁ F ·D ➔ DF' ·D
= λ xs. Λ X. λ a. a (fmap (fork (id ·D) (recD a)) xs) .

toDC ◁ Cast ·D ·(ι x: Top. DC x)
= intrCast -(λ x. [ β{ x } , (unrollD x).2 ]) -(λ x. β) .

inD ◁ F ·D ➔ D
= λ xs. rollD [ inD' xs , inC (elimCast -(monoFunctor toDC) xs) ] .
```

Fig. 53. Generic Parigot encoding (part 2) (`parigot/generic/encoding.ced`)

the Kleene trick so that the proof *inC* is definitionally equal to the untyped constructor *inU*. This permits us to use dependent intersection to form the refinement needed to type only canonical Parigot encodings.

Parigot encoding D**.** In Figure 53, DF' is the type scheme whose fixpoint is the solution to D in Figure 49, and DF is the refinement of DF' to those terms satisfying the reflection law. Since DF is a monotone type scheme (*monoDF*, proof omitted), we may define D as its fixpoint with rolling and unrolling operations *rollD* and *unrollD*. Following this is *recD*, the typed combinator for primitive recursion.

The constructor *inD* for D is defined in two parts. First, we define *inD*$'$ to construct a value of type $DF' \cdot D$ from $xs : F \cdot D$, with the definition being that which we obtained in Figure 49 from the computation law of the primitive recursion scheme. Then, with the auxiliary proof *toDC* that D is included into the type of untyped terms satisfying DC, we

```
import data-char/primrec ·F fmap -fmapId -fmapCompose ·D inD .

IndD ◁ D ➔ ⋆ = λ x: D. ∀ P: D ➔ ⋆. PrfAlgRec ·P ➔ P x .

DI ◁ ⋆ = ι x: D. IndD x .

recDI ◁ ∀ P: D ➔ ⋆. PrfAlgRec ·P ➔ Π x: DI. P x.1
= Λ P. λ a. λ x. x.2 a .

fromDI ◁ Cast ·DI ·D
= intrCast -(λ x. x.1) -(λ x. β) .

inDI' ◁ F ·DI ➔ D
= λ xs. inD (elimCast -(monoFunctor fromDI) xs) .

indInDI' ◁ Π xs: F ·DI. IndD (inDI' xs)
= λ xs. Λ P. λ a.
  ρ ς (fmapId ·DI ·D (λ x. proj1 (mksigma x.1 (recDI a x))) (λ x. β) xs)
    @x.(P (inD x))
- ρ ς (fmapCompose (proj1 ·D ·P) (λ x: DI. mksigma x.1 (recDI a x)) xs)
    @x.(P (inD x))
- a (fmap ·DI ·(Sigma ·D ·P) (λ x. mksigma x.1 (recDI a x)) xs) .

inDI ◁ F ·DI ➔ DI
= λ xs. [ inDI' xs , indInDI' xs ] .
```

Fig. 54. Generic Parigot encoding (part 3) (`parigot/generic/encoding.ced`)

define the constructor *inD* for *D* using the rolling operation and dependent intersection introduction. The definition is accepted by virtue of the following definitional equalities:

$$|inD'| =_{\beta\eta} |inU| =_{\beta\eta} |inC|$$
$$|xs| =_{\beta\eta} |elimCast\text{ -}(monoFunctor\ toDC)\ xs|$$

Finally, we can use Cedille to confirm that the typed recursion combinator and constructor are definitionally equal to the corresponding untyped operations.

```
_ ◁ { recD ≃ recU } = β .
_ ◁ { inD  ≃ inU }  = β .
```

Inductive Parigot encoding ***DI*****.** In Figure 54 we give the definition of *DI*, the type of the inductive subset of Parigot-encoded data, and *inDI*, its constructor. This definition begins by bringing *PrfAlgRec* into scope with an import to define the predicate *IndD*. For arbitrary $x : D$, the property *IndD* x states that, for all properties $P : D \to \star$, to prove $P\ x$ it suffices to give an (F, D)-proof algebra for P. Then, we define the inductive subset *DI* of Parigot encodings that satisfy the predicate *IndD* using dependent intersections. Definition *recDI* is the induction principle for terms of type D in this subset, and corresponds to the definition *recNatI* for Parigot naturals in Figure 41. With *recDI*, we can obtain the desired induction scheme for D if we can show D is included into *DI*.

We begin the proof of this type inclusion by defining the constructor *inDI* for the inductive subset. This is broken into three parts. First, *inDI′* constructs a value of type D from an F-collection of *DI* predecessors using the inclusion of type *DI* into D (*fromDI*). Next, with *indInDI′* we prove that the values constructed from *inDI′* satisfy the inductivity predicate *IndD* using the functor identity and composition laws.

In the body of *indInDI′*, we use equational reasoning to bridge the gap between the expected type $P\ (\mathit{inDI'}\ xs)$ and the type of the expression in the final line, which is:

$$P\ (\mathit{inD}\ (\mathit{fmap}\ (\mathit{proj1} \cdot D \cdot P)\ (\mathit{fmap} \cdot \mathit{DI} \cdot (\mathit{Sigma} \cdot D \cdot P)(\lambda\, x.\, \mathit{mksigma}\ x.1\ (\mathit{recDI}\ a\ x)))))$$

Observe that we can use the functor identity law to rewrite the expression *inDI′ xs* so that we introduce a use of *fmap* on an function of type $\mathit{DI} \to D$ that is definitionally equal to the identity function

$$\lambda\, x.\, \mathit{proj1}\ (\mathit{mksigma}\ x.1\ (\mathit{recDI}\ a\ x))$$

Applying the lifting of this function to *xs* results in a term of type $F \cdot D$, so in the expected type we exchange *inDI′* for *inD* as these two terms are definitionally equal. After this, we use the composition law to introduce two uses of *fmap*, one each for *proj1* and $\lambda\, x.\, \mathit{mksigma}\ x.1\ (\mathit{recDI}\ a\ x)$. This makes the rewritten type convertible with the type of the expression given. As *inDI′* and *indInDI′* are definitionally equal to each

```
reflectDI ◁ D → DI
= recD (λ xs. inDI (fmap ·(Pair ·D ·DI) ·DI (λ x. proj2 x) xs)) .

toDI ◁ Cast ·D ·DI
= intrCast -reflectDI -(λ x. (unrollD x).2) .

indD ◁ ∀ P: D → ⋆. PrfAlgRec ·D inD ·P → Π x: D. P x
= Λ P. λ a. λ x. recDI a (elimCast -toDI x) .
```

Fig. 55. Generic Parigot encoding (part 4) (`parigot/generic/encoding/ced`)

other (since $|\mathit{fork}\ \mathit{id}\ (\mathit{recD}\ a)| =_{\beta\eta} |\lambda\, x.\, \mathit{mksigma}\ x.1\ (\mathit{recDI}\ a\ x)|$), we can define the constructor *inDI* using the rolling operation and dependent intersection introduction.

Reflection and induction. We can now show an inclusion of the type D into the type DI by using the fact that terms of type D are canonical Parigot encodings, giving us induction. This is shown in Figure 55. First, we define the operation *reflectDI* which recursively rebuilds Parigot-encoded data with the constructor *inDI* for the inductive subset, producing a value of type DI. Then, since $|\mathit{reflectDI}| =_{\beta\eta} |\mathit{reflectU}|$, we can use *reflectDI* to witness the inclusion of D into DI, since every term x of type D is itself a proof that *reflectU* behaves extensionally as the identity function on x. From here, the proof *indD* of induction merely uses *recDI* in combination with this type inclusion.

6.3.3. *Computational and extensional character.* We now analyze the properties of our generic Parigot encoding. In particular, we wish to know the normalization guarantee for terms of type D and confirm that D admits an efficient and unique solution to the primitive recursion scheme, which can in turn be used to simulate case distinction and iteration. We omit the uniqueness proofs, which make heavy use of the functor laws and rewriting (see the code repository for this paper).

Normalization guarantee. In Figure 56, *normD* establishes the inclusion of type D into a function type $\mathit{AlgRec} \cdot D \cdot D \to D$. By Proposition 3, this guarantees call-by-name normalization of closed terms of type D.

```
import functor .
import cast .
import recType .
import utils .

module parigot/generic/props
  (F: ⋆ → ⋆) (fmap: Fmap ·F)
  {fmapId: FmapId ·F fmap} {fmapCompose: FmapCompose ·F fmap} .

import functorThms ·F fmap -fmapId -fmapCompose .
import parigot/generic/encoding ·F fmap -fmapId -fmapCompose .
import data-char/primrec-typing ·F .

normD ◁ Cast ·D ·(AlgRec ·D ·D → D)
= intrCast -(λ x. (unrollD x).1 ·D) -(λ x. β) .

import data-char/primrec ·F fmap -fmapId -fmapCompose ·D inD .

recDBeta ◁ PrimRecBeta recD
= Λ X. Λ a. Λ xs. β .

reflectD ◁ Π x: D. { recD (fromAlg inD) x ≃ x }
= λ x. (unrollD x).2 .

recDEta ◁ PrimRecEta recD = <..>
```

Fig. 56. Characterization of *recD* (`parigot/generic/props.ced`)

Primitive recursion scheme With proof *recDBeta*, we have that our solution *recD* (Figure 53) satisfies the computation law by definitional equality. By inspecting the definitions of *recD* and *inD*, and the erasures of *roll* and *unroll* (Figure 14), we can confirm that the computation law is satisfied in a constant number of steps under both call-by-name and call-by-value operational semantics.

Definition *recDEta* establishes that this solution is unique up function extensionality. The proof follows from induction, the functor laws, and a non-obvious use of the Kleene trick (Section 2.3). The reflection law is usually obtained as a consequence of uniqueness, but as the generic Parigot encoding has been defined with satisfaction of this law baked in, the proof *reflectD* proceeds by appealing to that fact directly.

