Mathematical Structures in Computer Science (2021), **31**, pp. 682–745
doi:10.1017/S0960129521000402

CAMBRIDGE UNIVERSITY PRESS

PAPER

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, IA, USA
*Corresponding author. Email: christopher-jenkins@uiowa.edu

(Received 23 December 2019; revised 5 October 2021; accepted 29 October 2021; first published online 10 December 2021)

Abstract

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. This derivation takes place within the preorder on Cedille types induced by *type inclusions*, a notion which is expressible within the theory itself. As applications, we use monotone recursive types to generically derive two recursive representations of data in lambda calculus, the Parigot and Scott encoding. For both encodings, we prove induction and examine the computational and extensional properties of their destructor, iterator, and primitive recursor in Cedille. For our Scott encoding in particular, we translate into Cedille a construction due to Lepigre and Raffalli (2019) that equips Scott naturals with primitive recursion, then extend this construction to derive a generic induction principle. This allows us to give efficient and provably unique (up to function extensionality) solutions for the iteration and primitive recursion schemes for Scott-encoded data.

Keywords: Recursive types; impredicative encodings; Scott encoding; recursion schemes; CDLE

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 $[\mu\, X.\, T/X]T$ (the substitution of $\mu\, X.\, T$ for X in 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 either 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.

Adding unrestricted recursive types 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 same 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 Author(s), 2021. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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 $\rightarrow$ preserves polarity in its codomain part and switches polarity in its domain part. For example, X occurs only positively in $(X \rightarrow Y) \rightarrow Y$, 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 $[T_1/Y]T_2$. For example, take T_1 to be X and T_2 to be $Y \rightarrow X$.

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):

$$\forall\ X.\ \forall\ Y.\ (X \rightarrow Y) \rightarrow F \cdot X \rightarrow F \cdot Y$$

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, 2021). 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.[1] The main benefit to this approach is that the existing meta-theoretic results for CDLE – confluence, logical consistency, 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 the predecessor function 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, the Parigot encoding readily supports the primitive recursion scheme but suffers from exponential representation size, whereas the Scott encoding has a linear representation size but only readily supports the case distinction scheme. For both encodings, we derive the primitive 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 sophisticated subtyping 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 representation (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 applications of recursive types in deriving lambda encodings with induction follow a common structure. This structure is outlined in Section 4, where we elaborate on the connection between structured recursion schemes and lambda encodings of datatypes using impredicative

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

polymorphism, and we explain the computational and extensional criteria we use to characterize implementations of the iteration scheme for the Scott and Parigot encodings. Section 5 covers the Scott encoding, introducing the case distinction scheme and giving first a concrete derivation of natural numbers supporting proof by cases, then the fully generic derivation. Section 6 covers the Parigot encoding, introducing the primitive recursion scheme and giving first a concrete example for Parigot-encoded naturals with an induction principle, then the fully generic derivation. Section 7 revisits the Scott encoding, showing concretely how to derive primitive recursion for Scott naturals, then generalizing to the derivation of the standard induction principle for generic Scott-encoded data. Section 8 gives a demonstration that our generic Scott encoding can be used for a wider class of datatypes than can our generic Parigot encoding. Finally, Section 9 discusses related work and Section 10 concludes and discusses future work.

2. Background: The Calculus of Dependent Lambda Eliminations

In this section, we review CDLE and its implementation in the Cedille programming language (Stump, 2017; Stump and Jenkins, 2021). 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, 2018*b*), 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 (2021) (Stump (2017) describes an earlier version); here, we have slightly simplified the typing rule for the ρ term construct.

To achieve compactness, CDLE is a pure typed lambda calculus. In particular, it 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 (2018*a*) showed how CDLE overcomes this fundamental difficulty by extending CC with three new typing constructs: the implicit products of Miquel (2001); the dependent intersections of Kopylov (2003); and equality over untyped terms. 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 $\Gamma \vdash T \stackrel{\rightarrow}{\in} \kappa$ 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 noncongruence rules for conversion for type constructors 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 $\rightsquigarrow_{\mathbf{n}}$ and $\rightsquigarrow_{\mathbf{n}}^*$ 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 corresponding term subexpressions are checked for $\beta\eta$-equivalence (written $=_{\beta\eta}$), modulo erasure (see Figure 4).

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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

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

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

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

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

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

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

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

Figure 1. Kind formation and kinding of types in CDLE.

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

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

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

Figure 2. Noncongruence rules for classifier convertibility.

functions that abstract over terms. The type constructs that CDLE inherits from CC 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 applications of type constructors to terms $T\ t$ and to other 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, for example, $\forall\, x : T.\ T'$ 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.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

$$\Gamma \vdash t \overset{\rightsquigarrow^*_{\mathbf{n}}}{\epsilon} T \;=\; \exists T'.\, (\Gamma \vdash t \overset{\rightarrow}{\epsilon} T') \wedge (T' \rightsquigarrow^*_{\mathbf{n}} T)$$

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

$$\frac{T \rightsquigarrow^*_{\mathbf{n}} \Pi x : T_1.\, T_2 \quad \Gamma, x : T_1 \vdash t \overset{\leftarrow}{\epsilon} T_2}{\Gamma \vdash \lambda x.\, t \overset{\leftarrow}{\epsilon} T} \qquad \frac{\Gamma \vdash t \overset{\rightsquigarrow^*_{\mathbf{n}}}{\epsilon} \Pi x : T'.\, T \quad \Gamma \vdash t' \overset{\leftarrow}{\epsilon} T'}{\Gamma \vdash t\, t' \overset{\rightarrow}{\epsilon} [t'/x]T}$$

$$\frac{T' \rightsquigarrow^*_{\mathbf{n}} \forall X : \kappa.\, T \quad \Gamma, X : \kappa \vdash t \overset{\leftarrow}{\epsilon} T}{\Gamma \vdash \Lambda X.\, t \overset{\leftarrow}{\epsilon} T'} \qquad \frac{\Gamma \vdash t \overset{\rightsquigarrow^*_{\mathbf{n}}}{\epsilon} \forall X : \kappa.\, T \quad \Gamma \vdash T' \overset{\rightarrow}{\epsilon} \kappa' \quad \kappa' \cong \kappa}{\Gamma \vdash t \cdot T' \overset{\rightarrow}{\epsilon} [T'/X]T}$$

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\epsilon} \star \quad \Gamma \vdash t \overset{\leftarrow}{\epsilon} T}{\Gamma \vdash \chi\, T \text{ - } t \overset{\rightarrow}{\epsilon} T} \qquad \frac{\Gamma \vdash T_1 \overset{\rightarrow}{\epsilon} \star \quad \Gamma \vdash t_1 \overset{\leftarrow}{\epsilon} T_1 \quad \Gamma, x : T_1 \vdash t_2 \overset{\leftarrow}{\epsilon} T_2}{\Gamma \vdash [x \lhd T_1 = t_1] \text{ - } t_2 \overset{\leftarrow}{\epsilon} T_2}$$

Figure 3. Standard term constructs.

2.2 Term constructs

Figure 3 gives the type inference rules for the standard term constructs of CDLE. These type inference rules, as well as those listed in Figures 5, 6, and 7, are *bidirectional* (c.f. Pierce and Turner, 2000): judgment $\Gamma \vdash t \overset{\rightarrow}{\epsilon} T$ indicates term t *synthesizes* type T under typing context Γ and judgment $\Gamma \vdash t \overset{\leftarrow}{\epsilon} T$ 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 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 $\Gamma \vdash t \overset{\rightsquigarrow^*_{\mathbf{n}}}{\epsilon} T$ (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. Local term definitions are given with

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

to be read "let x of type T_1 be t_1 in t_2," and global definitions are given with $x \lhd T = t$. (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 $|t|$.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

Figure 4. Erasure for annotated terms.

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

Figure 5. Implicit products.

$$
\frac{T \rightsquigarrow_{\mathbf{n}}^{*} \iota\, x : T_1.\, T_2 \quad \Gamma \vdash t_1 \overset{\leftarrow}{\in} T_1 \quad \Gamma \vdash t_2 \overset{\leftarrow}{\in} [t_1/x]T_2 \quad |t_1| =_{\beta\eta} |t_2|}{\Gamma \vdash [t_1, t_2] \overset{\leftarrow}{\in} T}
$$

$$
\frac{\Gamma \vdash t \overset{\rightsquigarrow_{\mathbf{n}}^{*}}{\in} \iota\, x : T_1.\, T_2}{\Gamma \vdash t.1 \overset{\rightarrow}{\in} T_1}
\qquad
\frac{\Gamma \vdash t \overset{\rightsquigarrow_{\mathbf{n}}^{*}}{\in} \iota\, x : T_1.\, T_2}{\Gamma \vdash t.2 \overset{\rightarrow}{\in} [t.1/x]T_2}
$$

Figure 6. Dependent intersections.

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 except modulo erasure.

The implicit product type $\forall\, x : T_1.\, T_2$ of Miquel (2001) (Figure 5) is the type for functions which accept an erased (computationally irrelevant) input of type T_1 and produce a result of type T_2. Implicit products are introduced with $\Lambda\, x.\, t$, and the type inference rule is the same as for ordinary function abstractions except for the side condition that x does not occur free in the erasure of the body t. Thus, the argument can play no computational role in the function and exists solely for the purposes of typing. 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 $t\ \text{-}t'$ 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 $T_1 \to T_2$ for $\Pi\, x : T_1.\, T_2$.

