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

Monotone Recursive Types and Recursive Data Representations in Cedille

Christopher Jenkins and Aaron Stump

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

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

Received 23 December 2019

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 type theory. As applications, we use monotone recursive types to generically derive two recursive representations of data in the lambda calculus, the Parigot and Scott encoding, together with constant-time destructors, a recursion scheme, and the standard induction principle.

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 of a recursive type to its one-step unrolling $[\mu\, X.\, T/X]T$ is the basis for the important distinction of *iso-* and *equi-recursive* types (Crary et al., 1999) (see also Section 20.2 of (Pierce, 2002)). With iso-recursive types, the two types are related by constant-time functions $\mathit{unroll} : \mu\, X.\, T \to [\mu\, X.\, T/X]T$ and $\mathit{roll} : [\mu\, X.\, T/X]T \to \mu\, X.\, T$ which are mutual inverses (composition of these two in any order produces a function that is ex-

tensionally the identity function). With equi-recursive types, the recursive type and its one-step unrolling are considered definitionally equal, and *unroll* and *roll* are not needed to pass between the two.

Without restrictions, adding recursive types as primitives to an otherwise terminating theory allows typing of diverging terms. For example, let B abbreviate $\mu\, X.\, (X \to X)$. Then, we see that B is equivalent to $B \to B$, allowing us to assign type $B \to B$ to $\lambda\, x.\, x\ x$. From that type equivalence, we see that we may also assign type B to this term, allowing us to type the diverging term $(\lambda\, x.\, x\ x)\lambda\, x.\, x\ x$.

Diverging terms usually must be avoided in type theory to retain soundness of the theory as a logic under the Curry-Howard isomorphism (Sørensen and Urzyczyn, 2006). The usual restriction on recursive types is to require that to form (alternatively, to introduce or to eliminate) $\mu\ X.\ T$, the variable X must occur only positively in T, where the function-type operator $\to$ preserves polarity in its codomain part and switches polarity in its domain part. For example, X occurs only positively in $(X \to Y) \to Y$, while Y occurs both positively and negatively. Since positivity is a syntactic condition, it is not compositional: if X occurs positively in T_1 and in T_2 containing also variable Y, this does not mean it will occur positively in $[T_1/Y]T_2$ (the substitution of T_1 for Y in T_2). For example, take T_1 to be X and T_2 to be $Y \to X$.

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

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

In Matthes's work, monotone recursive types are an addition to an underlying type theory, and the resulting system must be analyzed anew for such properties as subject

reduction, confluence, and normalization. In the present paper, we take a different approach by deriving monotone recursive types *within an existing type theory*, the Calculus of Dependent Lambda Eliminations (CDLE) (Stump, 2017; Stump, 2018b). Given any type scheme `F` satisfying a form of monotonicity, we show how to define a type `Rec ·F` together with constant-time functions `recRoll` and `recUnroll` witnessing the isomorphism between `Rec ·F` and `F ·(Rec ·F)`. The definitions are carried out in Cedille, an implementation of CDLE. The main benefit to this approach is that the existing metatheoretic results for CDLE – namely, confluence, logical soundness, and normalization for a class of types that includes ones defined here – apply, since they hold globally and hence perforce for the particular derivation of monotone recursive types.

Recursive representations of data in lambda calculi One important application of recursive types is their use in forming inductive datatypes, especially within a pure type theory where data must be encoded using λ-expressions. The most well-known method of lambda encoding is the *Church encoding*, or *iterative representation*, of data, which produces terms typable in unextended System F. The main deficiency of Church-encoded data is that data destructors, such as predecessor for naturals, can take no better than linear time to compute (Parigot, 1989; Splawski and Urzyczyn, 1999). As practical applications of Cedille's derived recursive types, we derive generically two *recursive representations* of data (described by (Parigot, 1992; Parigot, 1989)), the *Parigot encoding* and the *Scott encoding*, for which efficient destructors are known to exist (see (Stump and Fu, 2016) for discussion of the efficiency of these and other lambda encodings). For both encodings, we derive also a recursion scheme and induction principle. That this can be done for the Scott encoding in CDLE is itself quite a surprising result that builds on the derivations by (Lepigre and Raffalli, 2019; Parigot, 1988) of a strongly normalizing recursor for Scott naturals in resp. a Curry style type theory and logical framework.

Overview of this paper. We begin the remainder of this paper with a short introduction to CDLE (Section 2), before proceeding to the derivation of monotone recursive types (Section 3). Presentation of the applications of recursive types for deriving inductive datatypes with lambda encodings follows a common structure: Section 4 covers Scott encodings by first giving a concrete derivation of naturals with a weak induction principle, then the fully generic derivation; Section 5 gives a concrete example for Parigot naturals with the expected induction principle, then the fully generic derivation, and some important properties of the generic encoding (proven within Cedille); and Section 6 revists the Scott encoding, showing concretely how to derive the recursion principle for naturals, then generalizes to the derivation of the standard induction principle for generic Scott-encoded data, and shows that the same properties hold also for this derivation. Finally, Section 7 concludes by discussing related and future work. 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. CDLE, Cedille, and Lambda Encodings

The Calculus of Dependent Lambda Eliminations (CDLE), implemented in the Cedille proof assistant, is a logically consistent constructive type theory based on pure lambda calculus (Stump, 2017). Datatypes supporting an induction principle are derived within CDLE via λ-encodings like the well-known Church encoding. Geuvers proved that such derivations are impossible in pure second-order dependent type theory (Geuvers, 2001). To overcome this fundamental limitation, CDLE extends the Calculus of Constructions with three new type constructs (see below). Using these, induction was first derived for Church-encoded natural numbers (Stump, 2018a). Subsequently, derivations were carried out *generically*, both for the Church encoding and for the less well-known *Mendler encoding*: given a type scheme $F : \star \to \star$ satisfying certain properties, the inductive type

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

$$\frac{\Gamma \vdash FV(|t\ t'|) \subseteq dom(\Gamma)}{\Gamma \vdash \beta\langle t\rangle\{t'\} : \{t \simeq t\}} \qquad \frac{\Gamma \vdash q : \{t_1 \simeq t_2\} \quad \Gamma \vdash t : [t_2/x]T}{\Gamma \vdash \rho\ q\ -\ t : [t_1/x]T}$$

$$\frac{\Gamma \vdash q : \{t_1 \simeq t_2\} \quad \Gamma \vdash t_1 : T \quad FV(|t_2|) \subseteq dom(\Gamma)}{\Gamma \vdash \varphi\ q\ -\ t_1\{t_2\} : T} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t.1 : T}$$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : [t_1/x]T' \quad |t_1| = |t_2| \quad \Gamma \vdash \iota\, x{:}T.\, T' : \star}{\Gamma \vdash [t_1, t_2 @ x.T'] : \iota\, x{:}T.\, T'} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T.\, T'}{\Gamma \vdash t.2 : [t.1/x]T'}$$

Fig. 1. Introduction and elimination rules for additional type constructs

with its (categorical) constructor, destructor, and induction principle was derived (Firsov et al., 2018; Firsov and Stump, 2018).

Because it does not incorporate a datatype subsystem, a core version of Cedille ("Cedille Core") may be described very concisely, in 20 typing rules, occupying half a page (Stump, 2018c). These have been implemented in less than 1000 lines of Haskell in a checker that comes with Cedille. Cedille itself checks code written in a higher-level language, including support for inductive datatypes and a form of pattern-matching recursion, which elaborates down to Cedille Core.

We recapitulate the core ideas of Cedille. CDLE is an extrinsic (aka Curry-style) type theory, whose terms are exactly those of the pure untyped lambda calculus, with no additional constants or constructs. Cedille has a system of annotations for such terms, which contain sufficient information to type terms algorithmically. But these annotations play no computational role, and are erased both during compilation and by definitional equality. The latter is the congruential extension of $\beta\eta$-equality on erased terms and (at present) just β-equality on types.

CDLE extends the (Curry-style) Calculus of Constructions (CC) with three constructs: implicit products, primitive heterogeneous equality, and dependent intersection types.

$$
\begin{array}{llllll}
|\Lambda\, x{:}T.\, t| & = & |t| & |t\ -t'| & = & |t| \\
|\beta\langle t\rangle\{t'\}| & = & |t'| & |\rho\ q\ -\ t| & = & |t| \\
|\varphi\ q\ -\ t_1\{t_2\}| & = & |t_2| & |[t_1, t_2@x.T]| & = & |t_1| \\
|t.1| & = & |t| & |t.2| & = & |t|
\end{array}
$$

Fig. 2. Erasure rules for additional term annotations

Figure 1 shows the typing rules for annotated terms, for these constructs. The erasures of these annotations are given in Figure 2. In more detail, the additional constructs are:

The implicit product type ∀ `x: T. T'` of (Miquel, 2001). This can be thought of as the type for functions which accept an erased (computationally irrelevant) input `x` of type `T`, and produce a result of type `T'`. There are term constructs Λ `x. t` for introducing an implicit input `x`, and `t -t'` for instantiating such an input with `t'`. The implicit arguments exist just for purposes of typing. They play no computational role, and indeed definitional equality is defined only for erased terms (no implicit introductions or eliminations). When `x` is not free in `T'`, we allow ∀ `x: T. T'` to be written as `T` ⇒ `T'`, similarly to writing `T` → `T'` for Π `x: T. T'`.

An equality type $\{t_1 \simeq t_2\}$ on untyped terms. The terms t_1 and t_2 must have no undeclared free variables, but need not be typable. We introduce this with the term β`<t>{t'}`, which proves $\{t \simeq t\}$ and erases to (the erasure of) `t'`. If omitted, `t'` defaults to λ `x. x`. Combined with definitional equality, β can be used to prove $\{t_1 \simeq t_2\}$ for any $\beta\eta$-equal t_1 and t_2 whose free variables are all declared in the typing context. By allowing the β-term to erase to any closed (in context) term, we effectively add a top type to the language, since every term proves a true equation. We dub this the *Kleene trick*, as one may find the idea in Kleene's later definitions of numeric realizability, where any number is allowed as a realizer for a true atomic formula (Kleene, 1965).

We eliminate the equality type by rewriting, using the construct ρ `q - t`. If the expected type of the expression ρ `q - t` is `T`, and `q` proves $\{t_1 \simeq t_2\}$, then `t` is checked

against a type produced by replacing all occurrences of (terms convertible with) $\texttt{t}_1$ with $\texttt{t}_2$. For convenience, the Cedille tool also implements an enhanced variant of rewriting invoked by ρ+ where the expected type T is successively reduced and, for each reduction, the resulting type has all occurrences of $\texttt{t}_1$ replaced by $\texttt{t}_2$.

The construct φ q - $\texttt{t}_1${$\texttt{t}_2$} casts a term $\texttt{t}_2$ to type T, provided that $\texttt{t}_1$ has type T and q proves {$\texttt{t}_1 \simeq \texttt{t}_2$}. The term φ q - $\texttt{t}_1${$\texttt{t}_2$} erases to |$\texttt{t}_2$|. This is similar to the direct computation rule of NuPRL (see Section 2.2 of (Allen et al., 2006)).

The dependent intersection type ι x: T. T' of (Kopylov, 2003). This is the type for terms t which can be assigned both the type T and the type [t/x]T', the substitution instance of T' by t. In the annotated language, we introduce a value of ι x: T. T' by construct [t, t' @ x.T], where t has type T, t' has type [t/x]T', and the erasure |t| is definitionally equal to the erasure |t'|. The annotation @ x.T serves to specify the desired unsubstitution of the type of the second component. The T or [t.1/x]T' view of a term t of type ι x: T. T' is selected with t.1 and t.2, respectively.

We give two of the main meta-theoretic results of CDLE. For the full definition of the theory including kinding rules for types, as well as a semantics for types and proofs of the following theorems, see (Stump, 2018b):

Theorem 1 (Logical consistency). There is no term t such that $\vdash t : \forall\, X{:}\star.\, X$.

Theorem 2 (Call-by-name normalization of functions). Suppose $\Gamma \vdash t : T$, t is closed, and there exists a closed term t' which erases to $\lambda x.\ x$ and whose type is $T \to \Pi x : T_1.\ T_2$ for some T_1, T_2. Then $|t|$ is call-by-name normalizing.

In the code below, we elide annotations on the introduction forms for equalities and for dependent intersections, as they are inferred by Cedille. Cedille also infers many type arguments to functions; where needed, they are written with center dot.

3. Deriving Recursive Types in Cedille

To derive recursive types in Cedille, we implement a proof of Tarski's fixed-point theorem for monotone functions over a complete lattice. We recall here just the needed simple corollary of Tarski's more general result (cf. (Lassez et al., 1982)).

3.1. *Tarski's Theorem*

Theorem 3 ((Tarski, 1955)). Suppose f is a monotone operation on complete lattice $(A, \sqsubseteq, \sqcap)$. Let $Q = \{a \in A \mid f(a) \sqsubseteq a\}$ and $q = \sqcap Q$. Then $f(q) = q$.

Proof. First prove $f(q) \sqsubseteq q$. For this, it suffices to prove $f(q) \sqsubseteq a$ for every $a \in Q$, since this will imply that $f(q)$ is a lower bound of Q. Since q is the greatest lower bound of Q by definition of complete lattice, any other lower bound of Q (i.e., $f(q)$) must then be less than or equal to q. So assume $a \in Q$. We have $q \sqsubseteq a$ since q is a lower bound of Q. By monotonicity of f, we then obtain $f(q) \sqsubseteq f(a)$. Since $a \in Q$, we have $f(a) \sqsubseteq a$, and by transitivity of $\sqsubseteq$ we obtain $f(q) \sqsubseteq a$. From this, we obtain $q \sqsubseteq f(q)$ (hence showing both inclusions and thus equality of q and $f(q)$): from $f(q) \sqsubseteq q$ we obtain $f(f(q)) \sqsubseteq f(q)$ by monotonicity. Thus, $f(q)$ is in Q, and hence $q \sqsubseteq f(q)$. □

Notice in this proof *prima facie* impredicativity: we pick a fixed-point q of f by reference to a collection Q which contains q. We will see that this impredicativity carries over to Cedille. We may also observe that, actually, the above proof applies directly to show the following stronger statement (stronger because it holds with weaker assumptions):

Theorem 4. Suppose f is a monotone operation on a preorder $(A, \sqsubseteq)$, and that the set $Q = \{a \in A \mid f(a) \sqsubseteq a\}$ has a greatest lower bound q. Then $f(q) \sqsubseteq q$ and $q \sqsubseteq f(q)$.

We will need this strengthening – that $(A, \subseteq)$ need not form a complete lattice – to translate the proof to Cedille. But first, we must answer several questions:

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

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

— how should the meet operation $\sqcap$ be implemented?