```
import data-char/case-typing ·F .
import data-char/case ·F ·D inD .

caseD ◁ Case ·D
= Λ X. λ a. recD (fromAlgCase a) .

caseDBeta ◁ CaseBeta caseD
= Λ X. Λ a. Λ xs.
  ρ (fmapCompose ·D ·(Pair ·D ·X) ·D
       (λ x. proj1 x) (fork (id ·D) (caseD a)) xs)
    @x.{ a x ≃ a xs }
- ρ (fmapId ·D ·D (λ x. proj1 (fork (id ·D) (caseD a) x)) (λ x. β) xs)
    @x.{ a x ≃ a xs }
- β.

caseDEta ◁ CaseEta caseD = <..>
```

Fig. 57. Characterization of *caseD* (`parigot/generic/props.ced`)

Case-distinction scheme. In Figure 57, we define the candidate *caseD* for the operation giving case-distinction for D using *recD* and *fromAlgCase* (Figure 51). This definition satisfies the computation law only up to the functor laws. With definitional equality alone, $\mathit{caseD}\ a\ (\mathit{inD}\ \mathit{xs})$ reduces to

$$a\ (\mathit{fmap}\ \mathit{proj1}\ (\mathit{fmap}\ (\mathit{fork}\ \mathit{id}\ (\mathit{caseD}\ a))\ \mathit{xs}))$$

meaning we have introduced another traversal over the signature with *fmap*. As we have seen before, under call-by-value semantics this would also cause $\mathit{caseD}\ a$ to be needlessly computed for all predecessors. The proof of extensionality, *caseDEta*, follows from *recDEta*.

Destructor. In Figure 58 we define the destructor *outD* using case distinction. As such, the destructor inherits the caveat that the computation law, *lambek1D*, only holds up to the functor laws and is not efficient under call-by-name semantics. The extensionality law, *lambek2D*, holds by induction.

```
import data-char/destruct ·F ·D inD .

outD ◁ Destructor
= caseD (λ xs. xs) .

lambek1D ◁ Lambek1 outD
= λ xs. ρ (caseDBeta ·(F ·D) -(λ x. x) -xs) @x.{ x ≃ xs } - β .

lambek2D ◁ Lambek2 outD = <..>
```

Fig. 58. Characterization of destructor *outD* (`parigot/generic/props.ced`)

```
import data-char/iter-typing ·F .
import data-char/iter ·F fmap ·D inD .

foldD ◁ Iter ·D
= Λ X. λ a. recD (fromAlg a) .

foldDBeta ◁ IterBeta foldD
= Λ X. Λ a. Λ xs.
  ρ (fmapCompose ·D ·(Pair ·D ·X) ·X
       (λ x. proj2 x) (fork (id ·D) (foldD a)) xs)
    @x.{ a x ≃ a (fmap (foldD a) xs) }
- β .

foldDEta ◁ IterEta foldD
= <..>
```

Fig. 59. Characterization of *foldD* (`parigot/generic/props.ced`)

Iteration. The last property we confirm is that we may use *recD* to give a unique solution for the typing and computation laws of the iteration scheme. The proposed solution is *foldD*, given in Figure 59. The computation law, proven with *foldDBeta*, only holds by the functor laws as there are two traversals of the signature with *fmap* instead of one. Aside from this, this solution for *foldD* introduces constant-time overhead with every recursive call, since we form a tuple from the predecessor and previously computed result only to later discard the predecessor (see Figure 46 for the computation laws for dependent pairs).

6.3.4. *Example: Rose trees.* We conclude the discussion of Parigot-encoded data by using the generic derivation to define rose trees with induction. Rose trees are a datatype in which subtrees are contained within a list, meaning that nodes may have an arbitrary number of children. In Haskell, this datatype can be defined as:

```
data RoseTree a = Rose a [RoseTree a]
```

There are two motivations for this choice of example. First, while it is true that rose trees can be put into a form that reveals it is a strictly positive datatype by using containers (Abbott et al., 2003) and nested inductive definitions, we use impredicative encodings for datatypes and so the rose tree datatype we define is not syntactically strictly positive. Second, the expected induction principle for rose trees is known for being tricky to synthesize automatically: additional plugins (Ullrich, 2020) are required for Coq, and in the Agda standard library (The Agda Team, 2021) the induction principle is obtained by declaring rose trees as a size-indexed type (Abel, 2010).

The difficulty lies in giving users access to the inductive hypothesis for the sub-trees contained within the list. One work-around to address this that is available in both Coq and Agda is to forego reusing the list datatype and define rose trees as a mutually inductive type, thereby leveraging a mutual induction principle for the auxiliary "list of rose trees" datatype. However, this means that functionality defined for lists has to be re-implemented for this special-purpose type. In this section, we show that we can both reuse the list datatype in the definition of rose trees *and* derive a suitable induction principle for them in the style expected of a mutual inductive definition.

Lists. Figure 60 shows the definition of the datatype *List*, and the types of the list constructors (*nil* and *cons*) and induction principle (*indList*). These are defined using the generic derivation of inductive Parigot encodings. This derivation is brought into scope as "P", and the definitions within that module are accessed with the prefix "$P.$", e.g., "$P.D$" for the datatype. This is not to be confused with the predicate P (occurring

```
import utils.

module parigot/examples/list-data (A : ⋆).

import signatures/list ·A .

import parigot/generic/encoding as P
  ·ListF listFmap -listFmapId -listFmapCompose .

List ◁ ⋆ = P.D .

nil  ◁ List = <..>
cons ◁ A → List → List = <..>

indList
◁ ∀ P: List → ⋆. P nil → (Π hd: A. Π tl: List. P tl → P (cons hd tl)) →
  Π xs: List. P xs
= <..>

recList ◁ ∀ X: ⋆. X → (A → List → X → X) → List → X
= Λ X. indList ·(λ x: List. X) .
```

Fig. 60. Lists (`parigot/examples/list-data.ced`)

with no dot) which is quantified over in the type of *indList*. Note also that code in the figure refers to *List* as a datatype, not $List \cdot A$, since A is a parameter to the module.

For the sake of brevity, we omit the definitions of the list signature ($ListF$), its mapping operation (*listFmap*), and the proofs this mapping satisfies the functor laws (*listFmapId* and *listFmapCompose*). These appear in the figure as module arguments to the generic derivation. We also define the primitive recursion principle *recList* for lists as a non-dependent use of induction.

In Figure 61, we change module contexts (so we may consider lists with different element types, e.g. $List \cdot B$) to define the list mapping operation *listMap* by recursion. We also prove with *listMapId* and *listMapCompose* that this mapping operation obeys the functor laws.

Signature *TreeF***.** Figure 62 gives the signature *TreeF* for a datatype of trees whose branching factor is given by a functor F, generalizing the rose tree datatype. The proofs

```
import utils .
import functor .

module parigot/examples/list .

import parigot/examples/list-data .

listMap ◁ Fmap ·List
= Λ A. Λ B. λ f.
  recList ·A ·(List ·B) (nil ·B)
    (λ hd. λ tl. λ xs. cons (f hd) xs) .

listMapId ◁ FmapId ·List listMap = <..>
listMapCompose ◁ FmapCompose ·List listMap = <..>
```

Fig. 61. Map for lists (`parigot/examples/list.ced`)

```
import functor .
import utils .

module signatures/tree
  (A: ⋆) (F: ⋆ ➔ ⋆) (fmap: Fmap ·F)
  {fmapId: FmapId ·F fmap} {fmapCompose: FmapCompose ·F fmap} .

TreeF ◁ ⋆ ➔ ⋆ = λ T: ⋆. Pair ·A ·(F ·T) .

treeFmap ◁ Fmap ·TreeF
= Λ X. Λ Y. λ f. λ t. mksigma (proj1 t) (fmap f (proj2 t)) .

treeFmapId ◁ FmapId ·TreeF treeFmap = <..>
treeFmapCompose ◁ FmapCompose ·TreeF treeFmap = <..>
```

Fig. 62. Signature for F-branching trees (`signatures/tree.ced`)

that the lifting operation *treeFmap* respects identity and composition make use of the corresponding proofs for the given F.

Rose trees. In Figure 63, we instantiate the module parameters for the signature of F-branching trees with *List* and *listMap*, then instantiate the generic derivation of inductive Parigot encodings with *TreeF* and *treeFmap*. We define the standard constructor *rose* for rose trees using the generic constructor $P.inD$, and give a variant constructor $rose'$ which we use in the definition of the induction principle for rose trees. This variant uses

```
import utils .
import list-data .
import list .

module parigot/examples/rosetree-data (A: ⋆) .

import signatures/tree ·A ·List listMap -listMapId -listMapCompose .

import parigot/generic/encoding as P
  ·TreeF treeFmap -treeFmapId -treeFmapCompose .

RoseTree ◁ ⋆ = P.D .

rose ◁ A ➔ List ·RoseTree ➔ RoseTree
= λ x. λ t. P.inD (mksigma x t) .

rose' ◁ ∀ P: RoseTree ➔ ⋆. TreeF ·(Sigma ·RoseTree ·P) ➔ RoseTree
= Λ P. λ xs. P.inD (treeFmap (proj1 ·RoseTree ·P) xs) .