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) (Figure 6) is the type for terms t which can be assigned both type T_1 and type $[t/x]T_2$. It is a dependent generalization of intersection types (c.f. Barendregt et al., 2013, Part 3) in Curry-style theories, which allow one to express the fact that an untyped lambda calculus term can be assigned two different types. In Cedille's

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

$$\frac{T \rightsquigarrow_{\mathbf{n}}^{*} \{t_1 \simeq t_2\} \quad FV(t') \subseteq dom(\Gamma) \quad |t_1| =_{\beta\eta} |t_2|}{\Gamma \vdash \beta\{t'\} \overleftarrow{\in} T}$$

$$\frac{\Gamma \vdash t \overrightarrow{\in}^{*}_{\mathbf{n}} \{t_1 \simeq t_2\} \quad \Gamma \vdash [t_2/x]T' \overrightarrow{\in} \star \quad \Gamma \vdash t' \overleftarrow{\in} [t_2/x]T' \quad [t_1/x]T' \cong T}{\Gamma \vdash \rho\ t\ @x.T' - t' \overleftarrow{\in} T}$$

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

$$\frac{\Gamma \vdash t \overrightarrow{\in} T' \quad T' \cong \{\lambda\, x.\, \lambda\, y.\, x \simeq \lambda\, x.\, \lambda\, y.\, y\}}{\Gamma \vdash \delta - t \overleftarrow{\in} T} \quad \frac{\Gamma \vdash t \overrightarrow{\in}^{*}_{\mathbf{n}} \{t_1 \simeq t_2\}}{\Gamma \vdash \varsigma\ t \overrightarrow{\in} \{t_2 \simeq t_1\}}$$

Figure 7. Equality type.

annotated language, the introduction form for dependent intersections is written $[t_1, t_2]$ and 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 t_1 and t_2 are $\beta\eta$-equivalent modulo erasure. 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 $[t.1/x]T_2$.

Dependent intersections can be thought of as a dependent pair type where the two components are equal. Thus, 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. A consequence of this restriction is that the proof may be recovered, in the form of t itself, for use in computation.

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 $\rho\ t\ @x.T' - t'$ 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 $[t_1/x]T'$ is convertible with the expected type T. The entire expression erases to $|t'|$.

Example 1. Assume m and n have type *Nat*, *suc* and *pred* have type $Nat \to Nat$, 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 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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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 $isEven : Nat \to Bool$ a function returning *tt* if and only if its argument is even. Then, the type *Even* of even naturals can be defined as $\iota\, x : Nat.\, \{isEven\ x \simeq tt\}$. Since $|isEven\ zero| =_{\beta\eta} |tt|$, we can check $[zero, \beta\{zero\}]$ against type *Even*, and the expression erases to $|zero|$. More generally, if n is a *Nat* and t is a proof that $\{isEven\ n \simeq tt\}$, then $[n, \rho\ t\ @x.\{x \simeq tt\}\ -\ \beta\{n\}]$ 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 $\{isEven\ n \simeq tt\}$ to $\{tt \simeq tt\}$ 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 nonterminating 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 nontermination in inconsistent contexts. Assume there is a typing context Γ and term t such that $\Gamma \vdash t\ \vec{\in}\ \forall\, X : \star.\, X$, and let ω be the term:

$$\chi\ (\forall\, X : \star.\, X \to X)\ \text{-}\ \varphi\ (t \cdot \{\lambda\, x.\, x \simeq \lambda\, x.\, x\ x\})\ \text{-}\ (\Lambda\, X.\, \lambda\, x.\, x)\ \{\lambda\, x.\, x\ x\}$$

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

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

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).

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

Proposition 2. Stump and Jenkins, 2021.
There is no term t such that $\vdash t \vec{\in} \forall X : \star . X$.

Neither does nontermination from the Kleene trick nor 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, 2021. *Suppose that $\vdash t \vec{\in} T$, and that there exists t' such that $\vdash t' \vec{\in} T \to \Pi\, x : T_1 . T_2$ 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 terms involving 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

Having surveyed Cedille, we now turn to the derivation of recursive types within it. This is accomplished by implementing a version of Tarski's fixpoint theorem for monotone functions over a complete lattice. We 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 *when $f(x) \sqsubseteq x$.*

Theorem. (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 3: 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. 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. Since r is a lower bound of R, $r \sqsubseteq x$. 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 $r \sqsubseteq f(r)$.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash t \overset{\leftarrow}{\in} \mathit{Top}}{\Gamma \vdash \mathit{View} \cdot T\ t \overset{\rightarrow}{\in} \star} \qquad \frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash t \overset{\leftarrow}{\in} \mathit{Top} \quad \Gamma \vdash v \overset{\leftarrow}{\in} \mathit{View} \cdot T\ t}{\Gamma \vdash \mathit{elimView} \cdot T\ t\ \text{-}v \overset{\rightarrow}{\in} T}$$

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash t_1 \overset{\leftarrow}{\in} \mathit{Top} \quad \Gamma \vdash t_2 \overset{\leftarrow}{\in} T \quad \Gamma \vdash t \overset{\leftarrow}{\in} \{t_2 \simeq t_1\}}{\Gamma \vdash \mathit{intrView} \cdot T\ t_1\ \text{-}t_2\ \text{-}t \overset{\rightarrow}{\in} \mathit{View} \cdot T\ t_1}$$

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash t \overset{\leftarrow}{\in} \mathit{Top} \quad \Gamma \vdash v \overset{\leftarrow}{\in} \mathit{View} \cdot T\ t}{\Gamma \vdash \mathit{eqView} \cdot T\ \text{-}t\ \text{-}v \overset{\rightarrow}{\in} \{t \simeq v\}}$$

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

Figure 8. Internalized typing, axiomatically.

□

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 3.1 to the set of Cedille types – this is why we need to relax the assumption of Theorem 3 that S is a complete lattice. However, we must still answer several questions:

- how should the ordering $\sqsubseteq$ be implemented;
- how 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 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 the 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 8 summarizes the derivation of the *View* type family in Cedille, and Figure 9 gives its implementation. 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. It is defined using dependent intersection: $\mathit{View} \cdot T\ t = \iota\, x : T.\, \{x \simeq t\}$.

Axiomatic summary. The introduction form *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 expression $\mathit{intrView}\ t_1\ \text{-}t_2\ \text{-}t$ erases to $(\lambda\, x.\, x)\ |t_1|$. The elimination form *elimView* takes an untyped term t and

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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) t -(λ x. elimView β{ t x } -(v x)) -β .
```

Figure 9. Internalized typing (`view.ced`).

an erased argument v proving that t may be viewed as having type T and produces a term of type T. The crucial property of *elimView* is its erasure: *elimView* t $-v$ also erases to $(\lambda\, x.\, x)\ |t|$, and so the expression is definitionally equal to t itself. Finally, *eqView* provides a reasoning principle for views. It states that every proof v that t may be viewed as having type T is equal to t.

Implementation. We now turn to the Cedille implementation of *View* and its operations in Figure 9. The definition of *intrView* uses the φ axiom (Figure 7) and the Kleene trick (Section 2.3) so that the resulting $View \cdot T\ t_1$ erases to $|t_1|$ (see Figure 4 for erasure rules). Because of the Kleene trick, the requirement that a term both 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 uses φ to cast t to the type T of $v.1$ using the equality $\{v.1 \simeq t\}$ given by $v.2$. The reasoning principle *eqView* uses ρ to rewrite the expected type $\{t \simeq v\}$ with the equation $\{v.1 \simeq t\}$ (v and $v.1$ are convertible terms).

The last two definitions of Figure 9, *selfView* and *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 8. Definition *selfView* reflects the typing judgment that t has type T into the proposition that $\beta\{t\}$ can be viewed as having type T. Definition *extView* provides an extrinsic typing principle for functions: if we can show of an untyped term t that for all inputs x of type S we have that $t\ x$ can be viewed as having type T, then in fact t can be viewed as having type $S \to T$.

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 type S to type T will play the role of the ordering $S \sqsubseteq T$ in the proof of Theorem 3.1. We summarize the derivation of casts in Cedille axiomatically in Figure 10 and show their implementation in Figure 11.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

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

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

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

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

Figure 10. Casts, axiomatically.

```
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 .
```

Figure 11. Casts (`cast.ced`).

Axiomatic summary. The introduction form, *intrCast*, takes two erased arguments: a function $t : S \to T$ and a proof that t is *extensionally* the identity function for all terms of type S. In intrinsic type theories, 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, especially in the presence of the φ axiom. Indeed, by enabling the definition of *extView*, we will see that φ plays a crucial role in the implementation of *intrCast*.

The elimination form, *elimCast*, takes as an erased argument a proof c of the inclusion of type S into type T. Its crucial property is its erasure: in the Cedille implementation, *elimCast -c* erases to a term convertible with $\lambda x.\, x$. The last construct listed in Figure 10 is *eqCast*, the reasoning principle. It states that every witness of a type inclusion is provably equal to $\lambda x.\, x$.

Implementation. We now turn to the implementation of *Cast* in Figure 11. The type $\mathit{Cast} \cdot S \cdot T$ itself is defined to be the type of proofs that $\lambda x.\, x$ may be viewed as having type $S \to T$. The elimination form *elimCast* and reasoning principle *eqCast* are direct consequences of *elimView* and *eqView*, so we focus on the introduction form *intrCast*. Without the use of *extView*, we might

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 x. 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 .
```

Figure 12. Casts form a preorder (`cast.ced`).

expect the definition of *intrCast* to be of the form:

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

where • 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 are 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 *extrinsic typing* of 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 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 ascribing to $\lambda\, x.\, x$ the type $S \to T$.

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, that 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. *Cast* induces a preorder (or thin category) on Cedille types.

Proof. Figure 12 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 a lifting, for all types S and T, of any cast from S to T to a cast from $F \cdot S$ to $F \cdot T$. This is *Mono* in Figure 13. 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. Although these are somewhat burdensome to write by hand, their repetitive nature makes us optimistic such proofs can be automated for the signatures of standard datatypes.

We give an example in Figure 14 to illustrate the method. 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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
import cast .
module mono .

