Iteration and Coiteration Schemes for Higher-Order Nested Datatypes

Andreas Abel[a,*], Ralph Matthes[a], Tarmo Uustalu[b]

[a] *Institut für Informatik der Ludwig-Maximilians-Universität München Oettingenstraße 67, D-80538 München, Germany*

[b] *Institute of Cybernetics, Tallinn Technical University Akadeemia tee 21, EE-12618 Tallinn, Estonia*

Abstract

Several formulations of iteration and coiteration for inductive and coinductive constructors of higher kinds (higher-order nested datatypes) are put forth and studied. In particular, systems GMIt^ω and GIt^ω of generalized Mendler resp. conventional iteration are defined, which encompass the efficient folds of Martin et al. (2002) for rank-2 datatypes. In system GMIt^ω, the iterator and coiterator have a computational behavior similar to the general recursor, but their types guarantee termination. In system GIt^ω, monotonicity witnesses are used for a notion of monotonicity defined uniformly for all kinds. Strong normalization is proved by providing an embedding of the basic Mendler-style system $\underline{\mathsf{MIt}}^\omega$ into system F^ω.

Key words: Nested Datatypes, Generalized Folds, Efficient Folds, Iteration, Coiteration, System F^ω, Higher-Order Polymorphism, Strong Normalization

1 Introduction and Overview

1.1 Motivation

Iteration on natural numbers $\mathbb{N}$ is very well understood. One may define natural numbers by iteration on $\mathbb{N}$, hence gets primitive recursive functions, or

* Corresponding author.

Email addresses: `abel@informatik.uni-muenchen.de` (Andreas Abel), `matthes@informatik.uni-muenchen.de` (Ralph Matthes), `tarmo@cs.ioc.ee` (Tarmo Uustalu).

define even functionals by iteration on $\mathbb{N}$, arriving at primitive recursive function(al)s in the sense of Gödel. Primitive recursion extensionally does not give more function(al)s than iteration. A wealth of more advanced iteration principles such as course-of-value iteration, iteration with substitution of parameters, multiple iteration and combinations thereof can be reduced to plain iteration.

Natural numbers form the most fundamental datatype. Although in principle expressible via natural numbers, there is a huge body of useful datatypes, starting with the booleans, homogeneous lists (over some given datatype) and binary wellfounded trees. Many constructions are possible, such as wellfounded trees branching over a given datatype and labelled forests where the latter even require simultaneous datatypes. The datatypes can be classified according to the shape of the underlying fixed-point equation: polynomial, strictly positive, non-strictly positive. One can go even further and only require that the datatype arises as the least fixed-point of a monotone operator, hence discarding the analysis of shapes altogether.

What is iteration or primitive recursion over such a datatype? A category-theoretic view is most helpful: The notion of initial algebra yields the iteration principles, variations on it the principle of primitive recursion and many other notions which formerly were only known for natural numbers. For the sake of programming, these principles are truly different and cannot be reduced to basic iteration since the operational behavior cannot be simulated by the extensional reductions. Hence, the same functions are being programmed, but not the same algorithms are expressed for computing them. For example, the iterative predecessor on natural numbers takes linear time while the primitive recursive one is constant-time.

The theoretic framework for programming with these monotone inductive types is second-order lambda calculus. Higher datatypes come from using higher-order polymorphism, i. e., Girard's System F^ω. Already the rank-2 case of fixed-points of transformations of type transformers has only recently attracted the interest of the functional programming community. The least such fixed-points may be conceived as a uniform inductive definition of a family of types: All of the family members are simultaneously defined by an inductive definition, parametrically uniformly in the argument type by which the family is indexed. Note that this does not include definitions of a family of types by iteration on the build-up of the argument type, but it does allow a reference in the definition of the type, indexed by A, to the family member, indexed by, say $1 + A$, for all types A. One may wonder how to program with these data structures, using iteration mechanisms that have a clear logical underpinning (exploiting the well-known Curry-Howard isomorphism) and that extend to rank-n type constructors, for arbitrary n.

The main technical problem one faces with the formulation of an iteration scheme for higher-order datatypes, is the absence of a canonical definition of admissible constructor for forming the least fixed-points. The search for an optimal type-based approach leads to several plausible notions of monotonicity. The first thought would be to take the chosen notion of monotonicity as admissibility criterion for the formation of the least fixed-point. However, it suffices to require monotonicity when applying the iteration scheme, not for the formation itself, which will be unrestricted in our systems. In an even more radical line of thought, *every* constructor is allowed for the formation and iteration, but the typing requirements for the step term of the iteration will nevertheless guarantee termination. This is called Mendler's style, the first one conventional style.

The most clear-cut formulation does not work so well in typical examples, which led to the proposal of more refined programming schemes—generalized folds and efficient folds—whose difference from plain iteration is only visible at higher kinds. Hence, all of these variations on plain iteration are orthogonal to the variations listed above for the primitive recursive functions. Viewed from the first-order perspective, this article entirely confines itself to plain iteration—as far as it concerns least fixed-points.

In category theory, dualization is an easy task: one just reverses all arrows. For our datatypes, this yields "codatatypes" which represent greatest fixed-points, and whose logic-based programming discipline is called coiteration. In a lambda calculus, such as our framework, dualization is not as trivial as in category theory. Therefore, it is interesting to see the exact formulation of the coiteration schemes as well and get a feeling for them. This also greatly relies on the fact that, typically, the examples are not dualized: If one passes from the least fixed-point of some constructor to the greatest one of the same (and not the dual) constructor, new programming challenges arise.

1.2 Technical contribution

In this article, we take a close look at several possible extensions of system F with iteration and coiteration for (co)inductive constructors of higher kinds, each of which is strongly normalizable. We primarily work with two notions of constructor containment (corresponding to natural transformations in category theory), a basic and a more refined one, which imply different properties for the derived notions of monotonicity of a constructor. The systems we wish to promote most are the corresponding Mendler systems, called $\underline{\mathsf{MIt}}^\omega$ and GMIt^ω, with GMIt^ω being the most powerful system we discuss, in terms of direct expressivity. These systems capture the efficient folds of Martin et al. (2002), a generalized iteration scheme for rank-2 inductive constructors. But still we also look at the corresponding conventional-style systems, called $\underline{\mathsf{It}}^\omega$ and GIt^ω, which employ monotonicity explicitly. We relate the systems via typing- and reduction-preserving embeddings. In particular, $\underline{\mathsf{MIt}}^\omega$ is embedded into F^ω and GMIt^ω is embedded into $\underline{\mathsf{MIt}}^\omega$.

This article is an extended and reworked version of our conference paper Abel et al. (2003) which mostly discussed system GMIt^ω (called MIt^ω in that article). New in this article are the basic systems $\underline{\mathsf{MIt}}^\omega$ and $\underline{\mathsf{It}}^\omega$ as well as the discussion of GIt^ω and the definability of efficient folds within GIt^ω. The discussion of It^ω (a subsystem of GIt^ω) and some examples are taken from Abel and Matthes (2003).

1.3 Organization

The article is organized as follows. In Section 2, we recapitulate the system F^ω of higher-order polymorphism. Section 3 is devoted to an introduction to iteration à la Mendler and to a presentation of our basic Mendler-style system $\underline{\mathsf{MIt}}^\omega$, including its embedding into F^ω. In Section 4, we switch to a more refined notion of constructor containment, which takes us to system GMIt^ω of refined and generalized Mendler (co)iteration, embeddible into $\underline{\mathsf{MIt}}^\omega$. In Sections 5 through 7, we look at the corresponding conventional-style systems and discuss in detail the properties the corresponding notions of monotonicity. In Section 8, we give some advanced examples where we use the different systems in a combined manner. In Section 9, we present a syntactic version of the efolds of Martin et al. (2002) and show them to be straightforwardly definable within GIt^ω.

Examples form an important part of the article since they allow an intuitive comparison of the expressiveness of the systems. Therefore, the same programming tasks are dealt with several times. The summation of a powerlist is treated in the examples 4 (in $\underline{\mathsf{MIt}}^\omega$), 7 (in GMIt^ω), 10 (in $\underline{\mathsf{It}}^\omega$), 13 (in GIt^ω) and 19 (via efolds). Redecoration of triangular matrices first comes in the easier non-wellfounded case (example 6 within $\underline{\mathsf{MIt}}^\omega$ and 9 within GMIt^ω), in the advanced examples section for the wellfounded case (example 17). Free variable renaming of de Bruijn lambda terms is done in example 11 within It^ω, in example 21 with efolds and, in the advanced examples section, for the extension by explicit substitutions in example 15. The corresponding representation of substitution is achieved in example 14 within GIt^ω and again in example 21 with efolds. The other nine examples show streams of natural numbers, leaf retrieval in powerlists by help of streams, summation of elements of a "bushy list", powerlist reversal, the resolution of explicit substitutions and the calculation of the efolds schema for powerlists and de Bruijn terms.

2 System F^ω

Our development of higher-order datatypes takes place within system F^ω in Curry-style, enriched with binary sums and products, unit type and existential quantification over constructors. Although we choose a human-friendly notation of variables, we actually mean the nameless version à la de Bruijn which identifies α-equivalent terms. (Capture-avoiding) Substitution of an expression e for a variable x in expression f is denoted by $f[x := e]$.

Kinds are generated from the kind $*$ for types by the binary function kind constructor $\to$, and are denoted by the letter κ:

$$\kappa \quad ::= * \mid \kappa \to \kappa'$$

$$\begin{aligned} \mathsf{rk}(*) &:= 0 \\ \mathsf{rk}(\kappa \to \kappa') &:= \max(\mathsf{rk}(\kappa)+1, \mathsf{rk}(\kappa')) \end{aligned}$$

The *rank* of kind κ is computed by $\mathsf{rk}(\kappa)$. We introduce abbreviations for some special kinds: $\kappa 0 = *$, *types*, $\kappa 1 = * \to *$, *unary type transformers* and $\kappa 2 = (* \to *) \to (* \to *)$ *unary transformers of type transformers.* Then, $\mathsf{rk}(\kappa i) = i$ for $i \in \{0,1,2\}$.

Note that each kind κ' can be uniquely written as $\vec{\kappa} \to *$, where we write $\vec{\kappa}$ for the sequence $\kappa_1, \ldots, \kappa_n$ and set $\vec{\kappa} \to \kappa := \kappa_1 \to \ldots \to \kappa_n \to \kappa$, assuming that $\to$ associates to the right. Provided another sequence $\vec{\kappa}' = \kappa'_1, \ldots, \kappa'_n$ of the same length, i.e., $|\vec{\kappa}'| = |\vec{\kappa}|$, set $\vec{\kappa} \to \vec{\kappa}' := \kappa_1 \to \kappa'_1, \ldots, \kappa_n \to \kappa'_n$. This last abbreviation does not conflict with the abbreviation $\vec{\kappa} \to \kappa$ due to the required $|\vec{\kappa}'| = |\vec{\kappa}|$.

Constructors. (Denoted by uppercase latin letters.) Metavariable X ranges over an infinite set of constructor variables.

$$\begin{aligned} A, B, C, F, G ::= {} & X \mid \lambda X.F \mid F\,G \mid \forall X^\kappa.\,A \mid \exists X^\kappa.\,A \mid A \to B \\ & \mid A + B \mid A \times B \mid 1 \end{aligned}$$

Note that the constructors are given in Curry style although quantification is written with kind annotation. This is because the semantics of the quantifiers needs the kind κ. The constructor $\forall X^\kappa.\,A$ should be conceived as an abbreviation for $\forall^\kappa \lambda X.A$ where the lambda-abstracted variable is not kind annotated. Sometimes, "_" will be used as name of a constructor variable which never occurs free in a constructor, hence for void abstraction $\lambda\_.F$ and void quantification.

Objects (Terms). (Denoted by lowercase letters.) The metavariable x ranges over an infinite set of object variables.

$$\begin{aligned} r, s, t ::= {} & x \mid \lambda x.t \mid r\, s \mid \mathsf{inl}\, t \mid \mathsf{inr}\, t \mid \mathsf{case}\,(r,\ x.\, s,\ y.\, t) \\ & \mid \langle\rangle \mid \langle t_0, t_1 \rangle \mid r.0 \mid r.1 \mid \mathsf{pack}\, t \mid \mathsf{open}\,(r,\ x.\, s) \end{aligned}$$

Most term constructors are standard; "pack" introduces and "open" eliminates existential quantification, see below. The terms $r.0$ and $r.1$ are left and right projection of r, respectively. The identity $\lambda x.x$ will be denoted by id. We write $f \circ g$ for function composition $\lambda x.\, f\,(g\, x)$. Application $r\, s$ associates to the left, hence $r\vec{s} = (\ldots(r\, s_1)\ \ldots\ s_n)$ for $\vec{s} = s_1, \ldots, s_n$.

Contexts. Variables in a context Γ are assumed to be distinct.

$$\Gamma ::= \cdot \mid \Gamma, X^\kappa \mid \Gamma, x\!:\!A$$

Judgments. (The first two will be defined simultaneously, the third one based on these.)

$\Gamma\ \mathsf{cxt}$	Γ is a wellformed context
$\Gamma \vdash F : \kappa$	F is a wellformed constructor of kind κ in context Γ
$\Gamma \vdash t : A$	t is a wellformed term of type A in context Γ

Wellformed contexts. $\Gamma\ \mathsf{cxt}$

$$\frac{}{\cdot\ \mathsf{cxt}} \qquad \frac{\Gamma\ \mathsf{cxt}}{\Gamma, X^\kappa\ \mathsf{cxt}} \qquad \frac{\Gamma \vdash A : *}{\Gamma, x\!:\!A\ \mathsf{cxt}}$$

Hence, contexts declare kinds for constructor variables and types (not arbitrary constructors!) for object variables.

Wellkinded constructors. $\Gamma \vdash F : \kappa$

$$\frac{X^\kappa \in \Gamma \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash X : \kappa} \qquad \frac{\Gamma, X^\kappa \vdash F : \kappa'}{\Gamma \vdash \lambda X.F : \kappa \to \kappa'} \qquad \frac{\Gamma \vdash F : \kappa \to \kappa' \qquad \Gamma \vdash G : \kappa}{\Gamma \vdash F\, G : \kappa'}$$

$$\frac{\Gamma, X^\kappa \vdash A : *}{\Gamma \vdash \forall X^\kappa.\, A : *} \qquad \frac{\Gamma, X^\kappa \vdash A : *}{\Gamma \vdash \exists X^\kappa.\, A : *} \qquad \frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A \to B : *}$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A + B : *} \qquad \frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A \times B : *} \qquad \frac{\Gamma\ \mathsf{cxt}}{\Gamma \vdash 1 : *}$$

The rank of a constructor is given by the rank of its kind. If no kinds are given and cannot be guessed from the context of discourse, we assume $A, B, C, D : *$, $G, H, X, Y : \kappa 1$ and $F : \kappa 2$. If the context is clear (by default, we take the empty context "$\cdot$"), we write $F : \kappa$ for $\Gamma \vdash F : \kappa$. Write $\mathsf{Id} := \lambda X.X : \kappa \to \kappa$ for the identity constructor. Constructor application associates to the left, i. e., $F\,X_1\,\ldots\,X_n = (\ldots(F\,X_1)\,\ldots)\,X_n$. Setting $\vec{X} := X_1, \ldots, X_n$, the constructor $F\,X_1\,\ldots\,X_n$ is also written as $F\vec{X}$. Sums and products can be extended to all kinds: For $\kappa = \vec{\kappa} \to *$ with $|\vec{\kappa}| = n$, set

$$+^\kappa := \lambda F \lambda G \lambda \vec{X}.\ F\vec{X} + G\vec{X} \quad \text{and} \quad \times^\kappa := \lambda F \lambda G \lambda \vec{X}.\ F\vec{X} \times G\vec{X}.$$

Both these constructors have kind $\kappa \to \kappa \to \kappa$. If no ambiguity arises, the superscript is omitted. Finally, abbreviate constructor composition $\lambda Z.\, X\,(Y\,Z)$ by $X \circ Y$.

Equivalence on well-kinded constructors. Equivalence $F = F'$ for well-kinded constructors F and F' is given as the compatible closure of the following axiom.

$$(\lambda X.F)\,G =_\beta F[X := G]$$

We identify well-kinded constructors up to equivalence, which is a decidable relation due to normalization and confluence of simply-typed λ-calculus (where our constructors are the terms and our kinds are the types of that calculus).

As a consequence of this identification, well-kinded constructor composition $\circ$ is associative.

Welltyped terms. $\Gamma \vdash t : A$

$$\frac{(x\!:\!A) \in \Gamma \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x\!:\!A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\,s : B}$$

$$\frac{\Gamma, X^\kappa \vdash t : A}{\Gamma \vdash t : \forall X^\kappa.A} \qquad \frac{\Gamma \vdash t : \forall X^\kappa.A \qquad \Gamma \vdash F : \kappa}{\Gamma \vdash t : A[X := F]}$$

$$\frac{\Gamma \vdash t : A[X := F] \qquad \Gamma \vdash F : \kappa}{\Gamma \vdash \mathsf{pack}\,t : \exists X^\kappa.A} \qquad \frac{\Gamma \vdash r : \exists X^\kappa.A \qquad \Gamma, X^\kappa, x\!:\!A \vdash s : C}{\Gamma \vdash \mathsf{open}\,(r,\ x.\,s) : C}$$

$$\frac{\Gamma\ \mathsf{cxt}}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : *}{\Gamma \vdash \mathsf{inl}\,t : A + B} \qquad \frac{\Gamma \vdash t : B \qquad \Gamma \vdash A : *}{\Gamma \vdash \mathsf{inr}\,t : A + B}$$

$$\frac{\Gamma \vdash r : A + B \qquad \Gamma, x\!:\!A \vdash s : C \qquad \Gamma, y\!:\!B \vdash t : C}{\Gamma \vdash \mathsf{case}\,(r,\ x.\,s,\ y.\,t) : C}$$

$$\frac{\Gamma \vdash t_0 : A_0 \qquad \Gamma \vdash t_1 : A_1}{\Gamma \vdash \langle t_0, t_1\rangle : A_0 \times A_1} \qquad \frac{\Gamma \vdash r : A_0 \times A_1 \quad i \in \{0,1\}}{\Gamma \vdash r.i : A_i}$$

As for wellkinded constructors, we write $t : A$ for $\Gamma \vdash t : A$ if the context is clear (by default, we again take the empty context "$\cdot$").

Logical equivalence. If $\Gamma \vdash A : *$ and $\Gamma \vdash B : *$, then A and B are *logically equivalent* in context Γ iff there are terms r, s such that $\Gamma \vdash r : A \to B$ and $\Gamma \vdash s : B \to A$. If the context Γ is clear from the context of discourse, we just write $A \leftrightarrow B$ for logical equivalence of A and B.

Reduction. The one-step reduction relation $t \longrightarrow t'$ between terms t and t' is defined as the closure of the following axioms under all term constructors.

$$
\begin{array}{lll}
(\lambda x.t)\, s & \longrightarrow_\beta t[x := s] & \\
\mathsf{case}\,(\mathsf{inl}\, r,\ x.\, s,\ y.\, t) & \longrightarrow_\beta s[x := r] & \\
\mathsf{case}\,(\mathsf{inr}\, r,\ x.\, s,\ y.\, t) & \longrightarrow_\beta t[y := r] & \\
\langle t_0, t_1 \rangle.i & \longrightarrow_\beta t_i & \text{if } i \in \{0, 1\} \\
\mathsf{open}\,(\mathsf{pack}\, t,\ x.\, s) & \longrightarrow_\beta s[x := t] &
\end{array}
$$

We denote the transitive closure of $\longrightarrow$ by $\longrightarrow^+$ and the reflexive-transitive closure by $\longrightarrow^*$.

The defined system is a conservative extension of system F^ω. Reduction is type-preserving, confluent and strongly normalizing.

3 System $\underline{\mathsf{MIt}}^\omega$ of Basic Mendler Iteration and Coiteration

In this section we will introduce System $\underline{\mathsf{MIt}}^\omega$, a conservative extension of F^ω, which provides schemes of iteration and coiteration for higher-order datatypes, also called heterogeneous, nested or rank-n datatypes ($n \geq 2$). But first we will recall Mendler iteration for first-order (resp. homogeneous or rank-1) types which we then generalize to higher ranks.

3.1 Mendler Iteration for Rank-1 Inductive Types

Recall a standard example for homogeneous inductive types: the type of lists $\mathsf{List}(A) : *$ over some element type A, which has two constructors $\mathsf{nil} : \mathsf{List}(A)$ and $\mathsf{cons} : A \to \mathsf{List}(A) \to \mathsf{List}(A)$. Types like this one are called homogeneous because the argument to the type operator List is invariant in the type of the datatype constructors. In our case the argument is always the fixed type A. We will later see examples of heterogeneous types were the argument A to the type operator, call it T, varies in the different occurrences of T in the type of a constructor. The argument A can even contain T itself; in this case we speak of truly nested datatypes.