indRoseTree
◁ ∀ P: RoseTree ➔ ⋆. ∀ Q: List ·RoseTree ➔ ⋆.
  Q (nil ·RoseTree) ➔
  (Π t: RoseTree. P t ➔ Π ts: List ·RoseTree. Q ts ➔ Q (cons t ts)) ➔
  (Π x: A. Π ts: List ·RoseTree. Q ts ➔ P (rose x ts)) ➔
  Π t: RoseTree. P t
= Λ P. Λ Q. λ n. λ c. λ r.
  P.indD ·P (λ xs.
     indsigma xs ·(λ x: TreeF ·(Sigma ·RoseTree ·P). P (rose' x))
       (λ x. λ ts.
          [conv ◁ List ·(Sigma ·RoseTree ·P) ➔ List ·RoseTree
           = listMap (proj1 ·RoseTree ·P)]
        - [pf ◁ Q (conv ts)
           = indList ·(Sigma ·RoseTree ·P)
               ·(λ x: List ·(Sigma ·RoseTree ·P). Q (conv x))
               n (λ hd. λ tl. λ ih. c (proj1 hd) (proj2 hd) (conv tl) ih) ts]
        - r x (conv ts) pf)) .
```

Fig. 63. Rose trees (`parigot/examples/rosetree-data.ced`)

treeFmap to remove the tupled proofs that an inductive hypothesis holds for sub-trees that are introduced in the generic induction principle.

Finally, we give the induction principle for rose trees as *indRoseTree* in the figure. This is a mutual induction principle, with P the property one desires to show holds for all rose trees and Q the invariant that is maintained for collections of sub-trees. We require proofs that:

— P holds for *rose* x ts when Q holds for ts, bound as r;

— Q holds for *nil*, bound as n; and that

— if P holds for t and Q holds for ts then Q holds for *cons* t ts, bound as c.

In the body of *indRoseTree*, we use the induction principle $P.inD$ for the generic Parigot encoding ($P.indD$), then the induction principle *indsigma* (Figure 46) for pairs, revealing $x : RoseTree$ and $ts : List \cdot (Sigma \cdot RoseTree \cdot P)$. With auxiliary function *conv* to convert ts to a list of rose trees, we use list induction on ts to prove Q holds for *conv* ts. With this proved as pf, we can conclude by using r.

7. Lepigre-Raffalli encoding

We now revisit the issue of programming with Scott-encoded data. Neither the case-distinction scheme nor the weak induction principle we derived in Section 5 provide an obvious mechanism for recursion. In contrast, the Parigot encoding readily admits the primitive recursion scheme, as it can be viewed as a solution to that scheme. So despite its significant overhead in space representation, the Parigot encoding appears to have a clear advantage over the Scott encoding in total typed lambda calculi.

Amazingly, in some settings this deficit of the Scott encoding is *only* apparent. Working in a logical framework, Parigot (1988) showed how to derive with "metareasoning" a strongly normalizing recursor for Scott naturals. More recently, Lepigre and Raffalli (2019) demonstrated a well-typed recursor for Scott naturals in a Curry-style theory featuring a sophisticated form of subtyping utilizing "circular but well-founded" derivations. The Lepigre-Raffalli construction involves a novel impredicative encoding of datatypes, which we shall call the *Lepigre-Raffalli encoding*, that both supports recursion *and* is a supertype of the Scott encoding. In Cedille, we can similarly show a type inclusion of

Scott encodings into the type of Lepigre-Raffalli encodings using weak induction and the fact that our derived recursive types are least fixpoints.

We elucidate the construction of the Lepigre-Raffalli encoding by showing its relationship to the case-distinction scheme. The computation laws for case distinction over natural numbers (Figure 21) do not form a recursive system of equations. However, having obtained solutions for Scott naturals and *caseNat*, we can *introduce* recursion into the computation laws by observing that for all n, $|n| =_{\beta\eta} |\lambda\, z.\, \lambda\, s.\, \mathit{caseNat}\ z\ s\ n|$.

$$\begin{aligned} |\mathit{caseNat}\ t_1\ t_2\ \mathit{zero}| &=_{\beta\eta} t_1 \\ |\mathit{caseNat}\ t_1\ t_2\ (\mathit{suc}\ n)| &=_{\beta\eta} |t_2\ (\lambda\, z.\, \lambda\, s.\, \mathit{caseNat}\ z\ s\ n)| \end{aligned}$$

Viewing the computation laws this way, we see that t_2 is given a function which will make a recursive call on n when provided a suitable base and step case. We desire that these be the same base and step cases originally provided, i.e., that these in fact be t_1 and t_2 again. To better emphasize this new interpretation, we rename *caseNat* to *recLRNat*. By congruence of $\beta\eta$-equivalence, the two equations above give us:

$$\begin{aligned} |\mathit{recLRNat}\ t_1\ t_2\ \mathit{zero}\ t_1\ t_2| &=_{\beta\eta} |t_1\ t_1\ t_2| \\ |\mathit{recLRNat}\ t_1\ t_2\ (\mathit{suc}\ n)\ t_1\ t_2| &=_{\beta\eta} |t_2\ (\lambda\, z.\, \lambda\, s.\, \mathit{recLRNat}\ z\ s\ n)\ t_1\ t_2| \end{aligned}$$

To give types to the terms involved in these equations, we begin with the observation that we can use impredicative quantification to address the self-application occurring in the right-hand. Below, let the type T of the result we are computing be such that type variables Z and S are fresh with respect to its free variables, and let ? be a placeholder for a type.

$$\begin{aligned} t_1 &: \forall\, Z{:}\star.\, \forall\, S{:}\star.\, Z \to S \to T \\ t_2 &: \forall\, Z.\, \star\, \forall\, S{:}\star.\, ? \to Z \to S \to T \end{aligned}$$

This gives an interpretation of t_1 as being a constant function that ignores its two

$$\begin{array}{lcl} NatZ \cdot T & = & \forall\, Z{:}\star.\, \forall\, S{:}\star.\, Z \to S \to T \\ NatS \cdot T & = & \forall\, Z{:}\star.\, \forall\, S{:}\star.\, (Z \to S \to Z \to S \to T) \to Z \to S \to T \end{array}$$

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash t_1 \overset{\leftarrow}{\in} NatZ \cdot T \quad \Gamma \vdash t_2 \overset{\leftarrow}{\in} NatS \cdot T}{\Gamma \vdash recLRNat \cdot T\ t_1\ t_2 \overset{\rightarrow}{\in} Nat \to NatZ \cdot T \to NatS \cdot T \to T}$$

Fig. 64. Typing law for the Lepigre-Raffalli recursion scheme on *Nat*

arguments and returns a result of type T. For t_2, we must give a type for its first argument $\lambda\, z.\, \lambda\, s.\, recLRNat\ z\ s\ n$ that matches our intended reading that t_2 will instantiate the arguments z and s with t_1 and t_2. Now, t_2 is provided copies of t_1 and t_2 at the universally quantified types Z and S, so we make a further refinement to its type.

$$\forall\, Z{:}\star.\, \forall\, S{:}\star.\, (Z \to S \to\, ?) \to Z \to S \to T$$

We can complete the type of t_2 by observing that in the system of recursive equations for the computation law, *recLRNat* is a function of five arguments. Using η-expansion, we can rewrite the equation for the successor case to match this usage:

$$|recLRNat\ t_1\ t_2\ (suc\ n)\ t_1\ t_2| =_{\beta\eta} |t_2\ (\lambda\, z.\, \lambda\, s.\, \lambda\, z'.\, \lambda\, s'.\, recLRNat\ z\ s\ n\ z'\ s')\ t_1\ t_2|$$

where we understand that, from the perspective of t_2, instantiations of z and z' should have the universally quantified type Z and that instantiations of s and s' should have universally quantified type S. We thus obtain the complete definition of the type of t_2.

$$\forall\, Z{:}\star.\, \forall\, S{:}\star.\, (Z \to S \to Z \to S \to T) \to Z \to S \to T$$

Now we are able to construct a typing rule for our recursive combinator, shown in Figure 64. From this, we can obtain the following solution for the type of Lepigre-Raffalli

```
import cast.
import mono.
import recType.

import scott/concrete/nat as S .

module lepigre-raffalli/concrete/nat1 .

NatRec ◁ ⋆ ➔ ⋆ ➔ ⋆ ➔ ⋆
= λ X: ⋆. λ Z: ⋆. λ S: ⋆. Z ➔ S ➔ Z ➔ S ➔ X .

NatZ ◁ ⋆ ➔ ⋆
= λ X: ⋆. ∀ Z: ⋆. ∀ S: ⋆. Z ➔ S ➔ X .

NatS ◁ ⋆ ➔ ⋆
= λ X: ⋆. ∀ Z: ⋆. ∀ S: ⋆. NatRec ·X ·Z ·S ➔ Z ➔ S ➔ X .

Nat ◁ ⋆ = ∀ X: ⋆. NatRec ·X ·(NatZ ·X) ·(NatS ·X) .

recLRNat ◁ ∀ X: ⋆. NatZ ·X ➔ NatS ·X ➔ Nat ➔ NatZ ·X ➔ NatS ·X ➔ X
= Λ X. λ z. λ s. λ n. n z s .
```

Fig. 65. Primitive recursion for Scott naturals (part 1)
(`lepigre-raffalli/concrete/nat1.ced`)

naturals:

$$Nat = \forall\, X\!:\!\star.\, NatZ \cdot X \to NatS \cdot X \to NatZ \cdot X \to NatS \cdot X \to T$$

The remainder of this section is structured as follows. In Section 7.1, we show that the type of Lepigre-Raffalli encodings is a supertype of type of Scott encodings and derive the primitive recursion scheme for Scott naturals from the Lepigre-Raffalli recursion scheme we have just discussed. In Section 7.2, we modify the Lepigre-Raffalli encoding and derive induction for Scott naturals. Finally, in Section 7.3 we generalize this modification to the generic Lepigre-Raffalli encoding and derive induction for generic Scott encodings.

7.1. *Primitive recursion for Scott naturals, concretely*

Lepigre-Raffalli naturals. In Figure 65, we give in Cedille the definition for the type of Lepigre-Raffalli encodings we previously obtained. We use a qualified import of the

```
zero ◁ Nat
= Λ X. λ z. λ s. z ·(NatZ ·X) ·(NatS ·X) .

suc ◁ Nat → Nat
= λ n. Λ X. λ z. λ s.
  s ·(NatZ ·X) ·(NatS ·X) (λ z'. λ s'. recLRNat z' s' n) .

rollNat ◁ Cast ·(S.NatFI ·Nat) ·Nat
= intrCast
    -(λ n. n.1 zero suc)
    -(λ n. n.2 ·(λ x: S.NatF ·Nat. { x zero suc ≃ x }) β (λ m. β)) .

toNat ◁ Cast ·S.Nat ·Nat
= recLB -rollNat .
```

Fig. 66. Primitive recursion for Scott naturals (part 2)
(`lepigre-raffalli/concrete/nat1.ced`)

concrete encoding of Scott naturals (Section 5.1), so to access a definition from that development we use "S." as a prefix. The common shape $Z \to S \to Z \to S \to X$ has been refactored into the type family *NatRec*, used in the definitions of *NatS* and *Nat*. We also give the definition for the recursive combinator *recLRNat*, which we observe is definitionally equal to $S.caseNat$ (Figure 26):

```
_ ◁ { recLRNat ≃ S.caseNat } = β .
```

where "_" indicates an anonymous proof.

Inclusion of Scott naturals into Lepigre-Raffalli naturals. Figure 66 shows that Scott naturals ($S.Nat$) are a subtype of Lepigre-Raffalli naturals. This begins with the constructors *zero* and *suc*, whose definitions come from computation laws for *recLRNat*. In particular, for successor the first argument to the bound s is $\lambda z'. \lambda s'. \mathit{recLRNat}\ z'\ s'\ n$, the handle for making recursive calls on the predecessor n that awaits a suitable base and step case. Because the computation laws for *recLRNat* are derived from case distinction, we have that *zero* and *suc* are definitionally equal to $S.zero$ and $S.suc$.

