Iteration and Coiteration Schemes for Higher-Order and Nested Datatypes

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

[a] *Department of Computer Science, Chalmers University of Technology Rännvägen 6, SWE-41296 Göteborg, Sweden*

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

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

Abstract

This article studies the implementation of inductive and coinductive constructors of higher kinds (higher-order nested datatypes) in typed term rewriting, with emphasis on the choice of the iteration and coiteration constructions to support as primitive. We propose and compare several well-behaved extensions of System F^ω with some form of iteration and coiteration uniform in all kinds. In what we call Mendler-style systems, the iterator and coiterator have a computational behavior similar to the general recursor, but their types guarantee termination. In conventional-style systems, monotonicity witnesses are used for a notion of monotonicity defined uniformly for all kinds. Our most expressive systems GMIt^ω and GIt^ω of generalized Mendler resp. conventional (co)iteration encompass Martin, Gibbons and Bailey's efficient folds for rank-2 inductive types. Strong normalization of all systems considered is proved by providing an embedding of the basic Mendler-style system MIt^ω into System F^ω.

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

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

1 Introduction and Overview

This article studies the implementation of higher-order inductive and coinductive constructors in the setting of typed rewriting. For introducing inductive and coinductive *types* to typed lambda calculi, there are several well-known non-controversial solutions. (Co)inductive types can be either added to first-order simply typed lambda calculus together with (co)iteration or primitive (co)recursion, or alternatively, (co)inductive types with (co)iteration may be encoded in System F. The systems so obtained are all well-behaved. In particular, typed terms are strongly normalizable.

But besides inductive and coinductive types, in programming, one can also encounter inductive and coinductive constructors of higher kinds. In the mathematics of program construction community, there is a line of work devoted to programming with nested datatypes, or second-order inductive constructors, which are least fixed-points of *type transformer transformers*. Therefore, basically, a nested datatype is a datatype with a type parameter, hence a family of datatypes. But 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.

Therefore, the notion of nested datatype is more liberal than that of a family of inductive datatypes where each family member can be fully understood in isolation from the rest of the family. The typical example of this simpler situation is the regular datatype constructor List, where, for any type A, the type $\mathsf{List}\,A$ of finite lists of elements taken from A, is a separate inductive datatype. Clearly, List can also be viewed as a quite trivial nested datatype. What interests us, however, are nested datatypes whose family members are interwoven. Interesting examples of these nested datatypes, studied in the literature, include perfectly balanced trees, red-black trees, trie data structures and syntax with variable binding. As early as 1998, Hinze employs nested datatypes for efficient data structures. Hinze (2001) gives a more detailed explanation how these datatypes are constructed. This includes a nested datatype for red-black trees, where the invariants of red-black trees are ensured by the types—unlike the implementation of the operations for red-black trees of Okasaki (1999b). Kahrs (2001) shows that—with some refinement—one can even get back the efficiency of Okasaki's implementation, despite this additional guarantee. Hence, nested datatypes provide more information on the stored data, while they need not slow down computation.

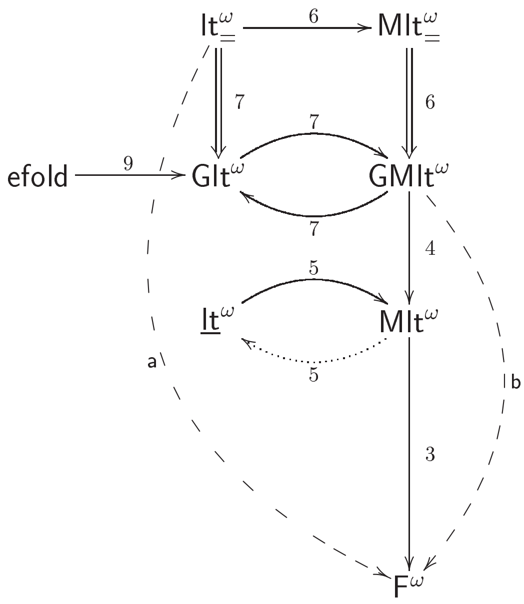
There already exist—see Section 9 for more bibliographical details—several

Table 1
Contents

suggestions concerning useful and well-behaved combinators for programming with them, which are different versions of folds, or iteration—demonstrating that, in higher kinds, it is not at all obvious what iteration should mean. Unfortunately, however, these works do not provide or hint at answers to questions central in rewriting such as reduction behaviours and, in particular, termination. This is because, in functional programming, the main motivation to program in a disciplined fashion (e.g., to program with structured (co)recursion combinators rather than with unstructured general recursion) is to be able to construct programs or optimize (e.g., deforest) them "calculationally", i.e., by means of equational reasoning.