We favor a view on inductive types motivated from category theory:[1] $\mathsf{List}(A)$ is defined as the least fixed-point of an operator $\mathsf{ListF}(A) := \lambda X.\, 1 + A \times X$ and we write $\mathsf{List}(A) := \mu(\mathsf{ListF}(A))$. There is just a single constructor $\mathsf{in} : \mathsf{ListF}(A)(\mathsf{List}(A)) \to \mathsf{List}(A)$ for lists. The functions nil and cons can be defined in terms of in:

$$\begin{array}{lll} \mathsf{nil} & := \mathsf{in}\,(\mathsf{inl}\,\langle\rangle) & : \mathsf{List}(A) \\ \mathsf{cons} & := \lambda a \lambda as.\, \mathsf{in}\,(\mathsf{inr}\,\langle a,\ as\rangle) & : A \to \mathsf{List}(A) \to \mathsf{List}(A) \end{array}$$

To define operations on lists we need a means of recursion. In our theory all functions should be total, hence we restrict to a scheme which we call *Mendler iteration* for reasons that become apparent later. One function which falls into this scheme is the general-purpose function foldr. It is part of the Haskell standard library and could be defined like this:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr c n = foldr'
    where foldr' []       = n
          foldr' (a : as) = c a (foldr' as)
```

The recursive part of this definition is the interior function `foldr'` which arises as the fixed-point of a functional $s = \lambda foldr'.$ <*body of* `foldr'`> such that the equation `foldr'` $= s$ `foldr'` holds. (The name "s" stands for "step term".) In general it is undecidable whether solutions of such fixed-point equations are total functions. However, totality can be guaranteed by standard methods if the term s has a special shape. In our case, a recursive call of `foldr'` occurs only with argument "`as`" which is a direct subterm of the input `(a : as)`. This already ensures termination of `foldr'`. Furthermore, the sublist "`as`" is only used as an argument to `foldr'`, hence the function is even *iterative*.

Mendler (1991) first observed that by a certain polymorphic typing of s one can determine the fixed-point of s to be an iterative function. The trick is to assign a fresh type X to the direct subcomponent "`as`" and restrict applications of the recursive function `foldr'` to arguments of this type. This has a twofold effect: Since X is a type we know nothing of, necessarily it holds that

(1) the component "`as`" can neither be further analyzed nor used in any way besides as an argument to `foldr'`, and
(2) the function `foldr'` cannot be called recursively unless applied to "`as`".

Mendler's trick is implemented by requiring s to be of type $(X \to B) \to$

[1] The study of categorical datatypes goes back to Hagino (1987).

$\mathsf{ListF}(A)X \to B$ for a fresh type variable X. The first parameter of s is the name *foldr′* for the recursive function whose application is now restricted to input of type X; the second parameter will later be bound to the term t of the input "$\mathsf{in}\,t$" of `foldr'`, but is now by its type $\mathsf{ListF}(A)X$ restricted to be either the canonical inhabitant $\langle\rangle$ of type 1 for the case of the empty list nil as input to `foldr'`, or a pair of a head element of type A and a tail of type X for the cons case. In the nil case, s somehow has to produce a result of type B, in the cons case, s can—among other possibilities—apply *foldr′* to the tail in order to arrive at a result of type B.

We call the respective fixed-point combinator which produces iterative functions *Mendler iterator*—written $\underline{\mathsf{MIt}}(s)$ for a step term s. It has the following reduction behavior:

$$\underline{\mathsf{MIt}}(s)\,(\mathsf{in}\,t) \longrightarrow_\beta s\,\underline{\mathsf{MIt}}(s)\,t$$

During reduction, the type variable X is substituted by the inductive type μF, in our example $\mathsf{List}(A)$. The fixed-point type μF is unrolled into $F\,(\mu F)$, hence, the constructor in is dropped. On the level of terms, this is exactly what distinguishes Mendler iteration from general recursion. The reduction can only take place when the constructor "in" is present. This and the fact that it is removed by reduction makes "in" act as a guard for unrolling recursion and ensures strong normalization, as we will prove later.

Summing up, we can augment the higher-order polymorphic lambda-calculus with rank-1 iteration by adding the following constants, typing and reduction rules:

Formation. $\quad \mu \,:\, (* \to *) \to *$

Introduction. $\quad \mathsf{in} : \forall F^{*\to*}.\, F\,(\mu F) \,\to\, \mu F$

Elimination.
$$\frac{\Gamma \vdash F : * \to * \qquad \Gamma \vdash B : * \qquad \Gamma \vdash s : \forall X^*.\, (X \to B) \to F\,X \to B}{\Gamma \vdash \underline{\mathsf{MIt}}(s) : \mu F \to B}$$

Reduction. $\quad \underline{\mathsf{MIt}}(s)\,(\mathsf{in}\,t) \longrightarrow_\beta s\,\underline{\mathsf{MIt}}(s)\,t$

3.2 Mendler Coiteration for Rank-1 Coinductive Types

In the previous section we considered least fixed points of recursive type equations. If we consider greatest fixed points instead, we obtain coinductive types νF which are dual to inductive types in the category-theoretic sense. Hence, obtaining rules for Mendler-style coinductive types is a matter of reversing some arrows.

$$
\begin{array}{ll}
\text{Formation.} & \nu \quad : (* \to *) \to * \\
\text{Elimination.} & \mathsf{out} : \forall F^{* \to *}.\ \nu F \ \to\ F\,(\nu F) \\
\text{Introduction.} & \dfrac{\begin{array}{l}\Gamma \vdash F : * \to * \\ \Gamma \vdash A : * \\ \Gamma \vdash s : \forall X^{*}.\ (A \to X) \to A \to F\,X\end{array}}{\Gamma \vdash \underline{\mathsf{MCoit}}(s) : A \to \nu F} \\
\text{Reduction.} & \mathsf{out}\ (\underline{\mathsf{MCoit}}(s)\ t) \longrightarrow_{\beta} s\ \underline{\mathsf{MCoit}}(s)\ t
\end{array}
$$

Dually to the general constructor in for inductive types, coinductive types possess a general destructor out which triggers unrolling of the coiterator $\underline{\mathsf{MCoit}}$ in the reduction rule. Since elements of coinductive types can be infinite objects, they need to be constructed by a recursive process—this gives some intuition why coinductive types are introduced by the coiterator.

Example 1 (Streams) The most popular coinductive type is the type of infinite streams over some element type. In the system of Mendler coiteration, it can be defined as follows:

$$
\begin{array}{lll}
\mathsf{Stream} & := \lambda A.\, \nu(\lambda X.\, A \times X) & : * \to * \\
\mathsf{head} & := \lambda r.\, (\mathsf{out}\, r).0 & : \forall A.\, \mathsf{Stream}\, A \to A \\
\mathsf{tail} & := \lambda r.\, (\mathsf{out}\, r).1 & : \forall A.\, \mathsf{Stream}\, A \to \mathsf{Stream}\, A
\end{array}
$$

Example 2 (Sequence of Natural Numbers) Assume a type Nat of natural numbers with addition "+" and numerals $0, 1, 2, \ldots$. We can define the sequence of all natural numbers starting at a number n as a stream using Mendler coiteration:

$$
\begin{array}{rl}
\mathsf{nats} := & \underline{\mathsf{MCoit}}(\lambda nats \lambda n.\, \langle n,\ nats\,(n+1)\rangle) \\
: & \mathsf{Nat} \to \mathsf{Stream}\,\mathsf{Nat}
\end{array}
$$

3.3 Heterogeneous Datatypes

In contrast to the polymorphic types given in the previous sections, there are recursive type constructors whose arguments vary in different occurrences in their defining equation. For instance, consider the following Haskell types:

```
PList a = Zero a | Succ (PList (a, a))
Bush  a = Nil    | Cons a (Bush (Bush a))
```