One possibility for these that is available in System F is to choose functions $A \to B$ as the ordering $A \sqsubseteq B$, and 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. This approach, described in e.g. (Wadler, 1990), is essentially a generalization of order theory to category theory, and recursive types are defined using the Church encoding. However, recursive types so derived in System F lack the crucial property that *roll* and *unroll* are constant-time operations. Before we consider the alternative choices for these used in this paper (Section 3.3), we must first introduce the (derived) notion of a cast in Cedille.

3.2. *Casts*

A cast is a function from A to B that is provably (intensionally) equal to λ `x. x` (cf. (Breitner et al., 2016), and (Firsov et al., 2018) for the related notion of "identity functions"). With types playing the role of elements of the preorder, existence of a cast from types A to B will play the role of the ordering $A \sqsubseteq B$ in the proof above. Let us now walk through the definitions given in Figure 3.

This first definition from Figure 3 makes `Cast ·A ·B` the type of terms `c` which are both functions from `A` to `B` and also witness the fact that they are equal to the identity function. Thanks to the Kleene trick any term can witness a true equality, so requiring that `c` witness that it is equal to λ `x. x` does not restrict the terms that can be casts

In intrinsic type theory, there would not be much more to say: identity functions cannot map from A to B unless A and B are convertible types. But in an extrinsic type theory like CDLE, there are many nontrivial casts. For example, (assuming types `List` and `Bool`) we may map from $\forall$ `A: ⋆. List ·A` to `List ·Bool` using the function

```
module cast.

Cast ◁ ⋆ → ⋆ → ⋆
= λ A: ⋆. λ B: ⋆. ι c: A → B. { c ≃ λ x . x }.

elimCast ◁ ∀ A: ⋆. ∀ B: ⋆. Cast ·A ·B ⇒ A → B
= Λ A. Λ B. Λ c. λ a. φ (ρ c.2 - β) - (c.1 a) { a }.

intrCast ◁ ∀ A: ⋆. ∀ B: ⋆. ∀ f: A → B. (Π a: A. {f a ≃ a}) ⇒ Cast ·A ·B
= Λ A. Λ B. Λ f. Λ e. [ λ x. φ (e x) - (f x) { x } , β ].

castRefl ◁ ∀ A: ⋆. Cast ·A ·A = Λ A. [ λ x. x , β ].

castTrans ◁ ∀ A: ⋆. ∀ B: ⋆. ∀ C: ⋆. Cast ·A ·B ⇒ Cast ·B ·C ⇒ Cast ·A ·C
= Λ A. Λ B. Λ C. Λ c1. Λ c2.
  intrCast -(λ x. elimCast -c2 (elimCast -c1 x)) -(λ x. β).
```

Fig. 3. Casts (`cast.ced`)

`λ l. l ·Bool`. This function erases to `λ l. l`, and hence is indeed a cast. For another example, we may cast from `ι x: A. B` to `A` using the function `λ x. x.1`. This function also erases to `λ x. x` and hence is also a cast.

Next from Figure 3: if we have a `Cast ·A ·B`, the eliminator `elimCast` allows us to convert something of type `A` to something of type `B`. This may seem unsurprising, since something of type `Cast ·A ·B` is a function from `A` to `B`. So of course one can turn an `A` into a `B`, just by applying that function.

But this is not how the definition of `elimCast` works. The cast itself is an erased input to `elimCast`; (using the erased-argument arrow), so `elimCast` *cannot* simply apply that function to turn an `A` into a `B`. Instead, we use the φ construct (strong direct computation). The term `φ (ρ c.2 - β) - (c.1 a) { a }` in the body of the definition of `cast` erases to `a`. But it has type `B`, the same type as `c.1 a`, because we can prove `{c.1 a ≃ a}` given that `c` equals `λ x. x`. This proof is the first subterm of φ (i.e., `ρ c.2 - β`). Note also that `elimCast` itself erases to `λ a. a`, because the φ term erases to `a`, and the Λ-abstractions are all erased.

Next from Figure 3: `intrCast` takes in a function `f` from `A` to `B`, together with a

proof that this function is extensionally the identity (expressed by Π `a: A. {f a ≃ a}`).
These arguments are both erased. Given these, `intrCast` produces a cast from `A` to `B` as follows. The cast has two parts, introduced with the square-bracket notation for dependent intersections:

1 a function from `A` to `B`, and

2 a proof that this function equals λ `x. x`.

One would think that the proof (`e` in the code) that `f` is extensionally the identity should be incorporated in the second part. The trick is to incorporate it in the first: the function we write from `A` to `B` is

```
λ x. φ (e x) - (f x) {x}
```

This function takes in `x` of type `A` and just returns it, using the proof `e` in the φ-term to show that `f x`, which has type `B` as desired, equals just `x`. This function erases to λ `x. x`, and is thus trivially shown by β in the second component of the proof to be intensionally the identity. As an aside, recall that by default β erases to λ `x. x`, so the two components of the square-bracket term indeed have the same erasure as required. So, even though Cedille lacks function extensionality, we may still define casts extensionally.

Finally, we may compose casts, and every type has an identity cast (`castTrans` and `castRefl` in Figure 3). Thus, we may think of `Cast` as a partial order on types, and it is with respect to this order that we may express monotonicity. Furthermore, `Cast` harmonizes with the notion of preorder: for any types `A` and `B`, there can exist at most one `Cast ·A ·B`, just as in a preorder there is at most one way in which $A \sqsubseteq B$.

There is no obvious way to express the greatest lower bound of an arbitrary (possibly infinite) set of types with respect to this partial order. So Theorem 3 cannot be applied. But we will see below that its restatement as Theorem 4 does apply.

```
module recType (F : ⋆ → ⋆).

import cast.

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

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

recCast ◁ ∀ X: ⋆. Cast ·(F ·X) ·X ⇒ Cast ·Rec ·X
= Λ X. Λ c. [ λ d. d ·X -c , β ].

recRoll ◁ Mono ⇒ Cast ·(F ·Rec) ·Rec
= Λ mono.
  intrCast
    -(λ xs. Λ X. Λ c. elimCast -(castTrans -(mono (recCast -c)) -c) xs)
    -(λ xs. β).

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

_ ◁ { recRoll   ≃ λ x. x } = β.
_ ◁ { recUnroll ≃ λ x. x } = β.
recIso1 ◁ { λ a. recRoll (recUnroll a) ≃ λ a. a} = β.
recIso2 ◁ { λ a. recUnroll (recRoll a) ≃ λ a. a} = β.
```

Fig. 4. Monotone recursive types derived in Cedille (`recType.ced`)

3.3. *Translating the proof of Theorem 4 to Cedille*

Figure 4 shows the translation of the proof of Theorem 4 to Cedille, deriving monotone recursive types. Cedille's module system allows us to parametrize the module shown in Figure 4 by the type scheme `F`. Monotonicity of `F` is expressed with respect to the partial order induced by `Cast`:

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

As noted in Section 3.1, it is enough to require that the set of f-closed sets has a greatest lower bound. Semantically, the meaning of an impredicative quantification `∀ X: ⋆. 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 4 thus expresses the intersection of the set of all `F`-closed types `X`. This `Rec` corresponds to `q` in the proof of Theorem 4. 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 becomes the intersection of all `F`-closed types, where `X`'s being `F`-closed means there is a cast from `F ·X` to `X`. We require just an erased argument of type `Cast ·(F ·X) ·X`. By making the argument erased, we express the idea that we are taking the intersection of sets satisfying a property (being `F`-closed).

Next from Figure 4: `recCast` implements the fact that if `X` is `F`-closed, then `Rec ·F` is less than or equal to `X`; it corresponds to the first part of the proof above that $f(q) \sqsubseteq a$ for any $a \in Q$. The function `recRoll` implements the part of the proof that establishes $f(q) \sqsubseteq q$. The function `recUnroll` implements the second part, that $q \sqsubseteq f(q)$. It is there that the impredicativity noted above appears. In `recCast`, casting the `Rec ·F` argument `d` to the type `X` involves instantiating the type argument of `d` to `X`; in `recUnroll`, the chosen instantiation is `F ·(Rec ·F)`. This would not be possible in predicative type theory.

Since `elimCast` erases (as noted in Section 3.2) to `λ a. a`, it is not hard to confirm by inspection what Cedille indeed checks, that `recRoll` and `recUnroll` both erase to `λ x. x` (proved using syntax `_` for anonymous definitions), and are thus constant-time operations. This makes the proofs `recIso1` and `recIso2` trivial.

4. Scott encoding

As a first application of monotone recursive types in Cedille – and as a warm-up for the more general derivations to come – we show how to derive Scott-encoded natural numbers supporting a weak form of induction, where by "weak" we mean that the inductive hypothesis is only available as an erased argument. In contrast to the Church encoding, which identifies datatypes with their associated iteration scheme, is derivable in System F, and which suffers from linear-time destructors (such as predecessor for naturals (Parigot, 1989)), the Scott encoding supports constant-time destructors by identifying datatypes with their case scheme, but it is not known how to express the type of Scott-encoded data

in System F ((Splawski and Urzyczyn, 1999) points towards a negative result). The Scott encoding was first described in unpublished lecture notes by Dana Scott (Scott, 1962), and appears also in (Parigot, 1989; Parigot, 1992) wherein it is referred to as a "recursive representation" of data, and Scott-encoded naturals are referred to as "stacks."

We illustrate the with a concrete example: Scott-encoded naturals are defined in the untyped λ-calculus by the following constructors:

$$\begin{aligned} Z &= \lambda\, z.\, \lambda\, s.\, z \\ S &= \lambda\, n.\, \lambda\, z.\, \lambda\, s.\, s\ n \end{aligned}$$

In System F extended with recursive types, the type $\mu\, N.\, \forall\, X.\, X \to (N \to X) \to X$ can be given to Scott naturals. The preceeding section provides the type-level fixpoint operator `Rec` that allows stating this type in Cedille. This, along with the definitions of several operations on, and weak induction principle for, Scott-encoded naturals is given in Figures 5 and 6 which we now describe in detail. In Section 6, we show that this weak form of induction can be used to define a recursor and standard induction principle for Scott naturals.

4.1. *Scott-encoded naturals, concretely*

`NatF`, `zeroF`, and `sucF`. The scheme `NatF` is the usual impredicative encoding of the signature functor for naturals. Terms `zeroF` and `sucF` are its constructors, quantifying over the parameter `N`; it is easy to confirm using the rules of Figure 2 that these erase to the untyped constructors for Scott naturals given above.

`WkInductiveNatF` and `NatFI`. Next, and following a similar recipe to (Stump, 2018a) for deriving inductive types in Cedille, we define predicate `WkInductiveNatF` (parameterized by a type `N`) over terms of type `NatF ·N`. `WkInductiveNatF ·N n` says that, to prove `P n` (for any property `P` over `NatF ·N`) for the given `n`, it suffices to show certain cases for

```
module scott/examples/nat.

import recType.
import cast.

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.

WkInductiveNatF : Π N: ⋆. NatF ·N → ⋆ = λ N: ⋆. λ n: NatF ·N.
  ∀ P: NatF ·N → ⋆. P (zeroF ·N) →
  (Π m: (ι x: N. NatF ·N). P m.2 ⇒ P (sucF m.1)) → P n.

NatFI ◁ ⋆ → ⋆ = λ N: ⋆. ι x: NatF ·N. WkInductiveNatF ·N x.

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

monoNatFI ◁ Mono ·NatFI = Λ X. Λ Y. λ c.
  intrCast
    -(λ n. [ elimCast -(monoNatF c) n.1
           , Λ P. λ z. λ s.
             n.2 ·(λ x: NatF ·X. P (elimCast -(monoNatF c) x)) z
               (λ r. s [ elimCast -c r.1, elimCast -(monoNatF c) r.2 ]) ])
    -(λ _. β).
```

Fig. 5. Scott naturals (part 1) (`scott/examples/nat.ced`)

`zeroF` and `sucF`. The case for `sucF` is somewhat tricky: it says that for any `m` whose type is the intersection of types `N` and `NatF ·N`, we may assume a proof that `P` holds of `m` (viewed as type `NatF ·N`) when showing `P` holds of the successor of `m` (viewed as type `N`). We are justified *ex post facto* in assuming that `m` has this intersection type by our future choice for the instantiation of `N` to `Nat`, defined further below in the figure. Notice also that the inductive hypothesis `P m.2` is *erased*, as functions defined over Scott naturals are only given direct (computationally relevant) access the predecessor, not to previously computed results. Finally, type scheme `NatFI` is formed as the intersection type of terms `x` of type `NatF ·N` with proofs `WkInductiveNatF ·N x` that `x` is (weakly) inductive.

`monoNatF` and `monoNatFI`. Term `monoNatF` is a proof that the type scheme `NatF` is monotonic (`Mono`, defined Section 3.3). Given types `X` and `Y` and a cast `c` between them, the goal is to form a cast between `NatF ·X` and `NatF ·Y`, which in turn is done by invoking `intrCast` with a function of type `NatF ·X` $\rightarrow$ `NatF ·Y`, and a proof that this equal to λ `x. x`. That the functional argument has the desired type is straightforward to see, so consider its *erasure*, which is λ `n. n`. The bound occurrence of `n` in this erased term can be η-expanded to λ `z.` λ `s. n z s` (the abstraction over type `Z` is erased), the bound occurrence of `s` η-expanded to λ `r. s r`, and finally the coercion `elimCast -c` may be inserted, as it does not change the $\beta\eta$-equivalence class of the erasure of the term. So, the second argument to `intrCast` does indeed prove that the first is extensionally equal to the identity (since it is intensionally so).

The definition of `monoNatFI` is more complex, as there are now in parts of the definition of the type `NatFI ·X` bound occurrences of `x: NatF ·X`, which must be coerced to type `NatF ·Y` (where again `X` and `Y` are arbitrary types with a cast between them). Since `NatFI` is defined as a dependent intersection, the body of the first argument to `intrCast` is dependent intersection introduction, where both components must be equal to the bound variable `n`. That this is true for the first component is easy to verify given the erasure of `elimCast` and of dependent intersection projection. The second component again sees `n` η-expanded (the type argument to `n.2` is erased, and so is abstraction over `P`), and the bound variable `s` is η-expanded. The bound variable `r` has type ι `x: X. NatF ·X`, easily coerced to type ι `x: Y. NatF ·Y`. Finally, the type argument to `n.2` is the kind-coercion of predicate `P: NatF ·Y` $\rightarrow \star$ to a predicate of kind `NatF ·X` $\rightarrow \star$ (type `Y` occurs contravariantly in the kind of `P`), ensuring the whole expression is well-typed.