In the typed lambda calculi and proof assistants communities, the motivation to rest on structured (co)recursion is more foundational—to ensure totality of definable functions or termination of typed programs, which, in systems of dependent types, is vital for type-checking purposes. This suggests the following research programme: take the solutions proposed within the functional programming community and see whether they solve the rewriting problems as well or, if they do not, admit elaborations which do. Another natural goal is to strive for solutions that scale up to higher-order (co)inductive constructors.

Contents. Table 1 displays the organization of the remainder of this article. In Section 2, we recapitulate System F^ω of higher-order polymorphism as a type assignment system for the λ-calculus with pairing (products), tagging (disjoint sums) and packing (existentials). In the following sections, we extend



Legend:	$\xRightarrow{n}$	special case of (see Section n)
	$\xrightarrow{n}$	definable in (see Section n)
	$\overset{5}{\dashrightarrow}$	embeddable into (see Section 5)
	$\overset{\mathsf{a}}{\dashrightarrow}$	direct definition (Abel and Matthes, 2003)
	$\overset{\mathsf{b}}{\dashrightarrow}$	direct definition (Abel, Matthes, and Uustalu, 2003)

Figure 1. Overview

F^ω by constants for inductive and coinductive constructors for arbitrary finite kinds together with iteration and coiteration schemes. Each scheme gives rise to a separate system which is named after its iterator. In the following, we will only talk about inductive constructors and iteration, although all systems support the dual concepts of coinductive constructors and coiteration as well. The superscript $^\omega$ indicates that inductive constructors of all higher kinds are available, which holds for all of our own systems, defined in sections 3–7.

Section 3 starts with an introduction to iteration à la Mendler for the first-order case, i.e., inductive types. The transfer of the central ideas to higher kinds by substituting natural transformations for functions results in the basic higher-order Mendler-style system MIt^ω which is embeddable into F^ω and generic enough to simulate all other systems. Figure 1 displays all embeddings carried out in the article. An arrow from A to B states that system A can be embedded into system B such that typing and reduction (i.e., computation) in system A are simulated in system B. The arrow styles indicate how direct the embedding is.

- A simple arrow from A to B, like from MIt^ω to F^ω, states that the new constants of system A, which form, introduce and eliminate inductive constructors, can be *defined* as terms of system B. Hence, system A simply provides new notation for existing constructs of system B. One also speaks of a shallow embedding.
- In contrast, a dotted arrow indicates a deep embedding. This means, all expressions of the source system have to be translated into appropriate expressions of the target system to simulate typing and reduction.
- Finally, the most direct embedding is displayed as a double arrow from A to B, meaning that system A is simply a restriction of system B. Consequently, the corresponding translation is the identity.

As Figure 1 illustrates, all other systems are definable in MIt^ω. But when it comes to practical usability, this system does not satisfy all wishes. As we will see in examples later, many programming tasks require a special pattern, namely Kan extensions, to be used in conjunction with Mendler iteration. Therefore, we have formulated the scheme of *generalized* Mendler iteration GIt^ω with hard-wired Kan extensions, presented in Section 4. The attribute *generalized* has been chosen in resemblance of Bird and Paterson's (1999a) generalized folds. System GMIt^ω has been first described in a previous publication (Abel, Matthes, and Uustalu, 2003) where a direct embedding into F^ω is given.

By now, the question how to generalize Mendler iteration—which resembles programming with general recursion—to higher kinds seems to be answered sufficiently. But what about conventional iteration, which is motivated by the view of inductive types as initial algebras in category theory? Can conventional iteration be generalized to higher kinds in the same way as Mendler iteration? What is the precise relationship of Mendler iteration and conventional iteration for higher kinds? These questions are addressed in the sections 5–7.