```
_ ◁ { zero ≃ S.zero } = β .
_ ◁ { suc  ≃ S.suc  } = β .
```

Recall that in Section 5.1.1, we saw that the function which rebuilds Scott naturals with its constructors behaves extensionally as the identity function. We can leverage this fact to define *rollNat*, which establishes an inclusion of $S.NatFI \cdot Nat$ into Nat by rebuilding a term of the first type with the constructors *zero* and *suc* for Lepigre-Raffalli naturals. The proof is given not by *wkIndNat*, but the even weaker pseudo-induction principle $S.WkIndNatF \cdot Nat\ n.1$,

$$\forall\, P{:}S.NatF \cdot Nat \to \star.\, P\ (S.zeroF \cdot Nat) \to (\Pi\, m{:}Nat.\, P\ (S.sucF\ m)) \to P\ n$$

given by $n.2$. We saw in Section 5.1 that $|S.zeroF| =_{\beta\eta} |S.zero|$ and $|S.sucF| =_{\beta\eta} |S.suc|$, so it follows that $|S.zeroF| =_{\beta\eta} |zero|$ and $|S.sucF| =_{\beta\eta} |suc|$.

With *rollNat*, we have that Nat is an $S.NatFI$-closed type. Since $S.Nat = Rec{\cdot}S.NatFI$ is the least such type with respect to type inclusion, using *recLB* (Figure 12) we have a cast from Scott naturals to Lepigre-Raffalli naturals.

Primitive recursion for Scott naturals. The last step in equipping Scott naturals with a primitive recursion scheme is to translate it to the Lepigre-Raffalli recursion scheme. This is done in three parts, shown in Figure 67. One complication that must be addressed is that the Lepigre-Raffalli recursion reinterprets the predecessor as a function for making recursive calls, but the primitive recursion scheme enables direct access to the predecessor. For now, we will duplicate the predecessor in order for it to serve in both roles. This means that if X is the type of results we wish to compute with primitive recursion, then we use Lepigre-Raffalli recursion to compute a function of type $S.Nat \to X$. In Section 7.2 we show how to avoid duplication by using intersection types.

If $x : X$ is the base case for primitive recursion, then *recNatZ* t is a constant polymorphic function that ignores its first three arguments and returns t_1. For the step case $f : S.Nat \to X \to X$, *recNatS* f produces a step case for Lepigre-Raffalli recursion, introducing:

— type variables Z and S,

— $r : \mathit{NatRec} \cdot (S.\mathit{Nat} \rightarrow X) \cdot Z \cdot S$, the handle for making recursive calls,

— z and s, the base and step cases at the abstracted types Z and S, and

— $m : S.Nat$, which we intend to be a duplicate of r.

1295 In the body of *recNatS*, f is given access to the predecessor m and the result recursively computed with r, where we decrement m as we pass through the recursive call. Finally, *recNat* gives us the primitive recursion scheme for Scott naturals by translating the base and step cases to the Lepigre-Raffalli style (and duplicating them), coercing the given Scott natural n to a Lepigre-Raffalli natural, and giving also the predecessor of n.

1300 With *recNatBeta1* and *recNatBeta2*, we use Cedille to confirm that the expected computation laws for the primitive recursion scheme hold by definition. To give a more complete understanding of how *recNat* computes, we show some intermediate steps involved

```
recNatZ ◁ ∀ X: ⋆. X → NatZ ·(S.Nat → X)
= Λ X. λ x. Λ Z. Λ S. λ z. λ s. λ m. x .

recNatS ◁ ∀ X: ⋆. (S.Nat → X → X) → NatS ·(S.Nat → X)
= Λ X. λ f. Λ Z. Λ S. λ r. λ z. λ s. λ m.
  f m (r z s z s (S.pred m)) .

recNat ◁ ∀ X: ⋆. X → (S.Nat → X → X) → S.Nat → X
= Λ X. λ x. λ f. λ n.
  recLRNat ·(S.Nat → X) (recNatZ x) (recNatS f)
    (elimCast -toNat n)
    (recNatZ x) (recNatS f) (S.pred n) .

recNatBeta1
◁ ∀ X: ⋆. ∀ x: X. ∀ f: S.Nat → X → X. { recNat x f S.zero ≃ x }
= Λ X. Λ x. Λ f. β .

recNatBeta2
◁ ∀ X: ⋆. ∀ x: X. ∀ f: S.Nat → X → X. ∀ n: S.Nat.
  { recNat x f (S.suc n) ≃ f n (recNat x f n) }
= Λ X. Λ x. Λ f. Λ n. β .
```

Fig. 67. Primitive recursion for Scott naturals (part 3)
(`lepigre-raffalli/concrete/nat1.ced`)

$$\begin{array}{ll}
 & \mathit{recNat}\ t_1\ t_2\ (S.\mathit{suc}\ n) \\[1ex]
\rightsquigarrow^*_\beta & \mathit{recLRNat}\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{elimCast}\ (S.\mathit{suc}\ n)) \\
 & \quad (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (S.\mathit{suc}\ n) \\[1ex]
\rightsquigarrow^*_\beta & S.\mathit{suc}\ n\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (S.\mathit{suc}\ n) \\[1ex]
\rightsquigarrow^*_\beta & \mathit{recNatS}\ t_2\ n\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (S.\mathit{suc}\ n) \\[1ex]
\rightsquigarrow^*_\beta & t_2\ (S.\mathit{pred}\ (S.\mathit{suc}\ n)) \\
 & \quad (n\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (S.\mathit{pred}\ (S.\mathit{suc}\ n))) \\[1ex]
\rightsquigarrow^*_\beta & t_2\ n\ (n\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ n) \\[1ex]
\leftsquigarrow^*_\beta & t_2\ n\ (\mathit{recLRNat}\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{elimCast}\ n)\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ n) \\[1ex]
\leftsquigarrow^*_\beta & t_2\ n\ (\mathit{recNat}\ t_1\ t_2\ n)
\end{array}$$

Fig. 68. Reduction of *recNat* for the successor case

for the step case for arbitrary untyped terms t_1, t_2, and n in Figure 68. In the figure, we omit types and erased arguments, and indeed it should be read as ordinary (full) β-reduction for untyped terms. In the last two lines of the figure, we switch the direction of reduction. Altogether, this shows that $|\mathit{recNat}\ t_1\ t_2\ (S.\mathit{suc}\ n)|$ and $|t_2\ n\ (\mathit{recNat}\ t_1\ t_2\ n)|$ are joinable in a constant number of reduction steps.

7.2. *Induction for Scott naturals, concretely*

In this section, we describe some modifications to the Lepigre-Raffalli encoding that allow us to equip Scott naturals with induction. For completeness, we show the full derivation, but as this development is very similar to what preceded we shall only highlight the differences. A byproduct of this modification is that we will no longer need to duplicate the predecessor: with dependent intersections, we can maintain the two different views of the predecessor at once.

In Figure 69, we begin our modification by making *NatRec* dependent: $\mathit{NatRec}{\cdot}P\ n\ {\cdot}Z{\cdot}T$

```
import cast.
import mono.
import recType.

import scott/concrete/nat as S .

module lepigre-raffalli/concrete/nat2 .

NatRec ◁ (S.Nat → ⋆) → S.Nat → ⋆ → ⋆ → ⋆
= λ P: S.Nat → ⋆. λ x: S.Nat. λ Z: ⋆. λ S: ⋆.
  Z → S → Z → S → P x.

NatZ ◁ (S.Nat → ⋆) → ⋆
= λ P: S.Nat → ⋆. ∀ Z: ⋆. ∀ S: ⋆. Z → S → P S.zero .

NatS ◁ (S.Nat → ⋆) → ⋆
= λ P: S.Nat → ⋆.
  ∀ Z: ⋆. ∀ S: ⋆. Π n: (ι x: S.Nat. NatRec ·P x ·Z ·S).
  Z → S → P (S.suc n.1) .

Nat ◁ ⋆
= ι x: S.Nat. ∀ P: S.Nat → ⋆. NatRec ·P x ·(NatZ ·P) ·(NatS ·P) .

recLRNat ◁ ∀ P: S.Nat → ⋆. NatZ ·P → NatS ·P → Π n: Nat. NatZ ·P → NatS ·P → P n.1
= Λ X. λ z. λ s. λ n. n.2 z s .

zero ◁ Nat
= [ S.zero , Λ X. λ z. λ s. z ·(NatZ ·X) ·(NatS ·X) ] .

suc ◁ Nat → Nat
= λ n.
  [ S.suc n.1
  , Λ P. λ z. λ s.
    s ·(NatZ ·P) ·(NatS ·P) [ n.1 , λ z. λ s. recLRNat z s n ] ] .

rollNat ◁ Cast ·(S.NatFI ·Nat) ·Nat
= intrCast
    -(λ n. n.1 zero suc)
    -(λ n. n.2 ·(λ x: S.NatF ·Nat. { x zero suc ≃ x }) β (λ m. β)) .

toNat ◁ Cast ·S.Nat ·Nat = recLB -rollNat .
```

Fig. 69. Induction for Scott naturals (part 1)
(`lepigre-raffalli/concrete/nat2.ced`)

```
indNatZ ◁ ∀ P: S.Nat → ⋆. P S.zero → NatZ ·P
= Λ X. λ x. Λ Z. Λ S. λ z. λ s. x .