The process of deriving monotonicity proofs such as `monoNatF` and `monoNatFI` is rather mechanical once the general idea is understood, so we omit such definitions from the remaining code listings.

```
Nat ◁ ⋆ = Rec ·NatFI.
rollNat   ◁ NatFI ·Nat → Nat = elimCast -(recRoll -monoNatFI).
unrollNat ◁ Nat → NatFI ·Nat = elimCast -(recUnroll -monoNatFI).

zeroFI ◁ WkInductiveNatF ·Nat (zeroF ·Nat) = Λ P. λ z. λ s. z.
zero ◁ Nat = rollNat [ zeroF ·Nat , zeroFI ] .

sucFI ◁ Π n: Nat. WkInductiveNatF ·Nat (sucF n)
= λ n. Λ P. λ z. λ s. s [ n , (unrollNat n).1 ] -((unrollNat n).2 z s).
suc ◁ Nat → Nat = λ n. rollNat [ sucF n , sucFI n ] .

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

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

LiftNat ◁ (Nat → ⋆) → NatF ·Nat → ⋆
= λ P: Nat → ⋆. λ x: NatF ·Nat. ∀ m: Nat. ∀ eq: {m ≃ x}. P (φ eq - m {x}).

wkInductionNat ◁ ∀ 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.
  (unrollNat n).2 ·(LiftNat ·P) (Λ _. Λ _. z)
    (λ r. Λ ih. Λ m. Λ eq. s r.1 -(ih -r.1 -β)) -n -β.
```

Fig. 6. Scott naturals (part 2) (`scott/examples/nat.ced`)

`Nat`, `zero`, `suc`, and `caseNat`. In Figure 6, the type `Nat` is defined as a fix-point of type scheme `NatFI`, with its associated rolling and unrolling operations, constructors, and predecessor function. If we consider now the assumed type of the predecessor in the successor case of `WkInductiveNatF`, we may confirm that a term of type `Nat` also has type `NatF ·Nat` (as `unrollNat` is defined by a cast, and every `NatFI ·Nat` is also a `NatF ·Nat`). Concerning the constructors: for `zero` it is easy to see that the two components of the intersection have the same erasure, as required; in the definition of `sucFI`, the λ-bound `s` is given one relevant argument `[ n , (unrollNat n).1 ]` (definitionally equal to `n` by erasure and that `unrollNat` is defined by a cast) and one irrelevant argument (a recursively computed proof of `P n`), which gives us again that both components of the intersection defining `suc` have the same erasure.

Lastly, we define the case-scheme `caseNat` for Scott naturals, and the predecessor `pred` in terms of it. Notice that `pred` enjoys the expected run-time behavior: `pred (suc x)` reducing to `x` in a constant number of steps for any `x` (we use the name `_` for anonymous proofs):

`_` $\triangleleft$ Π `x: Nat. { pred (suc x)` $\simeq$ `x }` = λ `x.` β .

`LiftNat` and `wkInductionNat`. To derive the weak induction principle for Scott naturals, we first define the type-level function `LiftNat` that transforms predicates over `Nat` to related predicates over `NatF ·Nat`. We require this because the proof principle `WkInductiveNatF ·Nat` associated with each `Nat` only supports proof of properties over `NatF ·Nat`. Furthermore, `rollNat` will not give us this predicate transformation, as it can only be used to convert terms of type `NatFI ·Nat` (and not terms of type `NatF ·Nat`) to `Nat`. So, `LiftNat ·P x` is the type of witnesses that, when given some `m` of type `Nat` that is equal to `x`, proves `P` holds for `x`, where `x` has been cast to the type of `m` using φ (see Figure 1 for the typing and erasure of φ). In effect, the definition of `LiftNat` leverages the Curry-style typing of our setting to condition our proof on the assumption that `x` also has type `Nat`.

The type given for `wkInductionNat` is the expected type of an induction principle for naturals, except that in the successor case the inductive hypothesis `P m` is given as an erased argument. In the body, we unroll the type of `n`, select the view of it as a proof of `WkInductiveNat ·Nat (unrollNat n).1`, and use this to prove `LiftNat ·P` by cases. In the base case, assumption `z` suffices, as its type is `P zero`, convertible (by the erasure of φ) with the expected type `P (`φ `eq - m {zeroF})`. In the successor case, we use `s` to prove `P (suc m.1)` (again convertible with the expected type), with the second (erased) argument a recursively computed proof that `P` holds for `m`. Finally, we must discharge the obligations introduced by `LiftNat` itself, so we provide some term of type `Nat` (`n`) and a proof that it is equal to `(unrollNat n).1` (provable by reflexivity: `unrollNat`

erases to `λ x. x`, and `n.1` erases to `n`). With `wkInductionNat` so defined, we have the pleasing result that the computational content of this proof principle for Scott naturals is precisely that of the case-scheme:

```
_ ◁ { wkInductionNat ≃ caseNat } = β .
```

Example We can use `wkInductionNat` to prove that the single-step rebuilding of a natural `n` by constructors is equal to `n`:

```
_ ◁ Π n: Nat. { n ≃ n zero suc } = λ n.
  wkInductionNat n ·(λ x: Nat. {x ≃ x zero suc}) β (λ m. Λ pf. β) .
```

Notice that the inductive hypothesis `pf` goes unused, as the predecessor is not itself recursively rebuilt with constructors. The question arises whether including an erased inductive hypothesis adds any power over simple "proof by cases," or more generally whether *anything* non-trivial can be computed from Scott-encoded data in Cedille. We return to this question in Section 6, answering the affirmative in the form of a derivation of a recursion and standard induction principle for them.

4.2. *Scott-encoded data, generically*

In this section we derive Scott-encoded data with a weak induction principle. This derivation is *generic*, in the sense that it works for any functor F. We begin with a general description of the *iteration scheme* and *case-scheme* for datatypes. An arbitrary inductive datatype S can be understood as the least fixpoint of a signature functor F, with generic constructor $in : F \cdot S \to S$ (for example, constructors `zero` and `suc` of Figure 5 can be merged together into a single constructor `inNat : NatFI ·Nat → Nat`). What separates inductive datatypes from the notion of (monotone) recursive types is that the latter need not be the least fixpoint. Within a type theory, this additional property translates to the existence of an *iterator fold* for S satisfying the following typing and reduction

rule (with *fmap* the usual lifting operation of functions $A \to B$ to $F \cdot A \to F \cdot B$ that respects identity and composition of functions):

$$\frac{X \text{ a type} \quad a : F \cdot X \to X \quad d : S}{\mathit{fold}\ a\ d : X} \qquad \mathit{fold}\ a\ (\mathit{in}\ \mathit{ds}) \rightsquigarrow a\ (\mathit{fmap}\ (\mathit{fold}\ a)\ \mathit{ds})$$

In category theory, this is captured by the notion of *initial F-algebras*. An F-algebra (S, in) is an object S (e.g., a type) together with morphism $in : F\ S \to S$ (e.g., a function), where F is again understood to be a functor. The algebra (S, in) is said to be *initial* when for every algebra (X, a) there is a unique morphism $(\!|\, a \,|\!) : S \to X$ such that $(\!|\, a \,|\!) \circ in = a \circ (F\ (\!|\, a \,|\!))$, or equivalently that the following diagram commutes:

$$\begin{array}{ccc} F\ S & \xrightarrow{in} & S \\ \downarrow{\scriptstyle F\ (\!|\, a \,|\!)} & & \downarrow{\scriptstyle (\!|\, a \,|\!)} \\ F\ X & \xrightarrow{a} & X \end{array}$$

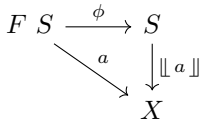
The iteration scheme (both its typing and computation law) for data in type theory is expressed in category theory as the guarantee of the existence of $(\!|\, a \,|\!)$, and the induction principle is expressed as the uniqueness of $(\!|\, a \,|\!)$ (c.f. (Jacobs and Rutten, 2011) for further discussion on this correspondence).

The case-scheme for datatype S in type theory is a function *case* (call this the *discriminator* for S) satisfying the following typing and reduction rule:

$$\frac{X \text{ a type} \quad a : F \cdot S \to X \quad d : S}{\mathit{case}\ a\ d : X} \qquad \mathit{case}\ a\ (\mathit{in}\ \mathit{ds}) \rightsquigarrow a\ \mathit{ds}$$

The case-scheme on its own is not a common subject of study in the categorical semantics of datatypes, so we invent some terminology. Call an algebra (S, ϕ) *discriminative* if for any morphism $a : F\ S \to X$ there exists a unique morphism $[\![\, a \,]\!] : S \to X$ such that

$[\![\, a \,]\!] \circ \phi = a$; equivalently, that the following diagram commutes:



We are unaware of any standard nomenclature for $[\![\, a \,]\!]$, so call this the *krisimorphism* (from the Greek $\kappa\rho\iota\sigma\eta$ meaning *judgment*, *decision*).

Using the iteration scheme for data, the typing rule for the case scheme can be satisfied by assigning *case a d* := *a* (*fold in d*), iteratively re-building data with constructor *in*. In category theory, the equality $a \circ (\!|\, in \,|\!) = a$ holds as a consequence of initiality, but in type theory this definition of the case-scheme does not satisfy the desired reduction rule. This is, in fact, a more general statement of the problem of linear run-time for computing predecessor for Church-encoded naturals.

For the derivation in this section, we use a modification of the case-scheme discussed above. This modification is due to subtle issues of *alignment* in Cedille – that is, ensuring that certain expressions are definitionally equal (after erasure) to each other. We describe this modification categorically: let $\widehat{X}$ denote the unitary product of object X, with $\langle f \rangle : Y \to \widehat{X}$ the unitary product of $f : Y \to X$ (for all such f, X, Y). It is clearly an equivalent condition to say that (S, ϕ) is discriminative iff for any morphism $a : F\ \widehat{S} \to X$ there exists a unique morphism $[\![\, a \,]\!]' : S \to X$ such that $[\![\, a \,]\!]' \circ \phi = a \circ (F\ \langle id \rangle)$, as $S \cong \widehat{S}$. To give informal intuition, in Cedille the unitary product provides space to "sneak in" an erased inductive hypothesis when defining the (weak) induction principle for datatype `S`.

Our generic derivation of Scott-encoded data is defined *directly* in terms of discriminative F-algebras, resulting in an efficient case-scheme. In particular, we make essential use of our derived recursive types to define (S, ϕ) in terms of triples (S, X, a) (where $a : F\ \widehat{S} \to X$). The remainder of this section gives some preliminary definitions and details the generic derivation. Proofs of essential properties (in particular, that the dis-

criminative algebra we define is also an initial algebra) are postponed until Section 6, wherein we derive the (non-weak) induction principle for datatype `S`.

```
module utils/wksigma .
import unit.

WkSigma ◁ Π A : ⋆ . (A → ⋆) → ⋆ = <..>

intrWkSigma ◁ ∀ A: ⋆. ∀ B: A → ⋆. Π a: A. ∀ b: B a. WkSigma ·A ·B = <..>
elimWkSigma ◁ ∀ A: ⋆. ∀ B: A → ⋆.
  WkSigma ·A ·B. ∀ X: ⋆. (Π a: A. ∀ b: B a. X) → X = <..>

wkproj1 ◁ ∀ A: ⋆. ∀ B: A → ⋆. WkSigma ·A ·B → A
= Λ A. Λ B. λ w. elimWkSigma w (λ a. Λ _. a).

indWksigma ◁ ∀ A: ⋆. ∀ B: A → ⋆. Π x: WkSigma ·A ·B.
  ∀ P: WkSigma ·A ·B → ⋆. (Π a: A. ∀ b: B a. P (intrWkSigma a -b)) → P x = <..>
etaWkSigma : ∀ A: ⋆. ∀ B: A → ⋆. Π p: WkSigma ·A ·B.
  {elimWkSigma p intrWkSigma ≃ p} = <..>

Wrap ◁ ⋆ → ⋆ = λ A: ⋆. WkSigma ·A ·(λ _: A. Unit).
wrap ◁ ∀ A: ⋆. A → Wrap ·A = Λ A. λ a. intrWkSigma a -unit.
unwrap ◁ ∀ A: ⋆. Wrap ·A → A = Λ A. λ w. wkproj1 w.
```

Fig. 7. Weak pairs and unitary products (`utils/wksigma.ced`)

4.2.1. *`WkSigma`, `Wrap`, and `Unit`* In Figure 7 we define the unitary product type `Wrap` in terms of the more general `WkSigma` (in code listings, `<..>` denotes an omitted definition). Type `WkSigma` is analogous to the impredicative encoding of a dependent pair, except that its second component is *erased*, and so for example is suitable for tupling together subdata with erased inductive hypotheses. Its constructor `intrWkSigma` takes an erased second argument, its eliminator `elimWkSigma` expects as an argument a function whose second argument is erased, and while the first projection `wkproj1` is easily definable, its second projection cannot be defined. For proofs: `indWkSigma` is an induction principle stating that, to prove a property `P` for some `WkSigma ·A ·B`, it suffices to show that the property holds of those weak pairs constructed using `intrWkSigma`; `etaWkSigma` proves that rebuilding a weak pair with its constructor reproduces the original pair. Type `Wrap`,

then, is defined by setting the second type argument to constant function returning `Unit`, the type with a single element (Figure 8). The induction principles for `WkSigma` and `Unit` can be derived in Cedille following the methods of (Stump, 2018a), and the respective extensionality principles (`etaWkSigma` and `etaUnit`) follow from these.

```
module utils/unit.
Unit ◁ ⋆ = <..>
unit ◁ Unit = <..>

indUnit ◁ Π x: Unit. ∀ P: Unit → ⋆. P unit → P x = <..>
etaUnit ◁ Π x: Unit. {x ≃ unit} = <..>
```

Fig. 8. The singleton type (`utils/unit.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 xs: F ·X . {fmap c xs ≃ xs}.

FmapCompose ◁ Fmap → ⋆ = λ fmap: Fmap.
  ∀ X: ⋆. ∀ Y: ⋆. ∀ Z: ⋆. Π f: Y → Z. Π g: X → Y.
  Π xs: F ·X. {fmap f (fmap g xs) ≃ fmap (λ x. f (g x)) xs}.

FmapExt ◁ Fmap → ⋆ = λ fmap: Fmap.
  ∀ X: ⋆. ∀ Y: ⋆. Π f: X → Y. Π g: X → Y. (Π x: X. {f x ≃ g x}) →
  Π xs: F ·X. {fmap f xs ≃ fmap g xs}.
```

Fig. 9. Functors (`functor.ced`)

4.2.2. *Functors* We define `Functor` and the associated functor identity and composition laws in Figure 9. Additionally, we define an optional property `FmapExt`, where `FmapExt fmap` expresses a kind of parametricity property of `fmap`. Precisely, it states that if functions `f` and `g` are extensionally equal, so too are `fmap f` and `fmap g`. This condition is required for showing in Section 5.2.3 that the recursive algebra we shall derive is unique.

Notice also that our definition of the identity law `FmapId` has a by-now familiar extrinsic

twist: the domain `X` and codomain `Y` of the lifted function `c` need not be convertible types in order for the constraint {`c x` $\simeq$ `x`} (for every `x: X`) to be satisfied. Phrasing the identity law in this way allows us derive a useful lemma (Figure 10) `monoFunctor` which shows every type scheme F that is a `Functor` is monotonic (`Mono`, defined in Section 3.3), yielding the utility function `fcast` (which is definitionally equal to λ `x. x`):

```
import functor.