```
Lam   a = Var a  | App (Lam a) (Lam a) | Abs (Lam (Maybe a))
```

The first definition, `PList`, is the type of *powerlists* (Bird et al., 2000) resp. *perfectly balanced, binary leaf trees* (Hinze, 2000). Notice that the argument to the type transformer `PList` on the right hand side is not simply the type variable "a", but `(a, a)` which is the Haskell notation for the Cartesian product `a` $\times$ `a`. This is why `PList` is called a *heterogeneous* type in contrast to *homogeneous* types like List where in the definition the argument is always the same type variable.

The second line defines "bushes" (Bird and Meertens, 1998) which are like lists except that the element type gets bigger as we are traversing the list from head to tail. On the right hand side of the defining equation the type transformer `Bush` occurs as part of the argument to itself. We will speak of a type with this property as a *truly nested type*, in contrast to the term *nested type* which in the literature denotes any heterogeneous type.

Finally, the third type `Lam a` is inhabited by de Bruijn representations of untyped lambda terms over a set of free variables "a". This type has been studied by Altenkirch and Reus (1999) and Bird and Paterson (1999b); a precursor of this type has been considered already by Pfenning and Lee (1989) and Pierce et al. (1989). The constructor for lambda-abstraction `Abs` expects a term over the extended set of free variables `Maybe a`, which is the Haskell representation of the sum type $1 + \mathsf{a}$. The disjoint sum reflects the choice for a bound variable under the abstraction: either it is the variable freshly bound (left injection into the unit set "1") or it is one of the variables that have been available already (right injection into "a").

We will encounter all of these datatypes in examples later. For now we are interested to encode these types in a suitable extension of F^ω. The encoding is possible if a combinator $\mu^{\kappa 1} : (\kappa 1 \to \kappa 1) \to \kappa 1$ for least fixed-point types of rank 1 is present. (Recall that $\kappa 1 := * \to *$.) In the following we give representations of these three types as least fixed points μF of type transformer transformers $F : \kappa 2$.

$$
\begin{array}{lll}
\mathsf{PListF} & := \lambda X \lambda A.\ A + X\,(A \times A) & : \kappa 2 \\
\mathsf{PList} & := \mu^{\kappa 1}\,\mathsf{PListF} & : \kappa 1 \\[1ex]
\mathsf{BushF} & := \lambda X \lambda A.\ 1 + A \times X\,(X\,A) & : \kappa 2 \\
\mathsf{Bush} & := \mu^{\kappa 1}\,\mathsf{BushF} & : \kappa 1 \\[1ex]
\mathsf{LamF} & := \lambda X \lambda A.\ A + (X\,A \times X\,A + X\,(1 + A)) & : \kappa 2 \\
\mathsf{Lam} & := \mu^{\kappa 1}\,\mathsf{LamF} & : \kappa 1
\end{array}
$$

Similarly to the rank-1 case we just have one general datatype constructor $\mathsf{in}^{\kappa 1}$ which rolls an inhabitant of $F\,(\mu^{\kappa 1} F)$ into the fixed point $\mu^{\kappa 1} F$. Note, however, that $\mu^{\kappa 1} F$ is not a type but a type constructor, hence, we need a polymorphic datatype constructor $\mathsf{in}^{\kappa 1} : \forall A.\, F\,(\mu^{\kappa 1} F)\, A \to \mu^{\kappa 1} F A$. Now, we are ready to define the usual datatype constructors for the considered heterogeneous datatypes:

$$
\begin{array}{lll}
\mathsf{zero} & := \lambda a.\, \mathsf{in}^{\kappa 1}\,(\mathsf{inl}\ a) & : \forall A.\, A \to \mathsf{PList}\ A \\
\mathsf{succ} & := \lambda l.\, \mathsf{in}^{\kappa 1}\,(\mathsf{inr}\ l) & : \forall A.\ \mathsf{PList}(A \times A) \to \mathsf{PList}\, A \\
\mathsf{bnil} & := \mathsf{in}^{\kappa 1}\,(\mathsf{inl}\,\langle\rangle) & : \forall A.\ \mathsf{Bush}\ A \\
\mathsf{bcons} & := \lambda a \lambda b.\, \mathsf{in}^{\kappa 1}\,(\mathsf{inr}\,\langle a, b\rangle) & : \forall A.\, A \to \mathsf{Bush}\,(\mathsf{Bush}\ A) \to \mathsf{Bush}\ A \\
\mathsf{var} & := \lambda a.\, \mathsf{in}^{\kappa 1}\,(\mathsf{inl}\ a) & : \forall A.\, A \to \mathsf{Lam}\ A \\
\mathsf{app} & := \lambda t_1 \lambda t_2.\, \mathsf{in}^{\kappa 1}\,(\mathsf{inr}\,(\mathsf{inl}\,\langle t_1, t_2\rangle)) & : \forall A.\ \mathsf{Lam}\ A \to \mathsf{Lam}\ A \to \mathsf{Lam}\ A \\
\mathsf{abs} & := \lambda r.\, \mathsf{in}^{\kappa 1}\,(\mathsf{inr}\,(\mathsf{inr}\ r)) & : \forall A.\ \mathsf{Lam}\,(1 + A) \to \mathsf{Lam}\ A
\end{array}
$$

Our aim is to define iteration for nested datatypes, a quest which recently has attracted some interest in the functional programming community (Bird and Paterson, 1999a; Hinze, 2000; Martin et al., 2002). In the remainder of this section we will show how to generalize Mendler iteration to higher ranks and point out some difficulties with this approach. In the remainder of this article we will present refined iteration schemes which overcome the shortcomings of plain Mendler iteration.

3.4 Mendler Iteration for Higher Ranks

To generalize Mendler iteration from types to type constructors, we introduce a syntactic notion of natural transformations $F \subseteq^{\kappa} G$ from constructor $F : \kappa$ to $G : \kappa$. Since every kind κ can be written in the form $\vec{\kappa} \to *$, natural transformations for kind κ can simply defined as follows (using a vector notation):

$$F \subseteq^{\vec{\kappa} \to *} G := \forall \vec{X}^{\vec{\kappa}}.\ F\vec{X} \to G\vec{X}$$

For types $F, G : *$, the type $F \subseteq^{*} G$ of natural transformations from F to G is just the ordinary function type $F \to G$. As an example, we observe that the general constructor $\mathsf{in}^{\kappa 1}$ from the last subsection is a natural transformation of type $F\,(\mu^{\kappa 1} F) \subseteq^{\kappa 1} \mu^{\kappa 1} F$. The superscript κ in "$\subseteq^{\kappa}$" will sometimes be omitted for better readability.

Generalizing Mendler iteration to higher kinds κ is now just a matter of replacing some arrows by natural transformations. We obtain the following constants, typing and reduction rules.

Formation. $\mu^\kappa : (\kappa \to \kappa) \to \kappa$

Introduction. $\mathsf{in}^\kappa : \forall F^{\kappa\to\kappa}.\, F\,(\mu^\kappa F) \subseteq^\kappa \mu^\kappa F$

Elimination.
$$\frac{\Gamma \vdash F : \kappa \to \kappa, \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : \forall X^\kappa.\, X \subseteq^\kappa G \to F\,X \subseteq^\kappa G}{\Gamma \vdash \underline{\mathsf{MIt}}^\kappa(s) : \mu^\kappa F \subseteq^\kappa G}$$

Reduction. $\underline{\mathsf{MIt}}^\kappa(s)\,(\mathsf{in}^\kappa\, t) \longrightarrow_\beta s\, \underline{\mathsf{MIt}}^\kappa(s)\, t$

Notice that for *every* constructor F of kind $\kappa \to \kappa$, $\mu^\kappa F$ is a constructor of kind κ. In Mendler's original system (Mendler, 1991) as well as its variant for the treatment of primitive (co-)recursion (Mendler, 1987), always positivity of F is required which is a very natural concept in the case $\kappa = *$. However, for higher kinds, there does not exist such a canonical syntactic restriction. Anyway, in Uustalu and Vene (1997) it has been observed that, in order to prove strong normalization, *there is no need for the restriction to positive inductive types*—an observation which has been the cornerstone for the treatment of monotone inductive types in Matthes (1998) and becomes even more useful for our higher-order nested datatypes.

It remains to show that we have obtained a sensible system. Subject reduction is easy to check for the new reduction rule; confluence is not jeopardized since there are no critical pairs; and strong normalization will be shown later by an embedding into F^ω. In the following we will try to evaluate whether with $\underline{\mathsf{MIt}}^\kappa$ we have obtained a sensible and usable device for programming.

Example 3 (Retrieving a Leaf of a Perfectly Balanced Tree) Let t : $\mathsf{PList}\,A$ be a binary leaf-labelled tree and p : $\mathsf{Stream\,Bool}$ a bit stream which acts as a path to one of the leaves a of t. Using $\underline{\mathsf{MIt}}^{\kappa 1}$ we can implement a function get such that $\mathsf{get}\,t\,p$ retrieves element a.

$$\begin{aligned} \mathsf{get} := {} & \underline{\mathsf{MIt}}^{\kappa 1}\Big(\lambda get \lambda t \lambda p.\, \mathsf{case}\,(t, \\ & \quad a.\, a, \\ & \quad l.\, \mathsf{if}\ (\mathsf{head}\, p)\ (get\, l\,(\mathsf{tail}\, p)).0\ (get\, l\,(\mathsf{tail}\, p)).1\ )\Big) \\ : {} & \forall A.\ \mathsf{PList}\, A \to \mathsf{Stream\,Bool} \to A \\ = {} & \mathsf{PList} \subseteq^{\kappa 1} (\lambda A.\, \mathsf{Stream\,Bool} \to A) \end{aligned}$$

Here we reused the type of streams defined in Example 1 and assumed a type of booleans $\mathsf{Bool} : *$ with constructors $\mathsf{true}, \mathsf{false} : \mathsf{Bool}$ and elimination $\mathsf{if} : \forall A.\, \mathsf{Bool} \to A \to A \to A$ which is, of course, definable in F^ω. To verify welltypedness, observe that the bound variables have the following types:

$$\begin{array}{ll} X & : \kappa 1 \quad \text{(not visible due to Curry-style)} \\ get & : \forall A.\, X\, A \to \mathsf{Stream\,Bool} \to A \\ t & : \mathsf{PListF}\, X\, A = A + X\,(A \times A) \\ p & : \mathsf{Stream\,Bool} \\ a & : A \\ l & : X\,(A \times A) \end{array}$$

In the recursive calls, the polymorphic type of get is instantiated with the product $A \times A$ which entails the typing $(get\, l\, (\mathsf{tail}\, p)) : A \times A$. It is now easy to check welltypedness of the whole function body. Note that $\underline{\mathsf{MIt}}^{\kappa 1}$ facilitates a kind of polymorphic recursion.

Example 4 (Summing up a Powerlist) Next, we want to define a function $\mathsf{sum} : \mathsf{PList\,Nat} \to \mathsf{Nat}$ which sums up all elements of a powerlist by iteration over its structure. In the case $\mathsf{sum}\,(\mathsf{zero}\ n)$ we can simply return n. The case $\mathsf{sum}\,(\mathsf{succ}\ t)$, however, imposes some challenge since sum cannot be directly applied to $t : \mathsf{PList}(\mathsf{Nat} \times \mathsf{Nat})$. The solution is to define a more general function sum' by polymorphic recursion, which has the following behavior.

$$\mathsf{sum}' : \forall A.\ \mathsf{PList}\, A \to (A \to \mathsf{Nat}) \to \mathsf{Nat}$$

$$\begin{array}{lll} \mathsf{sum}'\,(\mathsf{zero}\ a)\, f & \longrightarrow^+ & f\, a \\ \mathsf{sum}'\,(\mathsf{succ}\ l)\, f & \longrightarrow^+ & \mathsf{sum}'\ l\ (\lambda p.\, f\,(p.0) + f\,(p.1)) \end{array}$$

Here, the iteration process builds up a "continuation" f which in the end sums up the contents packed into a. Having found out the desired behavior of sum', its implementation using $\underline{\mathsf{MIt}}^{\kappa 1}$ is a mechanical process which results in the following definition:

$$\begin{array}{lll} \mathsf{sum}' & := & \underline{\mathsf{MIt}}^{\kappa 1}\Big(\lambda sum' \lambda t \lambda f.\, \mathsf{case}\,(t, \\ & & \qquad a.\, f\, a, \\ & & \qquad l.\, sum'\, l\ (\lambda p.\, f\,(p.0) + f\,(p.1)))\Big) \\ & : & \mathsf{PList} \subseteq^{\kappa 1} (\lambda A.\, (A \to \mathsf{Nat}) \to \mathsf{Nat}) \end{array}$$

The postulated reduction behavior is verified by a simple calculation. From sum', the summation function is obtained by $\mathsf{sum} := \mathsf{sum}'\ \mathsf{id}$.

Let us remark here that the result constructor $G' = \lambda A.\, (A \to \mathsf{Nat}) \to \mathsf{Nat}$ is an instance of a general scheme which is extremely useful for defining functions over heterogeneous datatypes. For the constant constructors $G = H = \lambda B.\, \mathsf{Nat}$, the result constructor G' is equivalent to $\lambda A \forall B.\, (A \to H\, B) \to G\, B$

which is a syntactic form of the *right Kan extension of* G *along* H. Kan extensions are so commonly used with nested datatypes that we will present an elimination scheme in Section 4 with hardwired Kan extensions.

Having completed these two examples we are confident that $\underline{\mathsf{MIt}}^\kappa$ is a useful iterator for higher-rank inductive types. In the following, we will again dualize our definition to handle also greatest fixed points of rank-n type constructors ($n \geq 2$).

3.5 Mendler Coiteration for Higher Ranks

Adding the following constructs, we obtain our System $\underline{\mathsf{MIt}}^\omega$, which is an extension of Mendler's system (1991) to finite kinds.

$$
\begin{array}{ll}
\text{Formation.} & \nu^\kappa \quad : (\kappa \to \kappa) \to \kappa \\
\text{Elimination.} & \mathsf{out}^\kappa : \forall F^{\kappa \to \kappa}.\, \nu^\kappa F \subseteq^\kappa F\,(\nu^\kappa F) \\
\text{Introduction.} & \dfrac{\begin{array}{l}\Gamma \vdash F : \kappa \to \kappa, \\ \Gamma \vdash G : \kappa \\ \Gamma \vdash s : \forall X^\kappa.\, G \subseteq^\kappa X \to G \subseteq^\kappa F\,X\end{array}}{\Gamma \vdash \underline{\mathsf{MCoit}}^\kappa(s) : G \subseteq^\kappa \nu^\kappa F} \\
\text{Reduction.} & \mathsf{out}^\kappa\,(\underline{\mathsf{MCoit}}^\kappa(s)\,t) \longrightarrow_\beta s\ \underline{\mathsf{MCoit}}^\kappa(s)\,t
\end{array}
$$

The reader is invited to check that no problems for subject reduction and confluence arise from these definitions. To demonstrate the usefulness of $\underline{\mathsf{MCoit}}^{\kappa 1}$ as a coiteration scheme, in the following we will develop a *redecoration* algorithm for infinite triangular matrices, which can be defined as a heterogeneous coinductive type. To this end, we fix a type $E : *$ of matrix elements. The type $\mathsf{Tri}\,A$ of triangular matrices with diagonal elements in A and ordinary elements E can be obtained as follows:

$$
\begin{array}{lll}
\mathsf{TriF} & := \lambda X \lambda A.\, A \times X(E \times A) & : \kappa 2 \\
\mathsf{Tri} & := \nu^{\kappa 1}\,\mathsf{TriF} & : \kappa 1
\end{array}
$$

We think of these triangles decomposed columnwise: The first column is a singleton of type A, the second a pair of type $E \times A$, the third a triple of type $E \times (E \times A)$, the fourth a quadruple of type $E \times (E \times (E \times A))$ etc. Hence, if some column has some type A' we obtain the type of the next column as $E \times A'$. This explains the definition of TriF. We can visualize triangles like this:

$$\begin{array}{c|c|c|c|l}
A & E & E & E & E \ldots \\
 & A & E & E & E \ldots \\
 & & A & E & E \ldots \\
 & & & A & E \ldots \\
 & & & & A \ldots
\end{array}$$

The vertical lines hint at the decomposition scheme.

Example 5 (Triangle Decomposition) The top element and the remainder, a trapezium in our visualization[2], can be obtained using the destructor for coinductive types:

$$\begin{aligned}
\mathsf{top} &:= \lambda t.\ (\mathsf{out}^{\kappa 1}\, t).0 &&:\ \forall A.\, \mathsf{Tri}\, A \to A \\
\mathsf{rest} &:= \lambda t.\ (\mathsf{out}^{\kappa 1}\, t).1 &&:\ \forall A.\, \mathsf{Tri}\, A \to \mathsf{Tri}\, (E \times A)
\end{aligned}$$

Cutting to top row of a trapezium $\mathsf{Tri}\,(E \times A)$ to obtain a triangle $\mathsf{Tri}\ A$ can be implemented using Mendler coiteration for rank 2:

$$\begin{aligned}
\mathsf{cut} &:= \underline{\mathsf{MCoit}}^{\kappa 1}\Big(\lambda cut \lambda t.\ \langle (\mathsf{top}\ t).1,\ \ cut\, (\mathsf{rest}\ t)\rangle\Big) \\
&:\ \ (\lambda A.\ \mathsf{Tri}\,(E \times A)) \subseteq^{\kappa 1} \mathsf{Tri}
\end{aligned}$$

Redecoration is an operation that takes a redecoration rule f (an assignment of B-decorations to A-decorated trees) and an A-decorated tree t, and returns a B-decorated tree t'. The return tree t' is obtained from t by B-redecorating every node based on the A-decorated subtree it roots, as instructed by the redecoration rule f. For streams, for instance redec : $\forall A \forall B.\, (\mathsf{Stream}\, A \to B) \to \mathsf{Stream}\, A \to \mathsf{Stream}\, B$ takes $f : \mathsf{Stream}\, A \to B$ and $t : \mathsf{Stream}\, A$ and returns $\mathsf{redec}\ \ f\ t$, which is a B-stream obtained from t by replacing each of its elements by what f assigns to the sub stream this element heads. Redecoration has some practical relevance, e.g., for the definition of digital filters—as a posting on the Haskell Mailing List (2002) shows.

Theoretically, redecoration is an operation dual to substitution in trees $T\, A$ over some label type A. Viewing T as a monad, substitution $(A \to T\, B) \to T\, A \to T\, B$ of B-labelled trees for A-labels is the monad multiplication operation. Viewing T as a comonad, redecoration $(T\, A \to B) \to T\, A \to T\, B$ becomes the new multiplication (Uustalu and Vene, 2002).

Example 6 (Triangle Redecoration) For triangles, redecoration works as

[2] Although elements of the type $\mathsf{Tri}(E \times A)$ *are* (infinite) triangles, we will speak of trapeziums due to our columnwise decomposition.

follows: In the triangle

$$\begin{array}{r} A\ E\ E\ E\ E\ \ldots \\ A\ E\ E\ E\ \ldots \\ \hline \underline{A}\ E\ E\ \ldots \\ A\ E\ \ldots \\ A\ \ldots \end{array}$$

the underlined A (as an example) gets replaced by the B assigned by the redecoration rule to the sub triangle cut out by the horizontal line; similarly, every other A is replaced by a B. Redecoration redec has type $\forall A \forall B.\, (\mathsf{Tri}\, A \to B) \to \mathsf{Tri}\, A \to \mathsf{Tri}\, B$. Therefore, it cannot be implemented directly using Mendler coiteration, but via an auxiliary function redec' with an isomorphic type in the proper format.

$$\begin{array}{rcl} \mathsf{redec} & := & \lambda f \lambda t.\ \mathsf{redec}'\,(\mathsf{pack}\,\langle f, t\rangle) \\ & : & \forall A \forall B.\ (\mathsf{Tri}\, A \to B) \to \mathsf{Tri}\, A \to \mathsf{Tri}\, B \end{array}$$

$$\begin{array}{rcl} \mathsf{redec}' & := & \underline{\mathsf{MCoit}}^{\kappa 1} \Big(\lambda \mathit{redec}' \lambda p.\ \mathsf{open}\,(p,\ \mathit{ft}. \\ & & \quad \langle \mathit{ft}.0\ (\mathit{ft}.1),\ \ \mathit{redec}'\ (\mathsf{pack}\,\langle \mathsf{lift}\ (\mathit{ft}.0),\ \ \mathsf{rest}\ (\mathit{ft}.1)\rangle)\rangle)\Big) \\ & : & (\lambda B \exists A.\, (\mathsf{Tri}\, A \to B) \times \mathsf{Tri}\, A) \subseteq^{\kappa 1} \mathsf{Tri} \end{array}$$

Here we make use of a function $\mathsf{lift} : \forall A \forall B.\ (\mathsf{Tri}\, A \to B) \to \mathsf{Tri}(E \times A) \to (E \times B)$ which lifts the redecoration rule to trapeziums such that it can be used with the trapezium $\mathsf{rest}\ (\mathit{ft}.1)$. [3]

$$\begin{array}{rcl} \mathsf{lift} & := & \lambda f \lambda t.\ \langle (\mathsf{top}\ t).0,\ \ f\,(\mathsf{cut}\ t)\rangle \\ & : & \forall A \forall B.\ (\mathsf{Tri}\, A \to B) \to \mathsf{Tri}(E \times A) \to E \times B \end{array}$$

Hence, if f is a redecoration rule, the new redecoration rule $\mathsf{lift}\, f$ for trapeziums takes a trapezium t of type $\mathsf{Tri}(E \times A)$ and yields a diagonal element of a trapezium in $\mathsf{Tri}(E \times B)$, viewed as a triangle! This means an element of $E \times B$. Since the elements outside the diagonal do not have to be transformed, the left component of type E is taken to be the upper component of the first column of t in the trapezium picture. The right component of type B comes from applying f to the triangle which results from cutting off the top row from t.

In the typing of redec', if the variable redec' receives the type $G' \subseteq^{\kappa 1} X$ with $G' := \lambda B \exists A.\, (\mathsf{Tri}\, A \to B) \times \mathsf{Tri}\, A$ and p gets type $G'B$, then ft gets type $(\mathsf{Tri}\, A \to B) \times \mathsf{Tri}\, A$, hence $\mathit{ft}.0\ (\mathit{ft}.1) : B$ and the term starting with redec' gets type $X(E \times B)$ because the argument to redec' gets type $G'(E \times B)$: The existential quantifier for A is instantiated with $E \times A$, the universal quantifiers for A and B in the type of lift are just instantianted by A and B themselves.

[3] Note that the definition of lift in Abel et al. (2003) was wrong albeit type-correct.

Apparently, one gets the following reduction behavior:

$$\mathsf{out}^{\kappa 1}(\mathsf{redec}'\,(\mathsf{pack}\,\langle f, t\rangle)) \longrightarrow^{+} \langle f\,t, \mathsf{redec}'\,(\mathsf{pack}\,\langle \mathsf{lift}\,f, \mathsf{rest}\;t\rangle)\rangle.$$

Hence, $\mathsf{top}\,(\mathsf{redec}\,f\,t) \longrightarrow^{+} f\,t$, and $\mathsf{rest}\,(\mathsf{redec}\,f\,t)$ and $\mathsf{redec}\,(\mathsf{lift}\,f)(\mathsf{rest}\;t)$ have a common reduct.

The source constructor G' of function redec' is a *left Kan extension of* Tri *along* Tri, since it is an instance of the general scheme $\lambda B \exists A.\,(H\,A \to B) \times G\,A$ with $G = H = \mathsf{Tri}$. In Section 4 we will introduce a generalized coiteration scheme with hardwired Kan extensions. This will allow us to define redec directly and not via an uncurried auxiliary function redec'.

3.6 *Embedding into* F^ω

In this subsection, we will show strong normalization for System $\underline{\mathsf{MIt}}^\omega$ by embedding it into F^ω. Since this shows that $\underline{\mathsf{MIt}}^\omega$ is just a conservative extension of F^ω, we can conclude that higher-order datatypes and Mendler iteration schemes are already present in F^ω. The quest for a precise formulation of this fact led to the definition of $\underline{\mathsf{MIt}}^\omega$ in which these concepts are isolated and given names.

Embeddings of (co)inductive constructors into F^ω can be obtained via the following recipe:

(1) Read off the encoding of (co)inductive constructors from the type of the (co)iterator.
(2) Find the encoding of the (co)iterator, which usually just consists of some lambda-abstractions and some shuffling resp. packing of the abstracted variables.
(3) Take the right-hand side of the reduction rule for (co)iteration as the encoding of the general constructor resp. destructor.