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

Figure 13. Monotonicity (`mono.ced`).

```
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 z (λ r. s (elimCast -c r)))
           -(λ n. β).
```

Figure 14. Monotonicity for *NatF*.

```
import cast .
import mono .
module recType (F : ⋆ → ⋆).

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

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

Figure 15. Monotone recursive types derived in Cedille (`recType.ced`).

cast from $NatF \cdot X$ to $NatF \cdot Y$. We do this using *intrCast* on a function that is *definitionally* equal ($\beta\eta$-convertible modulo erasure) to the identity function.

After assuming $n : NatF \cdot X$, we introduce a term of type $NatF \cdot Y$ by abstracting over type Z and terms $z : Z$ and $s : Y \to Z$. Instantiating the type argument of n with Z, the second term argument we provide must have type $X \to Z$. For this, we precompose s with the assumed cast c from X to Y.

3.4 Translating the proof of Theorem 3.1 to Cedille

Figure 15 shows the translation of the proof of Theorem 3.1 to Cedille, deriving monotone recursive types. Cedille's module system allows us to parametrize the module shown in Figure 15 by the type scheme F, and all definitions implicitly take F as an argument. For the axiomatic presentation in Figure 17, 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, 2021), types are interpreted as sets of ($\beta\eta$-equivalence classes of) closed terms, and in particular the meaning of an

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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 15 expresses the intersection of the set of all F-closed types. This *Rec* corresponds to r in the proof of Theorem 3.1. 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 $Cast \cdot (F \cdot X) \cdot X$. By making the argument erased, we express the idea of taking the intersection of sets of terms satisfying a property, and not a set of functions that take a proof of the property as an argument.

Theorem. *Rec* is the greatest lower bound of the set of F-closed types.

Proof. In Figure 15, definition *recLB* establishes that *Rec* is a lower bound of this set and *recGLB* establishes that it greater than or equal to any other lower bound, that is, for any other lower bound Y there is a cast from Y to *Rec*. For *recLB*, assume we have an F-closed type X and some $x : Rec$. It suffices to give a term of type X that is definitionally 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* that is definitionally equal to y. By the definition 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 15, *recRoll* implements part 1 of the proof of Theorem 3.1, 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 $F \cdot Rec$ and must show that $F \cdot Rec$ 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 (*recLB*) 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 $F \cdot Rec$ is F-closed, then use *recLB* to conclude. It is here we see the impredicativity noted earlier: in *recLB*, we instantiate the type argument of $x : Rec$ to the given type X; in *recUnroll*, the given type is $F \cdot Rec$. 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 16. The derivation of recursive types with these operators is summarized axiomatically in Figure 17.

Operations *roll* and *unroll* are implemented 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 are definitionally equal to $\lambda\, x.\, x$. This is show in Figure 16 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 laws.

Theorem. 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 16. □

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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.
```

Figure 16. Operators *roll* and *unroll* (`recType.ced`).

$$\frac{\Gamma \vdash F \overrightarrow{\in} \star \to \star}{\Gamma \vdash Rec \cdot F \overrightarrow{\in} \star}$$

$$\frac{\Gamma \vdash F \overrightarrow{\in} \star \to \star \quad \Gamma \vdash t \overleftarrow{\in} Mono \cdot F}{\Gamma \vdash roll \cdot F\ {-}t \overrightarrow{\in} F \cdot (Rec \cdot F) \to Rec \cdot F} \qquad \frac{\Gamma \vdash F \overrightarrow{\in} \star \to \star \quad \Gamma \vdash t \overleftarrow{\in} Mono \cdot F}{\Gamma \vdash unroll \cdot F\ {-}t \overrightarrow{\in} Rec \cdot F \to F \cdot (Rec \cdot F)}$$

$$\begin{aligned} |roll| &= (\lambda x.x)\ \lambda x.x \\ |unroll| &= (\lambda x.x)\ \lambda x.x \end{aligned}$$

Figure 17. 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, $Rec \cdot F$ 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 equation $roll\ (unroll\ t) = t$ holds by the operational semantics (however, the equation $unroll\ (roll\ t) = t$ should, unless one is satisfied with Church encodings). Instead, we view *Rec* as 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, $inD : F \cdot D \to D$, which we can understand as building 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 $zero : Nat$ and $suc : Nat \to Nat$ can be merged together into a single constructor $inNat : (1 + Nat) \to Nat$.

What separates our derived monotone recursive types (which also constructs a least 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.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

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

Figure 18. Generic iteration scheme.

$$\frac{\Gamma \vdash T \overset{\rightarrow}{\in} \star \quad \Gamma \vdash t_1 \overset{\leftarrow}{\in} T \quad \Gamma \vdash t_2 \overset{\leftarrow}{\in} T \to T}{\Gamma \vdash \mathit{foldNat} \cdot T\ t_1\ t_2 \overset{\rightarrow}{\in} \mathit{Nat} \to T}$$

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

Figure 19. Iteration scheme for *Nat*.

- The ordering corresponds to *morphisms*. Here, this means working with functions $S \to T$, not type inclusions $\mathit{Cast} \cdot S \cdot T$. 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 *functoriality*. 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 $F \cdot S \to F \cdot T$. This lifting must respect 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 18.

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 calls on the subdata of t' (accessed using *fmap*) then using t to compute a result from this. 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 19.

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 the computation laws for these schemes. Read Figure 18 as a collection of constraints with unknowns *D*, *foldD*, and *inD*. From these constraints, we can calculate an encoding of D in System F^{ω} that is a variant of the Church encoding. This is shown in Figure 20. For comparison, the figure also shows the familiar Church encoding of naturals, which we may similarly read from the iteration scheme for *Nat*.

For the typing law, 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 provide the given $a : F \cdot X \to X$ as an argument to the encoding. Finally, we use the right-hand side of the computation law to

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

$$
\begin{aligned}
D &= \forall X : \star . (F \cdot X \to X) \to X \\
foldD &= \Lambda X . \lambda a . \lambda x . x \cdot X\ a \\
inD &= \lambda x . \Lambda X . \lambda a . a\ (fmap \cdot D \cdot X\ (foldD\ \cdot X\ a)\ x)
\end{aligned}
$$

$$
\begin{aligned}
Nat &= \forall X : \star . X \to (X \to X) \to X \\
foldNat &= \Lambda X . \lambda z . \lambda s . \lambda n . n \cdot X\ z\ s \\
zero &= \Lambda X . \lambda z . \lambda s . z \\
suc &= \lambda n . \Lambda X . \lambda z . \lambda s . s\ (foldNat \cdot X\ z\ s\ n)
\end{aligned}
$$

Figure 20. Church encoding of *D* and *Nat*.

$$
\overline{\Gamma \vdash outD : D \to F \cdot D} \qquad |outD\ (inD\ t)| \rightsquigarrow |t|
$$

Figure 21. Generic destructor.

give a definition for constructor *inD*, and we can confirm that *inD* has type $F \cdot D \to D$. Thus, the Church encoding of *D* arises as a direct solution to the iteration scheme for *D* 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

$$
\begin{aligned}
&|foldD\ t\ (inD\ t')| \rightsquigarrow (\lambda x . x\ |t|)\ |inD\ t'| \rightsquigarrow |inD\ t'\ t| \\
&\rightsquigarrow (\lambda a . a\ |(fmap\ (foldD\ a)\ t')|)\ |t| \rightsquigarrow |t\ (fmap\ (foldD\ t)\ t')|
\end{aligned}
$$

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

The issue of inefficiency in computing predecessor for Church naturals has an analog for an arbitrary datatype *D* that only supports iteration. The destructor for datatype *D* is an operator *outD* satisfying the typing and computation law of Figure 21.

With *foldD*, we can define a candidate for the destructor that satisfies the desired typing as $outD = foldD\ (fmap\ inD)$. 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 a thorough characterization of our Scott and Parigot encodings. Specifically, in addition to the iteration scheme and destructor discussed above, we will later introduce the *case distinction* and *primitive recursion* schemes to explain the definitions of the Scott and Parigot encodings, respectively. Then, for each combination of encoding and scheme, we consider two properties: the efficiency (under call-by-name and call-by-value semantics) with which we can simulate the computation law of the scheme with the encoding, and the provability of the *extensionality* law for that scheme using the derived induction principle for the encoding.

We now detail the criteria we shall use, and the corresponding Cedille definitions, for the iteration scheme and destructor. This begins with Figure 22, 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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

Alg ◁ ⋆ → ⋆ = λ X: ⋆. F ·X → X .

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

Figure 22. Iteration typing (`data-char/itertyping.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 foldD: Iter ·D.
  ∀ X: ⋆. ∀ a: Alg ·X. AlgHom ·X a (foldD a) .

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

Figure 23. Iteration characterization (`data-char/iter.ced`).

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.

Iteration scheme. Figure 23 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 $a : Alg \cdot X$, we have that $foldD\ a\ (inD\ x)$ 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 (under both call-by-name and call-by-value operational semantics) by the implementation we give for that scheme.

The extensionality law, given by *IterEta*, expresses the property that a candidate for 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.

Destructor. In Figure 24, *Destructor* gives the type of the generic data destructor, and *Lambek1* and *Lambek2* together state that the property that candidate 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* (the computation law) is efficiently simulated by our solution for the encoding's destructor 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. Proofs of *Lambek2* (the extensionality law) will be given by induction.

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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 } .
```

Figure 24. Laws for the datatype destructor (`data-char/destruct.ced`).

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

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

Figure 25. Typing and computation laws for case distinction on *Nat*.

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 toward a negative result)[2]. Furthermore, it is not obvious how to define recursive functions over Scott encodings 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.

The Scott encoding can be seen as a solution to the case distinction scheme in polymorphic lambda calculi with recursive types. We give the typing and computation laws for this scheme for *Nat* in Figure 25. 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.

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

$$pred = caseNat \cdot Nat\ zero\ \lambda\ x.\ x$$

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

$$\begin{aligned} |pred\ zero| &\rightsquigarrow |zero| \\ |pred\ (suc\ n)| &\rightsquigarrow |(\lambda\ x.\ x)\ n| \rightsquigarrow |n| \end{aligned}$$

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 26. For the definition of *Nat*, the premises of the typing law mention *Nat* itself, so a direct solution requires some form of recursive types. For readability, the solution in Figure 26 uses iso-recursive types – so $|unroll(roll(t))| \rightsquigarrow |t|$ for all t. It is then a mechanical exercise to confirm that the computation laws are efficiently simulated by these definitions.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

Figure 26. Scott naturals in System F with iso-recursive types.