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

import cast.
import recType.

monoFunctor ◁ Mono ·F = Λ X. Λ Y. λ c.
  intrCast -(λ d. fmap c.1 d) -(λ a. fmapId c.1 (λ x . ρ c.2 - β) a).

fcast ◁ ∀ A: ⋆. ∀ B: ⋆. Cast ·A ·B ⇒ F ·A → F ·B
= Λ A. Λ B. Λ c. elimCast -(monoFunctor c).
```

Fig. 10. Functor theorems (`functorThms.ced`)

4.2.3. *Definition of generic datatype `S`* Figures 11 and 12 gives the definition of type `S` of generic Scott-encoded data and some operators. We walk through these figures in detail.

Type families `AlgS` and `SF` are our first steps to defining the Scott encoding generically. `AlgS` corresponds to the family of triples (S, X, a) we shall informally call *Scott-style pseudo F-algebras* (with $a : F\ \widehat{S} \to X$). The definition of `SF` is similar to the standard definition of the least fixpoint of `F` in polymorphic λ-calculi, but defined in terms of `AlgS` instead of usual F-algebras. Term `monoSF` is a proof that `SF` is monotonic.

`PreS`, `PrfS`, and `preIn`. Type family `PreS` is a "pre-definition" of the type of Scott-encoded data. As with the concrete derivation of Scott naturals, *ex post facto* the definition of `PreS` is justified by the coming definition of datatype `S`, from which there will be

```
import functor.
import utils.

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

import recType.
import cast.
import functorThms · F fmap -fmapId -fmapCompose.

AlgS ◁ ⋆ → ⋆ → ⋆ = λ S: ⋆. λ X: ⋆. F ·(Wrap ·S) → X.
SF ◁ ⋆ → ⋆ = λ S: ⋆. ∀ X: ⋆. AlgS ·S ·X → X.

monoSF ◁ Mono ·SF = <..>

PreS ◁ ⋆ → ⋆ = λ S: ⋆. ι x: S. SF ·S.
PrfS ◁ Π S: ⋆. (SF ·S → ⋆) → ⋆
= λ S: ⋆. λ P: SF ·S → ⋆. WkSigma ·(PreS ·S) ·(λ x: PreS ·S. P x.2).

preIn ◁ ∀ S: ⋆. F ·(PreS ·S) → SF ·S
= Λ S. λ xs. Λ X. λ alg. alg (fmap (λ x: PreS ·S. wrap x.1) xs).

monoPreS ◁ Mono ·PreS = <..>

monoPrfS ◁ ∀ X: ⋆. ∀ Y: ⋆. Π c: Cast ·X ·Y. ∀ P: SF ·Y → ⋆.
  Cast ·(PrfS ·X ·(λ x: SF ·X. P (elimCast -(monoSF c) x))) ·(PrfS ·Y ·P)
= <..>
```

Fig. 11. Generic datatype `S` and operations (part 1) (`scott/encoding.ced`)

a cast to the type `PreS ·S`. For any type `S` and predicate `P: SF ·S → ⋆`, `PrfS ·S ·P` is the type of weak pairs of some `x` of type `PreS ·S` and proofs that `P` holds for `x.2`. Definition `preIn` is similarly a "pre-definition" of the morphism component of a discriminative F-algebra; from the definition alone, it is clear that some `preCase` could be defined satisfying the desired computation law for the modified case-scheme (though not yet the typing law). Monotonicity for `PreS` and `PrfS` is given by `monoPreS` and `monoPrfS` (definitions omitted) – the latter requires the extensionality principle for `WkSigma` to show that eliminating a weak pair with its constructor re-builds the original weak pair.

```
WkPrfAlgS ◁ Π S: ⋆. (SF ·S → ⋆) → ⋆
= λ S: ⋆. λ P: SF ·S → ⋆. Π xs: F ·(PrfS ·S ·P).
  P (preIn (fmap (λ x: PrfS ·S ·P. wkproj1 x) xs)).

WkInductiveS ◁ Π S: ⋆. SF ·S → ⋆ = λ S: ⋆. λ x: SF ·S.
  ∀ P: SF ·S → ⋆. WkPrfAlgS ·S ·P → P x.

WkIF ◁ ⋆ → ⋆ = λ S: ⋆. ι x: SF ·S. WkInductiveS ·S x.

monoWkIF ◁ Mono ·WkIF = <..>

S ◁ ⋆ = Rec ·WkIF.
rollS   ◁ WkIF ·S → S = elimCast -(recRoll -monoISF).
unrollS ◁ S → WkIF ·S = elimCast -(recUnroll -monoISF).

toPreS ◁ Cast ·S ·(PreS ·S) = intrCast -(λ x. [ x , (unrollS x).1 ]) -(λ _. β).

case ◁ ∀ X: ⋆. AlgS ·S ·X → S → X = Λ X. λ a. λ x. (unrollS x).1 a.

in ◁ F ·S → S = λ xs.
  rollS [ preIn ·S (fcast -toPreS xs)
        , Λ P. λ a.
          ρ+ ς (fmapId (λ x: S. unwrap (wrap x)) (λ _ . β) xs) -
          ρ+ ς (fmapCompose (unwrap ·S) (wrap ·S) xs) -
           a (fmap ·S ·(PrfS ·S ·P)
                (λ x. intrWkSigma (toPreS.1 x) -((unrollS x).2 a)) xs) ].

out ◁ S → F ·S = case (λ xs. fmap (unwrap ·S) xs).

LiftS ◁ (S → ⋆) → SF ·S → ⋆ = λ P: S → ⋆. λ x: SF ·S.
  ∀ m: S. ∀ eq: {m ≃ x}. P (φ eq - m {x}).

wkInduction ◁ ∀ P: S → ⋆. WkPrfAlgS ·S ·(LiftS ·P) → Π x: S. P x
= Λ P. λ a. λ x. (unrollS x).2 ·(LiftS ·P) a -x -β.
```

Fig. 12. Generic datatype `S` and operations (part 2) (`scott/encoding.ced`)

`WkPrfAlgS`, `WkInductiveS`, and `WkIF`. In Figure 12, `WkPrfAlgS` is a Scott-style variant of the notion of a *P-proof F-algebra*, which itself was first described by (Firsov and Stump, 2018) as a dependently typed version of an F-algebra. For any type `S` and property `P` over `SF ·S`, `WkPrfAlgS ·S ·P` takes some `xs` in which all subdata (of type `PreS ·S`) are tupled together (using `PrfS`) with erased proofs that they satisfy `P`, and must return

a proof that `P` holds for the value constructed (by `preIn`) from `xs`, after removing the `WkSigma` wrapping.

Weak inductivity predicate `WkInductiveS ·S x` is the property of some type `S` and `x: SF ·S` that, for all properties `P: SF ·S`, to show that `P` holds of `x` it suffices to give a weak proof algebra `WkPrfAlgS ·S ·P`. `WkIF` is, finally, the type scheme whose fixpoint is the datatype `S` we wish to derive; it is defined as a family (over a type `S`) of the intersection type of terms `x` of type `SF ·S` which themselves satisfy `WkInductiveS ·S x`. Term `monoWkIF` proves that this type-scheme is monotonic.

`S`, `rollS`, `unrollS`, and `toPreS`. Type `S` is the type of generic Scott-encoded data, defined as a fixpoint of `WkIF`. Functions `rollS` and `unrollS` are the fixpoint rolling and unrolling operations for this type; because they are defined using casts, both are definitionally equal to λ `x. x`. This point is essential, as it allows us to define the cast `toPreS` from `S` to `PreS ·S`. In particular for any `x` of type `S`, `(unroll x).1` has type `SF ·S` and is definitionally equal to `x`.

`case`, `in`, and `out`. We can now define `case`, the *discriminator* for datatype `S`. In the body of the definition, `(unrollS x).1` has type `SF ·S` (convertible with $\forall$ `X: ⋆. AlgS ·S ·X` $\rightarrow$ `X`) and is given a suitable argument `a`.

For generic constructor `in`, the first component `preIn (fcast -toPreS xs)` of the introduced intersection is straightforward. The second component requires a proof of `WkInductive (preIn xs)` (for clarity we omit the inserted type coercion `fcast -toPreS` in the following discussion). So, after introducing `P: SF ·S` $\rightarrow$ $\star$ and `a: WkPrfAlgS ·S ·P` we rewrite the expected type,

```
P (preIn xs)
```

using the functor identity and composition laws to introduce additional wrapping and

unwrapping, producing

```
P (preIn (fmap unwrap (fmap wrap xs)))
```

Now we can invoke `a` on `xs` to produce a term of this expected type by first using `fmap` to produce, from the `S` subdata in `xs`, terms of type `PrfS ·S ·P` using `intrWkSigma`, where in particular the proofs of `P` are *recursively computed, but erased*. Because of this, and because `wrap` and `intrWkSigma` are definitionally equal, the two components of this intersection are indeed equal, and furthermore `in` is definitionally equal to `preIn`.

With the definitions of `in` and `case`, we have that both the expected typing and computation laws for our modified case scheme hold by definition (in Section 6, we show that the typing and computation laws for the usual case scheme hold by the functor laws). The definition of destructor `out` is relatively straightforward, using `case` and providing it a function that simply `unwrap`s all subdata.

`LiftS` and `wkInduction`. As with the concrete derivation of Scott naturals, before defining the weak form of induction for the Scott encoding we first define a type-level function `LiftS` that lifts properties over `S` to properties over `SF ·S`, as the proof principle of a term of type `S` works only for the latter. `LiftS ·P x` is the type of functions which, given an erased `m: S` and erased proof `eq` that `m` is equal to `x`, returns a proof that `P` holds of `x` after casting this to the type of `m`. Then, `wkInduction ·P a x` proves the expected `P x` by invoking the proof principle of `x` (after unrolling it to type `WkIF ·S`) to prove `LiftS ·P` and providing the proof algebra `a`, a term of type `S` (x) and proof it is equal to `(unrollS x).1`; the given type of the body, `P (φ β - x (unrollS x).1)` is convertible with the expected type.

We describe the properties that hold of our generic derivation of Scott-encoded data Section 6, where we derive their recursion and (full) induction principles.

5. Parigot encodings

In this section we derive inductive Parigot-encoded naturals, showing with a concrete example in Section 5.1 the main techniques used for the generic derivation of inductive Parigot-encoded data in Section 5.2. The Parigot encoding, first described by (Parigot, 1988; Parigot, 1992) for naturals and later for datatypes generally in (Geuvers, 2014) (wherein it is called the *Church-Scott encoding*), combines the approaches of the Church and Scott encoding to allow functions over data both access to previously computed results and efficient (under call-by-name operational semantics) access to immediate subdata. For example, in the untyped λ-calculus Parigot-encoded naturals are constructed as follows:

$$\begin{aligned} Z &= \lambda\, z.\, \lambda\, s.\, z \\ S &= \lambda\, n.\, \lambda\, z.\, \lambda\, s.\, s\ n\ (n\ z\ s) \end{aligned}$$

In System F extended with recursive types, the type $\mu\, N.\, \forall\, X.\, X \to (N \to X \to X) \to X$ can be given to Parigot naturals. The advantages of Parigot-encoding data are offset by a steep increase in the size required to represent them, with the encoding of natural n taking $O(2^n)$ space. Additionally, another deficit is that the type given above is not precise enough as it admits terms formed by nonsense constructors such as $S' = \lambda\, n.\, \lambda\, z.\, \lambda\, s.\, s\ Z\ (n\ z\ s)$.

As with Scott-encoded data, the recursive types derived in Section 3 allows us to state the type of Parigot naturals in Cedille. However, the approach taken for our derivation of them differs from the derivation of Scott naturals: we will "bake-in" to the definition of the type scheme the data's *reflection law*, i.e. that recursively rebuilding numbers with their constructors reproduces the same number. One consequence of this baking-in is that it rules out nonsense constructors such as S', leaving only the desired Z and S. To accomplish this, we find it convenient to use the *Kleene trick* (Section 2) to define a type `Top` of all (well-scoped) terms of the untyped λ-calculus; this is so that we may

reason directly about the computational behavior of terms before they could otherwise be defined. The definition of `Top` is given in Figure 13.

```
module utils/top.

Top ◁ ⋆ = {λ x. x ≃ λ x. x} .
```

Fig. 13. The `Top` type (`utils/top.ced`)

5.1. *Parigot-encoded naturals, concretely*

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

module parigot/examples/nat.

recNatU     ◁ Top = β{λ x. λ f. λ n. n x f}.
zeroU       ◁ Top = β{λ z. λ s. z}.
sucU        ◁ Top = β{λ n. λ z. λ s. s n (recNatU z s n)}.
reflectNatU ◁ Top = β{recNatU zeroU (λ _. sucU)}.

NatF ◁ ⋆ → ⋆ = λ N: ⋆.
  ι n: ∀ X: ⋆. X → (N → X → X) → X. {reflectNatU n ≃ n}.

monoNatF ◁ Mono ·NatF = <..>

Nat ◁ ⋆ = Rec ·NatF.
rollNat   ◁ NatF ·Nat → Nat = elimCast -(recRoll -monoNatF).
unrollNat ◁ Nat → NatF ·Nat = elimCast -(recUnroll -monoNatF).

zero ◁ Nat = rollNat [ Λ X. λ z. λ s. z , β{zeroU} ].
suc  ◁ Nat → Nat = λ n.
  rollNat [ Λ X. λ z. λ s. s n ((unrollNat n).1 z s)
          , ρ+ (unrollNat n).2 - β{sucU n} ].

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

pred ◁ Nat → Nat = recNat zero (λ m. λ _. m).
```

Fig. 14. Inductive Parigot naturals (part 1) (`parigot/examples/nat.ced`)

Figures 14 and 15 give the derivation of Parigot naturals supporting an induction principle. We describe this in detail.

`zeroU`, `sucU`, `recNatU`, and `reflectNatU`. Definition `recNatU` gives the untyped recursion principle for Parigot numerals, where the bound `x` is interpreted as the base case, `f` as the inductive case taking both the previously computed results and a predecessor value directly, and `n` as the numeral to recurse over. Following this are the untyped constructors `zeroU` and `sucU`. Term `reflectNatU` is the untyped version of a function that recursively rebuilds Parigot numerals with their constructors.

`NatF` and `Nat`. We can now define `NatF`, the type scheme whose fixpoint is the type of "inherently reflective" Parigot naturals. It is defined by a dependent intersection as the type of terms `n` which have the expected type, and for which rebuilding with constructors produces the same `n` (thanks to the Kleene trick, if this equation holds then it is trivial for `n` to be a proof of it). We have that `NatF` is monotonic by `monoNatF`, which allows us to define the type of Parigot-encoded data `Nat` as a fixpoint of `NatF` along with its rolling and unrolling operations (as they are defined by casts, both are equal to λ `x. x`).