Conventional iteration can be formulated for types which are obtained as the least fixed point of a *monotone* type transformer. A crucial task in finding conventional iterators for higher kinds will therefore be the formulation of higher-rank monotonicity. Section 5 investigates the most basic notion: A type constructor is monotone if it preserves natural transformations. The resulting system, $\underline{\mathsf{It}}^\omega$, is sufficiently strong to simulate Mendler iteration via a deep embedding, but the notion of monotonicity lacks important closure properties. Hence, a refined notion of monotonicity, which uses Kan extensions along the identity, is put forth in Section 6. The induced System $\mathsf{It}^\omega_{=}$ has been treated before (Abel and Matthes, 2003). It is definable in F^ω, but also in terms of

Sec.	No.	Example	System(s)	Page
2	1	Booleans: true, false, if	F^ω	12
	2	Maybe: bind	F^ω + patterns	14
3	3	Lists: map	Haskell	15
	4	Lists: map	MIt^ω	17
	5	Streams: head and tail	MIt^ω	18
	6	Streams: upfrom	MIt^ω	18
	7	Powerlists: get	MIt^ω	21
	8	Powerlists: sum	MIt^ω	22
	9	Infinite triangles: decomposition	MIt^ω	24
	10	Streams: redecoration	Haskell	25
	11	Infinite triangles: redecoration	MIt^ω	25
4	12	Powerlists: sum	GMIt^ω	31
	13	Bushes: sum	GMIt^ω	31
	14	Infinite Triangles: redecoration	GMIt^ω	32
5	15	Powerlists: sum	$\underline{\mathsf{It}}^\omega$	37
6	16	De Bruijn terms: map	$\mathsf{It}^\omega_\leq$	47
	17	Powerlists: reverse	$\mathsf{It}^\omega_\leq$	48
7	18	Powerlists: sum	GIt^ω	50
	19	De Bruijn terms: substitution	GIt^ω	51
8	20	De Bruijn + expl. subst.: map	$\mathsf{It}^\omega_\leq$	54
	21	De Bruijn + expl. subst.: eval	$\mathsf{It}^\omega_\leq$	56
	22	Finite triangles: redecoration	$\underline{\mathsf{It}}^\omega$, GMIt^ω, MIt^ω	56
9	23	Hofunctors	efold	61
	24	Powerlists: efold (Typing)	GIt^ω	63
	25	Powerlists: sum (Typing)	efold	64
	26	Powerlists: efold (Reduction)	GIt^ω	64
	27	Powerlists: sum (Reduction)	efold	65
	28	De Bruijn terms: efold	GIt^ω	65
	29	De Bruijn terms: substitution	efold	66

Table 2
Overview of examples

Mendler iteration via a cut-down version of GMIt^ω, called $\mathsf{MIt}^\omega_=$, which only uses Kan extensions along the identity. As shown in loc. cit., programming in $\mathsf{It}^\omega_=$ often requires a second layer of Kan extensions. This flaw is remedied in System GIt^ω, the conventional counterpart of generalized Mendler iteration.

After completing the definition of our systems, some more examples demonstrate the applicability of the different iteration schemes for different purposes (Section 8). The remainder of the article is devoted to related work. Section 9 compares our work with iteration schemes for nested datatypes found in the literature. Special attention is given to the efficient folds of Martin, Gibbons, and Bayley (2003); a type-theoretic adaptation of their work is shown to be definable in System GIt^ω. Section 10 relates this work to generic and dependently typed programming, type classes and other trends in functional programming and type theory. Finally, the main contributions of this article are summarized in Section 11.

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. Table 2 contains a complete list of examples together with the system in which they have been implemented. Our running example is summation for powerlists which is has been defined in almost all of our systems. For the conventional iteration systems, most examples are centered around the representation of untyped de Bruijn-style lambda terms as a nested datatype.

Relation to our previous work. This article is an extended and reworked version of our conference paper (Abel, Matthes, and Uustalu, 2003) which mostly discussed system GMIt^ω (called MIt^ω in that article). New in this article are the basic systems MIt^ω 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 $\mathsf{It}^\omega_=$ (a subsystem of GIt^ω) and some examples are taken from Abel and Matthes (2003). As in our previous work, the typing and reduction rules for iteration and coiteration are uniform in all kinds, in each of these systems.