```
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 = <..>
```

Figure 27. Scott naturals (part 1) (`scott/ concrete/nat.ced`).

```
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 .
```

Figure 28. Scott naturals (part 2) (`scott/concrete/nat.ced`).

5.1 Scott-encoded naturals, concretely

Our construction of Scott-encoded naturals supporting weak induction consists of three stages. In Figure 27, we give the definition of the noninductive datatype signature *NatF* with its constructors. In Figure 28, 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 29, the type *Nat* is given using recursive types and the desired weak induction principle for *Nat* is derived.

Signature *NatF*. In Figure 27, 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 erasures $\lambda z.\, \lambda s.\, z$ and $\lambda n.\, \lambda z.\, \lambda s.\, s\ n$ – these are the constructors for Scott naturals in untyped lambda calculus. The proof that *NatF* is monotone is omitted, indicated by `<..>` in the figure (we detailed the proof in Section 3.3.2).

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 n: NatF ·Nat.
  ∀ v: View ·Nat β{ n }. P (elimView β{ n } -v) .

wkIndNat ◁ ∀ P: Nat → ⋆. P zero → (Π m: Nat. P (suc m)) → 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) .
```

Figure 29. Scott naturals (part 3) (`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 28, 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 that elements of the datatype satisfy the datatype's induction principle, is an essential part of the recipe of Stump (2018*a*) for deriving inductive types in CDLE.

Nat, **the type of Scott naturals.** Figure 29 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 $WkIndNatF \cdot N$ (since *WkIndNatF* is not an equational constraint, this restriction is not trivialized by the Kleene trick). 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 specializing the rolling and unrolling operators to *NatFI* we obtain *rollNat* and *unrollNat*. The operators, along with the facts that $|zeroF| =_{\beta\eta} |zeroWkIndNatF|$ and $|sucF| =_{\beta\eta} |sucWkIndNatF|$, are then used to define the constructors *zero* and *suc*. From the fact that $|rollNat| =_{\beta\eta} \lambda\, x.\, x$, 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*.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

Weak induction for *Nat*. Weak induction for *Nat*, given by *wkIndNat* in Figure 29, allows us to prove $P\ n$ for arbitrary $n : Nat$ and $P : Nat \to \star$ if we can show that P holds of *zero* and that for arbitrary m we can construct a proof of $P\ (suc\ m)$. However, there is a gap between this proof principle and the proof principle $WkIndNatF \cdot Nat$ associated to n – the latter allows us to prove properties over terms of type $NatF \cdot Nat$, not terms of type *Nat*! To bridge this gap, we introduce a predicate transformer *LiftNat* that takes properties of kind $Nat \to \star$ to properties of kind $NatF \cdot Nat \to \star$. For any $n : NatF \cdot Nat$, the new property $LiftNat \cdot P\ n$ states that P holds for n if we have a way of viewing n at type *Nat* (*View*, Figure 8).

The key to the proof of weak induction for *Nat* is that by using *View*, the retyping operation of terms of type $NatF \cdot Nat$ to the type *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 $WkIndNatF \cdot Nat\ (unroll\ n).1$. Instantiating the predicate argument of this with $LiftNat \cdot P$ gives us three subgoals:

- $LiftNat \cdot P\ (zeroF \cdot Nat)$
  Assuming $v : View \cdot Nat\ \beta\{zeroF\}$, we wish to give a proof of $P\ (elimView\ \beta\{zeroF\}\ \text{-}v)$. This is convertible with the type $P\ zero$ of z, since $|elimView\ \beta\{zeroF\}\ \text{-}v| =_{\beta\eta} |zero|$.
- $\Pi\ m : Nat.\ LiftNat \cdot P\ (sucF\ m)$
  Assume we have such an m, and that v is a proof of $View \cdot Nat\ \beta\{sucF\ m\}$. We are expected to give a proof of $P\ (elimView\ \beta\{sucF\ m.1\}\ \text{-}v)$. The expression $s\ m$ has type $P\ (suc\ m)$, which is convertible with that expected type.
- $View \cdot Nat\ \beta\{(unrollNat\ n).1\}$
  This holds by *selfView n* of type $View \cdot Nat\ \beta\{n\}$, since $|\beta\{n\}| =_{\beta\eta} |\beta\{(unrollNat\ n).1\}|$.

The whole expression synthesizes type:

$$P\ (elimView\ \beta\{(unrollNat\ n).1\}\ \text{-}(selfView\ n))$$

which is convertible with the expected type $P\ n$.

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 case distinction. 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 distinction satisfies the corresponding *extensionality* law, that is, 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 30, along with predecessor *pred* (defined using *caseNat*) and proofs for both that they satisfy the desired computation laws (or β-laws) by definition. As both *rollNat* and *unrollNat* erase to $(\lambda\, x.\, x)\ (\lambda\, x.\, x)$ (Figure 17), our overhead in simulating case distinction is only a constant number of reductions. With an efficient operation for case distinction, we obtain an efficient predecessor *pred*.

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

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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 .
```

Figure 30. Computation laws for case distinction and predecessor (`scott/concrete/nat.ced`).

```
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) .
```

Figure 31. Extensional laws for case distinction (`scott/concrete/nat.ced`).

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 that which underlies the case distinction operator *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.

Extensional laws. Using weak induction, we can prove the extensionality law (or η-law) of the case distinction scheme. This is *caseNatEta* in Figure 31. 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 and is elaborated on in Section 6.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

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

Figure 32. Generic case distinction scheme.

$$\begin{aligned} D &= \mu\, D.\, \forall\, X : \star.\, (F \cdot D \to X) \to X \\ caseD &= \Lambda\, X.\, \lambda\, a.\, \lambda\, x.\, unroll(x) \cdot X\ a \\ inD &= \lambda\, x.\, roll(\Lambda\, X.\, \lambda\, a.\, a\ x) \end{aligned}$$

Figure 33. Generic Scott encoding of D in System F^{ω} with iso-recursive types.

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

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

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

Figure 34. Case distinction typing (`data-char/case-typing.ced`).

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 $inD : F \cdot D \to D$. Datatype D satisfies the case distinction scheme if there exists an operator *caseD* satisfying the typing and computation laws listed in Figure 32.

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, for case distinction 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 laws, we can define $outD : D \to F \cdot D$, the generic destructor that reveals the F-collection of predecessors used to construct a term of type D:

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

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

In System F^{ω} extended with iso-recursive types, we can read an encoding directly from these laws, resulting in the solutions for D, *caseD*, and *inD* given in Figure 33. This is the generic Scott encoding and is the basis for the developments in Section 5.2.2.

5.2.1 Characterization criteria

We formalize in Cedille the above description of the generic case distinction scheme in Figures 34 and 35. Definitions for the typing law of case distinction for datatype D are given in Figure 34, where the module is parametrized by a type scheme F that 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.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
import data-char/iter-typing .

module data-char/case
  (F: ⋆ → ⋆) (D: ⋆) (inD: Alg ·F ·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 caseD: Case ·D.
  ∀ X: ⋆. ∀ a: AlgCase ·D ·X. AlgCaseHom ·X a (caseD a) .

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

Figure 35. Case distinction characterization (`data-char/case.ced`).

In Figure 35, we take the signature F as well as the datatype D and its constructor *inD* as parameters. We import the definitions of Figure 22 without specifying that module's type scheme parameter, so in the type of *inD*, $Alg \cdot F \cdot D$, we give this explicitly as F. For a candidate $caseD : Case \cdot D$ for the operator for case distinction, the property *CaseBeta caseD* states that it satisfies the desired computation law, where the shape of the computation law is given by *AlgCaseHom*. *CaseEta caseD* is the property that *caseD* satisfies the extensionality law, that is, 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 *caseD a*.

5.2.2 Generic Scott encoding

We now detail the generic derivation of Scott-encoded data supporting weak induction. 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 36, we give the unrefined signature *DF* for Scott-encoded data and its constructor. In Figure 37, we define the predicate *WkIndDF* expressing that *DF* terms satisfy a certain weak induction principle and show that the constructor satisfies this predicate. In Figure 38, we take the fixpoint of the refined signature, defining the datatype *D*, and prove weak induction for *D*.

Signature *DF***.** In Figure 36, *DF* is the type scheme whose fixpoint is the solution to the equation for *D* in Figure 33. Definition *inDF* is the polymorphic constructor for signature *DF*, that is, the generic grouping together of the collection of constructors for the datatype signature (e.g., *zeroF* and *sucF*, Figure 27). Finally, *monoDF* is a proof that type scheme *DF* is monotonic (definition omitted, indicated by <. . .>).

Predicate *WkIndDF***.** In Figure 37, 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 $WkPrfAlg \cdot D \cdot P$, 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*.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 = <..>
```

Figure 36. Generic Scott encoding (part 1) (`scott/generic/encoding.ced`).

```
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 .
```

Figure 37. Generic Scott encoding (part 2) (`scott/generic/encoding.ced`).

In the concrete derivation of Scott naturals, the predicate *WkIndNatF* (Figure 28) required terms of the following types be given in proofs by weak induction:

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

We can understand *WkPrfAlg* as combing these types together into a single type, parametrized by the signature F:

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

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

The Scott-encoded datatype D**.** With the inductivity predicate *WkIndDF* and weak proof algebra *inWkIndDF* for it, we are now able to form a signature refining *DF* such that its fixpoint supports weak induction (proof by cases). Observe that the proof *inWkIndDF* in Figure 37 is definitionally equal to *inDF*:

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

This effectively allows us to use dependent intersections to form, for all D, the subset of the type $DF \cdot D$ whose elements satisfy $WkIndDF \cdot D$. This is *DFI* in Figure 38.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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) .
```

Figure 38. Generic Scott encoding (part 3) (`scott/generic/encoding.ced`).

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 *unrollD* that are definitionally equal to $\lambda\, x.\, x$. The constructor *inD* for D takes an F-collection of D predecessors *xs* and constructs 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 a property P holds for term $x : D$ 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 $WkIndDF \cdot D\ (unroll\ x).1$, given by $(unroll\ x).2$, 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 $|inDF| =_{\beta\eta} |inD|$. The whole expression, then, has type:

$$P\ (elimView\ \beta\{(unrollD\ x).1\}\ \text{-}(selfView\ x))$$

which is convertible with the expected type $P\ x$.

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 implementations of the case distinction scheme and destructor that both efficiently simulate their computation laws and provably satisfy their extensionality laws. Iteration and primitive recursion are treated in Section 7.3.1.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 .
```

Figure 39. Characterization of *caseD* (`scott/generic/props.ced`).