To implement this scheme, we start by performing some simple equivalence conversions on the type of $\lambda s.\,\underline{\mathsf{MIt}}^\kappa(s)$: (let $\kappa = \vec{\kappa} \to *$)

$$\begin{array}{rl}
& \forall F^{\kappa\to\kappa}\forall G^{\kappa}.\ (\forall X^{\kappa}.\ X \subseteq G \to F\,X \subseteq G) \to \mu^{\kappa}F \subseteq G \\
\equiv & \forall F^{\kappa\to\kappa}\forall G^{\kappa}.\ (\forall X^{\kappa}.\ X \subseteq G \to F\,X \subseteq G) \to \forall \vec{Y}^{\vec{\kappa}}.\ \mu^{\kappa}F\,\vec{Y} \to G\,\vec{Y} \\
\leftrightarrow & \forall F^{\kappa\to\kappa}\forall \vec{Y}^{\vec{\kappa}}.\ \mu^{\kappa}F\,\vec{Y} \to \forall G^{\kappa}.\ (\forall X^{\kappa}.\ X \subseteq G \to F\,X \subseteq G) \to G\,\vec{Y}
\end{array}$$

This equivalent type for $\lambda s.\,\underline{\mathsf{MIt}}^\kappa(s)$ states that there is a mapping of $\mu^\kappa F\,\vec{Y}$ into some other type. Now we simply *define* $\mu^\kappa F\,\vec{Y}$ to be that other type. The definitions of $\underline{\mathsf{MIt}}^\kappa(s)$ and in^κ then simply fall into place:

$$
\begin{array}{lll}
\mu^\kappa & : & (\kappa \to \kappa) \to \vec{\kappa} \to * \\
\mu^\kappa & := & \lambda F \lambda \vec{Y} \forall G^\kappa.\ (\forall X^\kappa.\ X \subseteq G \to F\,X \subseteq G) \to G\,\vec{Y} \\[1ex]
\underline{\mathsf{MIt}}^\kappa(s) & : & \mu^\kappa F \subseteq G \qquad \text{for } s : \forall X^\kappa.\ X \subseteq G \to F\,X \subseteq G \\
\underline{\mathsf{MIt}}^\kappa(s) & := & \lambda r.\ r\,s \\[1ex]
\mathsf{in}^\kappa & : & \forall F^{\kappa\to\kappa}.\ F\,(\mu^\kappa F) \subseteq \mu^\kappa F \\
\mathsf{in}^\kappa & := & \lambda t \lambda s.\ s\ \underline{\mathsf{MIt}}^\kappa(s)\ t
\end{array}
$$

Lemma 1 *With the definitions above,* $\underline{\mathsf{MIt}}^\kappa(s)\ (\mathsf{in}^\kappa\,t) \longrightarrow^+ s\ \underline{\mathsf{MIt}}^\kappa(s)\ t$ *in System* F^ω*.*

Proof. By simple calculation. □

To find an encoding of (co)inductive constructors, we consider the type of the (universal) coiterator $\lambda s.\,\underline{\mathsf{MCoit}}^\kappa(s)$:

$$
\begin{array}{ll}
 & \forall F^{\kappa\to\kappa} \forall G^\kappa.\ (\forall X^\kappa.\ G \subseteq X \to G \subseteq F\,X) \to G \subseteq \nu^\kappa F \\
\equiv & \forall F^{\kappa\to\kappa} \forall G^\kappa.\ (\forall X^\kappa.\ G \subseteq X \to G \subseteq F\,X) \to \forall \vec{Y}^{\vec{\kappa}}.\ G\,\vec{Y} \to \nu^\kappa F\,\vec{Y} \\
\leftrightarrow & \forall F^{\kappa\to\kappa} \forall \vec{Y}^{\vec{\kappa}}.\ (\exists G^\kappa.\ (\forall X^\kappa.\ G \subseteq X \to G \subseteq F\,X) \times G\,\vec{Y}) \to \nu^\kappa F\,\vec{Y}
\end{array}
$$

These considerations lead to the following definitions:

$$
\begin{array}{lll}
\nu^\kappa & : & (\kappa \to \kappa) \to \vec{\kappa} \to * \\
\nu^\kappa & := & \lambda F \lambda \vec{Y} \exists G^\kappa.\ (\forall X^\kappa.\ G \subseteq X \to G \subseteq F\,X) \times G\,\vec{Y} \\[1ex]
\underline{\mathsf{MCoit}}^\kappa(s) & : & G \subseteq \nu^\kappa F \qquad \text{for } s : \forall X^\kappa.\ G \subseteq X \to G \subseteq F\,X \\
\underline{\mathsf{MCoit}}^\kappa(s) & := & \lambda t.\ \mathsf{pack}\,\langle s,\, t\rangle \\[1ex]
\mathsf{out}^\kappa & : & \forall F^{\kappa\to\kappa}.\ \nu^\kappa F \subseteq F\,(\nu^\kappa F) \\
\mathsf{out}^\kappa & := & \lambda r.\ \mathsf{open}\,(r,\ st.\ (st.0)\ \underline{\mathsf{MCoit}}^\kappa(st.0)\ (st.1))
\end{array}
$$

Lemma 2 *With the definitions above,* $\mathsf{out}^\kappa\,(\underline{\mathsf{MCoit}}^\kappa(s)\ t) \longrightarrow^+ s\ \underline{\mathsf{MCoit}}^\kappa(s)\,t$ *in System* F^ω*.*

Proof. By simple calculation. □

Theorem 1 (Strong normalization) *System* $\underline{\mathsf{MIt}}^\omega$ *is strongly normalizing, i.e., for each welltyped term* t_0 *there is no infinite reduction sequence* $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$

Proof. By lemmata 1 and 2, such a reduction sequence would translate into the infinite sequence $t_0 \longrightarrow^+ t_1 \longrightarrow^+ t_2 \longrightarrow^+ \ldots$ of well-typed terms of System F^ω (here, the above definitions are meant to be unfolded), a contradiction to the strong normalization property of F^ω. $\square$

4 System GMIt^ω: Refined and Generalized Mendler (Co)Iteration

The system GMIt^ω of this section is a minor variant of the system MIt^ω in Abel et al. (2003). Its intension is to ease programming with Mendler iteration in case Kan extensions have to be used (cf. examples 4 and 6 in Section 3). In a sense, we are just hard-wiring the Kan extensions into the (co)iteration scheme of $\underline{\mathsf{MIt}}^\omega$.

4.1 Containment of Constructors

The key idea consists in identifying an appropriate containment relation for constructors of the same kind κ. For types, the canonical choice is implication. For an arbitrary kind κ, the easiest notion is "pointwise implication" $\subseteq^\kappa$, used in the previous section for the definition of $\underline{\mathsf{MIt}}^\omega$.

Refined containment A more refined notion $\leq^\kappa$ has been employed already in previous work (see Abel and Matthes, 2003) which studies iteration for monotone inductive constructors of higher kinds:

$$
\begin{array}{lcl}
F \leq^* \quad G & := & F \to G \\
F \leq^{\kappa \to \kappa'} G & := & \forall X^\kappa \forall Y^\kappa.\, X \leq^\kappa Y \to F\,X \leq^{\kappa'} G\,Y
\end{array}
$$

Hence, for $F, G : \kappa 1$, $F \leq^{\kappa 1} G = \forall A \forall B.(A \to B) \to FA \to GB$, first suggested to us by Peter Hancock in November 2000. Here, one does not only have to pass from F to G, but this has to be stable under changing the argument type from A to B.

This notion will give rise to a notion of monotonicity on the basis of which traditional-style iteration and coiteration can be extended to arbitrary ranks, see section 6.

Relativized refined containment In order to extend Mendler (co)iteration to higher kinds so that efficient generalized folds (Hinze, 2000; Martin et al., 2002) are directly covered, we have to relativize the notion $\leq^\kappa$, for $\kappa = \vec{\kappa} \to *$, to a vector $\vec{H}$ of constructors of kinds $\vec{\kappa} \to \vec{\kappa}$, i. e., $H_1 : \kappa_1 \to \kappa_1, H_2 : \kappa_2 \to \kappa_2, \ldots$ In addition to a variation of the argument constructor as in the definition of $\leq^{\kappa\to\kappa'}$, moreover, H_i is applied to the i-th "target argument" (below, another definition similarly modifies the "source argument").

For every kind $\kappa = \vec{\kappa} \to *$, define a constructor $\leq^\kappa_{(-)}$: $(\vec{\kappa} \to \vec{\kappa}) \to \kappa \to \kappa \to *$ by structural recursion on κ as follows:

$$\begin{array}{rcl} F \leq^* \quad G & := & F \to G \\ F \leq^{\kappa\to\kappa'}_{H,\vec{H}} G & := & \forall X^\kappa \forall Y^\kappa .\, X \leq^\kappa H\, Y \to F\, X \leq^{\kappa'}_{\vec{H}} G\, Y \end{array}$$

Note that, in the second line, H has kind $\kappa \to \kappa$. For $\vec{H}$ a vector of identity constructors, the new notion $\leq^\kappa_{\vec{H}}$ coincides with $\leq^\kappa$. Similarly, we define another constructor ${}_{(-)}{\leq^\kappa}$: $(\vec{\kappa} \to \vec{\kappa}) \to \kappa \to \kappa \to *$, where the base case is the same as before, hence no ambiguity with the notation arises.

$$\begin{array}{rcl} F \quad \leq^* \quad G & := & F \to G \\ F \;{}_{H,\vec{H}}{\leq^{\kappa\to\kappa'}}\; G & := & \forall X^\kappa \forall Y^\kappa .\, H\, X \leq^\kappa Y \to F\, X \;{}_{\vec{H}}{\leq^{\kappa'}}\; G\, Y \end{array}$$

As an example, for $F, G, H : \kappa 1$, one has

$$\begin{array}{rcl} F \;\leq^{\kappa 1}_H\; G & = & \forall A \forall B .\, (A \to H B) \to F A \to G B, \\ F \;{}_H{\leq^{\kappa 1}}\; G & = & \forall A \forall B .\, (H A \to B) \to F A \to G B. \end{array}$$

Even more concretely, we have (and will use them in the examples 4 and 9):

$$\begin{array}{rcl} \mathsf{PList} \leq^{\kappa 1}_{\lambda B.\mathsf{Nat}} \lambda B.\mathsf{Nat} & = & \forall A \forall B.(A \to \mathsf{Nat}) \to \mathsf{PList}\, A \to \mathsf{Nat} \\ \mathsf{Tri} \;{}_{\mathsf{Tri}}{\leq^{\kappa 1}}\; \mathsf{Tri} & = & \forall A \forall B.(\mathsf{Tri}\, A \to B) \to \mathsf{Tri}\, A \to \mathsf{Tri}\, B \end{array}$$

4.2 Definition of GMIt^ω

Now we are ready to define generalized Mendler-style iteration and coiteration, which specialize to ordinary Mendler-style iteration and coiteration in the case of (co)inductive types, and to a scheme encompassing generalized folds (Bird and Paterson, 1999a; Hinze, 2000; Martin et al., 2002) and the dual scheme for coinductive constructors of rank 2. The generalized scheme for coinductive constructors is a new principle of programming with non-wellfounded datatypes.

The system GMIt^ω is given as an extension of F^ω by wellkinded constructor constants μ^κ and ν^κ, and welltyped term constants $\mathsf{in}^\kappa, \mathsf{out}^\kappa$ as for $\underline{\mathsf{MIt}}^\omega$, and the elimination rule $\mathsf{GMIt}^\kappa(s)$ and the introduction rule $\mathsf{GMCoit}^\kappa(s)$ for every kind κ, and new term reduction rules.

Inductive constructors. Let $\kappa = \vec{\kappa} \to *$.

Formation. $\mu^\kappa : (\kappa \to \kappa) \to \kappa$

Introduction. $\mathsf{in}^\kappa : \forall F^{\kappa\to\kappa}.\, F\,(\mu^\kappa F) \subseteq^\kappa \mu^\kappa F$

Elimination.
$$\frac{\Gamma \vdash F : \kappa \to \kappa \quad \Gamma \vdash \vec{H} : \vec{\kappa} \to \vec{\kappa} \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : \forall X^\kappa.\, X \leq^\kappa_{\vec{H}} G \to F\,X \leq^\kappa_{\vec{H}} G}{\Gamma \vdash \mathsf{GMIt}^\kappa(s) : \mu^\kappa F \leq^\kappa_{\vec{H}} G}$$

Reduction. $\mathsf{GMIt}^\kappa(s)\,\vec{f}\,(\mathsf{in}^\kappa\,t) \longrightarrow_\beta s\,\mathsf{GMIt}^\kappa(s)\,\vec{f}\,t$

where $|\vec{f}| = |\vec{\kappa}|$.

Example 7 (Summing up a Powerlist, Revisited) The function sum' for powerlists (see Example 4) can be naturally implemented with $\mathsf{GMIt}^{\kappa 1}$. The difference to the original implementation confines itself to swapping the arguments t (powerlist) and f (continuation).

$$\begin{array}{rcl}
\mathsf{sum}' & := & \mathsf{GMIt}^{\kappa 1}\Big(\lambda sum'\lambda f\lambda t.\,\mathsf{case}\,(t, \\
& & \quad a.\, f\,a, \\
& & \quad l.\, sum'\ (\lambda p.\, f\,(p.0) + f\,(p.1))\ l)\Big) \\
& : & \forall A.\,(A \to \mathsf{Nat}) \to \mathsf{PList}\,A \to \mathsf{Nat}
\end{array}$$

The void universal quantification over B in $\mathsf{PList} \leq^{\kappa 1}_{\lambda B.\mathsf{Nat}} \lambda B.\mathsf{Nat}$ has been suppressed since it is irrelevant in our Curry-style system. The reduction behavior is trivially read off the definition:

$$\begin{array}{l}
\mathsf{sum}'\ f\,(\mathsf{zero}\ a) \longrightarrow^+ f\,a \\
\mathsf{sum}'\ f\,(\mathsf{succ}\ l) \longrightarrow^+ \mathsf{sum}'\,(\lambda p.\, f\,(p.0) + f\,(p.1))\ l
\end{array}$$

Example 8 (Summing up a Bush) Recall the nested datatype of "bushy lists" given in section 3.3:

$$\begin{array}{lll}
\mathsf{BushF} & = \lambda X\lambda A.\, 1 + A \times X\,(X\,A) & : \kappa 2 \\
\mathsf{Bush} & = \mu^{\kappa 1}\,\mathsf{BushF} & : \kappa 1 \\
\\
\mathsf{bnil} & = \mathsf{in}^{\kappa 1}\,(\mathsf{inl}\,\langle\rangle) & : \forall A.\ \mathsf{Bush}\,A \\
\mathsf{bcons} & = \lambda a\lambda b.\ \mathsf{in}^{\kappa 1}\,(\mathsf{inr}\,\langle a,\, b\rangle) & : \forall A.\, A \to \mathsf{Bush}\,(\mathsf{Bush}\ A) \to \mathsf{Bush}\ A
\end{array}$$

Similarly to powerlists, we can define a summation function sum' for bushes:

$$
\begin{array}{lcl}
\mathsf{bsum}' & := & \mathsf{GMIt}^{\kappa 1}\Big(\lambda bsum' \lambda f \lambda t.\, \mathsf{case}\,(t, \\
& & \quad u.\ 0, \\
& & \quad p.\ f(p.0) + bsum'\ (bsum'\ f)\ (p.1))\Big) \\
& : & \forall A.\,(A \to \mathsf{Nat}) \to \mathsf{Bush}\, A \to \mathsf{Nat}
\end{array}
$$

The 0 in the first case is the supposed zero in Nat. The types would be assigned as follows:

$$
\begin{array}{ll}
bsum' & : X \leq^{\kappa 1}_{\lambda B.\mathsf{Nat}} \lambda B.\mathsf{Nat} \\
f & : A \to \mathsf{Nat} \\
t & : 1 + A \times X(XA) \\
p & : A \times X(XA) \\
bsum' f & : XA \to \mathsf{Nat} \\
bsum'\,(bsum' f) & : X(XA) \to \mathsf{Nat}
\end{array}
$$

While termination of sum' for powerlists is already observable from the reduction behavior, this cannot be said for bsum' where the nesting in the datatype is reflected in the nesting of the recursive calls:

$$
\begin{array}{lcl}
\mathsf{bsum}'\ f\ \mathsf{bnil} & \longrightarrow^{+} & 0 \\
\mathsf{bsum}'\ f\,(\mathsf{bcons}\, a\, b) & \longrightarrow^{+} & f\, a + \mathsf{bsum}'\,(\mathsf{bsum}' f)\, b
\end{array}
$$

Note that the second argument seemingly decreases in an easy fashion: on the left-hand side, one has $\mathsf{bcons}\, a\, b$, one the right-hand side only b. But there is also the embedded call to $\mathsf{bsum}' f$ without specified second argument, hence to the whole function $\mathsf{bsum}' f$ about to be defined by iteration! Consequently, proving termination of bsum' using a term ordering seems to be a non-trivial issue, whereas in our system it is byproduct of type checking.

Coinductive constructors. Let $\kappa = \vec{\kappa} \to *$.

Formation. $\nu^\kappa \quad : (\kappa \to \kappa) \to \kappa$

Elimination. $\mathsf{out}^\kappa : \forall F^{\kappa \to \kappa}.\, \nu^\kappa F \ \subseteq^\kappa \ F\,(\nu^\kappa F)$

Introduction.
$$\frac{\begin{array}{l}\Gamma \vdash F : \kappa \to \kappa \\ \Gamma \vdash \vec{H} : \vec{\kappa} \to \vec{\kappa} \\ \Gamma \vdash G : \kappa \\ \Gamma \vdash s : \forall X^\kappa.\ G\ {}_{\vec{H}}{\leq^\kappa}\ X \to G\ {}_{\vec{H}}{\leq^\kappa}\ F\,X\end{array}}{\Gamma \vdash \mathsf{GMCoit}^\kappa(s) : G\ {}_{\vec{H}}{\leq^\kappa}\ \nu^\kappa F}$$

Reduction. $\mathsf{out}^\kappa(\mathsf{GMCoit}^\kappa(s)\,\vec{f}\,t) \longrightarrow_\beta s\ \mathsf{GMCoit}^\kappa(s)\,\vec{f}\,t$

where $|\vec{f}| = |\vec{\kappa}|$.

Example 9 (Triangle Redecoration, Revisited) Using $\mathsf{GMCoit}^{\kappa 1}$, triangle redecoration can be implemented much more concisely. In the context of Example 6 we obtain the function redec directly as follows.

$$\begin{array}{rl}\mathsf{redec} := & \mathsf{GMCoit}^{\kappa 1}\Big(\lambda \mathit{redec}\,\lambda f \lambda t.\ \langle f\,t, \mathit{redec}\,(\mathsf{lift}\,f)\,(\mathsf{rest}\,t)\rangle\Big) \\ : & \forall A \forall B.\ (\mathsf{Tri}\,A \to B) \to \mathsf{Tri}\,A \to \mathsf{Tri}\,B \ = \ \mathsf{Tri}\ {}_{\mathsf{Tri}}{\leq^{\kappa 1}}\ \mathsf{Tri}\end{array}$$

Thus, one can enforce the desired reduction behavior without any detours (in the design of the program and in the direction of the reduction, compare with the original version inside $\underline{\mathsf{MIt}}^\omega$):

$$\begin{array}{rcl}\mathsf{top}\,(\mathsf{redec}\,f\,t) & \longrightarrow^+ & f\,t \\ \mathsf{rest}\,(\mathsf{redec}\,f\,t) & \longrightarrow^+ & \mathsf{redec}\,(\mathsf{lift}\,f)(\mathsf{rest}\ t)\end{array}$$

4.3 Embedding GMIt^ω *into* $\underline{\mathsf{MIt}}^\omega$

The embedding of GMIt^ω into $\underline{\mathsf{MIt}}^\omega$ shown in this section preserves reductions. Hence, GMIt^ω inherits strong normalization from $\underline{\mathsf{MIt}}^\omega$. The embedding can even be read as a definition of $\mathsf{GMIt}^\kappa(s)$ and $\mathsf{GMCoit}^\kappa(s)$ within $\underline{\mathsf{MIt}}^\omega$. Therefore, we will later freely use these constructions within the system $\underline{\mathsf{MIt}}^\omega$.

For the sake of the embedding, we use a syntactic version of Kan extensions (see Mac Lane, 1998, chapter 10). This cannot come as a surprise to the readers of section 6.2 of Bird and Paterson (1999a) because that is one of the ways to justify generalized folds, i. e., elimination schemes for $\mu^{\kappa 1}$. Moreover, in the examples 4 and 6 within $\underline{\mathsf{MIt}}^\omega$, special Kan extensions were explicitly used, and the same programming tasks were accomplished directly within GMIt^ω in the examples 7 and 9. In the sequel, this will be clarified: Just by choosing the target constructor of iteration to be an appropriate Kan extension, one gets the behavior of $\mathsf{GMIt}^\kappa(s)$ within $\underline{\mathsf{MIt}}^\omega$, and similarly for the source constructor in the coinductive case.

Compared with Abel and Matthes (2003), Kan extensions "along" are now defined for all kinds, not just for rank 1. [4]

Right Kan extension along $\vec{H}$. Let $\kappa = \vec{\kappa} \to *$ and $\vec{\kappa}' = \vec{\kappa} \to \vec{\kappa}$ and define for $G : \kappa$, $\vec{H} : \vec{\kappa}'$ and $\vec{X} : \vec{\kappa}$ the type $(\mathsf{Ran}^{\kappa}_{\vec{H}}\, G)\, \vec{X}$ by iteration on $|\vec{\kappa}|$:

$$\mathsf{Ran}^{*}\, G := G$$
$$(\mathsf{Ran}^{\kappa_1 \to \tilde{\kappa}}_{H, \vec{H}}\, G)\, X \vec{X} := \forall Y^{\kappa_1}.\ X \leq^{\kappa_1} H Y \ \to\ (\mathsf{Ran}^{\tilde{\kappa}}_{\vec{H}}\, (G\, Y))\, \vec{X}$$

Left Kan Extension along $\vec{H}$. Let again $\kappa = \vec{\kappa} \to *$ and $\vec{\kappa}' = \vec{\kappa} \to \vec{\kappa}$ and define for $F : \kappa$, $\vec{H} : \vec{\kappa}'$ and $\vec{Y} : \vec{\kappa}$ the type $(\mathsf{Lan}^{\kappa}_{\vec{H}}\, F)\vec{Y}$ by iteration on $|\vec{\kappa}|$:

$$\mathsf{Lan}^{*}\, F := F$$
$$(\mathsf{Lan}^{\kappa_1 \to \tilde{\kappa}}_{H, \vec{H}}\, F)\, Y \vec{Y} := \exists X^{\kappa_1}.\ H X \leq^{\kappa_1} Y \ \times\ (\mathsf{Lan}^{\tilde{\kappa}}_{\vec{H}}\, (F X))\, \vec{Y}$$

We omit the index $\vec{H}$ if it is just a vector of identities Id.

Lemma 3 *Let $\kappa = \vec{\kappa} \to *$, $F, G : \kappa$ and $\vec{H} : \vec{\kappa} \to \vec{\kappa}$. The following pairs of types are logically equivalent:*

(1) $F \leq^{\kappa}_{\vec{H}} G$ and $F \subseteq^{\kappa} \mathsf{Ran}^{\kappa}_{\vec{H}}\, G$.
(2) $F\ {}_{\vec{H}}{\leq^{\kappa}}\ G$ and $\mathsf{Lan}^{\kappa}_{\vec{H}}\, F \subseteq^{\kappa} G$.

Proof. For $\kappa = *$, all these types are just $F \to G$. Otherwise, let $\vec{\kappa}' := \vec{\kappa} \to \vec{\kappa}$,

[4] The definitions are taken from Abel et al. (2003).

$n := |\vec{\kappa}|$ and define[5]

$$
\begin{array}{lll}
\mathsf{leqRan}^\kappa & := & \lambda g \lambda t \lambda \vec{f}.\, g\, \vec{f}\, t \\
& : & \forall F^\kappa \forall G^\kappa \forall \vec{H}^{\vec{\kappa}'}.\ F \le^\kappa_{\vec{H}} G \to F \subseteq^\kappa \mathsf{Ran}^\kappa_{\vec{H}}\, G \\[1ex]
\mathsf{ranLeq}^\kappa & := & \lambda h \lambda \vec{f} \lambda r.\, h\, r\, \vec{f} \\
& : & \forall F^\kappa \forall G^\kappa \forall \vec{H}^{\vec{\kappa}'}.\ F \subseteq^\kappa \mathsf{Ran}^\kappa_{\vec{H}}\, G \to F \le^\kappa_{\vec{H}} G \\[1ex]
\mathsf{leqLan}^\kappa & := & \lambda g \lambda x.\, \mathsf{open}\,(x,\, p_1.\, \mathsf{open}\,(p_1.1,\, p_2. \\
& & \quad \ldots \mathsf{open}\,(p_{n-1}.1,\, p_n.\, g (p_1.0) \ldots (p_n.0)(p_n.1)) \ldots)) \\
& : & \forall F^\kappa \forall G^\kappa \forall \vec{H}^{\vec{\kappa}'}.\ F \;{}_{\vec{H}}{\le^\kappa}\; G \to \mathsf{Lan}^\kappa_{\vec{H}}\, F \subseteq^\kappa G \\[1ex]
\mathsf{lanLeq}^\kappa & := & \lambda h \lambda \vec{f} \lambda t.\, h\, \mathsf{pack}\langle f_1, \mathsf{pack}\langle f_2, \ldots \mathsf{pack}\langle f_n, t\rangle \ldots\rangle\rangle \\
& : & \forall F^\kappa \forall G^\kappa \forall \vec{H}^{\vec{\kappa}'}.\ \mathsf{Lan}^\kappa_{\vec{H}}\, F \subseteq^\kappa G \to F \;{}_{\vec{H}}{\le^\kappa}\; G
\end{array}
\qquad \Box
$$

We can now simply define the new function symbols of GMIt^ω in $\underline{\mathsf{MIt}}^\omega$ in case $\kappa \neq *$ (otherwise, the typing and reduction rules are just the same for both systems):

$$
\begin{array}{ll}
\mathsf{GMIt}^\kappa(s) & := \mathsf{ranLeq}^\kappa(\underline{\mathsf{MIt}}^\kappa(\lambda h.\, \mathsf{leqRan}^\kappa(s(\mathsf{ranLeq}^\kappa\, h)))) \\
\mathsf{GMCoit}^\kappa(s) & := \mathsf{lanLeq}^\kappa(\underline{\mathsf{MCoit}}^\kappa(\lambda h.\, \mathsf{leqLan}^\kappa(s(\mathsf{lanLeq}^\kappa\, h))))
\end{array}
$$

Then $\mathsf{GMIt}^\kappa(s)$ and $\mathsf{GMCoit}^\kappa(s)$ have precisely the typing behavior as required for GMIt^ω. We only treat the inductive case, the coinductive one is analogous. Assume the step term s for $\mathsf{GMIt}^\kappa(s)$ of type $\forall X^\kappa.\ X \le^\kappa_{\vec{H}} G \to F\,X \le^\kappa_{\vec{H}} G$. Then,

$$
\hat{s} := \lambda h.\, \mathsf{leqRan}^\kappa(s(\mathsf{ranLeq}^\kappa\, h)) : \forall X^\kappa.\ X \subseteq^\kappa \mathsf{Ran}^\kappa_{\vec{H}}\, G \to F\,X \subseteq^\kappa \mathsf{Ran}^\kappa_{\vec{H}}\, G.
$$

Therefore, $\underline{\mathsf{MIt}}^\kappa(\hat{s}) : \mu^\kappa F \subseteq^\kappa \mathsf{Ran}^\kappa_{\vec{H}}\, G$, finally $\mathsf{GMIt}^\kappa(s) : \mu^\kappa F \le^\kappa_{\vec{H}} G$.

We calculate, using the above abbreviation $\hat{s}$,

$$
\begin{array}{rcl}
\mathsf{GMIt}^\kappa(s)\, \vec{f}\, (\mathsf{in}^\kappa\, t) & \longrightarrow^+ & \underline{\mathsf{MIt}}^\kappa(\hat{s})\, (\mathsf{in}^\kappa\, t)\, \vec{f} \\
& \longrightarrow & \hat{s}\ \underline{\mathsf{MIt}}^\kappa(\hat{s})\, t\, \vec{f} \\
& \longrightarrow^+ & s\ (\mathsf{ranLeq}^\kappa(\underline{\mathsf{MIt}}^\kappa(\hat{s})))\ \vec{f}\ t
\end{array}
$$

With a similar, but notationally more tedious calculation for the coinductive

[5] The definition for right Kan extension would even work for $\kappa = *$, the one for left Kan extension would be incorrect in that case.

case, we get, with these definitions, in $\underline{\mathsf{MIt}}^\omega$:

$$
\begin{aligned}
\mathsf{GMIt}^\kappa(s)\,\vec{f}\,(\mathsf{in}^\kappa\,t) \quad &\longrightarrow^+ s\ \mathsf{GMIt}^\kappa(s)\,\vec{f}\,t\\