2 System F^ω

Our development of higher-order datatypes takes place within the Curry-style version of system F^ω, extended with binary sums and products, unit type and existential quantification over type constructors. We employ the usual named-variables syntax, but identify α-equivalent expressions that is properly achieved in the nameless version à la de Bruijn. Capture-avoiding substitution of an expression e for a variable x in an expression f is denoted by $f[x := e]$.

2.1 The Syntax

In System F^ω, there are three categories of expressions: kinds, type constructors and terms.

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

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

The *rank* of kind κ is denoted by $\mathsf{rk}(\kappa)$. We introduce abbreviations for some special kinds: $\mathsf{k0} = *$, *types*, $\mathsf{k1} = * \to *$, *unary type transformers* and $\mathsf{k2} = (* \to *) \to (* \to *)$ *unary transformers of type transformers*. Then, $\mathsf{rk}(\mathsf{k}i) = i$ 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 $\vec{\kappa} \to \kappa := \kappa_1 \to \ldots \to \kappa_n \to \kappa$, letting $\to$ associate to the right. Provided another sequence $\vec{\kappa}' = \kappa'_1, \ldots, \kappa'_n$ of the same length, i.e., $|\vec{\kappa}'| = |\vec{\kappa}|$, set the sequence $\vec{\kappa} \to \vec{\kappa}' := \kappa_1 \to \kappa'_1, \ldots, \kappa_n \to \kappa'_n$. This last abbreviation does not conflict with the abbreviation $\vec{\kappa} \to \kappa$ due to the required $|\vec{\kappa}'| = |\vec{\kappa}|$.

Type constructors. (Denoted by uppercase latin letters.) Metavariable X ranges over an infinite set of type 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 type 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 type $\forall X^\kappa.\, A$ 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 variable which never occurs free in a type constructor, hence we write $\lambda\_.F$ for void abstraction.

Type constructor application associates to the left, i.e., $F\,G\,H$ stands for $(F\,G)\,H$. For $\vec{F} = F_1, \ldots, F_n$ a vector of constructors, we abbreviate $F\,F_1\,\ldots\,F_n$ as $F\vec{F}$. We write Id for $\lambda X.X$ and $F \circ G$ for $\lambda X.\,F\,(G\,X)$.

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

$$\begin{aligned} r, s, t ::= {} & x \mid \lambda x.t \mid r\,s \mid \mathsf{inl}\,t \mid \mathsf{inr}\,t \mid \mathsf{case}\,(r,\,x.\,s,\,y.\,t) \\ & \mid \langle\rangle \mid \langle t_1, t_2\rangle \mid \mathsf{fst}\,r \mid \mathsf{snd}\,r \mid \mathsf{pack}\,t \mid \mathsf{open}\,(r,\,x.\,s) \end{aligned}$$

Most term forms are standard; "pack" introduces and "open" eliminates existential quantification, see below. The term former fst, whenever it is used without argument, should be understood as the first projection function $\lambda x.\,\mathsf{fst}\,x$. This holds analogously for snd, inl, inr and pack. The identity $\lambda x.x$ will be denoted by id. We write $f \circ g$ for the 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 $\vec{s} = s_1, \ldots, s_n$.

Note that there is *no* distinguished form of recursion in the language of System F^ω. In the progression of this article, however, we will extend the system by different forms of iteration.

2.2 Kinding and Typing

In the following, we define judgments to identify the "good" expressions. All kinds are good by definition, but good type constructors need to be wellkinded and good terms need to be welltyped. As an auxiliary notion, we need to introduce contexts which record kinds resp. types of free variables.

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 type 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 type variables and types (not arbitrary type constructors!) for object variables.

Wellkinded type 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 type 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 $A, B, C, D : *$, $G, H, X, Y : \mathsf{k1}$ and $F : \mathsf{k2}$. If the context is clear (by default, we take the empty context "$\cdot$"), we write $F : \kappa$ for $\Gamma \vdash F : \kappa$. Sums and products can be extended to all kinds: For $\kappa = \vec{\kappa} \to *$ with $|\vec{\kappa}| = |\vec{X}| = n$, 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 type constructors have kind $\kappa \to \kappa \to \kappa$. If no ambiguity arises, the superscript is omitted.