Normalization guarantee. Recall that Proposition 3 guarantees call-by-name normalization for closed Cedille terms whose types can be included into some function type. The proof *normD* of Figure 39 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 35). For our solution *caseD* in Figure 39, *caseDBeta* proves it satisfies the computation law and *caseDEta* proves it satisfies the extensionality 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 17), 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 *caseDEta*, we proceed by weak induction and 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 40, we give the definition we proposed earlier 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 implementation of case distinction, 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.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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) .
```

Figure 40. Characterization of *outD* (`scott/generic/props.ced`).

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

$$|recNat \cdot T\ t_1\ t_2\ zero| \quad \rightsquigarrow |t_1|$$
$$|recNat \cdot T\ t_1\ t_2\ (suc\ n)| \rightsquigarrow |t_2\ n\ (recNat \cdot T\ t_1\ t_2\ n)|$$

Figure 41. Typing and computation laws for primitive recursion on *Nat*.

6. Parigot Encoding

In this section, we derive Parigot-encoded data with (non-weak) induction, illustrating with a concrete example in Section 6.1 the 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). It is a combination of the Church and Scott encoding: it directly supports recursion with access to previously computed results as well as to predecessors. For the inductive versions we derive in this section, this means that unlike the weakly 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 size of the encoding of natural number n is exponential in n.

The Parigot encoding can be seen as a direct 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 41. The significant feature of 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.

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 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.

In System F with iso-recursive types, 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 42.

Parigot naturals and canonicity. In addition to showing yet another application of derived recursive types in Cedille, this section serves two pedagogical purposes. First, the Parigot encoding more readily supports primitive recursion than 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 untyped lambda calculus in Cedille.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

Figure 42. Parigot naturals in System F with iso-recursive types.

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 above definition for 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 *Nat* 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 extensionality 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 only the canonical Parigot naturals. This characterization is the *reflection law*. As we will see by proving induction, the set of canonical Parigot naturals is precisely the set of terms *n* of type *Nat* satisfying the following equality:

$$\{recNat\ zero\ (\lambda\, m.\, suc)\ n \simeq n\}$$

As an example, the noncanonical Parigot natural $suc'\ (suc\ zero)$ does *not* satisfy this criterion: rebuilding it with the constructors *zero* and *suc* produces $suc\ (suc\ zero)$ (see also Ghani et al. (2012, Section 2), where it is shown that the reflection law together with dependent products guarantees induction for naturals).

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!

6.1 Parigot-encoded naturals, concretely

We split the derivation of inductive Parigot naturals into three parts. In Figure 43, we define untyped operations for Parigot naturals and prove that the untyped constructors preserve the reflection law. In Figure 44, we define the type *Nat* of canonical Parigot naturals and its constructors. Finally, in Figure 45, 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 43 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 42). The term *reflectNatU* 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 left-hand side is convertible with:

$$sucU\ (reflectNatU\ n)$$

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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 } .
```

Figure 43. Parigot naturals (part 1) (`parigot/concrete/nat.ced`).

```
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 ] .
```

Figure 44. Parigot naturals (part 2) (`parigot/concrete/nat.ced`).

Note that in addition to using the Kleene trick (Section 2.3) to have a type *Top* for terms of the untyped lambda calculus, 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 44, we first define the type scheme *NatF′* whose fixpoint over-approximates the type of Parigot naturals. Using dependent intersection, we then define

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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) .
```

Figure 45. Parigot naturals (part 3) (`parigot/concrete/nat.ced`).

the type scheme *NatF* to map 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 17).

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 type $NatF' \cdot Nat$ 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 proves

$$\{reflectNatU\ (sucU\ \beta\{n\}) \simeq sucU\ \beta\{n\}\}$$

by invoking the proof *sucC* with

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

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

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

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

***NatI*, the type of inductive Parigot naturals.** The definition of the inductive subset of Parigot naturals begins in Figure 45 with the predicate *IndNat* over *Nat*. For all $n : Nat$, *IndNat n* is the type of proofs 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 $P\ m$ as an inductive hypothesis. The type *NatI* is then defined with dependent intersection as the subset of *Nat* for which the predicate *IndNat* holds.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 .
```

Figure 46. Computation laws for primitive recursion and predecessor (`parigot/concrete/nat.ced`).

Definition *recNatI* brings us close to the derivation of an induction principle for *Nat*, but does not quite achieve it. With an inductive proof of 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 with proofs *indZero* and *indSuc* stating resp. that *zero* satisfies *IndNat* and, for every *n* in the inductive subset *NatI*, *suc n*.1 satisfies *IndNat*. We see that $|indZero| =_{\beta\eta} |zero|$, and also that $|indSuc| =_{\beta\eta} |suc|$ since $|recNatI| =_{\beta\eta} |recNat|$. Thus, 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*, that is, 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 $|reflectNatI| =_{\beta\eta} |reflectNatU|$, so we define a cast *toNatI* where the given proof:

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

has a type convertible with the expected type $\{reflectNatI\ n \simeq n\}$. So here we are using the full power of *intrCast* by exhibiting a type inclusion using a function that behaves extensionally like identity but is *not* intensionally the identity function. The proof *indNat* of the induction principle for Parigot naturals follows from *recNatI* and the use of *toNatI* to convert the given $n : Nat$ to the type *NatI*.

6.1.1 Computational and extensional character

We now give a characterization of *Nat*. From the code listing in Figure 44, it is clear *recNat* satisfies the typing law. Figure 46 shows the proofs of the computation laws (*recNatBeta1* and *recNatBeta2*), which hold by definitional equality. By inspecting the definitions of *recNat*, *zero*, and *suc*, and from the erasures of *roll* and *unroll*, we can confirm that the computation laws are simulated in a constant number of reduction steps under both call-by-name and call-by-value semantics.

Figure 46 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 do not obtain an efficient solution for the predecessor function under call-by-value operational semantics.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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) .
```

Figure 47. 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}.
```

Figure 48. Functors (`functor.ced`).

Finally, in Figure 47, 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 use the inductive hypothesis, which is unavailable with 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 our formulation 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 48. Analogous to monotonicity, functoriality 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 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 49, that establishes that every functor is a monotone type scheme (see Figure 13 for the definition of *Mono*).

Dependent pair types. Figure 50 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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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).
```

Figure 49. Functors and monotonicity (`functorThms`).

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

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

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

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

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

Figure 50. *Sigma*, axiomatically (`utils/sigma.ced`).

```
Pair ◁ ⋆ → ⋆ → ⋆
= λ S: ⋆. λ T: ⋆. Sigma ·S ·(λ _: S. T).

fork ◁ ∀ X: ⋆. ∀ S: ⋆. ∀ T: ⋆. (X → S) → (X → T) → X → Pair ·S ·T
= Λ X. Λ S. Λ T. λ f. λ g. λ x. mksigma (f x) (g x) .
```

Figure 51. *Pair* (`utils/sigma.ced`).

rules, we confirm that the projection functions and induction principle compute as expected over pairs formed from the constructor. In Figure 51, the type *Pair* is defined in terms of *Sigma* as the special case where the type of the second component does not depend on the first component. Additionally, the figure also gives the utility function *fork* for constructing nondependent pairs to help express the computation law of the primitive recursion scheme.

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 $fmap : Fmap \cdot F$ that satisfies the functor identity and composition laws.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

$$\frac{\Gamma \vdash T \overrightarrow{\in} \star \quad \Gamma \vdash t \overleftarrow{\in} F \cdot (Pair \cdot D \cdot T) \to T}{\Gamma \vdash recD \cdot T\ t \overrightarrow{\in} D \to T}$$

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

Figure 52. Generic primitive recursion scheme.

$$\begin{aligned} D &= \mu\, D.\, \forall\, X : \star.\, (F \cdot (Pair \cdot D \cdot X) \to X) \to X \\ recD &= \Lambda X.\, \lambda\, a.\, \lambda\, x.\, unroll(x) \cdot X\ a \\ inD &= \lambda\, x.\, roll(\Lambda X.\, \lambda\, a.\, a\ (fmap\ (fork\ (id \cdot D)\ (recD\ a))\ x)) \end{aligned}$$

Figure 53. Generic Parigot encoding of D in System F^{ω} with iso-recursive types.

The typing and computation laws for the generic primitive recursion scheme for datatype D with signature F and constructor inD are given in Figure 52. 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 affirmed by the computation law, which states that the action of $recD\ t$ over values built from $inD\ t'$ (for some $t' : F \cdot D$) is to apply t to the result of tupling each predecessor (accessed with $fmap$) with the result of $recD\ t$ (here id is the polymorphic identity function). Assuming t and $inD\ t'$ are typed according to the typing law, the right-hand side of the equation has type T with the following assignment of types to subexpressions:

- $id : D \to D$
- $fork\ id\ (recD\ t) : D \to Pair \cdot D \cdot T$
- $fmap\ (fork\ id\ (recD\ t)) : F \cdot D \to F \cdot (Pair \cdot D \cdot T)$

With primitive recursion, we can implement the datatype destructor $outD$:

$$outD = recD\ (fmap\ proj1)$$

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

$$|fmap\ proj1\ (fmap\ (fork\ id\ outD)\ t)|$$

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 canonical Parigot encodings in Figure 53. Similar to the case of the Scott encoding, we find that to give the definition of D we need some form of recursive types since D occurs in the premises of the typing law. For our derivation, we must further refine the type of D so that we only include canonical Parigot encodings (i.e., those built only from inD). We take the same approach used for Parigot naturals: *Top* and the Kleene trick help us express satisfaction of the reflection law for untyped terms, which is in turn used 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 54 and 55. Definitions for the typing law of the primitive recursion scheme are given in Figure 54, where parameter F gives the datatype signature. Type family *AlgRec* gives the shape

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 .
```