\mathsf{out}^\kappa(\mathsf{GMCoit}^\kappa(s)\,\vec{f}\,t) &\longrightarrow^+ s\ \mathsf{GMCoit}^\kappa(s)\,\vec{f}\,t
\end{aligned}
$$

Since one step of reduction in GMIt^ω is replaced by at least one reduction step of the encoding in $\underline{\mathsf{MIt}}^\omega$, GMIt^ω inherits strong normalization of $\underline{\mathsf{MIt}}^\omega$. Since the number of steps is fixed for every kind κ, we will in the examples just treat GMIt^ω as a subsystem of $\underline{\mathsf{MIt}}^\omega$ in the sense that we assume that both iteration and both coiteration schemes are present in $\underline{\mathsf{MIt}}^\omega$ together with their reduction rules. To put it differently, we take the above definitions as "macros" inside $\underline{\mathsf{MIt}}^\omega$ and neglect the difference in reduction steps between these "notational definitions" and the original system GMIt^ω.

5 Basic Conventional Iteration

We are looking for a system of conventional iteration into which we can embed $\underline{\mathsf{MIt}}^\omega$ in a way which sends Mendler iteration into conventional iteration.

Systems of conventional iteration, unlike Mendler-style systems, directly follow the idea of initial algebras in category theory. In that model, $F : \kappa \to \kappa$ would have to be an endofunctor on a category associated with κ. $\mu^\kappa F$ would be the object part of an initial F-algebra, and $\mathsf{in}^\kappa : F\,(\mu^\kappa F) \longrightarrow \mu^\kappa F$ be its morphism part. In the Mendler-style systems of this article, we have chosen to represent those morphisms by terms of type $F\,(\mu^\kappa F) \subseteq^\kappa \mu^\kappa F$. This decision for the type of the constructor in^κ will be fixed throughout the article.[6] The main open question is the choice of the syntactic representation of being a functor. Certainly, we cannot require the equational functor laws in our framework of F^ω, but have to concentrate on the types: If a functor F is applied to a morphism $s : A \longrightarrow B$, then the result is a morphism $Fs : FA \longrightarrow FB$. If F is a type-constructor, i. e., of kind $\kappa 1$, then this can be represented by the existence of a term m of type

$$\mathsf{mon}^{\kappa 1}\,F := \forall A \forall B.\ (A \to B) \to FA \to FB,$$

which is nothing but monotonicity of F. The constructor F would only represent the object mapping, and m the mapping on morphisms that would get the same name F in category theory. Then, $s : A \to B$ implies $m\,s : FA \to FB$.

The notion $\mathsf{mon}^{\kappa 1}\,F$ is the most logically-minded definition of rank-1-functors:

[6] Recently, Peter Aczel informed us of a different view on higher-rank datatypes with a type for the constructor that already incorporates some Kan extension.

It is free from the analysis of the shape of F (polynomial, strictly positive, non-strictly positive). Moreover, it is the only possible definition that is based on the existence of a *witness of monotonicity*, i. e., a term inhabiting the type expressing functoriality. This is no longer so for higher kinds. We will stick to the logical approach, but face several possible definitions of monotonicity expressing functoriality. In this section, we will start with *basic monotonicity*, in section 6, a more refined definition will be studied.

Basic Monotonicity Define

$$\underline{\mathsf{mon}}^{\kappa\to\kappa'} := \lambda F.\ \forall X^\kappa \forall Y^\kappa.\ X \subseteq^\kappa Y \to F\,X \subseteq^{\kappa'} F\,Y : (\kappa \to \kappa') \to *$$

This will be our notion of monotonicity of constructors $F : \kappa \to \kappa'$, and hence our representation of functoriality. Consequently, we only use our inductive constructor $\mu^\kappa F$ in the presence of some term m of type $\underline{\mathsf{mon}}^{\kappa\to\kappa} F$. As has been observed in Matthes (1998), there is no need to require a fixed term m beforehand. It is sufficient to give it as an argument to either the constructor or the iterator. It, moreover, does not need to be closed, hence giving rise to *conditional monotonicity* or (in the case m is just a variable) to *hypothetical monotonicity.* Anyhow, in loc. cit., it has been shown that strong normalization holds for inductive *types.* There, only $\mathsf{mon}^{\kappa 1}$ enters the definitions. Clearly, $\underline{\mathsf{mon}}^{\kappa 1}\, F = \mathsf{mon}^{\kappa 1}\, F$.

In the next subsection, we will see that a canonical formulation of (co)iteration, based on basic monotonicity $\underline{\mathsf{mon}}^{\kappa\to\kappa}$ for all kinds κ, is a subsystem of $\underline{\mathsf{MIt}}^\omega$.

5.1 Defining Basic Conventional Iteration in Terms of Mendler Iteration

Define the basic conventional iterators and coiterators by

$$\begin{aligned}
\underline{\mathsf{It}}^\kappa(m, s) &:= \underline{\mathsf{MIt}}^\kappa(\lambda it \lambda t.\ s\,(m\ it\ t)) \\
\underline{\mathsf{Coit}}^\kappa(m, s) &:= \underline{\mathsf{MCoit}}^\kappa(\lambda coit\ \lambda t.\ m\ coit\ (s\,t))
\end{aligned}$$

Then, one immediately gets the following derived typing rules

$$\frac{\Gamma \vdash F : \kappa \to \kappa \quad \Gamma \vdash m : \underline{\mathsf{mon}}^{\kappa\to\kappa}\, F \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : F\,G \subseteq^\kappa G}{\Gamma \vdash \underline{\mathsf{It}}^\kappa(m, s) : \mu^\kappa F \subseteq^\kappa G}$$

$$\frac{\Gamma \vdash F : \kappa \to \kappa \quad \Gamma \vdash m : \underline{\mathsf{mon}}^{\kappa\to\kappa}\, F \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : G \subseteq^\kappa F\,G}{\Gamma \vdash \underline{\mathsf{Coit}}^\kappa(m, s) : G \subseteq^\kappa \nu^\kappa F}$$

and reduction behavior as follows

$$\begin{array}{ll}\underline{\mathsf{It}}^\kappa(m,s)\,(\mathsf{in}^\kappa\,t) & \longrightarrow^+\ s\,(m\ \underline{\mathsf{It}}^\kappa(m,s)\,t)\\ \mathsf{out}^\kappa\,(\underline{\mathsf{Coit}}^\kappa(m,s)\,t) & \longrightarrow^+\ m\ \underline{\mathsf{Coit}}^\kappa(m,s)\,(s\,t)\end{array}$$

The interpretation of the typing rule for $\underline{\mathsf{It}}^\kappa(m,s)$ is as follows: Given a monotonicity witness m for F and an F-algebra s, i. e., a term s of type $F\,G \subseteq^\kappa G$, "initiality" of $\mu^\kappa F$ yields a "morphism" from $\mu^\kappa F$ to the underlying "object" G of the F-algebra s. This "morphism" is witnessed by $\underline{\mathsf{It}}^\kappa(m,s)$ of type $\mu^\kappa F \subseteq^\kappa G$.

Also the reduction behavior is as expected (see Matthes, 1999): If $\underline{\mathsf{It}}^\kappa(m,s)$ is given the constructor term $\mathsf{in}^\kappa\,t$, this reduces to the step term (the F-algebra) s, applied to the recursive call, where m organizes how the whole function $\underline{\mathsf{It}}^\kappa(m,s)$ is applied to the term t which was the argument to in^κ.

The rules for $\nu^\kappa F$ are just found by dualization.

Example 10 (Summing up a Powerlist, Conventional Style) We redo example 4 with $\underline{\mathsf{It}}^{\kappa 1}$. Again, we define a more general function

$$\mathsf{sum}' : \forall A.\ \mathsf{PList}\,A \to (A \to \mathsf{Nat}) \to \mathsf{Nat}\ =\ \mathsf{PList} \subseteq^{\kappa 1} G'$$

with $G' := \lambda A.\,(A \to \mathsf{Nat}) \to \mathsf{Nat} : \kappa 1$. This is done as follows:

$$\begin{array}{lll}\mathsf{PListF} & := \lambda X \lambda A.\ A + X\,(A \times A) & : \kappa 2\\ \mathsf{plistfb} & := \lambda s \lambda x.\,\mathsf{case}\,(x,\ a.\ \mathsf{inl}\,a,\ y.\ \mathsf{inr}\,(s\,y)) & : \underline{\mathsf{mon}}^{\kappa 2}\,\mathsf{PListF}\\ s & := \lambda t \lambda f.\,\mathsf{case}\,(t,\ a.\,f\,a,\ res.\ res\,(\lambda p.\,f\,(p.0) + f\,(p.1))) & : \mathsf{PListF}\,G' \subseteq^{\kappa 1} G'\\ \mathsf{sum}' & := \underline{\mathsf{It}}^{\kappa 1}(\mathsf{plistfb}, s) & : \mathsf{PList} \subseteq^{\kappa 1} G'\end{array}$$

An easy calculation shows that, as in example 4, we get the reduction behavior

$$\begin{array}{ll}\mathsf{sum}'\,(\mathsf{zero}\ a)\,f & \longrightarrow^+\ f\,a\\ \mathsf{sum}'\,(\mathsf{succ}\ l)\,f & \longrightarrow^+\ \mathsf{sum}'\ l\ (\lambda p.\,f\,(p.0) + f\,(p.1)),\end{array}$$

although the expansion of the definition of $\underline{\mathsf{It}}^{\kappa 1}(\mathsf{plistfb}, s)$ yields a more complicated term than in example 4. [7]

We want to isolate these means of basic conventional iteration and coiteration in the form of a system $\underline{\mathsf{It}}^\omega$ in order to make it the *target* of an embedding.

[7] One would need permutative conversions, also called commuting conversions, for case in order to recover the term found in example 4.

Let the system $\underline{\mathsf{It}}^\omega$ be given by the extension of system F^ω by the following constants, function symbols, typing and reduction rules for iteration and coiteration, starting with those for iteration:

Formation. $\underline{\mu}^\kappa : (\kappa \to \kappa) \to \kappa$

Introduction. $\underline{\mathsf{in}}^\kappa : \forall F^{\kappa\to\kappa}.\, F\,(\underline{\mu}^\kappa F) \subseteq^\kappa \underline{\mu}^\kappa F$

Elimination.
$$\frac{\Gamma \vdash F : \kappa \to \kappa, \quad \Gamma \vdash m : \underline{\mathsf{mon}}^{\kappa\to\kappa}\, F \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : F\,G \subseteq^\kappa G}{\Gamma \vdash \underline{\mathsf{It}}^\kappa(m, s) : \underline{\mu}^\kappa F \subseteq^\kappa G}$$

Reduction. $\underline{\mathsf{It}}^\kappa(m, s)\,(\underline{\mathsf{in}}^\kappa\, t) \longrightarrow_\beta s\,(m\,\underline{\mathsf{It}}^\kappa(m, s)\, t)$

As has been promised, this is nothing more than a system version of the definitions in the previous subsection. Since the embedding of $\underline{\mathsf{MIt}}^\omega$ into $\underline{\mathsf{It}}^\omega$ in the next subsection will even change the constructors (not only the terms), also the name of the fixed-point former μ^κ has been changed into $\underline{\mu}^\kappa$, as well as the name of the constructor (in the sense of constructing elements of fixed-points) in^κ, which has been changed to $\underline{\mathsf{in}}^\kappa$. Similar remarks apply to coiteration, given as follows:

Formation. $\underline{\nu}^\kappa : (\kappa \to \kappa) \to \kappa$

Elimination. $\underline{\mathsf{out}}^\kappa : \forall F^{\kappa\to\kappa}.\, \underline{\nu}^\kappa F \subseteq^\kappa F\,(\underline{\nu}^\kappa F)$

Introduction.
$$\frac{\Gamma \vdash F : \kappa \to \kappa, \quad \Gamma \vdash m : \underline{\mathsf{mon}}^{\kappa\to\kappa}\, F \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : G \subseteq^\kappa F\,G}{\Gamma \vdash \underline{\mathsf{Coit}}^\kappa(m, s) : G \subseteq^\kappa \underline{\nu}^\kappa F}$$

Reduction. $\underline{\mathsf{out}}^\kappa\,(\underline{\mathsf{Coit}}^\kappa(m, s)\, t) \longrightarrow_\beta m\,\underline{\mathsf{Coit}}^\kappa(m, s)\,(s\,t)$

Using the definitions in the previous subsection, $\underline{\mathsf{It}}^\omega$ embeds into $\underline{\mathsf{MIt}}^\omega$. The interesting result, however, is the embedding backwards: $\underline{\mathsf{MIt}}^\omega$ even embeds into $\underline{\mathsf{It}}^\omega$.

Here, we present a somewhat surprising embedding of $\underline{\mathsf{MIt}}^\omega$ into $\underline{\mathsf{It}}^\omega$. Certainly, there is the embedding through F^ω that polymorphically encodes the (co)inductive constructors (see section 3.6) and ignores the additional capabilities of $\underline{\mathsf{It}}^\omega$. An interesting embedding has to send the Mendler (co)iterators of $\underline{\mathsf{MIt}}^\omega$ to the conventional (co)iterators of $\underline{\mathsf{It}}^\omega$. Unlike the embedding of GMIt^ω into $\underline{\mathsf{MIt}}^\omega$, our embedding will not just be a "notational definition", but also transforms the (co)inductive constructors. As before, Kan extensions play a central role.

Naive Kan extensions along the identity. In the following, we define a naive form of Kan extensions (unlike in previous papers and above). Let $\kappa := \kappa_0 \to \vec{\kappa} \to *$, $F : \kappa$, $G : \kappa_0$ and $G_i : \kappa_i$ for $1 \leq |\vec{\kappa}|$. The types $(\underline{\mathsf{Ran}}^\kappa\, F)\, G\, \vec{G}$ and $(\underline{\mathsf{Lan}}^\kappa\, F)\, G\, \vec{G}$ are defined as follows:

$$(\underline{\mathsf{Ran}}^\kappa\, F)\, G\, \vec{G} := \forall Y^{\kappa_0}.\; G \subseteq^{\kappa_0} Y \;\to\; F\, Y\, \vec{G}$$

$$(\underline{\mathsf{Lan}}^\kappa\, F)\, G\, \vec{G} := \exists X^{\kappa_0}.\; X \subseteq^{\kappa_0} G \;\times\; F\, X\, \vec{G}$$

Alternatively, $\underline{\mathsf{Ran}}^\kappa$ and $\underline{\mathsf{Lan}}^\kappa$ can be seen as constructors of kind $\kappa \to \kappa$.

Notice that, trivially, always $\underline{\mathsf{Ran}}\, F \subseteq F$ and $F \subseteq \underline{\mathsf{Lan}}\, F$. More precisely, we can define for $\kappa \neq *$

$$\begin{array}{lll} \mathsf{ranId} : \underline{\mathsf{Ran}}^\kappa \subseteq^\kappa \mathsf{Id} & \text{by} & \mathsf{ranId} := \lambda x.x\,\mathsf{id} \\ \mathsf{lanId} : \mathsf{Id} \subseteq^\kappa \underline{\mathsf{Lan}}^\kappa & \text{by} & \mathsf{lanId} := \lambda x.\,\mathsf{pack}\langle \mathsf{id}, x\rangle \end{array}$$

Lemma 4 *We have the following logical equivalences (already in F^ω):*

(1) $(\forall X^\kappa.\; G \subseteq X \to G \subseteq F\, X) \leftrightarrow G \subseteq (\underline{\mathsf{Ran}}^{\kappa\to\kappa}\, F)\, G$ *for* $F : \kappa \to \kappa, G : \kappa$
(2) $(\forall X^\kappa \forall Y^\kappa.\; X \subseteq Y \to F\, X \subseteq F'\, Y) \leftrightarrow F \subseteq \underline{\mathsf{Ran}}^{\kappa\to\kappa'}\, F'$ *for* $F, F' : \kappa \to \kappa'$
(3) $\underline{\mathsf{mon}}^{\kappa\to\kappa'}\, F \leftrightarrow F \subseteq \underline{\mathsf{Ran}}^{\kappa\to\kappa'}\, F$ *for* $F : \kappa \to \kappa'$
(4) $(\forall X^\kappa.\; X \subseteq G \to F\, X \subseteq G) \leftrightarrow (\underline{\mathsf{Lan}}^{\kappa\to\kappa}\, F)\, G \subseteq G$ *for* $F : \kappa \to \kappa, G : \kappa$
(5) $(\forall X^\kappa \forall Y^\kappa.\; X \subseteq Y \to F\, X \subseteq F'\, Y) \leftrightarrow \underline{\mathsf{Lan}}^{\kappa\to\kappa'}\, F \subseteq F'$ *for* $F, F' : \kappa \to \kappa'$
(6) $\underline{\mathsf{mon}}^{\kappa\to\kappa'}\, F \leftrightarrow \underline{\mathsf{Lan}}^{\kappa\to\kappa'}\, F \subseteq F$ *for* $F : \kappa \to \kappa'$

Proof. For the first two equivalences, a proof for the direction from the left to the right is $\mathsf{mRan} := \lambda s \lambda t \lambda f.sft$. The reverse implications are proven by $\lambda s \lambda f \lambda t.stf$. For the fourth and fifth equivalence from left to right, the proof is $\mathsf{mLan} := \lambda s \lambda t.\mathsf{open}\,(t,\; p.\; s(p.0)(p.1))$, and $\lambda s \lambda f \lambda t.s\,\mathsf{pack}\langle f, t\rangle$ for the reverse direction. Apparently, (3) is an instance of (2), and (6) is an instance of (5).[8] □

The terms $\mathsf{mRan} := \lambda s\lambda t\lambda f.sft$ and $\mathsf{mLan} := \lambda s\lambda t.\mathsf{open}\,(t,\, p.\, s(p.0)(p.1))$ in the previous proof will be used in the embedding below.

The following is a crucial property of $\underline{\mathsf{It}}^\omega$: There is a uniform proof of monotonicity for all Kan extensions. (Recall that $\kappa \neq *$.)

$$\begin{array}{rcl} \underline{\mathsf{M}}^\kappa_{\mathsf{Lan}} & := & \lambda g\lambda p.\,\mathsf{open}\,(p,\ x.\ \mathsf{pack}\langle g\circ x.0,\ x.1\rangle) \\ & : & \forall G^\kappa.\ \underline{\mathsf{mon}}^\kappa\ (\underline{\mathsf{Lan}}^\kappa\ G) \\ \underline{\mathsf{M}}^\kappa_{\mathsf{Ran}} & := & \lambda g\lambda t\lambda f.\ t(f\circ g) \\ & : & \forall G^\kappa.\ \underline{\mathsf{mon}}^\kappa\ (\underline{\mathsf{Ran}}^\kappa\, G) \end{array}$$

The embedding is now straightforward. The system $\underline{\mathsf{MIt}}^\omega$ of Mendler iteration and coiteration can be defined in $\underline{\mathsf{It}}^\omega$ as follows (we only give the rules for the non-homomorphic definitions):

Formation. $\ulcorner\mu^\kappa\urcorner \ :\ (\kappa\to\kappa)\to\kappa$

$\ulcorner\mu^\kappa\urcorner \ :=\ \underline{\mu}^\kappa\circ\underline{\mathsf{Lan}}^{\kappa\to\kappa}$

Introduction. $\ulcorner\mathsf{in}^\kappa\urcorner \ :\ \forall F^{\kappa\to\kappa}.\ F\,\ulcorner\mu^\kappa F\urcorner \subseteq \underline{\mu}^\kappa(\underline{\mathsf{Lan}}^{\kappa\to\kappa}\,F)$

$\ulcorner\mathsf{in}^\kappa\urcorner \ :=\ \lambda t.\ \underline{\mathsf{in}}^\kappa\,(\mathsf{lanId}\ t)$

Elimination.
$$\frac{F:\kappa\to\kappa \qquad G:\kappa \qquad s:\forall X^\kappa.\ X\subseteq G\to F\,X\subseteq G}{\ulcorner\underline{\mathsf{MIt}}^\kappa(s)\urcorner:\ulcorner\mu^\kappa F\urcorner\subseteq\ulcorner G\urcorner}$$

$\ulcorner\underline{\mathsf{MIt}}^\kappa(s)\urcorner \ :=\ \lambda x.\ \underline{\mathsf{It}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}},\mathsf{mLan}\,\ulcorner s\urcorner)\ x$ where

$\mathsf{mLan}\,\ulcorner s\urcorner \ :\ \underline{\mathsf{Lan}}^{\kappa\to\kappa}\,\ulcorner F\urcorner\ulcorner G\urcorner\subseteq\ulcorner G\urcorner$

Reduction. $\underline{\mathsf{MIt}}^\kappa(s)\ (\mathsf{in}^\kappa\,t)\longrightarrow s\ \underline{\mathsf{MIt}}^\kappa(s)\ t$

[8] In fact, mRan even inhabits the type

$$\forall F', F^{\kappa\to\kappa'}\forall X^\kappa.\ (\forall Y^\kappa.\ X\subseteq Y\to F\,X\subseteq F'\,Y)\to F\,X\subseteq(\underline{\mathsf{Ran}}^{\kappa\to\kappa'}\,F')\,X,$$

the direction from left to right in the first equivalence is an instance of this type, and every lambda-abstraction inhabiting it also inhabits the direction from left to right in (2). The same holds with mLan for the fourth and fifth equivalence with type $\forall F', F^{\kappa\to\kappa'}\forall Y^\kappa.\ (\forall X^\kappa.\ X\subseteq Y\to F\,X\subseteq F'\,Y)\to(\underline{\mathsf{Lan}}^{\kappa\to\kappa'}\,F)\,Y\subseteq F'\,Y$.

This behavior is simulated by finitely many steps inside $\underline{\mathsf{It}}^\omega$:

$$
\begin{array}{rcl}
\ulcorner \underline{\mathsf{MIt}}^\kappa(s)\ (\mathsf{in}^\kappa\, t)\urcorner & \longrightarrow^+ & \underline{\mathsf{It}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}}, \mathsf{mLan}\ulcorner s\urcorner)\ (\underline{\mathsf{in}}^\kappa\,(\mathsf{lanId}\ulcorner t\urcorner)) \\
& \longrightarrow & \mathsf{mLan}\ulcorner s\urcorner\ (\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}}\ \underline{\mathsf{It}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}}, \mathsf{mLan}\ulcorner s\urcorner)\ (\mathsf{lanId}\ \ulcorner t\urcorner)) \\
& \longrightarrow & \mathsf{mLan}\ulcorner s\urcorner\ (\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}}\ \underline{\mathsf{It}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}}, \mathsf{mLan}\ulcorner s\urcorner)\ \ \mathsf{pack}\langle \mathsf{id},\ \ulcorner t\urcorner\rangle) \\
& \longrightarrow^+ & \mathsf{mLan}\ulcorner s\urcorner\ \ \mathsf{pack}\langle \underline{\mathsf{It}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}}, \mathsf{mLan}\ulcorner s\urcorner)\circ \mathsf{id}, \ulcorner t\urcorner\rangle \\
& \longrightarrow^+ & \ulcorner s\urcorner\ (\underline{\mathsf{It}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}}, \mathsf{mLan}\ulcorner s\urcorner)\circ \mathsf{id})\ \ulcorner t\urcorner \\
& \longrightarrow & \ulcorner s\urcorner\ (\lambda x.\ \underline{\mathsf{It}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}}, \mathsf{mLan}\ulcorner s\urcorner)\, x)\ \ulcorner t\urcorner \\
& = & \ulcorner s\ \underline{\mathsf{MIt}}^\kappa(s)\ t\urcorner
\end{array}
$$

Notice that the embedding employs a left Kan extension for the inductive case while the embedding of GMIt^ω into $\underline{\mathsf{MIt}}^\omega$ uses a right Kan extension for that purpose. Hence, there is a real need for the existential quantifier also for inductive constructors alone. We come to the coinductive case.

$$
\begin{array}{llcl}
\text{Formation.} & \ulcorner \nu^\kappa \urcorner & : & (\kappa\to\kappa)\to\kappa \\
& \ulcorner \nu^\kappa \urcorner & := & \underline{\nu}^\kappa \circ \underline{\mathsf{Ran}}^{\kappa\to\kappa} \\
\\
\text{Elimination.} & \ulcorner \mathsf{out}^\kappa \urcorner & : & \forall F^{\kappa\to\kappa}.\ \underline{\nu}^\kappa(\underline{\mathsf{Ran}}^{\kappa\to\kappa}\, F) \subseteq F\ \ulcorner \nu^\kappa F\urcorner \\
& \ulcorner \mathsf{out}^\kappa \urcorner & := & \lambda r.\ \mathsf{ranId}\,(\underline{\mathsf{out}}^\kappa\, r)
\end{array}
$$

$$
\text{Introduction.}\quad \frac{F:\kappa\to\kappa \qquad G:\kappa \qquad s:\forall X^\kappa.\ G\subseteq X\to G\subseteq F\,X}{\ulcorner \underline{\mathsf{MCoit}}^\kappa(s)\urcorner : \ulcorner G\urcorner \subseteq \ulcorner \nu^\kappa F\urcorner}
$$

$$
\begin{array}{lcll}
\ulcorner \underline{\mathsf{MCoit}}^\kappa(s)\urcorner & := & \lambda x.\ \underline{\mathsf{Coit}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Ran}}, \mathsf{mRan}\ulcorner s\urcorner)\ x & \text{where} \\
\mathsf{mRan}\ulcorner s\urcorner & : & \ulcorner G\urcorner \subseteq \underline{\mathsf{Ran}}^{\kappa\to\kappa}\ulcorner F\urcorner\ulcorner G\urcorner &
\end{array}
$$

$$
\text{Reduction.}\quad \mathsf{out}^\kappa\ (\underline{\mathsf{MCoit}}^\kappa(s)\ t) \longrightarrow s\ \underline{\mathsf{MCoit}}^\kappa(s)\ t
$$

$$
\begin{array}{rcl}
\ulcorner \mathsf{out}^\kappa\,(\underline{\mathsf{MCoit}}^\kappa(s)\,t)\urcorner & \longrightarrow^+ & \mathsf{ranId}\left(\underline{\mathsf{out}}^\kappa\,(\underline{\mathsf{Coit}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Ran}}, \mathsf{mRan}\ulcorner s\urcorner)\ulcorner t\urcorner)\right) \\