indNatS ◁ ∀ P: S.Nat → ⋆. (Π n: S.Nat. P n → P (S.suc n)) → NatS ·P
= Λ P. λ f. Λ Z. Λ S. λ r. λ z. λ s. f r.1 (r.2 z s z s) .

indNat
◁ ∀ P: S.Nat → ⋆. P S.zero → (Π m: S.Nat. P m → P (S.suc m)) →
  Π n: S.Nat. P n
= Λ P. λ x. λ f. λ n.
  recLRNat ·P (indNatZ x) (indNatS f) (elimCast -toNat n)
    (indNatZ x) (indNatS f) .
```

Fig. 70. Induction for Scott naturals (part 2)
(`lepigre-raffalli/concrete/nat2.ced`)

is the type of functions taking two arguments each of type Z and S and returning a proof that P holds for n. The next and most significant modification is to the definition *NatS*. We want that the handle n for invoking our inductive hypothesis will produce a proof that P holds for the predecessor — which is n itself! We can express the dual role of the predecessor as data (n) and function ($\lambda z. \lambda s. \mathit{recLRNat}\ z\ s\ n$) with dependent intersections, which recovers the view of the predecessor as a Scott natural without requiring duplication. This duality is echoed in the definition of *Nat*, which is defined with dependent intersections as the type of Scott naturals x which, for an arbitrary predicate P, will act as a function taking two base ($\mathit{NatZ} \cdot P$) and step ($\mathit{NatS} \cdot P$) cases each and produce a proof that P holds of x.

By its definition, the type *Nat* of the modified Lepigre-Raffalli encoding is a subtype of Scott encodings. Using the same approach as in Section 7.1, we can show a type inclusion in the other direction. We define the constructors *zero* and *suc*, noting that the innermost intersection introduction in *suc* again shows the dual role of the predecessor, then show that *Nat* is *S.NatFI* closed by rebuilding with these constructors.

Finally, we derive true induction for Scott naturals in Figure 70. Playing the same roles as *recNatZ* and *recNatS* (Figure 67), *indNatZ* and *indNatS* convert the usual base and step cases of inductive proof into forms suitable for Lepigre-Raffalli-style induction. In particular, note that in *indNatS* the bound r plays the role of both predecessor ($r.1$) and handle for the inductive hypothesis ($r.2$), avoiding duplication.

7.3. *Induction for Scott-encoded data, generically*

In this section, we formulate a *generic* variant of the Lepigre-Raffalli encoding and use this to derive induction for our generic Scott encoding. To explain generic encoding, we start by deriving a Lepigre-Raffalli recursion scheme from the observation that, for the solution *caseD* for the case-distinction scheme given in Figure 35, $|t| =_{\beta\eta} \lambda\, a.\, \mathit{caseD}\ a\ d$ for all t. This allows us to introduce recursion into the computation law for case distinction.

Let D be the type of Scott-encoded data whose signature is F, and let $t : \mathit{AlgCase} \cdot D \cdot T$ for some T and $t' : F \cdot D$. For the sake of exposition, we will for now assume that F is a functor. Using the functor identity law, we can form the following equation from the computation law of case distinction.

$$|\mathit{caseD}\ t\ (\mathit{inD}\ t')| \rightsquigarrow |t\ t'| =_{\mathit{FmapId}} |t\ (\mathit{fmap}\ (\lambda\, x.\, \lambda\, a.\, \mathit{caseD}\ a\ x)\ t')|$$

Renaming *caseD* to *recLRD*, we can read the above as a computational characterization of the generic Lepigre-Raffalli recursion scheme and use this to derive a suitable typing law. We desire that t should be a function which will be able to accept itself as a second argument in order to make recursive calls on predecessors.

$$|\mathit{recLRD}\ t\ (\mathit{inD}\ t')\ t| \rightsquigarrow |t\ t'\ t| =_{\mathit{FmapId}} |t\ (\mathit{fmap}\ (\lambda\, x.\, \lambda\, a.\, \mathit{recLRD}\ a\ x)\ t')\ t|$$

Under this interpretation, we are able to give the following type for terms t used in generic Lepigre-Raffalli recursion to compute a value of type T.

$$\forall\, Y{:}\star.\, F \cdot (Y \to Y \to T) \to Y \to T$$

$$\frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t \overleftarrow{\in} \forall\, Y : \star.\, F \cdot (Y \to Y \to T) \to Y \to T}{\Gamma \vdash \mathit{recLRD} \cdot T\ t \overrightarrow{\in} D \to (\forall\, Y : \star.\, F \cdot (Y \to Y \to T) \to Y \to T) \to T}$$

Fig. 71. Typing law for the generic Lepigre-Raffalli recursion scheme

Compare this to Lepigre-Raffalli recursion for naturals (Figure 64).

— Quantification over Y replaces quantification over Z and S for the base and step case of naturals. Here, we intend that Y will be impredicatively instantiated with the type $\forall\, Y : \star.\, F \cdot (Y \to Y \to T) \to Y \to T$ again.

— The single handle for making a recursive call on the natural number predecessor becomes an F-collection of handles of type $Y \to Y \to T$ for making recursive calls, obtained from the F-collection of D predecessors.

This leads to the typing law for *recLRD* listed in Figure 71. For the computation law, we desire it be precisely the same as that for *caseD* — meaning that we will not need to require F to be a functor for Lepigre-Raffalli recursion (or induction) over datatype D.

Unlike the other schemes we have considered, we are unaware of any standard criteria for characterizing Lepigre-Raffalli recursion, and the development of a categorical semantics for this scheme is beyond the scope of this paper. Instead, and under the assumption that F is a functor, we will show that from this scheme and the related induction principle we can give efficient and provably unique solutions to the iteration and primitive recursion schemes.

The derivations of this section are separated into three parts. In Figure 72, we give the type of our generic variant of the Lepigre-Raffalli encoding for a monotone signature. In Figure 73, we derive Lepigre-Raffalli induction for Scott encodings. Finally, in Figure 74 we assume the stronger condition that the datatype signature is a functor and derive a standard induction principle for Scott encodings.

```
import cast .
import mono .
import recType .

module scott-rec/generic/encoding
  (F: ⋆ ➔ ⋆) {mono: Mono ·F} .

import scott/generic/encoding as S ·F -mono .
import data-char/iter-typing ·F .
import data-char/primrec-typing ·F .

DRec ◁ (S.D ➔ ⋆) ➔ S.D ➔ ⋆ ➔ ⋆
= λ P: S.D ➔ ⋆. λ x: S.D. λ Y: ⋆. Y ➔ Y ➔ P x .

inDRec ◁ ∀ P: S.D ➔ ⋆. ∀ Y: ⋆. F ·(ι x: S.D. DRec ·P x ·Y) ➔ S.D
= Λ P. Λ Y. λ xs.
  [c ◁ Cast ·(ι x: S.D. DRec ·P x ·Y) ·S.D
   = intrCast -(λ x. x.1) -(λ x. β)]
- S.inD (elimCast -(mono c) xs) .

PrfAlgLR ◁ (S.D ➔ ⋆) ➔ ⋆
= λ P: S.D ➔ ⋆.
  ∀ Y: ⋆. Π xs: F ·(ι x: S.D. DRec ·P x ·Y). Y ➔ P (inDRec xs) .

D ◁ ⋆ = ι x: S.D. ∀ P: S.D ➔ ⋆. DRec ·P x ·(PrfAlgLR ·P) .

recLRD ◁ ∀ P: S.D ➔ ⋆. PrfAlgLR ·P ➔ Π x: D. PrfAlgLR ·P ➔ P x.1
= Λ P. λ a. λ x. x.2 a .
```

Fig. 72. Generic Lepigre-Raffalli-style induction for Scott encodings (part 1) (`lepigre-raffalli/generic/encoding.ced`)

Generic Lepigre-Raffalli encoding. In Figure 72, we begin by importing the generic Scott encoding, with the prefix "S." used to access definitions of that module. Type family *DRec* gives the shape of the types of handles for invoking an inductive hypothesis for a particular term x of type $S.D$ when giving a proof of some $P : S.D \to \star$ w-(compare this to *NatRec* in Figure 69).

Next, for all predicates P over Scott encodings $S.D$, *PrfAlgLR*$\cdot P$ is the type of Lepigre-Raffalli-style proof algebras for P, corresponding to *NatZ* $\cdot P$ and *NatS* $\cdot P$ together in Figure 69. A term of this type is polymorphic in a type Y (which we interpret as standing in for *PrfAlgLR* $\cdot P$ itself) and takes and F-collection *xs* of terms which, with the use

```
fromD ◁ Cast ·D ·S.D
= intrCast -(λ x. x.1) -(λ x. β) .

instDRec ◁ ∀ P: S.D → ⋆. Cast ·D ·(ι x: S.D. DRec ·P x ·(PrfAlgLR ·P))
= Λ P. intrCast -(λ x. [ x.1 , λ a. recLRD a x ]) -(λ x. β) .

inD ◁ F ·D → D
= λ xs.
  [ S.inD (elimCast -(mono fromD) xs)
  , Λ P. λ a.
    a ·(PrfAlgLR ·P) (elimCast -(mono (instDRec ·P)) xs) ].

rollD ◁ Cast ·(S.DFI ·D) ·D
= intrCast
    -(λ x. x.1 inD)
    -(λ x. x.2 ·(λ x: S.DF ·D. { x inD ≃ x }) (λ xs. β)) .

toD ◁ Cast ·S.D ·D
= recLB -rollD .

indLRD ◁ ∀ P: S.D → ⋆. PrfAlgLR ·P → Π x: S.D. PrfAlgLR ·P → P x
= Λ P. λ a. λ x. recLRD a (elimCast -toD x) .
```

Fig. 73. Generic Lepigre-Raffalli-style induction for Scott encodings (part 2) (`lepigre-raffalli/generic/encoding.ced`)

of dependent intersection types, are interpreted both as a predecessor and a handle for accessing the inductive hypothesis for that predecessor. To state that the result should be a proof that P holds for the value constructed from these predecessors, we need a variant constructor *inDRec* that first casts the predecessors to the type $S.D$ using monotonicity of F. Note that we have $|inDRec| =_{\beta\eta} |S.inD|$.

Type D is our generic Lepigre-Raffalli encoding, again defined with dependent intersection as the type for Scott encodings x also act as functions which, for all properties P, produce a proof that P holds of x when given two Lepigre-Raffalli-style proof algebras for P. Finally, *recLRD* is the Lepigre-Raffalli-style induction principle restricted to those Scott encodings which have type D. To obtain true Lepigre-Raffalli induction, it remains to show that *every* term of type $S.D$ has type D.

Lepigre-Raffalli induction. In Figure 73, we begin the process of demonstrating an inclusion of the type $S.D$ into D by defining the constructor *inD* for the generic Lepigre-Raffalli encoding. This definition crucially uses the auxiliary function *instDRec* to produce the two views of a given predecessor $x{:}D$ as subdata $(x.1)$ and as a handle for the inductive hypothesis associated to that predecessor $(\lambda\, a.\, recLRD\ a\ x)$; compare this to the definition of *suc* in Figure 69. As expected, we have that *inD* and $S.inD$ (and also $S.inDF$) are definitionally equal.