`zero` and `suc`. Next, we define the constructors `zero` and `suc` for Parigot naturals. The former is defined by rolling of a term of type `NatF ·Nat`, for which the second component β`{zeroU}` proves that `{reflectNatU zero` $\simeq$ `zero}`, as this equality holds β-equivalence. For the latter: in the first component of the introduced intersection we compute the second argument to the λ-bound `s` by unrolling the predecessor `n` and projecting out the view of it as having type $\forall$ `X: ⋆. X →` `(Nat → X → X) → X`; in the second component, we prove that for the first (which is definitionally equal to `sucU n`) the reflection law also holds by rewriting with the proof that this holds for `n`.

```
InductiveNat ◁ Nat → ⋆ = λ x: Nat.
  ∀ P: Nat → ⋆. P zero → (Π n: Nat. P n → P (suc n)) → P x.

NatI ◁ ⋆ = ι x: Nat. InductiveNat x.

zeroI ◁ NatI = [ zero , Λ P. λ z. λ s. z ].
sucI  ◁ NatI → NatI = λ n. [ suc n.1 , Λ P. λ z. λ s. s n.1 (n.2 z s) ].

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

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

inductionNat ◁ ∀ P: Nat → ⋆.
  P zero → (Π n: Nat. P n → P (suc n)) → Π n: Nat. P n
= Λ P. λ z. λ s. λ n. (elimCast -toNatI n).2 z s.
```

Fig. 15. Inductive Parigot naturals (part 2) (`parigot/examples/nat.ced`)

`recNat` and `pred`. The recursion principle for Parigot naturals is given by `recNat` by simply invoking the first component of the (unrolled) number argument `n` on the base and step cases for recursion `x` and `f`. Notice also that `recNat` is definitionally equal to `recNatU`. We use `recNat` to define `pred`, an efficient (under call-by-name operational semantics) predecessor which acts in the successor case by discarding the previously computed result and returning the previous number `m` directly. This is witnessed by the following definitionally true equality:

```
_ ◁ Π x: Nat. { pred (suc x) ≃ x } = λ x. β .
```

`InductiveNat` and `NatI` We now define `InductiveNat` (Figure 15), a predicate over `Nat`, with `InductiveNat x` stating that for all properties `P: Nat → ⋆`, to show `P x` it suffices to first show `P zero` and to show `P (suc m)` assuming some `m: Nat` and a proof `P m`. Then, `NatI` is the type of naturals which are also themselves proofs that they satisfy `InductiveNat`. From this we can define the inductive variant of the constructors for Parigot naturals, `zeroI` and `sucI` (for the latter, in the second component of the introduced intersection we must show `InductiveNat (suc n.1)`, which after abstracting over assumptions `P`, `z: P zero`, and `s: Π m: Nat. P m → P (suc m)` this goal is

discharged by invoking `s` and giving as a second argument `n.2 z s` of type `P n.1`). For both definitions, both first and second components are definitionally equal.

`reflectNat`, `toNatI`, and `inductionNat`. The purpose of baking-in the reflection law in the definition of `NatF` is now realized in the definition of `reflectNat`, which recursively re-builds its argument with the inductive variant of constructors. Since `reflectNat` is definitionally equal to `reflectNatU`, we can define the cast `toNatI` from `Nat` to `NatI` by using the proof associated with each `n: Nat` that {`reflectNatU n` $\simeq$ `n`} and by using the full power of `intrCast` to produce a function *intensionally* equal to identity from one that is only *extensionally* equal to it. identity Finally, we define `inductionNat`, the induction principle for Parigot naturals, by simply casting the given `Nat` to `NatI`. Pleasingly, the computational content of `inductionNat` is precisely that of `recNat`:

`_` $\triangleleft$ `{ inductionNat` $\simeq$ `recNat }` = β .

Examples For our Parigot naturals, we can define iterative functions such addition (`add`), recursive functions such as a summation of numbers $0...n$ (`sumFrom`), and prove by induction that `zero` is a right identity of addition (`addZRight`), shown below:

```
add ◁ Nat → Nat → Nat = λ m. λ n. recNat n (λ _. suc) m .
sumFrom ◁ Nat → Nat = recNat zero (λ m. λ s. add (suc m) s).

addZRight ◁ Π n: Nat. {add n zero ≃ n}
= inductionNat ·(λ x: Nat. {add x zero ≃ x}) β (λ m. λ ih. ρ+ ih - β).
```

5.2. *Parigot-encoded data, generically*

In this section we derive inductive Parigot-encoded data generically for a signature functor F. The Parigot encoding identifies datatypes with their *recursion scheme*, which allows functions defined over data to access both the previously computed result (as in

the iteration scheme given by the Church encoding) and all immediate subdata (as in the case scheme given by the Scott encoding). In type theory, that datatype P supports a recursion scheme translates to the existence of a recursor *rec* for P satisfying a certain typing and reduction rule: assuming *fmap* is the usual functorial lifting of a function $A \to B$ to $F \cdot A \to F \cdot B$, $A \times B$ the product type of types A and B introduced with (a, b) when a has type A and b has type B, and $in : F \cdot P \to P$ is the generic constructor of P, the typing and reduction rules are:

$$\frac{X \text{ a type} \quad a : F \cdot (P \times X) \to X \quad d : P}{rec\ a\ d : X} \qquad rec\ a\ (in\ ds) \rightsquigarrow a\ (fmap\ (\lambda\, x.\, (x, rec\ a\ x))\ ds)$$

Independently, (Geuvers, 1992) and (Mendler, 1991) defined *recursive F-algebras* to give a categorical semantics for the recursion scheme for datatypes (see Section 4.2 for a brief discussion of F-algebras and initial F-algebras). An algebra (P, ϕ) is *recursive* if for any morphism $a : F\ (P \times X) \to X$ there exists a morphism $(\!|\, a \,|\!) : P \to X$ such that $(\!|\, a \,|\!) \circ \phi = a \circ (F\ \langle id\,,\, (\!|\, a \,|\!)\rangle)$ (where $\langle id, (\!|\, a \,|\!)\rangle$ is the product-forming morphism of the identity id and $(\!|\, a \,|\!)$). This is depicted visually by the following commuting diagram:

$$\begin{array}{ccc} F\ P & \xrightarrow{\ \phi\ } & P \\ {\scriptstyle F\ \langle id, (\!|\, a \,|\!)\rangle}\downarrow & & \downarrow{\scriptstyle (\!|\, a \,|\!)} \\ F\ (P \times X) & \xrightarrow[\ a\ ]{} & X \end{array}$$

If (P, ϕ) is an initial F-algebra, and in the underlying category all pairs of objects have products, then it is also recursive, with $(\!|\, a \,|\!)$ a *paramorphism* (Meertens, 1992) uniquely defined (up to isomorphism) by the *catamorphism* (iteration). However, and as with the case scheme, in type theory this definition of the recursion scheme suffers from same inefficiency as plagues the predecessor for Church-encoded naturals.

As with the Scott encoding, our generic derivation of the Parigot encoding avoids this problem by being defined *directly* in terms of recursive F-algebras, using recursive

types to define (P, ϕ) in terms of triples (P, X, a), resulting in efficient data accessors under call-by-name operational semantics. We bake-in to our encoding the paramorphic reflection law $id = \langle\!|\, \phi \circ (F\ \pi_2) \,|\!\rangle$ (where $\pi_2 : A \times B \to B$ is the second projection for any product $A \times B$). In the generic development, the right-hand side of this equation describe a function `reflectI` which rebuilds data `P` to a type `IP` supporting an induction principle; that the equation holds allows us to define a cast `toIP` from `P` to `IP`.

The remainder of this section gives some preliminary definitions, details the generic derivation, and outlines properties of the data so derived including a proof that the morphism $\langle\!|\, a \,|\!\rangle$ is unique (that is, any other morphism $h : P \to X$ making the diagram commute is extensionally equal to $\langle\!|\, a \,|\!\rangle$), from which it is an easy corollary that (P, in) is an initial F-algebra.

5.2.1. *`Sigma` and `Pair`* Product type `Pair` and dependent product types `Sigma` (Figure 16) are derivable in Cedille via λ-encodings. We do not describe this here as the approach is essentially the same as in (Stump, 2018a). Instead, we simply give the type and kind signatures of the definitions we use, with `<..>` indicating omitted definitions.

```
module utils/sigma.

Sigma ◁ Π A : ⋆ . (A → ⋆) → ⋆ = <..>
Pair ◁ ⋆ → ⋆ → ⋆ = λ A: ⋆. λ B: ⋆. Sigma ·A ·(λ _: A. B).

mksigma ◁ ∀ A: ⋆. ∀ B: A → ⋆. Π a: A. B a → Sigma ·A ·B = <..>
mkpair  ◁ ∀ A: ⋆. ∀ B: ⋆. A → B → Pair ·A ·B = Λ A. Λ B. λ a. λ b. mksigma a b.

proj1 ◁ ∀ A: ⋆. ∀ B: A → ⋆. Sigma ·A ·B → A = <..>
fst ◁ ∀ A: ⋆. ∀ B: ⋆. Pair ·A ·B → A = Λ A. Λ B. λ p. proj1 p.

proj2 ◁ ∀ A : ⋆. ∀ B: A → ⋆. Π s: Sigma ·A ·B. B (proj1 s) = <..>
snd ◁ ∀ A: ⋆. ∀ B: ⋆. Pair ·A ·B → B = Λ A. Λ B. λ p. proj2 p.
```

Fig. 16. Product types (`utils/sigma.ced`)

```
import functor.
import utils.

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

import recType.
import cast.
import functorThms · F fmap -fmapId -fmapCompose.

recU     ◁ Top = β{λ alg. λ x. x alg}.
inU      ◁ Top = β{λ xs. λ alg. alg (fmap (λ x. mkpair x (recU alg x)) xs)}.
reflectU ◁ Top = β{recU (λ xs. inU (fmap snd xs))}.

AlgP ◁ ⋆ → ⋆ → ⋆ = λ P: ⋆. λ X: ⋆. F ·(Pair ·P ·X) → X.
PF  ◁ ⋆ → ⋆ = λ P: ⋆. ι x: ∀ X: ⋆. AlgP ·P ·X → X. {reflectU x ≃ x}.

monoPF ◁ Mono ·PF = <..>

P ◁ ⋆ = Rec ·PF.
rollP   ◁ PF ·P → P = elimCast -(recRoll -monoPF).
unrollP ◁ P → PF ·P = elimCast -(recUnroll -monoPF).

rec ◁ ∀ X: ⋆. AlgP ·P ·X → P → X = Λ X. λ alg. λ x. (unrollP x).1 alg.

inP1 ◁ F ·P → ∀ X: ⋆. AlgP ·P ·X → X =
  λ xs. Λ X. λ alg. alg (fmap (λ x: P. mkpair x (rec alg x)) xs).

reflectionInP1 ◁ Π xs: F ·P. {reflectU (inP1 xs) ≃ inP1 xs} =
  λ xs. ρ+ (fmapCompose ·P (snd ·P ·Top) (λ x. mkpair x (β{reflectU x})) xs) -
        ρ+ (fmapId ·P ·Top (λ x. β{reflectU x}) (λ x. (unrollP x).2) xs) -
        β.

in ◁ F ·P → P = λ xs. rollP [inP1 xs , ρ (reflectionInP1 xs) - β{inP1 xs}].
out ◁ P → F ·P = rec (fmap (fst ·P ·(F ·P))).
```

Fig. 17. Generic datatype type P and operations (parigot/encoding.ced)

5.2.2. *Definition of generic datatype P* Figures 17 and 18 give the definition of generic Parigot-encoded datatype `P` and the essential operations `recursion`, `in`, and `out`. We walk through these in detail.

`recU`, `inU`, and `reflectU` express the untyped versions of future definitions `rec`, `in`, and `reflectP`, respectively, and are best understood when we explain those. For now, it suffices to say that `recU` expresses the computational content of the recursion scheme for our data, `inU` is the datatype's generic constructor (equivalently, the morphism component of the recursive algebra), and `reflectU` recursively re-builds data with its (generic) constructor `inU`.

`AlgP`, `PF`, and `P`. Definition `AlgP` a type family corresponding to the triples (P, X, a) (with $a : F\ (P \times X) \to X$) by which we shall define recursive F-algebras, and which we informally dub *Parigot-style pseudo F-algebras*. `PF` is the type scheme whose fixpoint is the carrier of the recursive F-algebra. It is similar to the standard definition of the fixpoint of `F` in polymorphic λ-calculi, with two important differences; first, it uses `AlgP` instead of usual F-algebras; second, we use a dependent intersection (and the Kleene trick) to bake in the reflection law. Term `monoPF` proves that `PF` is monotonic, which entitles us to define our datatype `P` as its fixpoint, along with operations `rollR` and `unrollR` (both of which are definitionally equal to λ `x. x`) using the recursive types derived in Section 3.

`rec`, `in`, and `out`. Function `rec` is the datatype's recursion scheme, mapping a Parigot-style pseudo F-algebra (`P`, `X`, `alg`) to a function computing `X` from `P`. Its definition is straightforward, as the first component of the intersection which defines `PF ·P` (which we get by unrolling `x`) is a function which, given some `Alg ·P ·X` (for any `X`) returns something of type `X`. Notice also we have that `rec` and `recU` are definitionally equal (the syntax _ is used to give an anonymous proof):

```
_ ◁ {rec ≃ recU} = β.
```

The definition of `in` is more involved, and so is broken into three parts. The first term `inP1` computes from some `xs: F ·P` an expression whose type is the first part of the intersection type defining `PF ·P`. Its definition comes directly from the left-then-bottom path of the commuting diagram of the categorical definition of a recursive F-algebra. Given some type `X` and term `alg: AlgP ·P ·X`, we first `fmap` a function over `xs` that tuples together all subdata (`x`) with recursively computed results (`rec alg x`), producing an expression of type `F ·(Pair ·P ·X)` which is then given as an argument to `alg`.

The second definition `reflectionInP1` proves that an instance of the reflection law holds for data constructed from `inP1`, as required for the second component of the intersection type defining `PF ·P`. The equation to be proved is

```
{ reflectU (inP1 xs) ≃ inP1 xs }
```

Recalling the definition of `reflectU`, we see that the left-hand side of this equation is β-equivalent to an (untyped) expression containing the composition of `fmap snd` with `fmap (λ x. mkpair x (reflectU x))`. We invoke the functor law to rewrite this as a single mapping of `λ x. snd (mkpair x (reflectU x))`, and β-reducing this transforms the goal into proving

```
{ inP1 (fmap reflectU xs) ≃ inP1 xs }
```

Now, observe that all subdata of `xs` are "inherently reflective" (see the definition of `PF`), meaning that mapping `reflectU` over `xs` should be (extensionally) an identity operation. We finish the proof by using this fact together with the functor identity law.