& \longrightarrow & \mathsf{ranId}\left(\underline{\mathsf{M}}^\kappa_{\mathsf{Ran}}\ \underline{\mathsf{Coit}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Ran}}, \mathsf{mRan}\ulcorner s\urcorner)\ (\mathsf{mRan}\ulcorner s\urcorner\ulcorner t\urcorner)\right) \\
& \longrightarrow^+ & \mathsf{ranId}\left(\lambda f.\ \mathsf{mRan}\ulcorner s\urcorner\ulcorner t\urcorner\,(f\circ \underline{\mathsf{Coit}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Ran}}, \mathsf{mRan}\ulcorner s\urcorner))\right) \\
& \longrightarrow^+ & \mathsf{mRan}\ulcorner s\urcorner\ulcorner t\urcorner\,(\mathsf{id}\circ \underline{\mathsf{Coit}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Ran}}, \mathsf{mRan}\ulcorner s\urcorner)) \\
& \longrightarrow^+ & \ulcorner s\urcorner\,(\lambda x.\ \underline{\mathsf{Coit}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Ran}}, \mathsf{mRan}\ulcorner s\urcorner)\,x)\,\ulcorner t\urcorner \\
& = & \ulcorner s\ \underline{\mathsf{MCoit}}^\kappa(s)\ t\urcorner
\end{array}
$$

Although sufficient to cover all of $\underline{\mathsf{MIt}}^\omega$, our basic notion of monotonicity has several defects which will be overcome in the next section.

Lemma 5 *There is no closed term of type* $\underline{\mathsf{mon}}^{\kappa 2}(\lambda X.\, X \circ X)$ *in* F^ω*.*

Proof. Assume, to the contrary,

$$m : \forall X \forall Y.(\forall A.XA \to YA) \to \forall A.X(XA) \to Y(YA).$$

Assume a type variable Z and consider $X := \lambda A.A \to \bot$ with $\bot := \forall Z.Z$, and $Y := \lambda A.A \to Z$. Then, $s := \lambda t \lambda a.ta : \forall A.XA \to YA$. Therefore, $ms : X(XZ) \to Y(YZ) = \neg\neg Z \to (Z \to Z) \to Z$, with $\neg A := A \to \bot$. Hence, $\lambda u.m\, s\, u\, \mathsf{id} : \forall Z.\neg\neg Z \to Z$. This is impossible, since F^ω is not classical (to be proven using normalization). □

The lemma hinges on our definition of monotonicity: The notion of monotonicity studied in the next section *will* cover $\lambda X \lambda A.X(XA)$.

A motivation why the lemma holds can be given as follows: How would we go from $X(XA)$ to $Y(YA)$? Two possibilities seem to exist: either through $X(YA)$ or through $Y(XA)$. However, in the first case, we would badly need monotonicity (in the only possible sense) of X to exhibit $X(XA) \to X(YA)$, in the second case, we would need monotonicity of Y to pass from $Y(XA)$ to $Y(YA)$. If neither X nor Y are monotone, we cannot expect at all to succeed. The above lemma even gives an appropriate example.

The definition of monotonicity is naive in the sense that the following properties *do not hold* in general:

$$\begin{array}{lll} \underline{\mathsf{mon}}^{\kappa\to\kappa'}\, F, \underline{\mathsf{mon}}^{\kappa}\, X & \text{implies} \;\; \underline{\mathsf{mon}}^{\kappa'}(F\, X) & \text{[does not hold!]} \\ \underline{\mathsf{mon}}^{\kappa\to\kappa}\, F & \text{implies} \;\; \underline{\mathsf{mon}}^{\kappa}(\underline{\mu}^{\kappa} F) & \text{[does not hold!]} \end{array}$$

I.e., monotonicity of a least fixed-point $\underline{\mu}^\kappa F : \kappa$ is not inherited from monotonicity of $F : \kappa \to \kappa$. Still, this notion of monotonicity has been shown to be suitable to define iteration and coiteration in the sense of $\underline{\mathsf{It}}^\omega$.

Both non-implications are exemplified with the single example $F := \lambda\_\, \lambda A.\, \neg A$. Trivially, $\underline{\mathsf{mon}}^{\kappa 2}\, F$ is inhabited by $\lambda x.\, \mathsf{id}$. Certainly, $F\, X = \lambda A.\, \neg A$ is not monotone: If $m : \mathsf{mon}^{\kappa 1}(\lambda A.\, \neg A)$, then $m : (\bot \to A) \to (\bot \to \bot) \to \neg A$, hence $m\, \mathsf{id}\, \mathsf{id} : \forall A.\neg A$, hence logical inconsistency of F^ω ensues. Also $\underline{\mu}^{\kappa 1} F$ is not monotone because this type constructor is logically equivalent with

$\lambda A.\, \neg A$: The datatype constructor $\underline{\mathsf{in}}^{\kappa 1}$ yields one direction, the other comes from

$$\underline{\mathsf{It}}^{\kappa 1}(\lambda x.\ \mathsf{id}, \mathsf{id}) : \forall A.\, (\underline{\mu}^{\kappa 1} F)\, A \to \neg A.$$

Remark 1 (Another notion of monotonicity) Consider a modified notion of monotonicity which excludes some more constructors.

$$\underline{\mathsf{vmon}}^{\vec{\kappa} \to *}\, F := \forall \vec{X}^{\vec{\kappa}} \forall \vec{Y}^{\vec{\kappa}}.\ \vec{X} \subseteq^{\vec{\kappa}} \vec{Y} \to F \vec{X} \to F \vec{Y}$$

This notion is monotonicity preserving in a very strong sense: $\underline{\mathsf{vmon}}^{\kappa \to \kappa'}\, F$ alone implies $\underline{\mathsf{vmon}}^{\kappa'}(F\, X)$, but it fails to give a good target system for $\underline{\mathsf{MIt}}^{\omega}$. This is because for the appropriate modification of left Kan extension $\underline{\mathsf{Lan}}^{\kappa}$, (although a term $\underline{\mathsf{M}}_{\mathsf{Lan}}$ exist here as well,) the step term of $\underline{\mathsf{MIt}}^{\kappa}$ does not have a type isomorphic to $\underline{\mathsf{Lan}}^{\kappa \to \kappa}\, F\, G \subseteq^{\kappa} G$.

So in general, it is trivial that monotonicity of $F : \kappa \to \kappa$ fails to entail monotonicity of $\underline{\mu}^{\kappa} F$. In particular, however, if an $F : \kappa \to \kappa$ $(\kappa \neq *)$ which is monotone happens to preserve monotonicity, then $\underline{\mu}^{\kappa} F : \kappa$ is monotone, canonically.

Define

$$\underline{\mathsf{M}}^{\kappa}_{\mu}(p, m) := \lambda f \lambda t.\ \underline{\mathsf{It}}(m, \lambda t' \lambda f'.\ \underline{\mathsf{in}}^{\kappa}\ (m\ \ \mathsf{ranId}\ (p\ \underline{\mathsf{M}}^{\kappa}_{\mathsf{Ran}}\ f'\ t')))\ t\ f$$

Then $F : \kappa \to \kappa$, $p : \forall X^{\kappa}.\ \underline{\mathsf{mon}}^{\kappa}\, X \to \underline{\mathsf{mon}}^{\kappa}(F\ X)$ and $m : \underline{\mathsf{mon}}^{\kappa \to \kappa}\, F$ imply that the step term s has type $F\, (\underline{\mathsf{Ran}}^{\kappa}(\underline{\mu}^{\kappa} F)) \subseteq^{\kappa} \underline{\mathsf{Ran}}^{\kappa}(\underline{\mu}^{\kappa} F)$ and

$$\begin{array}{lcl}
\underline{\mathsf{M}}^{\kappa}_{\mu}(p, m) & : & \underline{\mathsf{mon}}^{\kappa}(\underline{\mu}^{\kappa} F) \\
\underline{\mathsf{M}}^{\kappa}_{\mu}(p, m)\ f\ (\underline{\mathsf{in}}^{\kappa}\ t) & \longrightarrow^{+} & \underline{\mathsf{in}}^{\kappa}\ (m\ \ \mathsf{ranId}^{\kappa}\ (p\ \underline{\mathsf{M}}^{\kappa}_{\mathsf{Ran}}\ f\ (m\ \ \underline{\mathsf{It}}^{\kappa}(m, s)\ t)))
\end{array}$$

For well-behaved F, p and m (such as the regular rank-2 constructors and their canonical monotonicity-preservation and monotonicity witnesses described in Sec. 9), the right-hand side is reducible further, and we get a common $\longrightarrow^{+}$-reduct for the terms $\underline{\mathsf{M}}^{\kappa}_{\mu}(p, m)\, f\, (\underline{\mathsf{in}}^{\kappa}\, t)$ and $\underline{\mathsf{in}}^{\kappa}\, (p\ \underline{\mathsf{M}}^{\kappa}_{\mu}(p, m)\, f\, t)$. For general F and m, however, such further simplification is obviously impossible (take F to be a constructor variable and m an object variable assumed to inhabit $\underline{\mathsf{mon}}^{\kappa \to \kappa}\, F$).

It is also true that if a monotone $F : \kappa \to \kappa$ preserves monotonicity, then $\underline{\nu}^{\kappa} F : \kappa$ is canonically monotone.

6 Refined Conventional Iteration

Let MIt^ω be the restriction of GMIt^ω where the vector $\vec{H} : \vec{\kappa}'$ of H's in the typing of GMIt^κ and GMCoit^κ only consists of identities Id. Consequently, one changes the name of GMIt^κ to MIt^κ and GMCoit^κ to MCoit^κ. The reduction rules does not change (except for the names just introduced).

We shall now proceed to the presentation of a system of conventional iteration corresponding to the system MIt^ω. The system will be called It^ω and is the system discussed in Abel and Matthes (2003). In section 7, arbitrary vectors $\vec{H}$ will be reintroduced and a system GIt^ω, corresponding to the full system GMIt^ω, will be studied.

As a first step, we have to employ a notion of monotonicity different from $\underline{\mathsf{mon}}$, with the basic containment notion $\subseteq$ replaced with the refined notion of containment $\leq$.

Refined monotonicity We define $\mathsf{mon}^\kappa := \lambda F.\ F \leq^\kappa F$, hence

$$\begin{aligned}\mathsf{mon}^{\kappa\to\kappa'} &= \lambda F.\ F \leq^{\kappa\to\kappa'} F\\ &= \lambda F.\ \forall X^\kappa \forall Y^\kappa.\ X \leq^\kappa Y \to F\,X \leq^{\kappa'} F\,Y : (\kappa \to \kappa') \to *\end{aligned}$$

The type $\mathsf{mon}^\kappa\, F$, seen as a proposition, asserts essentially that F is monotone in *all* argument positions, for *monotone* argument values. The same type is used in polytypic programming for generic map functions Hinze (2002); Altenkirch and McBride (2002). (Recall that, in contrast $\underline{\mathsf{mon}}^\kappa\, F$ asserts that F is monotone in its *first* argument position, for *all* argument values.)

Notice that for $\kappa 1 = * \to *$, mon coincides with $\underline{\mathsf{mon}}$. (mon is an extension to higher ranks of $\mathsf{mon}^{\kappa 1}$ from the beginning of the preceding section.)

For the monotonicity mon, differently from the previously used notion $\underline{\mathsf{mon}}$, we have that $\lambda X.\ X \circ X : (\kappa \to \kappa) \to \kappa \to \kappa$ is monotone:

$$\lambda g \lambda f.\ g\ (g\ f) : \mathsf{mon}^{(\kappa\to\kappa)\to\kappa\to\kappa}(\lambda X \lambda Y.\ X\ (X\ Y))$$

Also, if $F : \kappa \to \kappa'$ and $X : \kappa$ are monotone, then $F\ X : \kappa'$ is monotone as well:

$$m : \mathsf{mon}^{\kappa\to\kappa'}\ F \text{ and } n : \mathsf{mon}^\kappa\ X \text{ imply } m\ n : \mathsf{mon}^{\kappa'}(F\ X)$$

The following are the canonical *monotonicity witnesses* for some closed F

which we will need in examples later.

$$
\begin{array}{lll}
\mathsf{pair} & := \lambda f \lambda g \lambda p.\, \langle f\,(p.0),\ g\,(p.1)\rangle & : \mathsf{mon}^{*\to*\to*}\,(\lambda A \lambda B.\, A \times B) \\
\mathsf{fork} & := \lambda f.\, \mathsf{pair}\, f\, f & : \mathsf{mon}^{*\to*}\,(\lambda A.\, A \times A) \\
\mathsf{either} & := \lambda f \lambda g \lambda x.\, \mathsf{case}\,(x,\ a.\ \mathsf{inl}\,(f\,a),\ b.\ \mathsf{inr}\,(g\,b)) & : \mathsf{mon}^{*\to*\to*}\,(\lambda A \lambda B.\, A + B) \\
\mathsf{maybe} & := \mathsf{either}\ \mathsf{id} & : \mathsf{mon}^{*\to*}\,(\lambda A.\, C + A)
\end{array}
$$

In the definition of maybe, we assume that A does not occur free in C.

6.1 Definition of It^ω

The system It^ω is an extension of F^ω specified by the following typing and reduction rules.

Inductive constructors. Let $\kappa = \vec{\kappa} \to *$.

Formation. $\mu^\kappa\ : (\kappa \to \kappa) \to \kappa$

Introduction. $\mathsf{in}^\kappa\ : \forall F^{\kappa\to\kappa}.\, F\,(\mu^\kappa F)\ \subseteq^\kappa\ \mu^\kappa F$

Elimination.

$$
\frac{\Gamma \vdash F : \kappa \to \kappa, \quad \Gamma \vdash m : \mathsf{mon}^{\kappa\to\kappa}\, F \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : F\,G \subseteq^\kappa G}{\Gamma \vdash \mathsf{It}^\kappa(m,s) : \mu^\kappa F \leq^\kappa G}
$$

Reduction. $\mathsf{It}^\kappa(m,s)\,\vec{f}\,(\mathsf{in}^\kappa\, t) \longrightarrow_\beta s\,(m\ \mathsf{It}^\kappa(m,s)\,\vec{f}\,t)$

where $|\vec{f}| = |\vec{\kappa}|$.

Thus, this system is "conventional" in the sense $\underline{\mathsf{It}}^\omega$ is conventional: Iteration is only possible in the presence of a monotonicity witness m, being our representation of functoriality. And the argument s to It^κ (the "step term"), has the type $F\,G \subseteq^\kappa G$, the syntactic representation of (G,s) being an F-algebra. Somewhat surprisingly, the type of $\mathsf{It}^\kappa(m,s)$ is *not* $\mu^\kappa F \subseteq^\kappa G$, hence there seems to be a mismatch: The type of in^κ and the step term are based on the view of functors inhabiting types of the form $F_1 \subseteq^\kappa F_2$ but the result type of iteration is of the stronger form $F_1 \leq^\kappa F_2$. But this strengthening is needed to ensure subject reduction since the monotonicity witness m is applied to the iterator. It is also crucial for the following fact:[9]

[9] Therefore, one might accept the type of the iterator but would like to have the

For monotone F, we now get that μF is again monotone. Define

$$\mathsf{M}^{\kappa}_{\mu}(m) := \mathsf{It}^{\kappa}(m, \mathsf{in}^{\kappa}).$$

Then $m : \mathsf{mon}^{\kappa\to\kappa}\, F$ implies $\mathsf{M}^{\kappa}_{\mu}(m) : \mathsf{mon}^{\kappa}(\mu^{\kappa} F)$, and

$$\mathsf{M}^{\kappa}_{\mu}(m)\, \vec{f}\, (\mathsf{in}^{\kappa}\, t) \longrightarrow \mathsf{in}^{\kappa}\, (m\, \mathsf{M}^{\kappa}_{\mu}(m)\, \vec{f}\, t).$$

Coinductive constructors. Let $\kappa = \vec{\kappa} \to *$.

Formation.	$\nu^{\kappa} \quad : (\kappa \to \kappa) \to \kappa$
Elimination.	$\mathsf{out}^{\kappa} : \forall F^{\kappa\to\kappa}.\, \nu^{\kappa} F \ \subseteq^{\kappa}\ F\, (\nu^{\kappa} F)$
Introduction.	$\dfrac{\Gamma \vdash F : \kappa \to \kappa, \quad \Gamma \vdash m : \mathsf{mon}^{\kappa\to\kappa}\, F \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : G \ \subseteq^{\kappa}\ F\, G}{\Gamma \vdash \mathsf{Coit}^{\kappa}(m, s) : G \ \leq^{\kappa}\ \nu^{\kappa} F}$
Reduction.	$\mathsf{out}^{\kappa}\, (\mathsf{Coit}^{\kappa}(m, s)\, \vec{f}\, t) \longrightarrow_{\beta} m\ \mathsf{Coit}^{\kappa}(m, s)\, \vec{f}\, (s\, t)$ where $\lvert\vec{f}\rvert = \lvert\vec{\kappa}\rvert$.

As in the inductive case, νF is monotone if F is: Define

$$\mathsf{M}^{\kappa}_{\nu}(m) := \mathsf{Coit}^{\kappa}(m, \mathsf{out}^{\kappa}).$$

Then $m : \mathsf{mon}^{\kappa\to\kappa}\, F$ implies $\mathsf{M}^{\kappa}_{\nu}(m) : \mathsf{mon}^{\kappa}(\nu^{\kappa} F)$, and

$$\mathsf{out}^{\kappa}\, (\mathsf{M}^{\kappa}_{\nu}(m)\, \vec{f}\, t) \longrightarrow m\, \mathsf{M}^{\kappa}_{\nu}(m)\, \vec{f}\, (\mathsf{out}^{\kappa}\, t).$$

6.2 Examples

Two examples showing the use of $\mathsf{M}^{\kappa 1}_{\mu}$, i. e., the preservation of monotonicity under formation of least fixed-points, are given. More examples for programming in It^{ω}, also with $\mathsf{Coit}^{\kappa 1}$, are to be found in Abel and Matthes (2003).

---

mismatch removed by also using $\leq^{\kappa}$ for in^{κ} and the step term. However, this modification would be very impractical to work with since it would have an additional vector $\vec{g}$ of term arguments to in^{κ} in the reduction rule. In fact, this was the system underlying the original submission of Abel and Matthes (2003).

Example 11 (Free Variable Renaming for de Bruijn Terms) The free variables of a de Bruijn term may be renamed by the canonical monotonicity witness of Lam, called lam below.

$$\begin{aligned}\mathsf{lamf} &:= \lambda g \lambda f.\ \mathsf{either}\ \ f\ \ (\mathsf{either}\ (\mathsf{fork}\ (g\ \ f))\ (g\ \ (\mathsf{maybe}\ \ f))) &: \mathsf{mon}^{\kappa 2}\ \ \mathsf{LamF}\\ \mathsf{lam} &:= \mathsf{M}_{\mu}^{\kappa 1}(\mathsf{lamf}) &: \mathsf{mon}^{\kappa 1}\ \ \mathsf{Lam}\end{aligned}$$

The reduction behavior is

$$\begin{aligned}\mathsf{lam}\, f\, (\mathsf{var}\, a) &\longrightarrow^{+} \mathsf{in}^{\kappa 1}\, (\mathsf{inl}\, (f\ a))\\ \mathsf{lam}\, f\, (\mathsf{app}\, t_1 t_2) &\longrightarrow^{+} \mathsf{in}^{\kappa 1}\, (\mathsf{inr}\, (\mathsf{inl}\, \langle \mathsf{lam}\, f\, t_1, \mathsf{lam}\, f\, t_2\rangle))\\ \mathsf{lam}\, f\, (\mathsf{abs}\, r) &\longrightarrow^{+} \mathsf{in}^{\kappa 1}\, (\mathsf{inr}\, (\mathsf{inr}\, (\mathsf{lam}\, (\mathsf{maybe}\ \ f)\, r)))\end{aligned}$$

The right-hand sides are $\longrightarrow^{+}$-reducts of $\mathsf{var}\,(f\ a)$, $\mathsf{app}\,(\mathsf{lam}\,f\,t_1)(\mathsf{lam}\,f\,t_2)$ and $\mathsf{abs}\,(\mathsf{lam}\,(\mathsf{maybe}\,f)\,r)$, respectively. [10]

A special case of free variable renaming extends the free variable supply (the context) of a given term with a new variable (0) and renames the original free variable supply accordingly ($n \mapsto n+1$). We call the corresponding program weak (as it corresponds to the weaking rule of natural deduction).

$$\mathsf{weak} := \mathsf{lam}\ (\lambda a.\ \ \mathsf{inr}\ \ a) : \forall A.\ \ \mathsf{Lam}\ \ A \to \mathsf{Lam}\ (1 + A)$$

Example 12 (Reversing a Powerlist) A reversal program for powerlists is obtainable from the monotonicity witness of PList canonically generated from a noncanonical monotonicity witness of PListF.

The canonical monotonicity witness of PListF is

$$\mathsf{plistf} := \lambda g \lambda f.\ \ \mathsf{either}\, f\ \ (g\ \ (\mathsf{fork}\ \ f)) : \mathsf{mon}^{\kappa 2}\ \ \mathsf{PListF}$$

and the canonical monotonicity witness of PList is therefore

$$\mathsf{plist} := \mathsf{M}_{\mu}^{\kappa 1}(\mathsf{plistf}) : \mathsf{mon}^{\kappa 1}\ \ \mathsf{PList}$$

[10] The small nuisance that the desired terms are not *reducts* could be avoided by defining, e. g., $\mathsf{var}(a) := \mathsf{in}^{\kappa 1}\,(\mathsf{inl}\ a)$ as a one-place function symbol instead of the constant var. Another remedy would be definition of lam as $\mathsf{It}^{\kappa 1}(\mathsf{lamf}, s)$ with

$$s := \lambda x.\, \mathsf{case}\,(x,\ a.\ \mathsf{var}\, a,\ y.\, \mathsf{case}\,(y,\ p.\ \mathsf{app}\,(p.0)\,(p.1),\ r.\ \mathsf{abs}\ r)).$$

The reversal program, however, is manufactured as follows:

$$\begin{array}{lll}
\mathsf{swap} & := \lambda f \lambda p.\ \langle f\ (p.1), f\ (p.0)\rangle & : \mathsf{mon}^{\kappa 1}\ (\lambda A.A \times A) \\
\mathsf{revf}' & := \lambda g \lambda f.\ \mathsf{either}\, f\ (g\ (\mathsf{swap}\ \ f)) & : \mathsf{mon}^{\kappa 2}\ \ \mathsf{PListF} \\
\mathsf{rev}' & := \mathsf{M}_{\mu}^{\kappa 1}(\mathsf{revf}') & : \mathsf{mon}^{\kappa 1}\ \ \mathsf{PList} \\
\mathsf{rev} & := \mathsf{rev}'\ \mathsf{id} & : \mathsf{PList} \subseteq^{\kappa 1} \mathsf{PList}
\end{array}$$

Then, $\mathsf{rev}'\, f\, (\mathsf{succ}\ l)$ and $\mathsf{succ}\, (\mathsf{rev}'\, (\mathsf{swap}\, f)\, l)$ have a common reduct. Hence, $\mathsf{rev}'\, f$ reverses the powerlist and, moreover, applies f to each argument.

6.3 Embedding It^ω *into* MIt^ω

The iterator and coiterator of It^ω are definable within MIt^ω (see the beginning of this section) so that the typing rules are obeyed and reduction is simulated. We define:

$$\mathsf{It}^\kappa(m, s) \quad := \mathsf{MIt}^\kappa(\lambda \mathit{it} \lambda \vec{f} \lambda t.\, s\, (m\ \mathit{it}\ \vec{f}\, t))$$

$$\mathsf{Coit}^\kappa(m, s) := \mathsf{MCoit}^\kappa(\lambda \mathit{coit} \lambda \vec{f} \lambda t.\, m\ \mathit{coit}\ \vec{f}\, (s\, t))$$

Embedding MIt^ω into It^ω in a typing- and reduction-preserving way seems to be impossible, except for the uninformative embedding through F^ω.

7 Generalized Refined Conventional Iteration

Similar to MIt^ω, it is also possible to define a conventional-style counterpart to GMIt^ω. We will now present a system GIt^ω that accomplishes this. One important aspect is that the efficient folds of Martin et al. (2002) are directly definable in this system.

7.1 Definition of system GIt^ω

GIt^ω extends system It^ω in two directions: $\leq$ is generalized to $\leq_{\vec{H}}$, and another constructor $F' : \kappa \to \kappa$ allows even more freedom. Apart from the names of the (co)iterators, only the typing rules are changed; the reduction rules are the same.

Inductive constructors. Let $\kappa = \vec{\kappa} \to *$.

Formation. $\mu^\kappa \; : (\kappa \to \kappa) \to \kappa$

Introduction. $\mathsf{in}^\kappa : \forall F^{\kappa\to\kappa}.\, F\,(\mu^\kappa F) \;\subseteq^\kappa\; \mu^\kappa F$

Elimination.

$$\frac{\begin{array}{l}\Gamma \vdash F, F' : \kappa \to \kappa \\ \Gamma \vdash \vec{H} : \vec{\kappa} \to \vec{\kappa} \\ \Gamma \vdash m : \forall X^\kappa \forall Y^\kappa.\; X \leq^\kappa_{\vec{H}} Y \to F\,X \leq^\kappa_{\vec{H}} F'\,Y \\ \Gamma \vdash G : \kappa \\ \Gamma \vdash s : F'\,G \subseteq^\kappa G\end{array}}{\Gamma \vdash \mathsf{GIt}^\kappa(m,s) : \mu^\kappa F \leq^\kappa_{\vec{H}} \; G}$$

Reduction. $\mathsf{GIt}^\kappa(m,s)\,\vec{f}\,(\mathsf{in}^\kappa\,t) \longrightarrow_\beta s\,(m\,\mathsf{GIt}^\kappa(m,s)\,\vec{f}\,t)$

where $|\vec{f}| = |\vec{\kappa}|$.

Coinductive constructors. Let $\kappa = \vec{\kappa} \to *$.

Formation. $\nu^\kappa \quad : (\kappa \to \kappa) \to \kappa$

Elimination. $\mathsf{out}^\kappa : \forall F^{\kappa\to\kappa}.\, \nu^\kappa F \;\subseteq^\kappa\; F\,(\nu^\kappa F)$

Introduction.

$$\frac{\begin{array}{l}\Gamma \vdash F', F : \kappa \to \kappa \\ \Gamma \vdash \vec{H} : \vec{\kappa} \to \vec{\kappa} \\ \Gamma \vdash m : \forall X^\kappa \forall Y^\kappa.\; X \;{}_{\vec{H}}{\leq^\kappa}\; Y \to F'\,X \;{}_{\vec{H}}{\leq^\kappa}\; F\,Y \\ \Gamma \vdash G : \kappa \\ \Gamma \vdash s : G \;\subseteq^\kappa\; F'\,G\end{array}}{\Gamma \vdash \mathsf{GCoit}^\kappa(m,s) : G \;{}_{\vec{H}}{\leq^\kappa}\; \nu^\kappa F}$$

Reduction. $\mathsf{out}^\kappa\,(\mathsf{GCoit}^\kappa(m,s)\,\vec{f}\,t) \longrightarrow_\beta m\,\mathsf{GCoit}^\kappa(m,s)\,\vec{f}\,(s\,t)$

where $|\vec{f}| = |\vec{\kappa}|$.

Apparently, It^ω is (apart from different names for iterators and coiterators) just the special case with $F' = F$ and $\vec{H} = \vec{\mathsf{Id}}$.

7.2 Examples

Example 13 (Summing up a Powerlist, Revisited) In GIt^ω, the implementation of the summation of a powerlist in system GMIt^ω (example 7) can be closely mimicked. We can define

$$
\begin{aligned}
\mathsf{sum}' &:= \mathsf{GIt}^{\kappa 1}(m, s) \\
&\quad : \; \mathsf{PList} \leq^{\kappa 1}_{\lambda\_.\ \mathsf{Nat}} \lambda\_.\ \mathsf{Nat} = \forall A \forall\_.\ (A \to \mathsf{Nat}) \to \mathsf{PList}\ \ A \to \mathsf{Nat}