```
_ ◁ { inD ≃ S.inD } = β .
```

With the constructor defined, we show with *rollD* that D is an $S.DFI$-closed type ($S.DFI$ is defined in Figure 34) by giving a proof that rebuilding a term of type $S.DFI \cdot D$ with constructor *inD* reproduces the same term. As $S.D = Rec \cdot S.DFI$ is the least such type, we thus obtain a proof *toD* of an inclusion of the type $S.D$ into D using *recLB* (Figure 12). With this, we have Lepigre-Raffalli-style induction as *indLRD*.

Standard induction for Scott encodings. To derive the usual induction principle for Scott encodings using Lepigre-Raffalli induction, we change module contexts in Figure 74 and now assume that F is a functor (see Section 6.2 for the definitions of the functor laws). As we did for the derivation of induction for Scott naturals, our approach here is to convert a proof algebra of the form for standard induction

$$PrfAlgRec \cdot S.D\ \ S.inD \cdot P = \Pi\, xs{:}F \cdot (Sigma \cdot S.D \cdot P).\, P\ \ (S.inD\ \ (fmap\ \ proj1\ \ xs))$$

into one of the form for Lepigre-Raffalli induction.

Function *fromPrfAlgRec* gives the conversion of proof algebras, and its body is best read bottom-up. The bound *xs* is an F-collection of predecessors playing dual roles as subdata and inductive hypotheses that require proof algebras at the universally quantified type Y, and the bound $y{:}Y$ is "self-handle" of the Lepigre-Raffalli proof algebra we are defining that gives us access to those inductive hypotheses. With *applyDRec* we separate

```
import functor .
import cast .
import mono .
import utils .

module lepigre-raffalli/generic/induction
  (F: ⋆ ➔ ⋆) (fmap: Fmap ·F)
  {fmapId : FmapId ·F fmap} {fmapCompose: FmapCompose ·F fmap} .

import functorThms ·F fmap -fmapId -fmapCompose .

import scott/generic/encoding as S ·F -monoFunctor .
import lepigre-raffalli/generic/encoding ·F -monoFunctor .

import data-char/primrec-typing ·F .
import data-char/primrec ·F fmap -fmapId -fmapCompose ·S.D S.inD .

applyDRec ◁ ∀ P: S.D ➔ ⋆. ∀ Y: ⋆. Y ➔ (ι x: S.D. DRec ·P x ·Y) ➔ Sigma ·S.D ·P
= Λ P. Λ Y. λ y. λ x. mksigma x.1 (x.2 y y) .

fromPrfAlgRec
◁ ∀ P: S.D ➔ ⋆. PrfAlgRec ·P ➔ PrfAlgLR ·P
= Λ P. λ a. Λ Y. λ xs. λ y.
  ρ ς (fmapId ·(ι x: S.D. DRec ·P x ·Y) ·S.D
         (λ x. proj1 (applyDRec y x)) (λ x. β) xs)
    @x.(P (S.inD x))
- ρ ς (fmapCompose (proj1 ·S.D ·P) (applyDRec ·P y) xs)
    @x.(P (S.inD x))
- a (fmap (applyDRec ·P y) xs) .

indD ◁ ∀ P: S.D ➔ ⋆. PrfAlgRec ·P ➔ Π x: S.D. P x
= Λ P. λ a. λ x.
  indLRD (fromPrfAlgRec a) x (fromPrfAlgRec a) .
```

Fig. 74. Generic induction for Scott encodings
(`lepigre-raffalli/generic/induction.ced`)

these two roles, producing a dependent pair of type $Sigma \cdot S.D \cdot P$ as expected for the usual formulation of induction.

The type of the final line of *fromPrfAlgRec* is:

$$P\ (S.inD\ (fmap\ proj1\ (fmap\ (applyDRec\ y)\ xs)))$$

1420 Reading bottom-up, we use the functor composition law to fuse the two separate mappings of *proj1* and *applyDRec y*. Then, observing that this results in the mapping of a function which is definitionally equal $\lambda\, x.\, x$ (by the computation law of *proj1*, Figure 46), we use the functor identity law to remove the mapping completely. The resulting type is convertible with the expected type $P\ (inDRec\ xs)$ (since $|inDRec| =_{\beta\eta}\ |S.inD|$). With the conversion complete, in *indD* we equip Scott encodings with the standard induction principle by invoking Lepigre-Raffalli induction on two copies of the converted proof algebra.

7.3.1. *Computational and extensional character.* With the standard induction principle derived for Scott encodings, we can now show that Scott-encoded datatypes enjoy the same characterization up to propositional equality as Parigot-encoded datatypes. Concerning the efficiency of solutions to recursion schemes, we have already seen that Scott encodings offer a superior simulation of case distinction. We now consider primitive recursion and iteration. For the module listed in Figures 75 and 76, the generic Scott encoding is imported without qualification, and "*LR.*" qualifies the definitions imported from generic Lepigre-Raffalli encoding.

Primitive recursion. The solution *recD* in Figure 75 for the combinator for primitive recursion is a non-dependent instance of the standard induction scheme. As we saw for primitive recursion on Scott naturals in Section 7.1, the computation law for generic primitive recursion, proved by *recDBeta*, does not hold by reduction in the operational semantics alone, but *does* hold up to joinability using a constant number of β-reduction

```
import functor .
import cast .
import recType .
import utils .

module lepigre-raffalli/generic/propos
  (F: ⋆ → ⋆) (fmap: Fmap ·F)
  {fmapId: FmapId ·F fmap} {fmapCompose: FmapCompose ·F fmap} .

import functorThms ·F fmap -fmapId -fmapCompose .
import scott/generic/encoding ·F -monoFunctor .
import lepigre-raffalli/generic/encoding as LR ·F -monoFunctor .
import lepigre-raffalli/generic/induction ·F fmap -fmapId -fmapCompose .

import data-char/primrec-typing ·F .
import data-char/primrec ·F fmap -fmapId -fmapCompose ·D inD .

recD ◁ PrimRec ·D
= Λ X. λ a. indD ·(λ x: D. X) a .

recDBeta ◁ PrimRecBeta recD
= Λ X. Λ a. Λ xs. β .

recDEta ◁ PrimRecEta recD = <..>
```

Fig. 75. Characterization of *recD* (`lepigre-raffalli/generic/props.ced`)

steps. We illustrate for arbitrary (untyped) terms t and t'.

$$\begin{array}{ll}
& \mathit{recD}\ t\ (\mathit{inD}\ t') \\
\rightsquigarrow_\beta^* & \mathit{indLRD}\ (\mathit{fromPrfAlgRec}\ t)\ (\mathit{inD}\ t')\ (\mathit{fromPrfAlgRec}\ t) \\
\rightsquigarrow_\beta^* & \mathit{fromPrfAlgRec}\ t\ t'\ (\mathit{fromPrfAlgRec}\ t) \\
\rightsquigarrow_\beta^* & t\ (\mathit{fmap}\ (\mathit{applyDRec}\ (\mathit{fromPrfAlgRec}\ t))\ t') \\
\rightsquigarrow_\beta^* & t\ (\mathit{fmap}\ (\lambda\, x.\, \mathit{mksigma}\ x\ (x\ (\mathit{fromPrfAlgRec}\ t)\ (\mathit{fromPrfAlgRec}\ t)))\ t') \\
\leftsquigarrow_\beta^* & t\ (\mathit{fmap}\ (\lambda\, x.\, \mathit{mksigma}\ x\ (\mathit{recD}\ t))\ t') \\
\leftsquigarrow_\beta^* & t\ (\mathit{fmap}\ (\mathit{fork}\ \mathit{id}\ (\mathit{recD}\ a))\ t')
\end{array}$$

The extensionality law *recDEta*, whose proof is omitted in the figure, follows from induction.