Figure 54. Primitive recursion typing (`data-char/primrectyping.ced`).

```
import functor .
import utils .

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

module data-char/primrec
  (F: ⋆ → ⋆) (fmap: Fmap ·F) (D: ⋆) (inD: Alg ·F ·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)) .

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

Figure 55. Primitive recursion characterization (`data-char/primrec.ced`).

of the type functions used for primitive recursion, and *PrimRec* gives the shape of the type of operator *recD* itself.

In Figure 55, we now assume that F has an operation *fmap* for lifting functions, and we take additional module parameters D for the datatype and *inD* for its constructor. We import the definitions of the modules defined in Figures 23 and 35 without specifying the type scheme module parameter, so it is given explicitly for definitions exported from these modules (e.g., $Alg \cdot F \cdot D$). *AlgRecHom* gives the shape of the computation law for primitive recursion with respect to a particular $a : AlgRec \cdot D \cdot X$, *PrimRecBeta* is a predicate on candidates for the combinator for primitive recursion stating that they satisfy the computation law with respect to *all* such functions a, and *PrimRecEta* is a predicate stating that a candidate is the unique such solution up to function extensionality.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 } .
```

Figure 56. Generic Parigot encoding (part 1) (`parigot/generic/encoding.ced`).

The figure also lists *PrfAlgRec*, a dependent version of *AlgRec*. Read the type $PrfAlgRec \cdot P$ as the type of "(F, D)-proof algebras for P." It is the type of proofs that take an F-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 simulate both the iteration and case distinction schemes, the figure lists the helper functions *fromAlgCase* and *fromAlg*. Definition *fromAlgCase* converts a function for use in case distinction to one for use in primitive recursion 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 56, we give the computational characterization of canonical Parigot encodings as a predicate on untyped terms, then prove that the untyped constructor preserves this predicate; in Figure 57, we define the type of Parigot encodings, its primitive recursion combinator, and its constructors; in Figure 58, we define the inductive subset of Parigot encodings and its constructor; finally, in Figure 59, 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 56 are untyped versions of resp. the combinator for primitive recursion, the generic constructor, and the operation that builds

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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) ] .
```

Figure 57. Generic Parigot encoding (part 2) (`parigot/generic/encoding.ced`).

canonical Parigot encodings by recursively rebuilding the encoding with the constructor *inU* (compare to Figure 43 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 yet 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:

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

We rewrite by the functor composition law to fuse the mapping of *proj2* with that of *fork id reflectU*, and now the left-hand side of the resulting 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 the Kleene trick, so that the proof *inC* is definitionally equal to the untyped constructor *inU*. This allows us to use dependent intersection to form the refinement needed to type only canonical Parigot encodings.

Parigot encoding *D*. In Figure 57, DF' is the type scheme whose least fixpoint is the solution to *D* in Figure 53, 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 (see Figure 54 for the definition of *PrimRec*).

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 similar to that which we obtained in Figure 53. Then, with the auxiliary proof *toDC* that *D* is included into the type of untyped terms satisfying *DC*, we define the constructor *inD* for *D* using the rolling operation and dependent intersection

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 ] .
```

Figure 58. Generic Parigot encoding (part 3) (`parigot/generic/encoding.ced`).

```
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) .
```

Figure 59. Generic Parigot encoding (part 4) (`parigot/generic/encoding/ced`).

introduction. The definition is accepted by virtue of the following definitional equalities:

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

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 58, we give the definition of *DI*, the type for the inductive subset of Parigot-encoded data, and its constructor *inDI*. This definition begins by bringing *PrfAlgRec* into scope with an import, used to define the predicate *IndD*. For arbitrary $x : D$, the property *IndD x* states that, for all $P : D \to \star$, 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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

this subset and corresponds to the definition *recNatI* for Parigot naturals in Figure 45. With *recDI*, we can obtain the desired induction scheme for D if we 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 $P\ (inDI'\ xs)$ and the type of the expression in the final line, which is

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

The main idea here is that we first use the functor composition law to fuse the two lifted operations, then observe that this results in lifting a single function:

$$\lambda\, x.\, proj1\ (mksigma\ x.1\ (recDI\ a\ x)) : DI \to D$$

that is definitionally equal to the identity function (by the computation law for *proj1*, Figure 50, and the erasure of dependent intersection projections). We then use the functor identity law, exchanging *inD* for $inDI'$ in the rewritten type (these two terms are definitionally equal). As $inDI'$ and $indInDI'$ are definitionally equal to each other (since $|fork\ id\ (recD\ a)| =_{\beta\eta} |\lambda\, x.\, mksigma\ x.1\ (recDI\ a\ x)|$), 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, giving us induction, by using the fact that terms of type D are *canonical* Parigot encodings. This is shown in Figure 59. 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 $|reflectDI| =_{\beta\eta} |reflectU|$, 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 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 to confirm that *recD* is an efficient and unique solution to the primitive recursion scheme for D, 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 60, *normD* establishes the inclusion of type D into the function type $AlgRec \cdot D \cdot D \to D$. By Proposition 3, this guarantees call-by-name normalization of closed terms of type D.

Primitive recursion scheme With proof *recDBeta*, we have that our solution *recD* (Figure 57) satisfies the computation law by definitional equality. By inspecting the definitions of *recD* and *inD*, and the erasures of *roll* and *unroll* (Figure 17), 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 to function extensionality. The proof follows from induction, the functor laws, and a nonobvious use of the Kleene trick. 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.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 = <..>
```

Figure 60. Characterization of *recD* (`parigot/generic/props.ced`).

```
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 = <..>
```

Figure 61. Characterization of *caseD* (`parigot/generic/props.ced`).

Case distinction scheme. In Figure 61, we define the candidate *caseD* for the operation giving case distinction for *D* using *recD* and *fromAlgCase* (Figure 55). This definition satisfies the computation law only up to the functor laws. With definitional equality alone, *caseD a* (*inD xs*) is joinable with

$$a\ (fmap\ proj1\ (fmap\ (fork\ id\ (caseD\ a))\ xs))$$

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 *caseD a* to be needlessly computed for all predecessors. The proof of extensionality, *caseDEta*, follows from *recDEta*.

Destructor. In Figure 62, 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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 = <..>
```

Figure 62. 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 = <..>
```

Figure 63. Characterization of *foldD* (`parigot/generic/props.ced`).

laws and is not efficient under call-by-name semantics. The extensionality law, *lambek2D*, holds by induction.

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 (Figures 22 and 23). The proposed solution is *foldD*, given in Figure 63. 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 (*fromAlg*, defined in Figure 55, introduces the additional traversal).

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, also called finitely branching trees, are a datatype in which subtrees are contained within a list, meaning that nodes may have an arbitrary number of children. In Haskell, they are defined as:

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

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

The difficulty lies in giving users access to the inductive hypothesis for the subtrees contained within the list. To work around this, users of Coq or Agda can define a mutually inductive definition, forgoing reuse for the special purpose "list of rose trees" datatype, or prove the desired

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
import utils.

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

import signatures/list ·A .
import parigot/generic/encoding as R
  ·ListF listFmap -listFmapId -listFmapCompose .

List ◁ ⋆ = R.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) .
```

Figure 64. Lists (`parigot/examples/list-data.ced`).

induction principle manually for the definition that uses the standard list type. In this section, we use the induction principle derived for our generic Parigot encoding to take this second approach, proving an induction principle for rose trees in the style expected of a mutual inductive definition while re-using the list datatype. The presentation of a higher-level language, based on such a generic encoding, in which this induction principle could be automatically derived is a matter for future work.

Lists. Figure 64 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 "*R*", and the definitions within that module are accessed with the prefix "*R.*", for example, "*R.D*" for the datatype. 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*) (see `signatures/list.ced` in the code repository). In the figure, these are given as module arguments to the generic derivation. We also define the primitive recursion principle *recList* for lists as a nondependent use of induction.

In Figure 65, 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 66 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 that the lifting operation *treeFmap* respects identity and composition make use of the corresponding proofs for the given *F*.

Rose trees. In Figure 67, 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 *R.inD* and give a variant constructor *rose′* which we use in the definition of the induction principle for rose trees. This variant uses *treeFmap* to remove the tupled proofs that an inductive hypothesis holds for subtrees, introduced in the generic induction principle.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 = <..>
```

Figure 65. 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 = <..>
```

Figure 66. Signature for *F*-branching trees (`signatures/tree.ced`).

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 maintained for collections of subtrees. We require that:

- Q holds for *nil*, bound as n;
- if P holds for t and Q holds for ts then Q holds for $cons\ t\ ts$, bound as c; and that
- P holds for $rose\ x\ ts$ when Q holds for ts, bound as r.

In the body of *indRoseTree*, we use the induction principle *R.indD* for the generic Parigot encoding, then *indsigma* (Figure 50, Section 6.2), 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.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 R
  ·TreeF treeFmap -treeFmapId -treeFmapCompose .