\end{aligned}
$$

where

$$
\begin{aligned}
Q \; &:= \text{anything} : \kappa 1 \\
F' &:= \lambda X \lambda A.\ \mathsf{Nat} + X\ \ (Q\ \ A) : \kappa 2 \\
m \; &:= \lambda g \lambda f \lambda t.\ \mathsf{case}\Big(t,\ a.\ \mathsf{inl}\,(f\,a),\ l.\ \mathsf{inr}\Big(g\ \ (\lambda p.\ f\,(p.0) + f\,(p.1))\ \ l\Big)\Big) \\
&\quad : \; \forall X \forall Y.\ X \leq^{\kappa 1}_{\lambda\_.\ \mathsf{Nat}} Y \to \mathsf{PListF}\, X \leq^{\kappa 1}_{\lambda\_.\ \mathsf{Nat}} F'\ Y \\
&\quad = \; \forall X \forall Y.\ \Big(\forall A' \forall B.\ (A' \to \mathsf{Nat}) \to X\ \ A' \to Y\ \ B\Big) \\
&\qquad\qquad \to \forall A \forall B'.\ (A \to \mathsf{Nat}) \to A + X\ \ (A \times A) \to \mathsf{Nat} + Y\ \ (Q\ \ B') \\
s \; &:= \lambda t.\ \mathsf{case}\,(t,\ n.\, n,\ n.\, n) \\
&\quad : \; F'\ \ (\lambda\_.\ \mathsf{Nat}) \subseteq^{\kappa 1} \lambda\_.\ \mathsf{Nat} = \forall\_.\ \mathsf{Nat} + \mathsf{Nat} \to \mathsf{Nat}
\end{aligned}
$$

The reduction behavior is precisely that of sum' in example 7. One should pay attention to the division of work between m and s. The addition is done in m!

The previous example illustrates that one should not at all think of canonical monotonicity witnesses in GIt^ω. The m's do work specific to the programming task at hand. Example 12 of powerlist reversal by means of a noncanonical monotonicity witness is an easy precursor illustrating that idea.

Example 14 (Substitution for de Bruijn Terms) In GIt^ω, the following smooth definition of substitution for de Bruijn terms as an iteration is possible:

$$
\begin{aligned}
\mathsf{lift} \;\; &:= \lambda f \lambda x.\ \mathsf{case}\,(x,\ u.\ \mathsf{var}\ \ (\mathsf{inl}\ \ u),\ a.\ \mathsf{weak}\ \ (f\ \ a)) \\
&\quad : \; \forall A \forall B.\ (A \to \mathsf{Lam}\ \ B) \to 1 + A \to \mathsf{Lam}\,(1 + B) \\
\mathsf{subst} &:= \mathsf{GIt}^{\kappa 1}(m, s) \\
&\quad : \; \mathsf{Lam} \leq^{\kappa 1}_{\mathsf{Lam}} \mathsf{Lam} = \forall A \forall B.\ (A \to \mathsf{Lam}\ \ B) \to (\mathsf{Lam}\ \ A \to \mathsf{Lam}\ \ B)
\end{aligned}
$$

where

$$
\begin{aligned}
F' &:= \lambda X \lambda A.\ \mathsf{Lam}\ A + (X\ A \times X\ A + X\ (1+A)) : \kappa 2\\
m &:= \lambda g \lambda f \lambda t.\ \mathsf{case}\,(t,\ a.\ \mathsf{inl}\ (f\ a),\ t'.\ \mathsf{inr}\ (\mathsf{case}\,(t',\\
&\qquad p.\ \mathsf{inl}\ \langle g\ f\ (p.0), g\ f\ (p.1)\rangle,\ r.\ \mathsf{inr}\ (g\ (\mathsf{lift}\ f)\ r))))\\
&\ :\ \forall X \forall Y.\ X \le^{\kappa 1}_{\mathsf{Lam}} Y \to \mathsf{LamF}\, X \le^{\kappa 1}_{\mathsf{Lam}} F'\ Y\\
&\ =\ \forall X \forall Y.\ (\forall A \forall B.\ (A \to \mathsf{Lam}\ B) \to X\ A \to Y\ B)\\
&\qquad \to \forall A \forall B.\ (A \to \mathsf{Lam}\ B)\\
&\qquad\quad \to A + (X\ A \times X\ A + X\ (1+A))\\
&\qquad\qquad \to \mathsf{Lam}\ B + (Y\ B \times Y\ B + Y\ (1+B))\\
s &:= \lambda t.\ \mathsf{case}\,(t,\ u.\, u,\ t'.\, \mathsf{case}\,(t',\ p.\ \mathsf{app}\ (p.0)\ (p.1),\ r.\ \mathsf{abs}\ \ r))\\
&\ :\ F'\ \mathsf{Lam} \subseteq^{\kappa 1} \mathsf{Lam}\\
&\ =\ \forall A.\ \mathsf{Lam}\ A + (\mathsf{Lam}\ A \times \mathsf{Lam}\ A + \mathsf{Lam}\,(1+A)) \to \mathsf{Lam}\ A
\end{aligned}
$$

Notice that we use weakening weak of example 11 in the definition of lifting lift, implicitly embedding It^ω into GIt^ω. subst has exactly the expected reduction behavior ($f : A \to \mathsf{Lam}\, B$ is the substitution rule, $\mathsf{subst}\ f : \mathsf{Lam}\, A \to \mathsf{Lam}\, B$ is the actual substitution function replacing the variables of its de Bruijn term argument according to the rule f):

$$
\begin{aligned}
&\mathsf{subst}\, f\,(\mathsf{var}\, a) &&\longrightarrow^+\ f\, a\\
&\mathsf{subst}\, f\,(\mathsf{app}\ t_1\, t_2) &&\longrightarrow^+\ \mathsf{app}\,(\mathsf{subst}\, f\, t_1)\,(\mathsf{subst}\, f\, t_2)\\
&\mathsf{subst}\, f\,(\mathsf{abs}\ r) &&\longrightarrow^+\ \mathsf{abs}\,(\mathsf{subst}\,(\mathsf{lift}\, f)\, r)
\end{aligned}
$$

Alternatively, we could replace s above by $\lambda t.\ \mathsf{case}\,(t,\ u.\, u,\ t'.\, \mathsf{in}^{\kappa 1}(\mathsf{inr}\ t'))$. Its drawback would be the same as in example 11: Only reducts of the desired right-hand sides would be achieved (again only due to the fact that the datatype constructors such as var are defined as constants, not as function symbols taking arguments).

7.3 Embeddings between GIt^ω *and* GMIt^ω

GIt^ω embeds into GMIt^ω much the same way as It^ω embeds into MIt^ω:

$$\mathsf{GIt}^\kappa(m,s) \quad := \mathsf{GMIt}^\kappa(\lambda it \lambda \vec{f} \lambda t.\, s\,(m\ it\ \vec{f}\, t))$$

$$\mathsf{GCoit}^\kappa(m,s) := \mathsf{GMCoit}^\kappa(\lambda coit \lambda \vec{f} \lambda t.\, m\ coit\ \vec{f}\,(s\, t))$$

The embedding of GMIt^ω into GIt^ω is much more interesting and again just a definitional embedding.

$$\begin{aligned}\mathsf{GMIt}^\kappa(s) &:= \mathsf{GIt}^\kappa(\mathsf{lanLeq}^{\kappa\to\kappa}\ \mathsf{id},\ \mathsf{leqLan}^{\kappa\to\kappa}\ s)\\ \mathsf{GMCoit}^\kappa(s) &:= \mathsf{GCoit}^\kappa(\mathsf{ranLeq}^{\kappa\to\kappa}\ \mathsf{id},\ \mathsf{leqRan}^{\kappa\to\kappa}\ s)\end{aligned}$$

Here, we used the definitions in the proof of lemma 3. It is easy to check that, with these definitions,

$$\begin{aligned}\mathsf{GMIt}^\kappa(s)\,\vec{f}\,(\mathsf{in}^\kappa\,t) &\longrightarrow^+ s\ \mathsf{GMIt}^\kappa(s)\,\vec{f}\,t\\ \mathsf{out}^\kappa(\mathsf{GMCoit}^\kappa(s)\,\vec{f}\,t) &\longrightarrow^+ s\ \mathsf{GMCoit}^\kappa(s)\,\vec{f}\,t\end{aligned}$$

Hence, the reductions are simulated. However, type-preservation has not yet been addressed.

Lemma 6 *Assume $\kappa = \vec{\kappa} \to *$, $n = |\vec{\kappa}|$, $\vec{\kappa}' = \vec{\kappa} \to \vec{\kappa}$, $F : \kappa \to \kappa$ and $\vec{H} : \vec{\kappa}'$. Define the constructor*

$$F' := \lambda Y \lambda \vec{Y} \exists X^\kappa.\ X \leq^\kappa_{\vec{H}} Y \to \mathsf{Lan}^\kappa(F\,X)(\vec{H}\,\vec{Y}) : \kappa \to \kappa.$$

Here, $(\vec{H}\,\vec{Y})$ means $(H_1\,Y_1)\ \ldots\ (H_n\,Y_n)$. Then, we have the following typings:

$$\begin{aligned}\mathsf{lanLeq}^{\kappa\to\kappa} &: \forall X^\kappa \forall Y^\kappa.\ F'Y \subseteq^\kappa F'Y \to X \leq^\kappa_{\vec{H}} Y \to F\,X \leq^\kappa_{\vec{H}} F'\,Y\\ \mathsf{leqLan}^{\kappa\to\kappa} &: \forall G^\kappa.\ (\forall X^\kappa.\ X \leq^\kappa_{\vec{H}} G \to F\,X \leq^\kappa_{\vec{H}} G) \to F'\,G \subseteq^\kappa G\end{aligned}$$

Redefine *the constructor F' to be*

$$F' := \lambda X \lambda \vec{X} \forall Y^\kappa.\ X\ {}_{\vec{H}}{\leq^\kappa}\ Y \to \mathsf{Ran}^\kappa(F\,Y)(\vec{H}\,\vec{X}).$$

Then, types can be assigned as follows:

$$\begin{aligned}\mathsf{ranLeq}^{\kappa\to\kappa} &: \forall X^\kappa \forall Y^\kappa.\ F'X \subseteq^\kappa F'X \to X\ {}_{\vec{H}}{\leq^\kappa}\ Y \to F'\,X\ {}_{\vec{H}}{\leq^\kappa}\ F\,Y\\ \mathsf{leqRan}^{\kappa\to\kappa} &: \forall G^\kappa.\ (\forall X^\kappa.\ G\ {}_{\vec{H}}{\leq^\kappa}\ X \to G\ {}_{\vec{H}}{\leq^\kappa}\ F\,X) \to G \subseteq^\kappa F'\,G\end{aligned}$$

Proof. By simple unfolding of the definitions of Lan^κ and Ran^κ. Observe that the $\vec{H}$ enter only the arguments; the Kan extensions are not formed along the $\vec{H}$. The premisses $F'Y \subseteq^\kappa F'Y$ and $F'X \subseteq^\kappa F'X$ are just there for perfect fit with the definitions of lanLeq and ranLeq. They will later always be instantiated by id. □

With the lemma at hand, the above-defined embedding is easily seen to be type-preserving. Certainly, the name F' has been chosen to name the additional constructor which can freely be chosen in GIt^ω. For the inductive case, it is the definition involving Lan^κ, for the coinductive case, F' needs Ran^κ.

While all of GMIt^ω can be embedded into GIt^ω, using canonical definitions of F' and canonical terms m which do not have an interesting operational meaning, we have seen in the example 13 that the term m can really be problem-specific and even do the essential part of the computation. Many more such terms will be shown in section 9. They will be found in a systematic way by induction on the build-up of regular rank-2-constructors. Hence, they are still generic but much less uniform than those constructed in the embedding shown above.

8 Advanced Examples

Since all of the systems considered in this article definitionally embed into $\underline{\mathsf{MIt}}^\omega$, we do the following examples in $\underline{\mathsf{MIt}}^\omega$ and freely use the iteration schemes from everywhere (since, for $\longrightarrow^+$, there is no difference between the original systems and the embeddings). Therefore, we can also use every definition from the previous examples.

Example 15 (De Bruijn Terms with Explicit Substitutions) The examples 11 and 14 have shown that de Bruijn terms constitute a Kleisli triple $(\mathsf{Lam}, \mathsf{var}, \mathsf{subst})$ with unit $\mathsf{var} : \forall A.\, A \to \mathsf{Lam}\, A$ and bind operation

$$\mathsf{subst} : \forall A \forall B.\ (A \to \mathsf{Lam}\ B) \to (\mathsf{Lam}\ A \to \mathsf{Lam}\ B).$$

From the "Kleisli triple" formulation of Lam we mechanically get the "monad" formulation $(\mathsf{Lam}, \mathsf{var}, \mathsf{flatten})$ with $\mathsf{flatten}$ the join or multiplication operation of the monad, since $\mathsf{flatten} : \forall A.\ \mathsf{Lam}\,(\mathsf{Lam}\ A) \to \mathsf{Lam}\ A$ can be obtained from subst as $\mathsf{flatten} := \mathsf{subst}\ \mathsf{id}$.

Consider now an extension of de Bruijn terms with explicit flattening which is a special form of explicit substitution. This truly nested datatype is definable as follows:

$$\begin{aligned}
\widehat{\mathsf{LamF}} &:= \lambda X \lambda A.\ \mathsf{LamF}\ X\ A + X\,(X\ A) : \kappa 2\\
\widehat{\mathsf{Lam}} &:= \mu^{\kappa 1}\, \widehat{\mathsf{LamF}} : \kappa 1\\
\widehat{\mathsf{var}}\,(a) &:= \mathsf{in}^{\kappa 1}\,(\mathsf{inl}\,(\mathsf{inl}\ a))\\
\widehat{\mathsf{app}}\,(t_1, t_2) &:= \mathsf{in}^{\kappa 1}\,(\mathsf{inl}\,(\mathsf{inr}\,(\mathsf{inl}\,\langle t_1, t_2\rangle)))\\
\widehat{\mathsf{abs}}\,(r) &:= \mathsf{in}^{\kappa 1}\,(\mathsf{inl}\,(\mathsf{inr}\,(\mathsf{inr}\ r)))\\
\widehat{\mathsf{flat}}\,(e) &:= \mathsf{in}^{\kappa 1}\,(\mathsf{inr}\ e)
\end{aligned}$$

Then, one has the following derived typings:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \widehat{\mathsf{var}}\,(a) : \widehat{\mathsf{Lam}}\ A} \qquad \frac{\Gamma \vdash t_1 : \widehat{\mathsf{Lam}}\ A \quad \Gamma \vdash t_2 : \widehat{\mathsf{Lam}}\ A}{\Gamma \vdash \widehat{\mathsf{app}}\,(t_1, t_2) : \widehat{\mathsf{Lam}}\ A}$$

$$\frac{\Gamma \vdash r : \widehat{\mathsf{Lam}}\,(1 + A)}{\Gamma \vdash \widehat{\mathsf{abs}}\,(r) : \widehat{\mathsf{Lam}}\ A} \qquad \frac{\Gamma \vdash e : \widehat{\mathsf{Lam}}\,(\widehat{\mathsf{Lam}}\ A)}{\Gamma \vdash \widehat{\mathsf{flat}}\,(e) : \widehat{\mathsf{Lam}}\ A}$$

Renaming of free variables in a term is implemented by the canonical monotonicity witness of $\widehat{\mathsf{Lam}}$, derived from the following generic monotonicity witness $\widehat{\mathsf{lamf}}$ for the datatype functor $\widehat{\mathsf{LamF}}$, using lamf from example 11:

$$\begin{aligned} \widehat{\mathsf{lamf}} &:= \lambda g \lambda f.\ \mathsf{either}\,(\mathsf{lamf}\ g\ f)\,(g\,(g\ f)) : \mathsf{mon}^{\kappa 2}\ \widehat{\mathsf{LamF}} \\ \widehat{\mathsf{lam}} &:= \mathsf{M}_{\mu}^{\kappa 1}(\widehat{\mathsf{lamf}}) = \mathsf{It}^{\kappa 1}(\widehat{\mathsf{lamf}}, \mathsf{in}^{\kappa 1}) \qquad : \mathsf{mon}^{\kappa 1}\ \widehat{\mathsf{Lam}} \end{aligned}$$

Note that the treatment of explicit flattening in the definition of $\widehat{\mathsf{lamf}}$ would be impossible with basic monotonicity $\underline{\mathsf{mon}}^{\kappa 1}$, see lemma 5. The following reduction behavior immediately follows:

$$\begin{aligned} \widehat{\mathsf{lam}}\ f\ \ \widehat{\mathsf{var}}\,(a) &\longrightarrow^{+} \widehat{\mathsf{var}}\,(f\ a) \\ \widehat{\mathsf{lam}}\ f\ \ \widehat{\mathsf{app}}\,(t_1, t_2) &\longrightarrow^{+} \widehat{\mathsf{app}}\,(\widehat{\mathsf{lam}}\ f\ t_1, \widehat{\mathsf{lam}}\ f\ t_2) \\ \widehat{\mathsf{lam}}\ f\ \ \widehat{\mathsf{abs}}\,(r) &\longrightarrow^{+} \widehat{\mathsf{abs}}\,(\widehat{\mathsf{lam}}\,(\mathsf{maybe}\ f)\ r) \\ \widehat{\mathsf{lam}}\ f\ \ \widehat{\mathsf{flat}}\,(e) &\longrightarrow^{+} \widehat{\mathsf{flat}}\,(\widehat{\mathsf{lam}}\,(\widehat{\mathsf{lam}}\ f)\ e) \end{aligned}$$

As was the case with the summation for the truly nested datatype Bush in example 8, the termination of $\widehat{\mathsf{lam}}$ is not obvious at all: In the recursive call in the last line, the parameter f is changed to $\widehat{\mathsf{lam}}\ f$, hence using the whole iteratively defined function $\widehat{\mathsf{lam}}\ f$. Nevertheless, termination holds by strong normalization of $\underline{\mathsf{MIt}}^{\omega}$.

Using $\widehat{\mathsf{lam}}$, we can represent full *explicit* substitution

$$\begin{aligned} \mathsf{esubst} &: \ \ \forall A \forall B.\ (A \to \widehat{\mathsf{Lam}}\ B) \to (\widehat{\mathsf{Lam}}\ A \to \widehat{\mathsf{Lam}}\ B) \\ \mathsf{esubst} &:= \lambda f \lambda t.\ \widehat{\mathsf{flat}}\,(\widehat{\mathsf{lam}}\ f\ t) \end{aligned}$$

esubst is explicit substitution in the sense that a term of the form $\widehat{\mathsf{flat}}\,(r)$ is returned for $\mathsf{esubst}\ f\ t$, hence only renaming but no substitution is carried out.

Example 16 (Resolution of Explicit Substitutions) The set of de Bruijn terms Lam can be embedded into the set of de Bruijn terms $\widehat{\mathsf{Lam}}$ with explicit flattening. The embedding function $\mathsf{emb} : \forall A.\ \mathsf{Lam}\ A \to \widehat{\mathsf{Lam}}\ A$ can be defined by iteration in a straightforward manner. The other direction is handled by a function $\mathsf{eval} : \forall A.\ \widehat{\mathsf{Lam}}\ A \to \mathsf{Lam}\ A$ which has to resolve the explicit flattenings. With the help of $\mathsf{It}^{\kappa 1}$, this is defined by

$$\begin{aligned} \mathsf{eval}' &:= \mathsf{It}^{\kappa 1}(\widehat{\mathsf{lam}}\mathsf{f}, s) : \widehat{\mathsf{Lam}} \leq^{\kappa 1} \mathsf{Lam} \\ \mathsf{eval} &:= \mathsf{eval}'\ \mathsf{id} \qquad : \widehat{\mathsf{Lam}} \subseteq^{\kappa 1} \mathsf{Lam} \end{aligned}$$

where $s := \lambda t.\ \mathsf{case}\,(t,\ t'.\,\mathsf{in}^{\kappa 1}\ t',\ e.\ \mathsf{flatten}\,e) : \widehat{\mathsf{LamF}}\,\mathsf{Lam} \subseteq^{\kappa 1} \mathsf{Lam}$, with $\mathsf{flatten}$ taken from the previous example. The most interesting case of the reduction behavior is

$$\mathsf{eval}'\ f\ \ \widehat{\mathsf{flat}}\,(e) \longrightarrow^{+} \mathsf{flatten}\,(\mathsf{eval}'\,(\mathsf{eval}'\,f)\,e).$$

More specifically, $\mathsf{eval}\ \ \widehat{\mathsf{flat}}\,(e) \longrightarrow^{+} \mathsf{flatten}\,(\mathsf{eval}'\ \mathsf{eval}\ e)$, with the following types: $e : \widehat{\mathsf{Lam}}\,(\widehat{\mathsf{Lam}}\ A)$, $\mathsf{eval} : \widehat{\mathsf{Lam}}\ A \to \mathsf{Lam}\ A$, and finally $\mathsf{eval}'\ \mathsf{eval} : \widehat{\mathsf{Lam}}\,(\widehat{\mathsf{Lam}}\ A) \to \mathsf{Lam}\,(\mathsf{Lam}\ A)$.

To conclude, $\mathsf{eval}\,(\mathsf{esubst}\,f\,t) \longrightarrow^{+} \mathsf{flatten}\,(\mathsf{eval}'\ \mathsf{eval}\ (\widehat{\mathsf{lam}}\,f\,t))$.

Example 17 (Redecoration of Wellfounded Triangles) Although we will pass from infinite triangular matrices to finite (well-founded) ones, the following example is *not* a dualization of example 6 since

- the underlying rank-2 constructor TriF is not dualized but only modified in order to avoid trivialities arising from taking the least fixed-point, and
- the programming task is not dualized from redecoration to substitution, but redecoration is kept.

Again, fix a type $E : *$ of matrix elements. The type $\mathsf{FTri}\,A$ of finite triangular matrices with diagonal elements in A and ordinary elements E can be obtained as follows:

$$\begin{aligned} \mathsf{FTriF} &:= \lambda X \lambda A.\ A \times (1 + X(E \times A)) : \kappa 2 \\ \mathsf{FTri} &:= \mu^{\kappa 1}\,\mathsf{FTriF} \qquad\qquad : \kappa 1 \end{aligned}$$

The columnwise decomposition and visualization of elements of type $\mathsf{FTri}\ A$ is done as for the infinite triangles of type $\mathsf{Tri}\ A$. Finiteness arises from taking the least fixed-point. By taking the left injection into the sum $1 + \ldots$, one can construct elements without further recurrence, hence the type $\mathsf{FTri}\ A$ is not empty unless A is. More generally, elements of type $\mathsf{FTri}\ A$ are constructed by means of

$$\begin{aligned} \mathsf{sg}\,(a) &:= \mathsf{in}^{\kappa 1}\,\langle a, \mathsf{inl}\,\langle\rangle\rangle \\ \mathsf{cons}\,(a,t) &:= \mathsf{in}^{\kappa 1}\,\langle a, \mathsf{inr}\ t\rangle \end{aligned}$$

Then, one gets the derived typing rules

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{sg}\,(a) : \mathsf{FTri}\ A} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash t : \mathsf{FTri}\,(E \times A)}{\Gamma \vdash \mathsf{cons}\,(a,t) : \mathsf{FTri}\ A}$$

There are two monotonicity witnesses for FTriF, for $\underline{\mathsf{mon}}^{\kappa 2}$ and for $\mathsf{mon}^{\kappa 2}$:

$$\begin{array}{lll}\underline{\mathsf{ftrif}} & := \lambda g.\ \mathsf{pair\ id\ (maybe}\ g) & : \underline{\mathsf{mon}}^{\kappa 2} \\ \mathsf{ftrif} & := \lambda g \lambda f.\ \mathsf{pair}\ f\ (\mathsf{maybe}\ (g\ (\mathsf{pair\ id}\ f))) & : \mathsf{mon}^{\kappa 2} \\ \mathsf{ftri} & := \mathsf{M}^{\kappa 1}_{\mu}(\mathsf{ftrif}) & : \mathsf{mon}^{\kappa 1}\ \mathsf{FTri}\end{array}$$

(The last definition uses means of It^{ω}.) By help of $\underline{\mathsf{It}}^{\omega}$, define the function $\mathsf{ftop} : \forall A.\ \mathsf{FTri}\ A \to A$ that yields the topmost diagonal element:

$$\mathsf{ftop} := \underline{\mathsf{It}}^{\kappa 1}(\underline{\mathsf{ftrif}}, \pi_0)$$

with $\pi_0 := \lambda p.\ (p.0)$. Also set $\pi_1 := \lambda p.\ (p.1)$. As reduction behavior, we get

$$\begin{array}{ll}\mathsf{ftop\ \ sg}\,(a) & \longrightarrow^{+} a \\ \mathsf{ftop\ \ cons}\,(a, t) & \longrightarrow^{+} a\end{array}$$

(An alternative definition of ftop in It^{ω} would be $\mathsf{It}^{\kappa 1}(\mathsf{ftrif}, \pi_0)\ \mathsf{id}$.)

We need to define a function fcut that cuts off the top row of a trapezium $\mathsf{FTri}\,(E \times A)$ to obtain a triangle $\mathsf{FTri}\ A$. This does not fit very well with any of our iteration schemes. Aiming at using GMIt^{ω}, we want to define a more general function $\mathsf{fcut}' : \mathsf{FTri} \le^{\kappa 1}_{H} \mathsf{FTri}$ with $H := \lambda A.\ E \times A$. Then,

$$\mathsf{fcut} := \mathsf{fcut}'\ \mathsf{id} : \forall A.\ \mathsf{FTri}\,(E \times A) \to \mathsf{FTri}\ A$$

Definition and reduction behavior[11] of fcut' are as follows:

$$\begin{array}{lll}\mathsf{fcut}' & := & \mathsf{GMIt}^{\kappa 1}\Big(\lambda fcut' \lambda f \lambda x.\,\mathsf{case}\,\Big(x.1, \\ & & \quad u.\ \mathsf{sg}\,((f\ (x.0)).1), \\ & & \quad v.\,\mathsf{cons}\,((f\ (x.0)).1, fcut'(\mathsf{pair\ id}\ f)\ v)\Big)\Big) \\ & : & \forall A \forall B.\,(A \to E \times B) \to \mathsf{FTri}\ A \to \mathsf{FTri}\ B \\ \mathsf{fcut}'\ f\ \ \mathsf{sg}\,(a) & \longrightarrow^{+} & \mathsf{sg}\,((f\,a).1) \\ \mathsf{fcut}'\ f\ \mathsf{cons}\,(a, t) & \longrightarrow^{+} & \mathsf{cons}\,((f\,a).1, \mathsf{fcut}'\,(\mathsf{pair\ id}\ f)\,t)\end{array}$$

For the definition of redecoration, we will again need a means of lifting a redecoration rule on triangles to one on trapeziums. It is defined precisely as

[11] If one added a tiny bit of extensionality, one would have extensional equality of $\mathsf{pair\ id\ id}$ and id, and hence get extensional equality of $\mathsf{fcut\ \ sg}\,(\langle e, a\rangle)$ and a, as well as that of $\mathsf{fcut\ cons}\,(\langle e, a\rangle, t)$ and $\mathsf{cons}\,(a, \mathsf{fcut}\,t)$.

in example 6, but with the new auxiliary functions.

$$\begin{aligned}\mathsf{flift} &:= \lambda f \lambda t.\, \langle (\mathsf{ftop}\ t).0,\ f\,(\mathsf{fcut}\ t)\rangle \\ &:\quad \forall A \forall B.\ (\mathsf{FTri}\ A \to B) \to \mathsf{FTri}\,(E \times A) \to E \times B\end{aligned}$$

Finally, we can define redecoration fredec. Its description is the same as that for redec that works on infinite triangles in example 6. Setting

$$G := \lambda A \forall B.\, (\mathsf{FTri}\ A \to B) \to \mathsf{FTri}\ B,$$

its type will be

$$\mathsf{FTri} \subseteq^{\kappa 1} G = \forall A \forall B.\ \mathsf{FTri}\, A \to (\mathsf{FTri}\, A \to B) \to \mathsf{FTri}\, B.$$

(For convenience, we changed the two argument types w. r. t. redec.) Unfortunately, G is not a right Kan extension (which might have allowed a direct definition of fredec inside GMIt^ω), but $G = (\mathsf{Ran}^{\kappa 1}_{\mathsf{Id}}\, \mathsf{FTri}) \circ \mathsf{FTri}$. Moreover, fredec essentially will need primitive recursion, not just iteration. Since the present article confines itself to iteration, the standard trick with products is adopted to represent primitive recursion: We will define a more general function $\mathsf{fredec}' : \mathsf{FTri} \subseteq^{\kappa 1} \mathsf{FTri} \times^{\kappa 1} G$. It seems that any hardwired Kan extensions in the system would only complicate the following definition which is done in plain $\underline{\mathsf{MIt}}^\omega$.

$$\begin{aligned}\mathsf{fredec}' := \underline{\mathsf{MIt}}^{\kappa 1}\Big(\lambda \mathit{fredec}' \lambda t.\, \Big\langle &\mathsf{in}^{\kappa 1}\,(\underline{\mathsf{ftrif}}\ (\pi_0 \circ \mathit{fredec}')\ t), \\ \lambda f.\, \mathsf{in}^{\kappa 1}\Big\langle &f\,(\mathsf{in}^{\kappa 1}\,(\underline{\mathsf{ftrif}}\ (\pi_0 \circ \mathit{fredec}')\ t)), \\ &\mathsf{maybe}\ (\lambda x.\, (\pi_1 \circ \mathit{fredec}')\ x\ (\mathsf{flift}\, f))\ (t.1)\Big\rangle\Big\rangle\Big)\end{aligned}$$