Finally, to define `in` we combine these two definitions via intersection to form an expression of type `PF ·P` and use `rollP` to get an expression of type `P`, where in the second component of the introduced intersection we use the Kleene trick again to allow

`inP1 xs` to have the type `{reflectU (inP1 xs) ≃ inP1 xs }`. Notice also that our definition of `in` is β-equivalent (after erasure) to `inU`:

`_ ◁ {in ≃ inU} =` β

695 The last definition `out` is straightforward, computed by recursion over some term of type `P`: the Parigot-style F-algebra we give takes some expression of type `F ·(Pair ·P ·(F ·P))`, simply selects the first component of the tupled results (the immediate subdata) and discards the second (the previously computed results). Under a call-by-name operational semantics, the discarded computation will not be evaluated, ensuring that `out` incurs no
700 unnecessary run-time penalty.

```
PrfAlgP ◁ (P → ⋆) → ⋆ = λ Q: P → ⋆.
  Π xs: F ·(Sigma ·P ·Q). Q (in (fmap (proj1 ·P ·Q) xs)).

Inductive ◁ P → ⋆ = λ x: P. ∀ Q: P → ⋆. PrfAlgP ·Q → Q x.
IP ◁ ⋆ = ι x: P. Inductive x.

inIP1 ◁ F ·IP → P = λ xs. in (fcast -(intrCast ·IP ·P -(λ x. x.1) -(λ x. β)) xs).

inIP2 ◁ Π xs: F ·IP. Inductive (inIP1 xs) = λ xs. Λ Q. λ alg.
  ρ ≤ (fmapId (λ x: IP. proj1 (mksigma x.1 (x.2 alg))) (λ _. β) xs) -
  ρ ≤ (fmapCompose (proj1 ·P ·Q) (λ x: IP. mksigma x.1 (x.2 alg)) xs) -
  alg (fmap (λ x: IP. mksigma x.1 (x.2 alg)) xs).

inIP ◁ F ·IP → IP = λ xs. [ inIP1 xs , inIP2 xs].

reflectP ◁ P → IP = λ x. rec (λ xs. inIP (fmap (snd ·P ·IP) xs)) x.

toIP ◁ Cast ·P ·IP = intrCast -reflectP -(λ r. (unrollP r).2).

induction ◁ ∀ Q: P → ⋆. PrfAlgP ·Q → Π x: P. Q x =
  Λ Q. λ alg. λ x. (elimCast -toIP x).2 alg.
```

Fig. 18. Generic inductive datatype `IP` (`parigot/encoding.ced`)

`PrfAlgP`, `Inductive`, and `IP`. In Figure 18 we define resp. *Parigot-style Q-proof F-algebras*, an inductivity predicate `Inductive` for terms of type `P`, and the type of terms `x` which both have type `P` and themselves realize the predicate `Inductive x`. The notion

of Q-proof F-algebras was first described by (Firsov and Stump, 2018) as a dependently typed version of an F-algebra. For the Parigot-style version of this, we have as carrier Q, a property over P, and the inductive hypothesis `xs` of type `F ·(Sigma ·P ·Q)` (wherein all subdata are paired together with proofs that `Q` holds of it). From this, it must be shown that `Q` holds of the data constructed by `in` of `xs` (after projecting out just the subdata of `xs`).

Type `Inductive x` is a property saying that, for some particular `x: P`, in order to show `Q x` (for any property `Q`) it suffices to show `PrfAlg ·Q`. Type `IP` is the type (formed by intersection) of data of type `P` that proves it is itself inductive.

`inIP`. The constructor `inIP` for type `IP` is also defined in three parts, one for each side of the intersection defining `IP` (`inIP1` and `inIP2`, resp.), and one (`inIP`) defined by combining these. Definition `inIP1` constructs an element of `P` from `xs: F ·IP` by simply casting `xs` to type `F ·P` and invoking `in`. For `inIP2` we must show every `P` constructed by `inIP1` is also inductive. After introducing arguments `xs`, `Q`, and `alg` (the proof algebra), the goal is to prove `Q (inIP1 xs)`. We start by rewriting by the functor identity law to introduce an additional mapping over `xs` which first tuples together subdata (coerced to type `P`) with recursively computed results, then immeidately selects just the subdata. The rewritten type is now:

```
Q (in (fmap (λ x: IP. proj1 (mksigma x.1 (x.2 alg))) xs))
```

Next, we separate this into two mappings over `xs` using the functor composition law:

```
Q (in (fmap proj1 (fmap (λ x: IP. mksigma x.1 (x.2 alg)) xs)))
```

and this is the type of the expression `alg (fmap (λ x: IP. mksigma x.1 (x.2 alg)) xs)` given in `inIP2`.

Again, we note that `inIP` is definitionally equal to `in`, as are `inIP1` and `inIP2`:

`_ ◁ { in ≃ inIP } = β .`

`reflectP` and `induction`. The derivation of the induction principle for `P` concludes in three short steps. First, we define `reflectP`, a function constructing an element of `IP` from the data `F ·(Pair ·P ·IP)` by recursively re-building with constructor `inIP`. Since its erasure is equal to `reflectU`, we may use it to define a cast `toIP` from `P` to `IP` using the baked-in reflection law of `P`. Then, the definition of `induction` is simple: take the `x` of type `P`, cast it to `IP`, select the view of this as a proof of `Inductive (unrollP x).1`, and give as argument to this the proof-algebra `alg`. This leaves us with the pleasing result that the computational behavior of `induction` is precisely that of `rec`:

`_ ◁ { induction ≃ rec } = β .`

5.2.3. *Properties of `P`* Our generically derived inductive Parigot-encoded data satisfies the expected *cancellation law*, *reflection law*, *Lambek's lemma*, and (conditional) uniqueness of the universal mapping property of the recursive algebra (`P`, `in`), and closed terms of type `P` are call-by-name normalizing. Each of these has been proven within Cedille, shown in Figure 19.

— *Normalization* is shown by `norm` and by appealing to Theorem 2: there exists a cast from `P` to some function type, so any closed annotated term `t` of type `P` is normalizing under a call-by-name operational semantics.

— *The cancellation law* proves that the diagram describing recursive F-algebras at the beginning of this section commutes, giving the computation of `rec` over data constructed by `in`.

— *The reflection law* has been discussed already in the derivation. As it is built into the datatype, its proof is trivial.

— *Lambek's lemma* states that `out` and `in` are mutual inverses; `lambek1` holds by the

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

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

import parigot/encoding ·F fmap -fmapId -fmapCompose.
import functorThms ·F fmap -fmapId -fmapCompose.

norm ◁ Cast ·P ·(AlgP ·P ·Unit → Unit)
= intrCast -(λ x. (unrollP x).1 ·Unit) -(λ _. β) .

cancellation ◁ ∀ X: ⋆. Π xs: F ·P. Π a: AlgP ·P ·X.
  { rec a (in xs) ≃ a (fmap (λ r. mkpair r (rec a r)) xs) } = <..>

reflection ◁ Π x: P. {reflectP x ≃ x} = λ x. (unrollP x).2.

lambek1 ◁ Π xs: F ·P. {out (in xs) ≃ xs} = <..>
lambek2 ◁ Π r: P. {in (out r) ≃ r} = <..>

unique ◁ FmapExt ·F fmap → ∀ X: ⋆. Π a: AlgP ·P ·X. Π h: P → X.
          Π hom: Π xs: F ·P. {h (in xs) ≃ a (fmap (λ r. mkpair r (h r)) xs)}.
          Π x: P. {h x ≃ rec a x} = <..>

fold ◁ ∀ X: ⋆. (F ·X → X) → P → X
= Λ X. λ a. rec (λ xs. a (fmap (snd ·P ·X) xs)) .

initial ◁ FmapExt ·F fmap → ∀ X: ⋆. Π a: F ·X → X. Π h: P → X.
          Π hom: Π xs: F ·P. {h (in xs) ≃ a (fmap h xs)}.
          Π x: P. {h x ≃ fold a x}
= λ fext. Λ X. λ a. λ h. λ hom.
  unique fext ·X
    (λ xs. a (fmap (snd ·P ·X) xs)) h
    (λ xs. ρ+ (fmapCompose (snd ·P ·X) (λ x: P. mkpair x (h x)) xs) - hom xs) .
```

Fig. 19. Properties of `P` (`parigot/props.ced`)

functor laws alone, whereas `lambek2` additionally requires the induction principle (in the proof the induction hypothesis is not used, merely dependent case-analysis).

The second to last proof `unique` shows uniqueness of the universal mapping property of the recursive algebra (`P`, `in`) – that is, for any Parigot-style F-algebra `a` with carrier `X`, if there is some other morphism `h: P → X` which makes the following diagram commute:

$$\begin{array}{ccc} F\ P & \xrightarrow{in} & P \\ {\scriptstyle F\ \langle id,h\rangle}\downarrow & & \downarrow{\scriptstyle h} \\ F\ (P \times X) & \xrightarrow[a]{} & X \end{array}$$

then `h` is (extensionally) equal to `rec a`. Proof of this fact requires an additional condition `FmapExt ·F fmap`: that is, that invoking `fmap` with extensionally equal functions produces extensionally equal functions (see Section 4.2.2 Figure 9 for the definition). Having this, it is easy to show under the same condition that (`P`, `in`) is an initial F-algebra: we simply define the iterator (catamorphism) `fold` in terms of the recursor (paramorphism) `rec` and appeal to `unique`.

5.2.4. *Example: Parigot-encoded lists* We conclude the discussion of Parigot-encoded data by instantiating the generic derivation of Section 5.2 to define Parigot-encoded lists. In doing so, we show that the expected induction principle for lists is derivable from the generic induction principle, and that the additional parametricity condition `FmapExt` required for showing initiality in Section 5.2.3 can be satisfied by simple datatypes.

```
module utils/sum.

Sum ◁ ⋆ → ⋆ → ⋆ = <..>

in1 ◁ ∀ A: ⋆. ∀ B: ⋆. A → Sum ·A ·B = <..>
in2 ◁ ∀ A: ⋆. ∀ B: ⋆. B → Sum ·A ·B = <..>

indSum ◁ ∀ A: ⋆. ∀ B: ⋆. Π x: Sum ·A ·B.
  ∀ Q: Sum ·A ·B → ⋆. (Π a: A. Q (in1 a)) → (Π b: B. Q (in2 b)) → P x = <..>
```

Fig. 20. Coproducts (`utils/sum.ced`)

We begin with a brief description `Sum`, the coproduct type in Cedille. Type `Sum ·A ·B` represents the disjoint union of types `A` and `B`, which can be formed either with a term of type `A` using `in1` or a term of type `B` using `in2`. The induction principle `indSum` states that, in order to prove that a property `Q: Sum ·A ·B → ⋆` holds for some `x: Sum ·A ·B`, it suffices to show that `Q` holds of the coproduct constructed with either `in1` or `in2`,

given any argument suitable for these. The definitions of Figure 20 can be derived within Cedille using standard techniques (c.f. (Stump, 2018a)) so their definitions are omitted (indicated by `<..>`).

```
module utils/listFunctor.

import functor.
import sum.
import sigma.
import unit.

ListF ◁ ⋆ → ⋆ → ⋆ = λ A: ⋆. λ L: ⋆. Sum ·Unit ·(Pair ·A ·L).

ListFmap ◁ ∀ A: ⋆. Fmap ·(ListF ·A) = <..>
ListFmapId ◁ ∀ A: ⋆. FmapId ·(ListF ·A) (ListFmap ·A) = <..>
ListFmapCompose ◁ ∀ A: ⋆. FmapCompose ·(ListF ·A) (ListFmap ·A) = <..>

ListFmapExt ◁ ∀ A: ⋆. FmapExt ·(ListF ·A) (ListFmap ·A)
= Λ A. Λ X. Λ Y. λ f. λ g. λ ext. λ l.
  indSum l ·(λ x: ListF ·A ·X. {ListFmap f x ≃ ListFmap g x})
    (λ _. β)
    (λ p. indPair p ·(λ x: Pair ·A ·X. {ListFmap f (in2 x) ≃ ListFmap g (in2 x)})
            (λ hd. λ tl. ρ+ (ext tl) - β)).
```

Fig. 21. Signature functor for `List` (`utils/listFunctor.ced`)

Figure 21 defines `ListF`, the signature functor for lists, in the standard way as the coproduct of the `Unit` type (Figure 8) and the product (Figure 16) of `A` and `L`, where `A` is the type of elements of the list and `L` is the stand-in for recursive occurrences of the datatype. The definitions of the functorial lifting of a function `ListFmap` and proofs that this respects identity and composition (resp. `ListFmapId` and `ListFmapCompose`) are omitted.

To show the additional constraint `ListFmapExt` on `ListFmap`, we assume two functions `f` and `g` of type `X → Y` (for any `X` and `Y`), and a proof `ext` that for every `x: X`, we have {`f x ≃ g x`}. We further assume an arbitrary `l: ListF ·A ·X`, and must show that {`ListFmap f l ≃ ListFmap g l`}. This proof obligation is discharged with `indSum`: in the case the list is empty, there is no subdata of type `X` to invoke `f` or `g` on, and the proof

is trivial; otherwise, we further invoke `indPair`, the induction principle for products, to reveal the head `hd: A` and tail `tl: X` of the list and appeal to `ext` to show that {`in2 (mkpair hd (f tl)) ≃ in2 (mkpair hd (g tl))`}.

```
import utils.

module parigot/examples/list (A : ⋆).
import parigot/encoding as P
  ·(ListF ·A) (ListFmap ·A) -(ListFmapId ·A) -(ListFmapCompose ·A).