RoseTree ◁ ⋆ = R.D .

rose ◁ A → List ·RoseTree → RoseTree
= λ x. λ t. R.inD (mksigma x t) .

rose' ◁ ∀ P: RoseTree → ⋆. TreeF ·(Sigma ·RoseTree ·P) → RoseTree
= Λ P. λ xs. R.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.
  R.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)) .
```

Figure 67. Rose trees (`parigot/examples/rosetree-data.ced`).

Amazingly, in some settings, this deficit of the Scott encoding is *only* apparent. Working with 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 which utilizes "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 an inclusion of the type of Scott encodings into the type of Lepigre–Raffalli encodings by using weak induction together with the fact that our derived recursive types are least fixpoints.

Lepigre–Raffalli recursion. 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 25) 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, $|n| =_{\beta\eta} |\lambda\, z.\, \lambda\, s.\, caseNat\ z\ s\ n|$:

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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

$$NatZ \cdot T = \forall Z : \star. \forall S : \star. Z \to S \to T$$
$$NatS \cdot T = \forall Z : \star. \forall S : \star. (Z \to S \to Z \to S \to T) \to Z \to S \to T$$

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

Figure 68. Typing law for the Lepigre–Raffalli recursion scheme on *Nat*.

Viewing the computation laws this way, we see in the *suc* case 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, that is, 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:

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

To give types to the terms involved in these equations, observe that we can use impredicative quantification to address the self-application occurring in the right-hand sides. 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:

$$t_1 : \forall Z : \star. \forall S : \star. Z \to S \to T$$
$$t_2 : \forall Z : \star. \forall S : \star. \ ? \to Z \to S \to T$$

This gives an interpretation of t_1 as a constant function that ignores its two arguments and returns a result of type T. For t_2, "?" holds the place of the type of its first argument, $\lambda z.\, \lambda s.\, recLRNat\ z\ s\ n$. We are searching for a type 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:

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

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 68. From this, we obtain the type for Lepigre–Raffalli 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 naturals is a supertype of the type of Scott naturals 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 and derive induction for generic Scott encodings.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 .
```

Figure 69. Primitive recursion for Scott naturals (part 1) (`lepigre-raffalli/concrete/nat1.ced`).

```
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 .
```

Figure 70. Primitive recursion for Scott naturals (part 2) (`lepigre-raffalli/concrete/nat1.ced`).

7.1 Primitive recursion for Scott naturals, concretely

Our derivation of primitive recursion for Scott naturals is split into three parts. In Figure 69, we define the type of Lepigre–Raffalli naturals and the combinator for Lepigre–Raffalli recursion. In Figure 70, we prove that Scott naturals are a subtype of Lepigre–Raffalli naturals, giving us Lepigre–Raffalli recursion over them. Finally, in Figure 71, we implement primitive recursion for Scott naturals using Lepigre–Raffalli recursion.

Lepigre–Raffalli naturals. In Figure 69, we give in Cedille the definition for the type of Lepigre–Raffalli encodings we previously obtained. We use a qualified import of the concrete encoding of Scott naturals (Section 5.1), so to access a definition from that development we use "*S.*" as a prefix (not to be confused with the quantified type variable *S* that appears with no period). 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 30):

Inclusion of Scott naturals into Lepigre–Raffalli naturals. Figure 70 shows that Scott naturals (*S.Nat*) are a subtype of Lepigre–Raffalli naturals. This begins with the constructors *zero* and *suc*,

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 .
```

Figure 71. Primitive recursion for Scott naturals (part 3) (`lepigre-raffalli/concrete/nat1.ced`).

whose definitions come from computation laws we derived for *recLRNat*. In particular, for successor the first argument to the bound s is $\lambda\, z'.\, \lambda\, s'.\, recLRNat\ z'\ s'\ n$, 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 $S.WkIndNatF \cdot Nat\ n.1$,

$$\forall\, P : S.NatF \cdot Nat \to \star.\, P\ (S.zeroF \cdot Nat) \to (\Pi\, m : Nat.\, P\ (S.sucF\ m)) \to P\ n.1$$

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 a lower bound of all such types with respect to type inclusion, using *recLB* (Figure 15) we have a cast from Scott naturals to Lepigre–Raffalli naturals.

Primitive recursion for Scott naturals. The last step in equipping Scott naturals with primitive recursion is to translate this scheme to Lepigre–Raffalli recursion. This is done in three parts, shown in Figure 71. One complication that must be addressed is that Lepigre–Raffalli recursion reinterprets the predecessor as a function for making recursive calls, but primitive recursion enables direct access to the predecessor. So that it may serve both roles, we duplicate the predecessor. This means that if T is the type of results we wish to compute with primitive recursion, then we use Lepigre–Raffalli recursion to compute a function of type $S.Nat \to T$.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{recNat}\ t_1\ t_2\ (\mathit{S.suc}\ n) \\
\rightsquigarrow_\beta^* & \ \mathit{recLRNat}\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{elimCast}\ (\mathit{S.suc}\ n)) \\
& \qquad (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{S.pred}\ (\mathit{S.suc}\ n)) \\
\rightsquigarrow_\beta^* & \ \mathit{S.suc}\ n\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ n \\
\rightsquigarrow_\beta^* & \ \mathit{recNatS}\ t_2\ n\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ n \\
\rightsquigarrow_\beta^* & \ t_2\ n\ (n\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{S.pred}\ n)) \\
\leftsquigarrow_\beta^* & \ t_2\ n\ (\mathit{recLRNat}\ (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{elimCast}\ n) \\
& \qquad (\mathit{recNatZ}\ t_1)\ (\mathit{recNatS}\ t_2)\ (\mathit{S.pred}\ n)) \\
\leftsquigarrow_\beta^* & \ t_2\ n\ (\mathit{recNat}\ t_1\ t_2\ n)
\end{aligned}
$$

Figure 72. Reduction of *recNat* for the successor case.

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

- type variables Z and S,
- $r : \mathit{NatRec} \cdot (\mathit{S.Nat} \to T) \cdot Z \cdot S$, the handle for making recursive calls,
- z and s, the base and step cases at the abstracted types Z and S, and
- $m : \mathit{S.Nat}$, which we intend to be a duplicate of r.

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.

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 for the step case for arbitrary untyped terms t_1, t_2, and n in Figure 72. 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 $|\mathit{recNat}\ t_1\ t_2\ (\mathit{S.suc}\ n)|$ 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 modifications to the Lepigre–Raffalli encoding allowing us to equip Scott naturals with induction. For completeness, we show the full derivation, but as this development is similar to what preceded, we shall only highlight the differences.

In Figure 73, we begin our modification by making *NatRec* dependent: $\mathit{NatRec} \cdot P\ n \cdot Z \cdot S$ 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 type family *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. This duality is echoed in *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 and produce a proof that P holds of x.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 .
```

Figure 73. Induction for Scott naturals (part 1) (`lepigre-raffalli/concrete/nat2.ced`).

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 proving that rebuilding a term of type $S.NatFI \cdot Nat$ with these constructors reproduces the original term at type *Nat*.

Finally, we derive true induction for Scott naturals in Figure 74. Playing the same roles as *recNatZ* and *recNatS* (Figure 71), *indNatZ* and *indNatS* convert the usual base and step cases of an 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).

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 the generic encoding, we start by

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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) .
```

Figure 74. Induction for Scott naturals (part 2) (`lepigre-raffalli/concrete/nat2.ced`).

deriving a Lepigre–Raffalli recursion scheme from the observation that, for the solution *caseD* for the case distinction scheme given in Figure 39, $|t| =_{\beta\eta} |\lambda\, a.\, caseD\ a\ t|$ 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, let $t : AlgCase \cdot D \cdot T$ for some T, and let $t' : F \cdot D$. 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:

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

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:

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

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:

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

Compare this to Lepigre–Raffalli recursion for naturals (Figure 68).

- 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 75. 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 76, we give the type of our generic variant of the Lepigre–Raffalli encoding for a monotone signature. In Figure 77,

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

Figure 75. Typing law for the generic Lepigre–Raffalli recursion scheme.

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

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

import scott/generic/encoding as S ·F -mono .

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 .
```

Figure 76. Generic Lepigre–Raffalli-style induction for Scott encodings (part 1) (`lepigre-raffalli/generic/encoding.ced`).

we derive Lepigre–Raffalli induction for Scott encodings. Finally, in Figure 78, we assume the stronger condition that the datatype signature is a functor and derive a standard induction principle for Scott encodings.

Generic Lepigre–Raffalli encoding. In Figure 76, we begin by importing the generic Scott encoding, using prefix "*S.*" to access definitions in 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 $S.D$ and predicate $P : S.D \to \star$ (compare this to *NatRec* in Figure 73). Next, for all predicates P over Scott encodings $S.D$, $\mathit{PrfAlgLR} \cdot P$ is the type of Lepigre–Raffalli-style proof algebras for P, corresponding to $\mathit{NatZ} \cdot P$ and $\mathit{NatS} \cdot P$ together in Figure 73. A term of this type is polymorphic in a type Y (which we interpret as standing in for $\mathit{PrfAlgLR} \cdot P$ itself) and takes an F-collection xs of terms which, with the use of dependent intersection types, are each interpreted both as a predecessor and a handle for accessing the inductive hypothesis for that predecessor. The argument of type Y is the step case to be given to these handles. 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 $|\mathit{inDRec}| =_{\beta\eta} |S.\mathit{inD}|$.

Type D is our generic Lepigre–Raffalli encoding, again defined with dependent intersection as the type for Scott encodings x that also act as functions that, 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.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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) .
```

Figure 77. Generic Lepigre–Raffalli-style induction for Scott encodings (part 2) (`lepigre-raffalli/generic/encoding.ced`).

Lepigre–Raffalli induction. In Figure 77, 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 with that predecessor ($\lambda\, a.\, recLRD\ a\ x$) for a given predicate *P*; compare this to the definition of *suc* in Figure 73. As expected, we have that *inD* and *S.inD* (and also *S.inDF*) are definitionally equal.

With the constructor defined, we show with *rollD* that *D* is an *S.DFI*-closed type (*S.DFI* is defined in Figure 38) 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 a lower bound of all such types, we thus obtain a proof *toD* of an inclusion of the type *S.D* into *D* using *recLB* (Figure 15). 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 78 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 handles for 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 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

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

We use the functor composition law to fuse the two mappings of *proj1* and *applyDRec y*. Then, observing that this results in the mapping of a function that is definitionally equal $\lambda\, x.\, x$ (by

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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) .
```

Figure 78. Generic induction for Scott encodings (`lepigre-raffalli/generic/induction.ced`).

the computation law of *proj1*, Figure 50), 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 do 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 79 and 80, 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 79 for the combinator for primitive recursion is a nondependent instance of standard induction. 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 steps. We illustrate for arbitrary (untyped) terms t

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

```
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 = <..>
```

Figure 79. Characterization of *recD* (`lepigre-raffalli/generic/props.ced`).

```
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 = <..>
```

Figure 80. Characterization of *foldD* (`lepigre-raffalli/generic/props.ced`).

and t':

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

The extensionality law *recDEta* (proof omitted) follows from induction.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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 80 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 dual-role predecessors, 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 55).