In the typing derivation, one would assign types as follows:

$$\begin{array}{ll} \mathit{fredec}' & : X \subseteq^{\kappa 1} \mathsf{FTri} \times^{\kappa 1} G \\ \pi_0 \circ \mathit{fredec}' & : X \subseteq^{\kappa 1} \mathsf{FTri} \\ t & : \mathsf{FTriF}\ X\ A \\ \mathsf{in}^{\kappa 1}\,(\underline{\mathsf{ftrif}}\ (\pi_0 \circ \mathit{fredec}')\ t) & : \mathsf{FTri}\ A \\ f & : \mathsf{FTri}\ A \to B \\ \pi_1 \circ \mathit{fredec}' & : X\,(E \times A) \to G\,(E \times A) \\ \lambda x.\, (\pi_1 \circ \mathit{fredec}')\ x\ (\mathsf{flift}\, f) & : X\,(E \times A) \to \mathsf{FTri}\,(E \times B) \end{array}$$

Set $\mathsf{fid} := \pi_0 \circ \mathsf{fredec}' : \mathsf{FTri} \subseteq^{\kappa 1} \mathsf{FTri}$ and $\mathsf{fredec} := \pi_1 \circ \mathsf{fredec}' : \mathsf{FTri} \subseteq^{\kappa 1} G$.

Finally, observe the following reduction behavior:

$$\begin{array}{ll} \mathsf{fredec}'\ \ \mathsf{sg}\,(a) & \longrightarrow^+ \langle \mathsf{sg}\,(a), \lambda f.\ \mathsf{sg}\,(f\ \ \mathsf{sg}\,(a))\rangle \\ \mathsf{fredec}'\ \ \mathsf{cons}\,(a,t) & \longrightarrow^+ \langle \mathsf{cons}\,(a, \mathsf{fid}\,t), \lambda f.\,\mathsf{cons}\,(f\ \ \mathsf{cons}\,(a, \mathsf{fid}\,t), \mathsf{fredec}\ t\,(\mathsf{flift}\,f))\rangle \end{array}$$

Expressed solely in terms of fid and fredec, this gives

$$\begin{array}{ll} \mathsf{fid}\ \ \mathsf{sg}\,(a) & \longrightarrow^+ \mathsf{sg}\,(a) \\ \mathsf{fid}\ \mathsf{cons}\,(a,t) & \longrightarrow^+ \mathsf{cons}\,(a, \mathsf{fid}\,t) \\ \mathsf{fredec}\ \ \mathsf{sg}\,(a)\ f & \longrightarrow^+ \mathsf{sg}\,(f\ \ \mathsf{sg}\,(a)) \\ \mathsf{fredec}\ \ \mathsf{cons}\,(a,t)\ f & \longrightarrow^+ \mathsf{cons}\,(f\ \ \mathsf{cons}\,(a, \mathsf{fid}\,t), \mathsf{fredec}\ t\,(\mathsf{flift}\,f)) \end{array}$$

Apparently, fid is an iteratively defined identity, and $\mathsf{fid}\,x$ with x a variable does not reduce to x. Apart from this deficiency due to mere iteration, fredec behaves as specified.

9 Efficient Folds

Martin et al. (2002) have proposed what they call efolds (in shorthand for efficient folds) as a generalized iteration scheme for a certain syntactically delimited class of constructors in $\kappa 2$ which we call regular. We will now introduce a variant of the efolds and show that they are wittily constructed rather general instances of $\mathsf{GIt}^{\kappa 1}$.

Define regularity of a constructor $F : \kappa 2$ inductively as follows:

- if $Q : \kappa 1$, $M_Q : \mathsf{mon}^{\kappa 1}\ Q$, then $\lambda\_\lambda A.\ Q\ A$ is regular (constant functor),
- $\lambda X \lambda A.\ X\ A$ is regular (identity),
- if F_0, F_1 are regular, then $F_0 +^{\kappa 2} F_1$ is regular (sum),
- if F_0, F_1 are regular, then $F_0 \times^{\kappa 2} F_1$ is regular (product),
- if $Q : \kappa 1$, $M_Q : \mathsf{mon}^{\kappa 1}\ Q$ and F_0 is regular, then $\lambda X.\ Q \circ (F_0\ X)$ is regular,
- if F_0 is regular, then $F = \lambda X.\ X \circ (F_0\ X)$ is regular.

Note that the second case is actually covered by the sixth and the first (with $Q := \mathsf{Id}$), so strictly speaking it is not needed. (There is even further redundancy in the definition, but we shall ignore this here.)

Elimination rule:

$$\frac{\begin{array}{l} F : \kappa 2 \text{ regular} \\ H : \kappa 1 \\ \vec{d} : \vec{\mathsf{D}}^F_H \\ s : \mathsf{R}^F_H\ G \subseteq^{\kappa 1} G \end{array}}{\mathsf{efold}^F_H(\vec{d}, s) : \mu^{\kappa 1} F \leq^{\kappa 1}_H C}$$

Reduction rule:

$$\mathsf{efold}^F_H(\vec{d}, s)\ f\ (\mathsf{in}^{\kappa 1}\ t) \longrightarrow s\ (\mathsf{M}^F_H(\vec{d})\ \mathsf{efold}^F_H(\vec{d}, s)\ f\ t)$$

Here

$$\begin{array}{ll} \mathsf{R}^F_H & : \kappa 2 \\ \vec{\mathsf{D}}^F_H & : * \\ \mathsf{M}^F_H(\vec{d}) & : \forall X \forall Y.\ X \leq^{\kappa 1}_H Y \to F\ X \leq^{\kappa 1}_H \mathsf{R}^F_H\ Y \quad \text{if } \vec{d} : \vec{\mathsf{D}}^F_H \end{array}$$

are defined by induction on the regular construction of F which also settles $|\vec{\mathsf{D}}^F_H| = |\vec{d}|$.

Independent of the actual definition of R^F_H, $\vec{\mathsf{D}}^F_H$, $\mathsf{M}^F_H(\cdot)$, it is immediate from what we have already stated that efold is definable in terms of $\mathsf{GIt}^{\kappa 1}$ in a typing- and reduction-preserving way:

$$\mathsf{efold}^F_H(\vec{d}, s) := \mathsf{GIt}^{\kappa 1}(\mathsf{M}^F_H(\vec{d}), s)$$

Definition of R^F_H, $\vec{\mathsf{D}}^F_H$, $\mathsf{M}^F_H(\cdot)$ (inductive on the regular construction of F):

- if $Q : \kappa 1$, $M_Q : \mathsf{mon}^{\kappa 1}\ Q$, $F = \lambda\_\lambda A.\ Q\ A$, then

$$\begin{array}{ll} \mathsf{R}^F_H & := \lambda\_.\ Q \circ H \\ \vec{\mathsf{D}}^F_H & := () \\ \mathsf{M}^F_H() & := \lambda g \lambda f \lambda x.\ M_Q\ f\ x \end{array}$$

- if $F = \lambda X \lambda A.\ X\ A$, then

$$\begin{array}{ll} \mathsf{R}^F_H & := F \\ \vec{\mathsf{D}}^F_H & := (H \subseteq^{\kappa 1} H) \\ \mathsf{M}^F_H(d) & := \lambda g \lambda f \lambda x.\ g\ (d \circ f)\ x) \end{array}$$

- if F_0, F_1 are regular and $F = F_0 +^{\kappa 2} F_1$, then

$$\begin{aligned}
\mathsf{R}_H^F &:= \mathsf{R}_H^{F_O} +^{\kappa 2} \mathsf{R}_H^{F_1} \\
\vec{\mathsf{D}}_H^F &:= (\vec{\mathsf{D}}_H^{F_0}, \vec{\mathsf{D}}_H^{F_1}) \\
\mathsf{M}_H^F(\vec{d_0}, \vec{d_1}) &:= \lambda g \lambda f \lambda x.\ \mathsf{case}\,(x,\ y.\ \mathsf{inl}\ (\mathsf{M}_H^{F_0}(\vec{d_0})\ g\ f\ y),\ y.\ \mathsf{inr}\ (\mathsf{M}_H^{F_1}(\vec{d_1})\ g\ f\ y))
\end{aligned}$$

- if F_0, F_1 are regular and $F = F_0 \times^{\kappa 2} F_1$, then

$$\begin{aligned}
\mathsf{R}_H^F &:= \mathsf{R}_H^{F_O} \times^{\kappa 2} \mathsf{R}_H^{F_1} \\
\vec{\mathsf{D}}_H^F &:= (\vec{\mathsf{D}}_H^{F_0}, \vec{\mathsf{D}}_H^{F_1}) \\
\mathsf{M}_H^F(\vec{d_0}, \vec{d_1}) &:= \lambda g \lambda f \lambda x.\ \langle \mathsf{M}_H^{F_0}(\vec{d_0})\ g\ f\ (x.0), \mathsf{M}_H^{F_1}(\vec{d_1})\ g\ f\ (x.1) \rangle
\end{aligned}$$

- if $Q : \kappa 1$, $M_Q : \mathsf{mon}^{\kappa 1}\ Q$, $F = \lambda X.\ Q \circ (F_0\ X)$, then

$$\begin{aligned}
\mathsf{R}_H^F &:= \lambda X.\ Q \circ H \circ (F_0\ X) \\
\vec{\mathsf{D}}_H^F &:= (\forall X.\ \mathsf{R}_H^{F_0}\ X \subseteq^{\kappa 1} H \circ (F_0\ X), \vec{\mathsf{D}}_H^{F_0}) \\
\mathsf{M}_H^F(d, \vec{d_0}) &:= \lambda g \lambda f \lambda x.\ M_Q\ (d\ \circ (\mathsf{M}_H^{F_0}(\vec{d_0})\ g\ f))\ x
\end{aligned}$$

  Notice that, here, R_H^F is defined via F_0, not recursively via $\mathsf{R}_H^{F_0}$.
- if F_0 is regular and $F = \lambda X.\ X \circ (F_0\ X)$, then

$$\begin{aligned}
\mathsf{R}_H^F &:= F \qquad \text{(this is no typo!)} \\
\vec{\mathsf{D}}_H^F &:= (\forall X.\ \mathsf{R}_H^{F_0}\ X \subseteq^{\kappa 1} H \circ (F_0\ X), \vec{\mathsf{D}}_H^{F_0}) \\
\mathsf{M}_H^F(d, \vec{d_0}) &:= \lambda g \lambda f \lambda x.\ g\ (d\ \circ (\mathsf{M}_H^{F_0}(\vec{d_0})\ g\ f))\ x
\end{aligned}$$

Example 18 (Efolds for Powerlists) We apply the definitions of R_H^F, $\vec{\mathsf{D}}_H^F$, $\mathsf{M}_H^F(\vec{d})$ to $F := \mathsf{PListF} = \lambda X \lambda A.\ A + X(A \times A) : \kappa 2$. It is regular because it is a sum of $\lambda X \lambda A.\, A$ which is covered by the first case with $Q := \mathsf{Id}$, and of $\lambda X \lambda A.\ X(A \times A)$ which is derived from the last case since $\lambda X \lambda A.\, A \times A$ is again an instance of the first case. Setting $M_{\mathsf{Id}} := \lambda f \lambda x.\, f\, x$ for the canonical monotonicity witness of $\mathsf{Id} : \kappa 1$, we get

$$\begin{aligned}
\mathsf{R}_H^{\mathsf{PListF}} &= \lambda X \lambda A.\ H\ A + X\ (A\ \times\ A) \\
\vec{\mathsf{D}}_H^{\mathsf{PListF}} &= (\forall_\forall A.\ H\ A \times H\ A \to H\ (A \times A)) \\
\mathsf{M}_H^{\mathsf{PListF}}(d) &\longrightarrow^+ \lambda g \lambda f \lambda x.\ \mathsf{case}\Big(x,\ y.\ \mathsf{inl}\,(M_{\mathsf{Id}}\ f\ y),\ y.\ \mathsf{inr}\,(g\ (d \circ \mathsf{fork}\ \ f)\ y)\Big)
\end{aligned}$$

Hence we get

$$\begin{gathered}
d : \forall_\forall A.\ H\ A \times H\ A \to H\ (A \times A) \\
s : \forall A.\ H\ A + G\ (A \times A) \to G\ A \\
\vdots \\
\mathsf{efold}_H^{\mathsf{PListF}}(d, s) : \forall A \forall B.\ (A \to H\ B) \to \mathsf{PList}\ A \to G\ B
\end{gathered}$$

and

$$\begin{aligned}
\mathsf{efold}_H^{\mathsf{PListF}}(d, s)\ f\ (\mathsf{zero}\ \ a) &\longrightarrow^+ s\ (\mathsf{inl}\ (M_{\mathsf{Id}}\ f\ a)) \\
&\longrightarrow^+ s\ (\mathsf{inl}\ (f\ a)) \\
\mathsf{efold}_H^{\mathsf{PListF}}(d, s)\ f\ (\mathsf{succ}\ \ l) &\longrightarrow^+ s\ (\mathsf{inr}\ (\mathsf{efold}_H^{\mathsf{PListF}}(d, s)\ (d \circ \mathsf{fork}\ \ f)\ l))
\end{aligned}$$

Example 19 (Summing up a Powerlist) The definition of sum' in terms of $\mathsf{efold}_{\lambda\_.\ \mathsf{Nat}}^{\mathsf{PListF}}$ is our goal. Recall that $\mathsf{R}_H^{\mathsf{PListF}} = \lambda X \lambda A.\ H\ A + X\ (A\ \times\ A)$, so $\mathsf{R}_{\lambda\_.\ \mathsf{Nat}}^{\mathsf{PListF}} = \lambda X \lambda A.\ \mathsf{Nat} + X\ (A \times A)$.

$$\mathsf{sum}' := \mathsf{efold}_{\lambda\_.\ \mathsf{Nat}}^{\mathsf{PListF}}(d, s) : \forall A \forall\_.\ (A \to \mathsf{Nat}) \to \mathsf{PList}\ \ A \to \mathsf{Nat} \quad \text{, where}$$

$$\begin{aligned}
d &:= \lambda p.\ p.0\ +\ p.1 && : \forall_\forall_.\mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \\
s &:= \lambda x.\, \mathsf{case}\, (x,\ n.\, n,\ n.\, n) && : \forall_.\mathsf{Nat} + \mathsf{Nat} \to \mathsf{Nat}
\end{aligned}$$

The reduction behaviour is precisely that of example 7, as expected.

Writing our definition directly in terms of $\mathsf{GIt}^{\kappa 1}$ in Example 13, we saw that we could use $F' := \lambda X \lambda A.\ \mathsf{Nat} + X\ (Q\ A)$ with Q any constructor in $\kappa 1$. In $F' := \mathsf{R}_{\lambda\_.\ \mathsf{Nat}}^{\mathsf{PListF}}$, which is forced, when switching to $\mathsf{efold}_{\lambda\_.\ \mathsf{Nat}}^{\mathsf{ListF}}$, we have $Q := \lambda A.\ A \times A$.

Example 20 (Efolds for de Bruijn Terms) Regularity of $F := \mathsf{LamF} = \lambda X \lambda A.\ A + (X\ A \times X\ A + X(1 + A)) : \kappa 2$ can again easily be established by help of the first three rules and the last rule of the definition. Some calculation will give (maybe is the chosen monotonicity witness for $\lambda A.\ 1 + A$)

$$\begin{aligned}
\mathsf{R}_H^{\mathsf{LamF}} \quad &= \quad \lambda X \lambda A.\ H\ A + (X\ A \times X\ A + X(1 + A)) \\
\vec{\mathsf{D}}_H^{\mathsf{LamF}} \quad &= \quad (\forall A.\ H\ A \to H\ A,\ \forall A.\ H\ A \to H\ A, \\
&\qquad\qquad \forall_\forall A.\ 1 + H\ A \to H\ (1 + A)) \\
\mathsf{M}_H^{\mathsf{LamF}}(d_1, d_2, d) &\longrightarrow^+ \lambda g \lambda f \lambda x.\ \mathsf{case}\Big(x,\ a.\ \mathsf{inl}\, (M_{\mathsf{Id}}\ f\ a),\ y.\ \mathsf{inr}\ \mathsf{case}\Big(y, \\
&\qquad p.\ \mathsf{inl}\Big\langle g\ (d_1 \circ f)\ (p.0), g\ (d_2 \circ f)\ (p.1)\Big\rangle, \\
&\qquad r.\ \mathsf{inr}\, (g\ (d \circ (\mathsf{maybe}\, f))\ r)\Big)\Big)
\end{aligned}$$

Hence

$$
\begin{array}{c}
d_1 : \forall A.\ H\ A \to H\ A \\
d_2 : \forall A.\ H\ A \to H\ A \\
d : \forall\_\forall A.\ 1 + H\ A \to H\ (1 + A) \\
s : \forall A.\ H\ A + (G\ A \times G\ A + G\ (1 + A)) \to G\ A \\
\vdots \\
\mathsf{efold}^{\mathsf{LamF}}_{H}(d_1, d_2, d, s) : \forall A \forall B.\ (A \to H\ B) \to \mathsf{Lam}\ A \to G\ B
\end{array}
$$

and

$$
\begin{array}{lll}
\mathsf{efold}^{\mathsf{LamF}}_{H}(d_1, d_2, d, s)\ f\ (\mathsf{var}\ a) & \longrightarrow^{+} & s\ (\mathsf{inl}\ (M_{\mathsf{Id}}\ f\ a)) \\
& \longrightarrow^{+} & s\ (\mathsf{inl}\ (f\ a)) \\
\mathsf{efold}^{\mathsf{LamF}}_{H}(d_1, d_2, d, s)\ f\ (\mathsf{app}\ t_1\ t_2) & \longrightarrow^{+} & s\ (\mathsf{inr}\ (\mathsf{inl}\ \langle \mathsf{efold}^{\mathsf{LamF}}_{H}(d_1, d_2, d, s)\ (d_1 \circ f)\ t_1, \\
& & \qquad \mathsf{efold}^{\mathsf{LamF}}_{H}(d_1, d_2, d, s)\ (d_2 \circ f)\ t_2 \rangle)) \\
\mathsf{efold}^{\mathsf{LamF}}_{H}(d_1, d_2, d, s)\ f\ (\mathsf{abs}\ r) & \longrightarrow^{+} & s\ (\mathsf{inr}\ (\mathsf{inr}\ (\mathsf{efold}^{\mathsf{LamF}}_{H}(d_1, d_2, d, s) \\
& & \qquad (d \circ (\mathsf{maybe}\, f))\ r)))
\end{array}
$$

Example 21 (Renaming and Substitution for de Bruijn Terms) The functions lam and subst can be given the definitions

$$
\begin{array}{lll}
\mathsf{lam} & := \mathsf{efold}^{\mathsf{LamF}}_{\mathsf{Id}}(\mathsf{id}, \mathsf{id}, \mathsf{id}, \mathsf{in}^{\kappa 1}) & : \mathsf{Lam} \leq^{\kappa 1}_{\mathsf{Id}} \mathsf{Lam} \\
\mathsf{weak} & := \mathsf{lam}\ (\lambda a.\ \mathsf{inr}\ a) & : \forall A.\ \mathsf{Lam}\ A \to \mathsf{Lam}\ (1 + A) \\
\mathsf{subst} & := \mathsf{efold}^{\mathsf{LamF}}_{\mathsf{Lam}}(\mathsf{id}, \mathsf{id}, \mathsf{lift}\ \mathsf{id}, s) & : \mathsf{Lam} \leq^{\kappa 1}_{\mathsf{Lam}} \mathsf{Lam},
\end{array}
$$

where we use lift and s from example 14.

Since composition with the identitiy does an η-expansion of the term, the reduction behavior of lam is not exactly as in example 11. This problem is even more visible for subst:

$$
\begin{array}{lll}
\mathsf{subst}\ f\,(\mathsf{var}\ a) & \longrightarrow^{+} & f\, a \\
\mathsf{subst}\ f\,(\mathsf{app}\ t_1\, t_2) & \longrightarrow^{+} & \mathsf{app}\,(\mathsf{subst}\,(\lambda x.\ f\, x)\, t_1)\,(\mathsf{subst}\,(\lambda x.\ f\, x)\, t_2) \\
\mathsf{subst}\ f\,(\mathsf{abs}\ r) & \longrightarrow^{+} & \mathsf{abs}\,(\mathsf{subst}\,(\lambda x.\, \mathsf{lift}\ \mathsf{id}\,(\mathsf{maybe}\ f\, x))\, r)
\end{array}
$$

We would have liked to see $\mathsf{lift}\, f$ instead of $\lambda x.\, \mathsf{lift}\ \mathsf{id}\,(\mathsf{maybe}\ f\, x)$. Extensionally, they are equal: Both $\longrightarrow^{+}$-reduce to $\mathsf{var}\,(\mathsf{inl}\ u)$ for argument $\mathsf{inl}\ u$, and both $\longrightarrow^{+}$-reduce to $\mathsf{weak}\,(f\, a)$ for argument $\mathsf{inr}\, a$. Certainly, also $\lambda x.\, f\, x$ and f are extensionally equal.

In much the same way as the efficient folds of Martin et al. (2002) are an instance of generalized refined iteration, the original generalized folds of Bird and Paterson (1999a) are an instance of a scheme of generalized basic conventional iteration. This scheme, however, has a certain deficiency to it, and we have therefore chosen to not describe it here. We only mention that the deficiency in question is also the root cause to the "efficiency problem" of generalized folds for which the efficient folds were invented as a remedy.

10 Conclusion

We have put forth and studied several possible formulations of iteration and coiteration for (co)inductive constructors of higher kinds. The main conclusion is that there is no single best solution.

Of the systems considered here, GMIt^ω and GIt^ω are clearly the most advanced in terms of direct expressive power. In particular, this is witnessed by the fact that the efolds of Martin et al. (2002) are very straightforwardly defined in GIt^ω. But for many applications, the more basic $\underline{\mathsf{It}}^\omega$ and $\underline{\mathsf{MIt}}^\omega$ are perfectly sufficient and invoking GMIt^ω or GIt^ω to program these would occur as an exaggeration. There are many interesting cases where the more basic systems $\underline{\mathsf{It}}^\omega$ and $\underline{\mathsf{MIt}}^\omega$ do not suffice to express the algorithmic idea appropriately, but where the freedom in choosing the additional parameters $\vec{H}$ in both GMIt^ω and GIt^ω, and parameter F' in GIt^ω, is not needed. These more rigid typings are embodied in the intermediary systems MIt^ω and It^ω, which have exactly the reduction behavior of their "generalized" versions GMIt^ω and GIt^ω but are easier to typecheck in practice.

In GMIt^ω and $\underline{\mathsf{MIt}}^\omega$, where the iterator and coiterator are Mendler-style, their computational behavior is very close to `letrec`, except that termination of computations is guaranteed.

As demonstrated with the advanced examples, in practice, an eclectic view is most helpful: in principle, we would program in $\underline{\mathsf{MIt}}^\omega$, but constantly use the expressive capabilities of the other systems, viewed as macro definitions on top of $\underline{\mathsf{MIt}}^\omega$.

Acknowledgments Peter Hancock deserves thanks for suggesting to us the definition $\mathsf{mon}\ F := F \leq F$. We also highly appreciate the the feedback we got from Peter Aczel at TYPES'03 in Turin, April/May 2003.

The first author acknowledges the support by the PhD Programme Logic in Computer Science (GKLI) of the *Deutsche Forschungs-Gemeinschaft.* The third author was partially supported by the Estonian Science Foundation under grants no. 4155 and 5677.

All three authors received support also from the FP5 thematic network project TYPES (IST-1999-29001).

References

Abel, A., Matthes, R., 2003. (Co-)iteration for higher-order nested datatypes. In: Geuvers, H., Wiedijk, F. (Eds.), Types for Proofs and Programs, International Workshop, TYPES 2002. Vol. 2646 of Lecture Notes in Computer Science. Springer, Berg en Dal, The Netherlands, pp. 1–20. URL `http://www.tcs.informatik.uni-muenchen.de/~abel/types02.pdf`

Abel, A., Matthes, R., Uustalu, T., April 2003. Generalized iteration and coiteration for higher-order nested datatypes. In: Gordon, A. (Ed.), Foundations of Software Science and Computation Structures (FoSSaCS 2003). Vol. 2620 of Lecture Notes in Computer Science. Springer, Warsaw, Poland, pp. 54–69.

Altenkirch, T., McBride, C., 2002. Generic programming within dependently typed programming, to appear in the proceedings of WCGP'02.

Altenkirch, T., Reus, B., 1999. Monadic presentations of lambda terms using generalized inductive types. In: Flum, J., Rodríguez-Artalejo, M. (Eds.), Computer Science Logic, 13th International Workshop, CSL '99, Proceedings. Vol. 1683 of Lecture Notes in Computer Science. Springer, pp. 453–468.

Bird, R., Gibbons, J., Jones, G., 2000. Program optimisation, naturally. In: Millenial Perspectives in Computer Science. Palgrave. URL `http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/index.html#naturally`

Bird, R., Meertens, L., 1998. Nested datatypes. In: Jeuring, J. (Ed.), Mathematics of Program Construction, MPC'98, Proceedings. Vol. 1422 of Lecture Notes in Computer Science. Springer, pp. 52–67.

Bird, R., Paterson, R., 1999a. Generalised folds for nested datatypes. Formal Aspects of Computing 11 (2), 200–222. URL `citeseer.nj.nec.com/bird99generalised.html`

Bird, R. S., Paterson, R., 1999b. De Bruijn notation as a nested datatype. Journal of Functional Programming 9 (1), 77–91. URL `citeseer.nj.nec.com/bird98de.html`

Hagino, T., 1987. A typed lambda calculus with categorical type constructors. In: Pitt, D. H., Poigné, A., Rydeheard, D. E. (Eds.), Category Theory and Computer Science. Vol. 283 of Lecture Notes in Computer Science. Springer, pp. 140–157.

Haskell Mailing List, July 25 2002. Markus schnell's message `slide: useful function?` URL `http://www.mail-archive.com/haskell@haskell.org/`

Hinze, R., July 2000. Efficient generalized folds. In: Jeuring, J. (Ed.), Proceedings of the Second Workshop on Generic Programming, WGP 2000. Ponte de Lima, Portugal.

Hinze, R., 2002. Polytypic values possess polykinded types. Science of Computer Programming, MPC Special Issue 43, 129–159.
URL `http://www.informatik.uni-bonn.de/~ralf/publications.html`

Mac Lane, S., 1998. Categories for the Working Mathematician, 2nd Edition. Vol. 5 of Graduate Texts in Mathematics.

Martin, C., Gibbons, J., Bayley, I., April 2002. Disciplined, efficient, generalised folds for nested datatypes, submitted for publication.
URL `http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/index.html#wcgp`

Matthes, R., 1998. Extensions of system F by iteration and primitive recursion on monotone inductive types. Doktorarbeit (PhD thesis), University of Munich, available via the homepage `http://www.tcs.informatik.uni-muenchen.de/~matthes/`.

Matthes, R., 1999. Monotone (co)inductive types and positive fixed-point types. Theoretical Informatics and Applications 33 (4/5), 309–328.

Mendler, N. P., 1987. Recursive types and type constraints in second-order lambda calculus. In: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, N.Y. IEEE Computer Society Press, pp. 30–36.

Mendler, N. P., 1991. Inductive types and type constraints in the second-order lambda calculus. Annals of Pure and Applied Logic 51 (1–2), 159–172.

Pfenning, F., Lee, P., 1989. LEAP: A language with eval and polymorphism. In: TAPSOFT, Vol.2. Vol. 352 of Lecture Notes in Computer Science. Springer, pp. 345–359.

Pierce, B., Dietzen, S., Michaylov, S., 1989. Programming in higher-order typed lambda-calculi. Tech. rep., Carnegie Mellon University.

Spławski, Z., Urzyczyn, P., 1999. Type Fixpoints: Iteration vs. Recursion. SIGPLAN Notices 34 (9), 102–113, proceedings of the 1999 International Conference on Functional Programming (ICFP), Paris, France.

Uustalu, T., Vene, V., 1997. A cube of proof systems for the intuitionistic predicate μ-, ν-logic. In: Haveraaen, M., Owe, O. (Eds.), Selected Papers from the 8th Nordic Workshop on Programming Theory (NWPT '96), Oslo, Norway, December 1996. Vol. 248 of Research Reports, Department of Informatics, University of Oslo. pp. 237–246.

Uustalu, T., Vene, V., 2002. The dual of substitution is redecoration. In: Hammond, K., Curtis, S. (Eds.), Trends in Functional Programming 3. Intellect, Bristol / Portland, OR, pp. 99–110.
URL `http://www.cs.ioc.ee/ tarmo/papers/sfp01-book.ps.gz`