List ◁ ⋆ = P.P .
nil ◁ List = P.in (in1 unit) .
cons ◁ A → List → List = λ hd. λ tl. P.in (in2 (mkpair hd tl)) .

indList ◁ ∀ Q: List → ⋆. Q nil →
  (Π hd: A. Π tl: List. Q tl → Q (cons hd tl)) → Π l: List. Q l = <..>
```

Fig. 22. Parigot-encoded lists (`parigot/examples/list.ced`)

Finally, we define the type `List` of Parigot-encoded lists in Figure 22. The module in which it is defined takes a type parameter `A` for the type of elements of the list, and imports the generic derivation (qualified with module name `P`) with the definitions of Figure 21 instantiated with this parameter. Thus, `List` is defined directly as `P.P` (where the signature functor is `ListF ·A`). Constructors `nil` and `cons` are defined in terms of the generic constructor `P.in` (which expects as an argument some `ListF ·A ·List`), and the standard induction principle `indList` is defined (omitted) in terms of the generic `P.induction` as well as the induction principles for `Sum` and `Pair`.

6. Recursive Scott encoding

The induction principle `wkInductionNat` we showed for Scott naturals in Section 4.1 is weak. In a proof of some property using this principle, in the case that that natural number in question is of the form `suc m`, the inductive hypothesis `P m` is *erased* and so cannot assist in computing a proof of `P (suc m)`. This situation reflects the usual

criticism of the Scott encoding: that it is not inherently iterative. There does not appear to be any way, for example, to define a recursor over Scott naturals of type:

$\forall$ `X: ⋆. X → (Nat → X → X) → Nat → X`

Amazingly, in some settings this deficit of the Scott encoding is only apparent. (Parigot, 1988) showed how to derive using "metareasoning" a strongly normalizing recursor for Scott encoded naturals with a similar type to the one above. More recently, (Lepigre and Raffalli, 2019) also showed how the same recursor can be given the above type in a Curry-style type system featuring a sophisticated form of subtyping utilizing "circular but well-founded" typing derivations.

Knowing this, we revisit our earlier question from the end of Section 4.1: does access to an erased inductive hypothesis add any power over mere proof by cases? We answer in the affirmative by showing how to type the recursor for Scott naturals in Cedille using `wkInductionNat` together with an ingenious type definition used by (Lepigre and Raffalli, 2019) (therein attributed to Parigot) for the same purpose.

The main idea behind this derivation of the recursor for Scott naturals is noticing that the untyped lambda terms encoding zero and successor for Scott naturals may be η-expanded in such a way that the interpretations for these constructors – the "base" and "step" cases of a function computed over a Scott natural – may be passed copies of themselves:

$$\begin{array}{rcll} Z & = & \lambda\, z.\, \lambda\, s.\, z & \simeq \quad \lambda\, z_1.\, \lambda\, s_1.\, \lambda\, z_2.\, \lambda\, s_2.\, z_1\ z_2\ s_2 \\ S & = & \lambda\, n.\, \lambda\, z.\, \lambda\, s.\, s\ n & \simeq \quad \lambda\, n.\, \lambda\, z_1.\, \lambda\, s_1.\, \lambda\, z_2.\, \lambda\, s_2.\, s_1\ n\ z_2\ s_2 \end{array}$$

A usage $n\ t_1\ t_2\ t_1\ t_2$ based on this understanding should: ensure t_1 is a constant function ignoring its first two arguments and returning the intended result for the base case; and ensure t_2 is a function taking as arguments the predecessor n' and another copy each of t_1 and t_2 for making recursive calls invoked by $n'\ t_1\ t_2\ t_1\ t_2$.

The type that supports this intended usage for Scott naturals is far from obvious,

but can be defined in (unextended, impredicative) System F. This is given as `NatR` in Figure 23, which is itself a minor generalization of the one presented by (Lepigre and Raffalli, 2019). We describe the figures in detail in Section 6.1; the definitions rely upon previous ones given in Section 4.1, Figures 5 and 6. In Section 6.2, we generalize this type definition (and thus the entire derivation) in two orthogonal ways, making it (1) *generic*, working for any datatype signature functor F; and (2) *dependent*, transforming the recursor into the standard induction principle.

6.1. *Recursor for Scott naturals, concretely*

`NatRZ`, `NatRS`, and `NatR`. The type definition of `NatR` is rather tricky, so we endeavor to provide some intuition for its construction. Compared to the foregoing discussion, the Scott naturals that `NatR` classifies have been η-expanded once more so that they may take *themselves* as arguments (at type `Nat`). `NatR` can be seen as a *supertype* of `Nat`, a fact we shall soon demonstrate by deriving a proof of `Cast ·Nat ·NatR`. It relies on two additional definitions: `NatRZ` (a type family of "base cases" for recursion); and `NatRS` (a type family of "step cases"). In these two definitions, quantification over `Z` and `S` is used to hide recursive references to `NatRZ` and `NatRS`, respectively. The intended use of a term r of type `NatR` is:

— r was produced as a type coercion of some n of type `Nat`;

— its two arguments of type `NatRZ ·X` are copies of the same "base case" term;

— its two arguments of type `NatRS ·X` are copies of the same "step case" term; and

— its `Nat` argument is n (that is, itself)

The functions `zeroR` and `sucR` give an operational understanding of the subset of terms of type `NatR` that are also Scott naturals. In `zeroR` (which by η-contraction and erasure is equal `zero`) we take two copies each of the base (`z1` and `z2`, both of type `NatRZ ·X`) and step cases (`s1` and `s2`, both of type `NatRS ·X`) and apply `z1` to the second copy of each

```
import cast.
import recType.

module scott-rec/examples/nat-rec.
import scott/examples/nat.

NatRZ ◁ ⋆ → ⋆ = λ X: ⋆. ∀ Z: ⋆. ∀ S: ⋆. Z → S → Nat → X.
NatRS ◁ ⋆ → ⋆ = λ X: ⋆.
  ∀ Z: ⋆. ∀ S: ⋆. (Z → S → Z → S → Nat → X) → Z → S → Nat → X.
NatR ◁ ⋆ = ∀ X: ⋆.
  NatRZ ·X → NatRS ·X → NatRZ ·X → NatRS ·X → Nat → X.

zeroR ◁ NatR = Λ P. λ z1. λ s1. λ z2. λ s2. λ m. z1 z2 s2 m.
sucR ◁ NatR → NatR = λ n. Λ X. λ z1. λ s1. λ z2. λ s2. λ m. s1 (n ·X) z2 s2 m.

toNatR' ◁ Π n: Nat. ι x: NatR. {x ≃ n} =
  wkInductionNat ·(λ y: Nat. ι x: NatR. {x ≃ y})
    [zeroR , β{zero}]
    (λ m. Λ r. [ sucR (φ r.2 - r.1 {m} ) , β{suc m} ]).

toNatR ◁ Cast ·Nat ·NatR
= intrCast -(λ n. (toNatR' n).1) -(λ n. (toNatR' n).2).

recNatBase ◁ ∀ X: ⋆. X → NatRZ ·X
= Λ X. λ x. Λ Z. Λ S. λ z. λ s. λ n. x.

recNatStep ◁ ∀ X : ⋆. X → (Nat → X → X) → NatRS ·X
= Λ X. λ x. λ f. Λ Z. Λ S. λ n. λ z. λ s. λ m.
  f (pred m) (n z s z s (pred m)).

recNat ◁ ∀ X: ⋆. X → (Nat → X → X) → Nat → X
= Λ X. λ x. λ f. λ n.
  (elimCast -toNatR n) (recNatBase x) (recNatStep x f) (recNatBase x) (recNatStep x f) n.

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

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

Fig. 23. Recursive Scott naturals (`scott-rec/examples/nat-rec.ced`)

argument; for the recursor we shall define, `z1` will always be instantiated as a constant function ignoring its arguments. In the definition of `sucR` (which also η-contracts and erases to `suc`) the λ-bound `s1` expects first a term of type `NatRZ ·X → NatRS ·X → NatRZ ·X → NatRS ·X → Nat → X` (which is the type of the given `n ·X`), and it is also given the secondary copies `z2` and `s2`; in defining the recursor, `s1` and `s2` (resp. `z1` and `z2`) will always be instantiated with the same term, so in effect this gives `s1` a way to make recursive calls (via `z2` and `s2`) at each predecessor by passing down `z2` and `s2` as arguments to the predecessor `n ·X`, potentially to be further duplicated.

`toNatR'` and `toNatR`. We now prove `NatR` is a supertype of `Nat`. The conversion function `toNatR'` takes some number `n` and produces a term that both has type `NatR` and proves itself equal to `n`; recall that the Kleene trick (Section 2) allows any term to be a witness for a true equation. The conversion function is defined using the weak induction principle for Scott naturals: in the base case the goal is ι `x: NatR. {x ≃ zero}`, readily proven by `[ zeroR ,` β`{zero} ]`; in the successor case, the goal is to prove ι `x: NatR. {x ≃ suc m}`. In the first component of the intersection, we *make use of the erased induction hypothesis* `r` (whose type is ι `x: NatR. {x ≃ m}`) in order to cast `m` to the type `NatR` of `r.1` using φ and equation `r.2`, then apply `sucR` to this, resulting in an expression whose erasure is definitionally equal to the erasure of `suc m`. With `toNatR'` we may readily define `toNatR`, the cast from `Nat` to `NatR`. In the definition, for the second argument of `intrCast` we assume some `n` and must prove `{(toNatR n).1 ≃ n}`, which is given directly by `(toNatR n).2`.

`recNatBase`, `recNatStep`, and `recNat`. We can now define the recursor `recNat` for Scott naturals. Helper function `recNatBase` takes some `x`, a result for the base case, ignores its other three arguments, and simply returns `x`. Function `recNatStep` takes a base case `x` and function `f` of type `Nat → X → X`, and must produce an expression of

type `NatRS ·X`, which is itself a polymorphic function type. The quantified type variables `Z` and `S` in `NatRS ·X` hide resp. the occurrences of `NatRZ ·X` and `NatRS ·X` in the types of `n` (`Z → S → Z → S → Nat → X`), `z` (`Z`), and `s` (`S`); the last argument `m` of type `Nat` is intended to always be instantiated as the successor of `n`. We invoke `f` on the predecessor of `m` and the recursively computed result produced by invoking `n` on `z`, `s`, and `pred m`.

Finally, we put these definitions together in `recNat`. In its body, we cast the natural argument `n` to the type `NatR`, and for arguments provide it two copies each of `recNatBase x` and `recNatStep x f`, and a copy of itself. Notice, for example, that if `n` is non-zero then the first `recNatStep f` argument will be given a copy of itself (referred to by the λ-bound `s` in `recNatStep`). The recursor `recNat` satisfies the desired computation laws `recNatCompZ` and `recNatCompS` by definition (though only by β-equivalence, and not β-reduction alone).

Example As with the Parigot naturals defined in Section 5.1, we can define for recursive Scott naturals iterative functions such as addition (`add`) and recursive functions such as a summation of numbers $0...n$ (`sumFrom`):

```
add ◁ Nat → Nat → Nat = λ m. λ n. recNat n (λ p. λ s. suc s) m.
sumFrom ◁ Nat → Nat = λ m. recNat zero (λ p. λ s. suc (add p s)) m.
```

6.2. *Full induction for Scott-encoded data, generically*

In this section, we generalize the technique used to derive a recursor for Scott naturals in the previous section in two orthogonal ways: making it generic in a functor F, and making it *dependent* in order to support an induction principle. The code listing is given in Figures 24 and 25, which we walk through in detail.

`IndS` and `PrfAlg'`. As we did for Scott naturals in Section 6.1, the first step towards defining recursion principle for type `S` is to define some family of types capturing the

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

module scott-rec/induction
  (F : ⋆ → ⋆) (fmap : Fmap · F)
  {fmapId : FmapId · F fmap}{fmapCompose : FmapCompose · F fmap}.

import functorThms · F fmap -fmapId -fmapCompose.
import scott/encoding ·F fmap -fmapId -fmapCompose.

IndS ◁ (S → ⋆) → ⋆ → ⋆ = λ P: S → ⋆. λ Y: ⋆. ι x: S. Y → Y → P x.

unwrapFIndS ◁ ∀ P: S → ⋆. ∀ Y: ⋆. F ·(Wrap ·(IndS ·P ·Y)) → F ·S
= Λ P. Λ Y. fmap ·(Wrap ·(IndS ·P ·Y)) (λ w. (unwrap w).1).

PrfAlg' ◁ (S → ⋆) → ⋆ → ⋆ = λ P: S → ⋆.
  ∀ Y: ⋆. Π xs: F ·(Wrap ·(IndS ·P ·Y)). Y → P (in (unwrapFIndS xs)).

InductiveS ◁ S → ⋆ = λ s: S. ∀ P: S → ⋆. PrfAlg' ·P → PrfAlg' ·P → P s.

I ◁ ⋆ = ι x: S. InductiveS x.
fromI ◁ Cast ·I ·S = intrCast -(λ x. x.1) -(λ x. β).

wrapIndS ◁ ∀ P: S → ⋆. I → Wrap ·(IndS ·P ·(PrfAlg' ·P))
= Λ P. λ s. wrap [ s.1 , s.2 ·P ].

inI ◁ F ·I → I = λ xs.
  [ in (fcast -fromI xs)
  , Λ P. λ a1. λ a2.
    ρ ς (fmapId (λ x: I. unwrap (wrap x)) (λ _. β) xs) -
    ρ ς (fmapCompose (unwrap ·I) (wrap ·I) xs) -
    a1 (fmap (wrapIndS ·P) xs) a2 ] .
```

Fig. 24. Generic datatype `I` and operations (part 1) (`scott-rec/induction.ced`)

notion of a datatype taking two copies of interpretations for its constructors. This is done with the definition `IndS`, whose first parameter `P` is a property over `S` and whose second parameter `Y` shall always be instantiated with `PrfAlg' ·P`, a proof-algebra variant which recursively refers to `IndS` itself. Comparing to Figure 23, `Y` should be understood as an algebraic "grouping together" of the quantified variables `Z` and `S` for the base and step cases of recursion on naturals (in `PrfAlg'` we re-quantify over `Y`). As the goal is to

find an appropriate instantiation for `Y` so that every `x: S` can be cast to the type `Y` $\rightarrow$ `Y` $\rightarrow$ `P x`, we make use of this *ex ante* observation to define `IndS ·P ·Y` as a dependent intersection of these two types.

The definition of `PrfAlg'` (which corresponds to the type families `NatRZ` and `NatRS` together in Figure 23) describes a family of functions paramaterized by some property `P` over `S`; `PrfAlg' ·P` quantifies over `Y` (hiding recursive occurrences of `PrfAlg' ·P`), assumes some collection of subdata `xs` of type `F ·(Wrap ·(IndS ·P ·Y))`, and take an additional `Y` argument (to be given to subdata for further recursive calls), and will return a proof that `P` holds for the data constructed with `in` of `xs` (after unwrapping and projecting out the view of the subdata as having type `S` with `unwrapFIndS`).

`InductiveS` and `I`. With predicate `InductiveS`, we commit to the instantiation of `PrfAlg' ·P` (generalizing over `P`) for the parameter `Y` of `IndS`. With datatype `I`, we make the now-expected step of forming a dependent intersection type of terms `x: S` which also prove themselves `InductiveS`; it is clear that `I` is isomorphic to the type $\forall$ `P: S` $\rightarrow$ $\star$. `IndS ·P ·(PrfAlg' ·P)`. Finally, it is easy to define the cast `fromI` converting terms of type `I` to `S`.

Constructor `inI`. Next, we define the generic constructor `inI` for recursive Scott naturals. Given some collection `xs` of type `F ·I` (easily cast to the type `F ·S`), in the second component of the intersection defining `inI` we must show `InductiveS (in xs)`. To do this, for any `P` we assume `a1` and `a2` of type `PrfAlg' ·P`, and now goal is to show `P (in xs)`. Using the functor identity law, we rewrite this to

```
P (in (fmap (λ x. unwrap (wrap x)) xs))
```

Then, using the functor composition law this is further transformed to

```
P (in (fmap unwrap (fmap wrap xs)))
```

convertible with the type of the given expression `a1 (fmap (wrapIndS ·P) xs) a2`:

```
P (in (unwrapFIndS (fmap wrapIndS xs)))
```

In essence, we are exploiting definitional equality to exchange `wrap` and `unwrap` with the versions mentioned in the return type `a1`. These versions commit the proof principle of the second component of `I` to proving property `P` (`a1` expects to work with subdata of type `Wrap ·(IndS ·(PrfAlg' ·P) ·P)`). The last argument to `a1` is `a2`; we shall always instantiate these with the same term, meaning `a1` is given the capability of making recursive calls of itself via `a2`. Finally, notice that `inI` is definitionally equal to `in`.