```
import data-char/iter-typing ·F .
import data-char/iter ·F fmap ·D inD .

lrFromAlg ◁ ∀ X: ⋆. Alg ·X → LR.PrfAlgLR ·(λ x: D. X)
= Λ X. λ a. Λ Y. λ xs. λ y.
  a (fmap ·(ι x: D. LR.DRec ·(λ x: D. X) x ·Y) ·X (λ x. x.2 y y) xs) .

foldD ◁ Iter ·D
= Λ X. λ a. λ x. LR.indLRD ·(λ x: D. X) (lrFromAlg a) x (lrFromAlg a) .

foldDBeta ◁ IterBeta foldD
= Λ X. Λ a. Λ xs. β .

algHomLemma
◁ ∀ X: ⋆. ∀ a: Alg ·X. ∀ h: D → X. AlgHom ·X a h → AlgRecHom ·X (fromAlg a) h
= <..>

foldDEta ◁ IterEta foldD = <..>
```

Fig. 76. Characterization of *foldD* (`lepigre-raffalli/generic/props.ced`)

Iteration. While primitive recursion can be used to simulate iteration, we saw in Section 6.3.3 that this results in a definition of *foldD* that obeys the expected computation law only up to the functor laws. We now show in Figure 76 that with Lepigre-Raffalli recursion, we can do better and obtain a solution obeying the computation law by definitional equality alone. The first definition, *lrFromAlg*, converts a function of type $Alg \cdot X$ (used in iteration) to a function for use in Lepigre-Raffalli recursion. It maps over the F-collection of predecessors having dual roles, applying each to two copies of the handle y specifying the next step of recursion. For the solution *foldD* for iteration, we use Lepigre-Raffalli recursion on two copies of the converted $a : Alg \cdot X$.

As was the case for *recD*, with *foldD* the left-hand and right-hand sides of the computation law for iteration are joinable using a constant number of full β-reductions. This means that the proof *foldDBeta* holds by definitional equality alone. The proof of the extensionality law, *foldDEta*, follows as a consequence of *recDEta* and a lemma that any function $h : D \to X$ which satisfies the computation law for iteration with respect to some $a : Alg \cdot X$ also satisfies the computation law for primitive recursion with respect to *fromAlg a* (Figure 51).

7.3.2. *Scott encoding vs. Parigot encoding.* Satisfaction of the above properties by the Scott and Parigot encoding establishes that both are adequate representations of inductive datatypes in Cedille. However, there are compelling reasons for preferring the Scott encoding. First, and as mentioned before, the Parigot encoding suffers from significant space overhead: Parigot naturals require exponential space compared to the linear-space Scott naturals. Second, efficiency of the destructor for Scott encodings does not depend on the choice of evaluation strategy and the computation law for iteration is satisfied by definitional equality. Finally, not all monotone type schemes in Cedille are functors. For such a type scheme F, we cannot use F as a datatype signature for the generic Parigot encoding. However, we *can* use F in this way for the generic Scott encoding, and although we cannot obtain the standard induction principle for the resulting datatype, we still may still use Lepigre-Raffalli-style induction.

How significant a limitation for the Parigot encoding is this in practice? Consider a datatype for infinitely branching trees, which in Haskell would be defined as follows.

```
data ITree = Leaf | Node (Nat -> ITree)
```

ITree is a positive datatype, however in the presence of the δ axiom (see Figure 3), we can in fact *prove* that the usual mapping operation for the signature of infinitary trees does not satisfy the functor laws as we formulated them in Figure 44.

In Figure 77, we give an axiomatic summary of derivable sum (coproduct) types with induction in Cedille. The constructors are *in1* and *in2*, and the induction principle is *indsum* and follows the expected computation laws. In Figure 78 we define *ITreeF*, the signature for infinitely branching trees, and *itreeFmap*, its corresponding mapping operation (here *Unit* is an alias for the type of the polymorphic identity function). Following this, we prove with *monoITreeF* that this type scheme is monotonic. The function from

$$\frac{\Gamma \vdash S \overrightarrow{\in} \star \quad \Gamma \vdash T \overrightarrow{\in} \star}{\Gamma \vdash \mathit{Sum} \cdot S \cdot T \overrightarrow{\in} \star}$$

$$\frac{\Gamma \vdash \mathit{Sum} \cdot S \cdot T \overrightarrow{\in} \star \quad \Gamma \vdash s \overleftarrow{\in} S}{\Gamma \vdash \mathit{in1} \cdot S \cdot T\ s \overrightarrow{\in} \mathit{Sum} \cdot S \cdot T} \quad \frac{\Gamma \vdash \mathit{Sum} \cdot S \cdot T \overrightarrow{\in} \star \quad \Gamma \vdash t \overleftarrow{\in} T}{\Gamma \vdash \mathit{in2} \cdot S \cdot T\ t \overrightarrow{\in} \mathit{Sum} \cdot S \cdot T}$$

$$\frac{\begin{array}{c}\Gamma \vdash s \overrightarrow{\in} \mathit{Sum} \cdot S \cdot T \quad \Gamma \vdash P \overrightarrow{\in} \mathit{Sum} \cdot S \cdot T \to \star \\ \Gamma \vdash t_1 \overleftarrow{\in} \Pi\, x{:}T.\, P\ (\mathit{in1}\ x) \quad \Gamma \vdash t_2 \overleftarrow{\in} \Pi\, x{:}S.\, P\ (\mathit{in2}\ x)\end{array}}{\Gamma \vdash \mathit{indsum}\ s \cdot P\ t_1\ t_2 \overrightarrow{\in} P\ s}$$

$$\begin{array}{lcl} |\mathit{indsum}\ (\mathit{in1}\ s)\ t_1\ t_2| & =_{\beta\eta} & |t_1\ t| \\ |\mathit{indsum}\ (\mathit{in2}\ t)\ t_1\ t_2| & =_{\beta\eta} & |t_2\ t| \end{array}$$

Fig. 77. *Sum*, axiomatically (`utils/sum.ced`)

ITreeF $\cdot X$ to *ITreeF* $\cdot Y$ that realizes the type inclusion is *itreeFmap* (*elimCast* -*c*), and the proof that it behaves as the identity function follows by induction on *Sum*.

For the counterexample *itreeFmapIdAbsurd*, we consider two terms of type *ITreeF*$\cdot$*Nat*: the first, t_1, is defined using the second coproduct injection on the identity function for *Nat*, and the first, t_2, is the result of mapping *caseNat zero suc* over t_1. From *reflectNat* (Section 5.1.1), we know that *caseNat zero suc* behaves as the identity function on *Nat*. If *itreeFmap* satisfied the functor identity law, we would thus obtain a proof that t_1 and t_2 are propositionally equal. However, the erasures of t_1 and t_2 are closed untyped terms which are $\beta\eta$-inequivalent, so we use δ to derive a contradiction. This establishes that our generic derivation of Scott-encoded data, together with Lepigre-Raffalli-style recursion and induction, allow for programming with a strictly larger set of inductive datatypes than does our generic derivation of Parigot-encoded data.

8. Related Work

Monotone inductive types. Matthes (2002) employs Tarski's fixpoint theorem to motivate the construction of a typed lambda calculus with monotone recursive types. The

```
module signatures/itree .

import functor .
import cast .
import mono .
import utils .

import scott/concrete/nat .

ITreeF ◁ ⋆ → ⋆ = λ X: ⋆. Sum ·Unit ·(Nat → X) .

itreeFmap ◁ Fmap ·ITreeF
= Λ X. Λ Y. λ f. λ t.
  indsum t ·(λ _: ITreeF ·X. ITreeF ·Y)
    (λ u. in1 u) (λ x. in2 (λ n. f (x n))) .

monoITreeF ◁ Mono ·ITreeF
= Λ X. Λ Y. λ c.
  intrCast
    -(itreeFmap (elimCast -c))
    -(λ t. indsum t ·(λ x: ITreeF ·X. { itreeFmap (elimCast -c) x ≃ x })
             (λ u. β) (λ x. β)) .

t1 ◁ ITreeF ·Nat = in2 (λ x. x) .
t2 ◁ ITreeF ·Nat = itreeFmap (caseNat zero suc) t1 .