8. Scott Encoding vs. Parigot Encoding

Satisfaction of the computation and extensionality laws of iteration, the destructor, case distinction, and primitive recursion by both 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 discussed earlier, the Parigot encoding suffers from significant space overhead: Parigot naturals are represented in 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 as a signature for the generic Scott encoding, and although we cannot obtain the (generic) standard induction principle for the resulting datatype, we still may still use Lepigre–Raffalli induction.

With our final example, we demonstrate that this last concern is not hypothetical: the set of monotone type schemes is a *strict* superset of the set of functorial type schemes. Consider a datatype for infinitely branching trees, which in Haskell would be defined as:

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

The signature of *ITree* is positive. However, because the recursive occurrence of *ITree* in the node constructor's argument type occurs within an arrow, we cannot prove that the functor identity law as formulated in Figure 48 holds.

To see why this is the case, assume we have $f : Nat \to S$ and $g : S \to T$. Lifting g over the type scheme $\lambda\, X : \star.\, Nat \to X$ and applying the result to f, we obtain the expression:

$$\lambda\, x.\, g\ (f\ x) : Nat \to T$$

To prove the functor identity law, we must show that the expression above is propositionally equal to f while assuming only that g behaves extensionally like the identity function on terms of type S. Since Cedille's equality is intensional, we cannot conclude from this last assumption that g is itself equal to $\lambda\, x.\, x$.

In fact, in the presence of the δ axiom, we can *prove* that the covariant mapping for function types, and thus the mapping for the signature of infinitary trees does not satisfy the functor laws. Recall that δ - t (Figure 7) can be checked against any type if t proves an absurd equation, and that the Cedille implementation uses the Böhm-out algorithm to determine when an

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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

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

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

$$|indsum\ (in1\ s)\ t_1\ t_2| =_{\beta\eta} |t_1\ s|$$
$$|indsum\ (in2\ t)\ t_1\ t_2| =_{\beta\eta} |t_2\ t|$$

Figure 81. *Sum*, axiomatically (`utils/sum.ced`).

```
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) reflectNat t1]
- δ - pf .
```

Figure 82. Counterexample: a monotone type scheme which is not a functor (`signatures/itree.ced`).

equation is absurd. The counterexample proceeds by picking closed instances of f and g such that g behaves extensionally as the identity function, but nonetheless f and $\lambda x.g\ (f\ x)$ are Böhm-separable.

Not every monotone scheme is functorial. In Figure 81, 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 82, we define *ITreeF*, the signature for infinitely branching trees, and *itreeFmap*, its corresponding mapping

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

operation (here *Unit* is the single-element type). Following this, we prove with *monoITreeF* that this type scheme is monotonic. The function from $ITreeF \cdot X$ 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*. Note that for the node case in particular, we know that the function we are mapping (*elimCast -c*) is *definitionally* equal to the identity function.

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 second, 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 Parigot encoding. Though the generic formulation of the standard induction principle is not derivable without assuming functoriality of the datatype signature, we observe that in some cases that Lepigre–Raffalli induction can be used to give an ordinary (datatype-specific) induction principle. For example, for *ITree*, we can derive

```
indITree ◁ ∀ P: ITree → ⋆. P leaf →
 (Π f: Nat → ITree. (Π n: Nat. P (f n)) → P (node f)) → Π x: ITree. P x
= <..>
```

where *leaf* and *node* are the constructors for *ITree* (see `lepigre-raffalli/examples/itree.ced` in the code repository).

9. 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 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 metatheoretic 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 datatype recursion schemes, 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 computation laws for the primitive recursion scheme for datatypes. The

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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 scheme and the Scott encoding, and the primitive (co)recursion scheme and the 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. This allows 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 nondependent impredicative encoding we used to equip Scott 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 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 cyclic subtyping derivation with an internally realized proof of the fact that our derived recursive types are least fixpoints of type schemes.

Our twofold 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 with 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 case distinction serves a mostly pedagogical role; we leave as future work the task of providing a more semantic (e.g., category-theoretic) account of Lepigre–Raffalli recursion.

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 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 have shown Lambek's lemma and both the computation *and* extensionality laws for the case distinction, iteration, and primitive recursion schemes.

10. 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

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

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 used the derived monotone recursive types to derive two recursive representations of data in lambda calculus, the Parigot and Scott encoding, *generically* in a signature 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 positive datatypes, including infinitary trees and the usual formulation of rose trees. However, more care is needed for checking monotonicity of nested datatypes (Abel et al., 2005; Bird and Meertens, 1998).

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, which would complete the characterization of Scott-encoded datatypes whose signatures are positive but nonfunctorial. 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. The authors gratefully acknowledge NSF support under award 1524519, and DoD support under award FA9550-16-1-0082 (MURI program).

Competing Interests. The authors declare none.

Notes

1 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

2 In an unpublished note, Abadi et al. (1993) claim to present a type for Scott naturals in System F that lacks an efficient predecessor. Our view is that this type is a form of Church encoding.

References

Abadi, M., Cardelli, L. and Plotkin, G. (1993). Types for the Scott numerals. Unpublished note.

Abbott, M. G., Altenkirch, T. and Ghani, N. (2003). Categories of containers. In: Gordon, A. D. (ed.), *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*, vol. 2620. Lecture Notes in Computer Science. Springer, 23–38.

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

Abel, A., Matthes, R. and Uustalu, T. (2005). Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science* **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. *Journal of Applied Logic* **4** (4) 428–469.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

Atkey, R. (2018). Syntax and semantics of quantitative type theory. In: Dawar, A. and Grädel, E. (eds.), *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09–12, 2018*. ACM, 56–65.

Barendregt, H. P., Dekkers, W. and Statman, R. (2013). *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press.

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*, 52–67.

Breitner, J., Eisenberg, R. A., Jones, S. P. and Weirich, S. (2016). Safe zero-cost coercions for Haskell. *Journal of Functional Programming* **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)*, New York, NY, USA, ACM, 50–63.

Dybjer, P. and Palmgren, E. (2016). Intuitionistic type theory. In: E. N. Zalta (ed.), *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. (eds.), *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*, vol. 10895. Lecture Notes in Computer Science. Springer, 235–252.

Firsov, D. and Stump, A. (2018). Generic derivation of induction for impredicative encodings in cedille. In: Andronick, J. and Felty, A. P. (eds.), *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*. ACM, 215–227.

Geuvers, H. (1992). Inductive and coinductive types with iteration and recursion. In: *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Bastad*. Bastad, Chalmers University of Technology, 183–207.

Geuvers, H. (2001). Induction is not derivable in second order dependent type theory. In: Abramsky, S. (ed.), *Typed Lambda Calculi and Applications (TLCA)*, vol. 2044. Lecture Notes in Computer Science. Springer, 166–181.

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

Ghani, N., Johann, P. and Fumex, C. (2012). Generic fibrational induction. *Logical Methods in Computer Science* **8** (2).

Kleene, S. (1965). Classical extensions of intuitionistic mathematics. In: Bar-Hillel, Y. (ed.), *LMPS 2*. North-Holland Publishing Company, 31–44.

Kopylov, A. (2003). Dependent intersection: A new way of defining records in type theory. In: *18th IEEE Symposium on Logic in Computer Science (LICS)*, 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)*. IEEE Computer Society, 460–469.

Lepigre, R. and Raffalli, C. (2019). Practical subtyping for Curry-style languages. *ACM Transactions on Programming Languages and Systems* **41** (1) 5:1–5:58.

Matthes, R. (1999). Monotone fixed-point types and strong normalization. In: Gottlob, G., Grandjean, E. and Seyr, K. (eds.), *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings*, vol. 1584. Lecture Notes in Computer Science. Springer, 298–312.

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*, 173–184.

Miquel, A. (2001). The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. In: Abramsky, S. (ed.), *Typed Lambda Calculi and Applications*, vol. 2044. Lecture Notes in Computer Science. Springer, 344–359.

Parigot, M. (1988). Programming with proofs: a second order type theory. In: Ganzinger, H. (ed.), *European Symposium on Programming (ESOP)*, vol. 300. Lecture Notes in Computer Science. Springer, 145–159.

Parigot, M. (1989). On the representation of data in lambda-calculus. In Börger, E., Büning, H. and Richter, M. (eds.), *Computer Science Logic (CSL)*, vol. 440. Lecture Notes in Computer Science. Springer, 309–321.

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 Transactions on Programming Languages and Systems* **22** (1) 1–44.

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press

Santocanale, L. (2002). A calculus of circular proofs and its categorical semantics. In: Nielsen, M. and Engberg, U. (eds.), *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*, vol. 2303. Lecture Notes in Computer Science. Springer, 357–371.

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, Vol. 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: Rémy, D. and Lee, P. (eds.), *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP 1999), Paris, France, September 27-29, 1999*. ACM, 102–113.

Stump, A. (2017). The calculus of dependent lambda eliminations. *Journal of Functional Programming* **27** e14.

Stump, A. (2018a). From realizability to induction via dependent intersection. *Annals of Pure and Applied 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. *Journal of Functional Programming* **26** e3.

Stump, A. and Jenkins, C. (2021). 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.

Cite this article: Jenkins C and Stump A (2021). Monotone recursive types and recursive data representations in Cedille. *Mathematical Structures in Computer Science* **31**, 682–745. https://doi.org/10.1017/S0960129521000402

https://doi.org/10.1017/S0960129521000402 Published online by Cambridge University Press