```
toI' ◁ Π x: S. ι y: I. {y ≃ x} = λ x.
  wkInduction ·(λ x: S. ι y: I. {y ≃ x})
    (λ xs. Λ s. Λ eq.
    [mkI ◁ PrfS ·S ·(LiftS ·(λ x: S. ι y: I. {y ≃ x})) → I
      = λ p. elimWkSigma p (λ z. Λ ih.
        [eqz  ◁ {ih ≃ z} = (ih -z.1 -β).2] -
        [ind  ◁ InductiveS z.1 = ρ ς eqz - (ih -z.1 -β).1.2 ] -
        [ z.1 , φ eqz - ind {z} ])] -
    [ inI (fmap mkI xs) , β{in (fmap unwrap xs)} ])
    x .

toI ◁ Cast ·S ·I = intrCast -(λ x. (toI' x).1) -(λ x. (toI' x).2).

repackIndS ◁ ∀ P: S → ⋆. ∀ Y: ⋆. Y → Wrap ·(IndS ·P ·Y) → Sigma ·S ·P
= Λ P. Λ Y. λ y. λ w. mksigma (unwrap w).1 ((unwrap w).2 y y).

PrfAlg ◁ (S → ⋆) → ⋆ = λ P: S → ⋆.
  Π d: F ·(Sigma ·S ·P). P (in (fmap (proj1 ·S ·P) d)).

fromPrfAlg ◁ ∀ P: S → ⋆. PrfAlg ·P → PrfAlg' ·P
= Λ P. λ a. Λ Y. λ xs. λ y.
  ρ+ ς (fmapCompose (proj1 ·S ·P) (repackIndS ·P y) xs)
  - a (fmap (repackIndS ·P y) xs).

induction ◁ ∀ P: S → ⋆. PrfAlg ·P → Π s: S. P s
= Λ P. λ a. λ s. (elimCast -toI s).2 (fromPrfAlg a) (fromPrfAlg a).

rec ◁ ∀ X: ⋆. (F ·(Pair ·S ·X) → X) → S → X = Λ X. induction ·(λ x: S. X).
```

Fig. 25. Generic datatype `I` and operations (part 2) (`scott-rec/induction.ced`)

`toI'` and `toI`. We now establish that `I` is also a supertype of `S` (`fromI` shows that it is a subtype, so the two types classify precisely the same set of terms). Conversion function `toI'` (analogous to `toNatR'` in Figure 23) uses the weak induction principle of `S` (and the *Kleene trick*, c.f. Section 2) to return from some `x: S` some `I` that is equal to `x`. Within the body of the proof, `eq` proves that `x: S` is equal to `in (fmap unwrap xs)`, and the collection `xs` has subdata tupled together with erased proofs of the (lifted, see Figure 12) property that these are equal to terms of type `I`. Local definition `mkI` constructs from each such weak pair a term of type `I` (the Cedille construct `[`$x \triangleleft T$ `= t] - t'` introduces a local binding) – and, furthermore, is definitionally equal to `unwrap` (Figure 7), as it erases to λ `p. elimWkSigma p (`λ `s. s)`. Given the underlying `z: PreS ·S` and erased proof `ih: LiftS ·(`λ `x: S.` ι `y: I. {y` $\simeq$ `x}) z`, we show

— `z` and `ih` must be equal (`eqz`); and

— `z` must be `InductiveS` (`ind`, erasing to `ih`)

which together allows us to form a term of type `I`, where in the second component of the introduced intersection we use φ (Figure 1) to cast `z` to the type of `ind` via `eqz`.

This makes the definition of the cast `toI` easy, as `toI'` returns in a single argument the components required from each argument to `intrCast`.

`PrfAlg`, `fromPrfAlg`, and `induction`. Next, we show that we can convert any instance of the more mundane *Parigot-style P-proof F-algebra* `PrfAlg` (described in Section 5.2.2) to a `PrfAlg'`. This is done with `fromPrfAlg`. First, we assume `a: PrfAlg ·P`, type `Y`, subdata collection `xs: F ·(Wrap ·(IndS ·P ·Y))`, and `y: Y` (our handle for making recursive calls with the `PrfAlg' ·P` we are defining). The goal is now to prove:

```
P (in (unwrapFIndS xs))
```

which is convertible with the type:

```
P (in (fmap (λ x. proj1 (repackIndS y x)) xs))
```

Helper function `repackIndS` converts between the wrapping and unwrapping done by `PrfAlg` (which uses `Sigma`) and `PrfAlg'` (which uses `Wrap`), tupling together each sub-component with the recursively computed results (by providing the sub-component with two copies of `y`). Finally, we rewrite the expected type by the functor composition law:

```
P (in (fmap proj1 (fmap (repackIndS y) xs)))
```

which is the type of the given expression `a (fmap (repackIndS ·P y) xs)`.

Having `fromPrfAlg`, defining the induction principle `induction` is straightforward. Given some `PrfAlg ·P`, cast the given `s: S` to type `I` and provide it with two copies of `fromPrfAlg a`. The recursion principle `rec` is even simpler, a non-dependent usage of `induction`.

6.2.1. *Properties of `S`* Our generically derived inductive Scott-encoded data enjoys the same properties we showed for Parigot-encoded data (Section 5.2.3, wherein they are further elaborated upon): *call-by-name normalization* for closed terms (`norm`), the *cancellation laws* (now given for the standard formulation of the case-scheme as well as for the recursion scheme), *reflection law*, and *Lambek's lemma*. The second to last proof `unique` shows uniqueness of the universal mapping property of recursive algebra (`S`, `in`), from which it is an easy consequence that (`S`, `in`) is an *initial F-algebra*. Each of these has been proven within Cedille (Figure 26).

We conclude the discussion of Scott-encoded data by observing that particular datatypes can be defined using this generic derivation in almost exactly the same way as with the generic Parigot encoding. For instance, the definition Scott-encoded lists proceeds as described in Figure 22 of Section 5.2.4, modulo module imports and name qualifications.

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

module scott-rec/props
  (F: ⋆ → ⋆) (fmap: Fmap ·F)
  {fmapId: FmapId ·F fmap} {fmapCompose: FmapCompose ·F fmap}.

import functorThms ·F fmap -fmapId -fmapCompose.

import scott/encoding  ·F fmap -fmapId -fmapCompose.
import scott-rec/induction ·F fmap -fmapId -fmapCompose.

norm ◁ Cast ·S ·(AlgS ·S ·Unit → Unit)
= intrCast -(λ x. (unrollS x).1 ·Unit) -(λ _. β) .

case' ◁ ∀ X: ⋆. (F ·S → X) → S → X
= Λ X. λ a. case (λ xs. a (fmap (unwrap ·S) xs)).

-- cancellation for case scheme
caseComp' ◁ ∀ X: ⋆. Π a: F ·S → X. Π xs: F ·S. {case' a (in xs) ≃ a xs} = <..>

cancellation ◁ ∀ P: S → ⋆. Π xs: F ·S. Π a: PrfAlg ·P.
  {induction a (in xs) ≃ a (fmap (λ x. mksigma x (induction a x)) xs)} = <..>

reflect ◁ S → S = λ x: S. rec (λ xs. in (fmap (snd ·S ·S) xs)) x.
reflection ◁ Π x: S. {reflect x ≃ x} = <..>

lambek1 ◁ Π xs: F ·S. {out (in xs) ≃ xs} = <..>
lambek2 ◁ Π x: S. {in (out x) ≃ x} = <..>

unique ◁ FmapExt ·F fmap → ∀ X: ⋆. Π a: F ·(Pair ·S ·X) → X. Π h: S → X.
         Π hom: Π xs: F ·S. {h (in xs) ≃ a (fmap (λ x. mkpair x (h x)) xs)}.
         Π x: S. {h x ≃ rec a x} = <..>

fold ◁ ∀ X: ⋆. (F ·X → X) → S → X
= Λ X. λ a. λ x. rec (λ xs. a (fmap (snd ·S ·X) xs)) x .

initial ◁ FmapExt ·F fmap → ∀ X: ⋆. Π a: F ·X → X.
          Π h: S → X. Π hom: Π xs: F ·S. {h (in xs) ≃ a (fmap h xs)}.
          Π x: S. {h x ≃ fold a x}
= λ fext. Λ X. λ a. λ h. λ hom.
  unique fext ·X (λ xs. a (fmap (snd ·S ·X) xs)) h
    (λ xs. ρ+ (fmapCompose (snd ·S ·X) (λ x: S. mkpair x (h x)) xs) - hom xs).
```

Fig. 26. Properties of S (`scott-rec/props.ced`)

7. Related Work

Monotone inductive types. In (Matthes, 2002), Matthes employs Tarski's fixpoint theorem to motivate the construction of a typed λ-calculus with monotone recursive types. The gap between this order-theoretic result and the type theory is bridged by way of category theory, with the evidence that a type scheme is monotonic corresponding to the morphism-mapping component 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, then strong normalization of System F is preserved by extension with monotone iso-recursive types. Otherwise, he shows a counterexample to normalization.

In contrast, our approach can be characterized as an embedding of preoder theory *within* a type theory, with evidence of monotonicity corresponding to the mapping of a zero-cost cast over a type scheme. As mentioned in the introduction, deriving monotone recursive types within the type theory of Cedille has the benefit of guaranteeing that they enjoy precisely the same meta-theoretic properties as enjoyed by Cedille itself – no additional work is required.

Recursive F-algebras. 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, what is usually sought after is an efficient *recursion scheme* for such data, and the derivation in Section 3.3 does not on its own provide this. Independently, (Mendler, 1991; Geuvers, 1992) developed recursive F-algebras to give a category-theoretic semantics of the recursion scheme for inductive data, and (Geuvers, 1992; Matthes, 2002) use this notion in extending a typed λ-calculus with typing and reduction rules for an efficient datatype recursor. In our generic derivation of Parigot-encoded data, our weaker notion of recursive types (lacking

as it is either a recursion or iteration scheme) is sufficient for defining datatypes *directly* in terms of recursive F-algebras, guaranteeing an efficient recursor.

Recursor for Scott-encoded data. The type definition used for the (non-dependent) strongly normalizing recursor for Scott-encoded naturals in Section 6.1 is due to (Lepigre and Raffalli, 2019). The type system in which they 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 proofs" in a typing derivation. Roughly, the correspondence between their type system and that of Cedille's is so: both theories are Curry-style, enabling a rich subtyping relation, which in Cedille is internalized as `Cast`; and in defining recursor for Scott naturals, we replace the circular subtyping derivation with an inductive proof within Cedille itself that the subtyping relation holds. Section 6.2 generalizes their construction of an appropriate supertype for Scott-encoded data by making it *generic* (in an arbitrary functor F) and *dependent*.

We leave as future work the task of providing a more semantic (e.g. category-theoretic) account of the derivation of a recursor for Scott-encoded data.

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. In (Firsov et al., 2018), the approach for the Mendler encoding was refined 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.

Conclusion

We have shown in this paper how monotone recursive types with constant-time *roll* and *unroll* operations can be derived within the type theory of Cedille by applying Tarski's fixpoint theorem to a preorder on types constructed from zero-cost type coercions. As applications, we use the derived monotone recursive types to derive two recursive representations of data, the Parigot-style and Scott-style lambda encoding, *generically* in a signature functor F. These recursive representations enjoy constant-time data accessors, making them of practical significance. Furthermore, we gave for each encoding an induction principle and proof of a collection of properties arising from the categorical semantics of datatypes as initial F-algebras. That this can be achieved for the Scott encoding is itself rather remarkable, and the derivation uses an inductive proof that a zero-cost type coercion holds between the type of Scott-encoded data and a suitable supertype, described by (Lepigre and Raffalli, 2019).

Financial Aid

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

References

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

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

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

Firsov, D., Blair, R., and Stump, A. (2018). Efficient Mendler-Style Lambda-Encodings in Cedille. In Avigad, J. and Mahboubi, A., editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 235–252. Springer.

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

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

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

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

Jacobs, B. and Rutten, J. (2011). An introduction to (co) algebra and (co) induction.

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

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

Lassez, J.-L., Nguyen, V., and Sonenberg, E. (1982). Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112 – 116.

Lepigre, R. and Raffalli, C. (2019). Practical subtyping for Curry-style languages. *ACM Trans. Program. Lang. Syst.*, 41(1):5:1–5:58.

Matthes, R. (1999). Monotone Fixed-Point Types and Strong Normalization. In Gottlob, G., Grandjean, E., and Seyr, K., editors, *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings*, volume 1584 of *Lecture Notes in Computer Science*, pages 298–312. Springer.

Matthes, R. (2002). Tarski's fixed-point theorem and lambda calculi with monotone inductive types. *Synthese*, 133(1-2):107–129.

Meertens, L. (1992). Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424.

Mendler, N. P. (1991). Predictive type universes and primitive recursion. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 173–184.

Miquel, A. (2001). The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In Abramsky, S., editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359. Springer.

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

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

Parigot, M. (1992). Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356.

Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

Scott, D. (1962). A system of functional abstraction (1968). lectures delivered at university of california, berkeley. *Cal*, 63.

Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., New York, NY, USA.

Splawski, Z. and Urzyczyn, P. (1999). Type fixpoints: Iteration vs. recursion. In *ICFP*.

Stump, A. (2017). The Calculus of Dependent Lambda Eliminations. *J. Funct. Program.*, 27:e14.

Stump, A. (2018a). From realizability to induction via dependent intersection. *Ann. Pure Appl. Logic*, 169(7):637–655.

Stump, A. (2018b). Syntax and semantics of Cedille. *CoRR*, abs/1806.04709.

Stump, A. (2018c). Syntax and typing for Cedille core. *CoRR*, abs/1811.01318.

Stump, A. and Fu, P. (2016). Efficiency of lambda-encodings in total type theory. *J. Funct. Program.*, 26:e3.

Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309.

Wadler, P. (1990). Recursive types for free! unpublished.