itreeFmapIdAbsurd ◁ FmapId ·ITreeF itreeFmap → ∀ X: ⋆. X
= λ fid. Λ X.
  [pf ◁ { t2 ≃ t1 } = fid (caseNat zero suc) (caseNatEta) t1]
- δ - pf .
```

Fig. 78. Counter-example: a monotone type scheme which is not a functor (`signatures/itree.ced`)

gap between this order-theoretic result and type theory is bridged using category theory, with evidence that a type scheme is monotonic corresponding to the morphism-mapping rule of a functor. Matthes shows that as long as the reduction rule eliminating an *unroll* of a *roll* incorporates the monotonicity witness in a certain way, strong normalization of System F is preserved by extension with monotone iso-recursive types. Otherwise, he shows a counterexample to normalization.

In contrast, we establish that type inclusions (zero-cost casts) induce a preorder *within* the type theory of Cedille, and carry out a modification of Tarski's order-theoretic result directly within it. Evidence of monotonicity is given by an operation lifting type inclusions, not arbitrary functions, over a type scheme. As mentioned in the introduction, deriving monotone recursive types within the type theory of Cedille has the benefit of guaranteeing that they enjoy precisely the same meta-theoretic properties as enjoyed by Cedille itself – no additional work is required.

Impredicative encodings and datatype recursion schemes. Our use of casts in deriving recursive types guarantees that the *rolling* and *unrolling* operations take constant time, permitting the definition of efficient data accessors for inductive datatypes defined with them. However, when using recursive types to encode datatypes one usually desires efficient solutions to recursion schemes for datatypes, and the derivation in Section 3.4 does not on its own provide this.

Independently, Mendler (1991) and Geuvers (1992) developed the category-theoretic notion of recursive F-algebras to give the semantics of the primitive recursion scheme for inductive datatypes, and Geuvers (1992) and Matthes (2002) use this notion in extending a typed lambda calculus with typing and reduction rules for an efficient primitive recursion scheme for inductive datatypes. The process of using a particular recursion scheme to directly obtain an impredicative encoding is folklore knowledge (c.f. Abel et al., 2005, Section 3.6). Geuvers (2014) used this process to examine the close connection between the (co)case-distinction (resp. primitive (co)recursion) scheme and the Scott (resp. Parigot) encoding for (co)inductive types in a theory with primitive positive recursive types. Using derived monotone recursive types in Cedille, we follow the same approach but for encodings of datatypes with induction principles, allowing us to establish within Cedille that the solution to the primitive recursion scheme is unique (i.e., that we obtain *initial* recursive F-algebras).

Recursor for Scott-encoded data. The non-dependent impredicative encoding we used to equip Scott-encoded naturals with primitive recursion in Section 7.1 is based on a result by Lepigre and Raffalli (2019). We thus dub it the *Lepigre-Raffalli* encoding, though they report that the encoding is in fact due to Parigot. In earlier work, Parigot (1988) demonstrated the lambda term that realizes the primitive recursion scheme for Scott naturals, but the typing of this term involved reasoning outside of the logical framework being used. The type system in which Lepigre and Raffalli (2019) carry out this construction has built-in notions of least and greatest type fixpoints and a sophisticated form of subtyping that utilizes ordinals and well-founded circular typing derivations based on cyclic proof theory (Santocanale, 2002). Roughly, the correspondence between their type system and that of Cedille's is so: both theories are Curry-style, enabling a rich subtyping relation which in Cedille is internalized as *Cast*; and in defining recursor for Scott naturals, we replace the circular subtyping derivation with an internally realized proof of the fact that our derived recursive types are least fixpoints of type schemes.

Our two-fold generalization of the Lepigre-Raffalli encoding (making it both generic *and* dependent) is novel. To the best of our knowledge, the observation that the Lepigre-Raffalli-style recursion scheme associated to this encoding can be understood as introducing recursion into the computation laws for the case-distinction scheme is also novel. This characterization informs both our minor modification to the encoding in Section 7.1 for natural numbers (we quantify over types for both base and step cases of recursion, instead of quantifying over only the latter as done by Lepigre and Raffalli (2019)) and the generic formulation. Presently, this connection between Lepigre-Raffalli recursion and the case-distinction scheme serves a pedagogical role; we leave as future work the task of providing a more semantic (e.g., category-theoretic) account of the Lepigre-Raffalli recursion scheme.

Lambda encodings in Cedille. Work prior to ours describes the generic derivation of induction for lambda-encoded data in Cedille. This was first accomplished by Firsov and Stump (2018) for the Church and Mendler encodings, which do not require recursive types as derived in this paper. The approach used for the Mendler encoding by was then further refined by Firsov et al. (2018) to enable efficient data accessors, resulting in the first-ever example of a lambda encoding in type theory with derivable induction, constant-time destructor, and whose representation requires only linear space. To the best of our knowledge, this paper establishes that the Scott encoding is the second-ever example of a lambda encoding enjoying these same properties. Firsov and Stump (2018) and Firsov et al. (2018) also provide a computational characterization for their encodings, limited to Lambek's lemma and the computation law for Mendler-style iteration. For the Parigot and Scott encodings we have presented, we showed Lambek's lemma and both the computation *and* extensionality laws for the case-distinction, iteration, and primitive recursion schemes.

Conclusion and Future Work

We have shown how to derive monotone recursive types with constant-time *roll* and *unroll* operations within the type theory of Cedille by applying Tarski's least fixpoint theorem to a preorder on types induced by an internalized notion of type inclusion. By deriving recursive types within a theory, rather than extending it, we do not need to rework existing meta-theoretic results to ensure logical consistency or provide a normalization guarantee. As applications, we use the derived monotone recursive types to derive two recursive representations of data in lambda calculus, the Parigot and Scott encoding, *generically* in a signature functor F. For both encodings, we derived induction and gave a thorough characterization of the solutions they admit for case distinction, iteration, and primitive recursion. In particular, we showed that with the Scott encoding all three of these schemes can be efficiently simulated. This derivation, which builds on a result described by Lepigre and Raffalli (2019), crucially uses the fact that recursive types in Cedille provide *least* fixpoints of type schemes.

In the authors' opinion, we have demonstrated that lambda encodings in Cedille provide an adequate basis for programming with inductive datatypes. Building from this basis to a convenient surface language requires the generation of monotonicity witnesses from datatype declarations. Our experience coding such proofs in Cedille leads us to believe they can be readily mechanized for a large class of nonstrictly positive datatypes, including infinitary trees and the usual formulation of rose trees. However, more care is needed for checking monotonicity of nested datatypes (Bird and Meertens, 1998; Abel et al., 2005).

Finally, we believe our developments have raised two interesting questions for future investigation. The first of these is the development of a categorical semantics for Lepigre-Raffalli recursion, allowing us to complete the characterization of Scott-encoded datatypes whose signatures are positive but non-functorial. Second, in the derivation of primitive recursion for Scott encodings leastness of the fixpoint formed from our derived recursive type operator plays a similar role to the cyclic subtyping rules of Lepigre and Raffalli (2019). If this correspondence generalizes, some subset of their type system might be translatable to Cedille, opening the way to a surface language with a rich notion of subtyping in the presence of recursive types that is based on internalized type inclusions.

Financial Aid

We gratefully acknowledge NSF support under award 1524519, and DoD support under award FA9550-16-1-0082 (MURI program).

References

Abbott, M. G., Altenkirch, T., and Ghani, N. (2003). Categories of containers. In Gordon, A. D., editor, *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2620 of *Lecture Notes in Computer Science*, pages 23–38. Springer.

Abel, A. (2010). MiniAgda: Integrating sized and dependent types. In Komendantskaya, E., Bove, A., and Niqui, M., editors, *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010*, volume 5 of *EPiC Series*, pages 18–33. EasyChair.

Abel, A., Matthes, R., and Uustalu, T. (2005). Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.*, 333(1-2):3–66.

Allen, S. F., Bickford, M., Constable, R. L., Eaton, R., Kreitz, C., Lorigo, L., and Moran, E. (2006). Innovations in computational type theory using Nuprl. *J. Applied Logic*, 4(4):428–469.

Atkey, R. (2018). Syntax and semantics of quantitative type theory. In Dawar, A. and Grädel, E., 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.

Bird, R. S. and Meertens, L. G. L. T. (1998). Nested datatypes. In *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, pages 52–67.

Breitner, J., Eisenberg, R. A., Jones, S. P., and Weirich, S. (2016). Safe zero-cost coercions for Haskell. *J. Funct. Program.*, 26:e15.

Böhm, C., Dezani-Ciancaglini, M., Peretti, P., and Rocca, S. D. (1979). A discrimination algorithm inside λ-β-calculus. *Theoretical Computer Science*, 8(3):271 – 291.

Crary, K., Harper, R., and Puri, S. (1999). What is a Recursive Module? In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, New York, NY, USA. ACM.

Dybjer, P. and Palmgren, E. (2016). Intuitionistic Type Theory. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2016 edition.

Firsov, D., Blair, R., and Stump, A. (2018). Efficient Mendler-Style Lambda-Encodings in Cedille. In Avigad, J. and Mahboubi, A., 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. Springer.

Firsov, D. and Stump, A. (2018). Generic Derivation of Induction for Impredicative Encodings in Cedille. In Andronick, J. and Felty, A., editors, *Certified Programs and Proofs (CPP)*.

Geuvers, H. (1992). Inductive and coinductive types with iteration and recursion. In *WORKSHOP ON*. Bastad, Chalmers University of Technology.

Geuvers, H. (2001). Induction Is Not Derivable in Second Order Dependent Type Theory. In Abramsky, S., editor, *Typed Lambda Calculi and Applications (TLCA)*, volume 2044 of *Lecture Notes in Computer Science*, pages 166–181. Springer.

Geuvers, H. (2014). The Church-Scott representation of inductive and coinductive data. Unpublished manuscript.

Jenkins, C. and Stump, A. (2018). Spine-local Type Inference. *CoRR*, abs/1805.10383. Accepted for post-proceedings of Implementation of Functional Languages (IFL) 2018.

Kleene, S. (1965). Classical Extensions of Intuitionistic Mathematics. In Bar-Hillel, Y., editor, *LMPS 2*, pages 31–44. North-Holland Publishing Company.

Kopylov, A. (2003). Dependent intersection: A new way of defining records in type theory. In *18th IEEE Symposium on Logic in Computer Science (LICS)*, pages 86–95.

Lambek, J. (1968). A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103(2):151–161.

Lassez, J.-L., Nguyen, V., and Sonenberg, E. (1982). Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112 – 116.

Leivant, D. (1983). Reasoning about functional programs and complexity classes associated with type disciplines. In *24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 460–469. IEEE Computer Society.

Lepigre, R. and Raffalli, C. (2019). Practical subtyping for Curry-style languages. *ACM Trans. Program. Lang. Syst.*, 41(1):5:1–5:58.

Matthes, R. (1999). Monotone Fixed-Point Types and Strong Normalization. In Gottlob, G., Grandjean, E., and Seyr, K., editors, *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings*, volume 1584 of *Lecture Notes in Computer Science*, pages 298–312. Springer.

Matthes, R. (2002). Tarski's fixed-point theorem and lambda calculi with monotone inductive types. *Synthese*, 133(1-2):107–129.

The Coq development team (2018). *The Coq proof assistant reference manual.* LogiCal Project. Version 8.7.2.

Mendler, N. P. (1991). Predictive type universes and primitive recursion. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 173–184.

Miquel, A. (2001). The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In Abramsky, S., editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359. Springer.

Parigot, M. (1988). Programming with proofs: a second order type theory. In Ganzinger, H., editor, *European Symposium On Programming (ESOP)*, volume 300 of *Lecture Notes in Computer Science*, pages 145–159. Springer.

Parigot, M. (1989). On the representation of data in lambda-calculus. In Börger, E., Büning, H., and Richter, M., editors, *Computer Science Logic (CSL)*, volume 440 of *Lecture Notes in Computer Science*, pages 309–321. Springer.

Parigot, M. (1992). Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356.

Pierce, B. C. (2002). *Types and programming languages.* MIT Press.

Pierce, B. C. and Turner, D. N. (2000). Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44.

Santocanale, L. (2002). A calculus of circular proofs and its categorical semantics. In Nielsen, M. and Engberg, U., editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 357–371. Springer.

Scott, D. (1962). A system of functional abstraction. Lectures delivered at University of California, Berkeley.

Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* Elsevier Science Inc., New York, NY, USA.

Splawski, Z. and Urzyczyn, P. (1999). Type fixpoints: Iteration vs. recursion. In *ICFP*.

Stump, A. (2017). The Calculus of Dependent Lambda Eliminations. *J. Funct. Program.*, 27:e14.

Stump, A. (2018a). From realizability to induction via dependent intersection. *Ann. Pure Appl. Logic*, 169(7):637–655.

Stump, A. (2018b). Syntax and typing for Cedille core. *CoRR*, abs/1811.01318.

Stump, A. and Fu, P. (2016). Efficiency of lambda-encodings in total type theory. *J. Funct. Program.*, 26:e3.

Stump, A. and Jenkins, C. (2018). Syntax and semantics of Cedille. *CoRR*, abs/1806.04709.

Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309.

The Agda Team (2021). The Agda standard library, v1.5. `https://github.com/agda/agda-stdlib`.

Ullrich, M. (2020). Generating induction principles for nested inductive types in metacoq. Bachelor's thesis.

Uustalu, T. and Vene, V. (1999). Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*, 10(1):5–26.

Wadler, P. (1990). Recursive types for free! Unpublished manuscript.