Equivalence on wellkinded type constructors. The notion of β-equivalence $F = F'$ for wellkinded type constructors F and F' is given as the compatible closure (i.e. closure under all type constructor forming operations) of the following axiom.

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

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

As a consequence of this identification, wellkinded type constructor composition $\circ$ is associative.

Welltyped terms. $\Gamma \vdash t : A$. The following chart recapitulates the typing rules for pure Curry-style F^ω.

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

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

More rules are needed for introduction and elimination of the System F^ω extensions: unit type, binary sum and product types, and existential types.

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

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

$$\frac{\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B} \qquad \frac{\Gamma \vdash r : A \times B}{\Gamma \vdash \mathsf{fst}\,r : A} \qquad \frac{\Gamma \vdash r : A \times B}{\Gamma \vdash \mathsf{snd}\,r : B}$$

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

As for wellkinded type 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. Let $\Gamma \vdash A : *$ and $\Gamma \vdash B : *$. We say 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.

2.3 Operational Semantics

Terms of System F^ω denote functional programs whose operational meaning is given by the following reduction relation.

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 formers.[1]

$$
\begin{array}{ll}
(\lambda x.t)\, s & \longrightarrow_\beta t[x := s] \\[1ex]
\mathsf{case}\,(\mathsf{inl}\, r,\ x.\, s,\ y.\, t) & \longrightarrow_\beta s[x := r] \\
\mathsf{case}\,(\mathsf{inr}\, r,\ x.\, s,\ y.\, t) & \longrightarrow_\beta t[y := r] \\[1ex]
\mathsf{fst}\langle t_1, t_2\rangle & \longrightarrow_\beta t_1 \\
\mathsf{snd}\langle t_1, t_2\rangle & \longrightarrow_\beta t_2 \\[1ex]
\mathsf{open}\,(\mathsf{pack}\, t,\ x.\, s) & \longrightarrow_\beta s[x := t]
\end{array}
$$

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.

Example 1 (Booleans) We can encode the datatype of booleans $\mathsf{Bool} : *$ in System F^ω as $\mathsf{Bool} := 1 + 1$, with data constructors $\mathsf{true} := \mathsf{inl}\langle\rangle$ and $\mathsf{false} := \mathsf{inr}\langle\rangle$. Elimination of booleans is done by if-then-else, which is encoded as $\mathsf{if} := \lambda b \lambda t \lambda e.\, \mathsf{case}\,(b,\ \_.\, t,\ \_.\, e)$. The reader is invited to check the typings $\mathsf{true} : \mathsf{Bool}$, $\mathsf{false} : \mathsf{Bool}$ and $\mathsf{if} : \mathsf{Bool} \to \forall A.\ A \to A \to A$ as well as the operational behavior $\mathsf{if}\ \mathsf{true}\ t\ e \longrightarrow^+ t$ and $\mathsf{if}\ \mathsf{false}\ t\ e \longrightarrow^+ e$.

2.4 Syntactic Sugar

The term language of System F^ω is concise and easy to reason about, but for programming, what we intend to do to a certain extent in this article, a little too spartan. To make programs more readable, we introduce let binding and pattern matching as a meta notation in this section. These new constructs should not be regarded as extensions to System F^ω; we formally describe a transformation relation $\rightsquigarrow$ which eliminates all syntactic sugar.

Non-recursive let bindings. As implemented in some functional programming languages, e. g., Scheme or Ocaml, $\mathsf{let}\ x = r\ \mathsf{in}\ s$ shall denote the β-redex $(\lambda x.s)\, r$. This must not be confused with a recursive let; in our case, the variable x bound by let cannot be used in r. Formally, let-bindings can be removed

[1] This means, we may apply one of the $\longrightarrow_\beta$-rules to an arbitrary subterm of t in order to obtain one step of reduction. This especially includes the ξ-rule which says that $t \longrightarrow t'$ implies $\lambda x.\, t \longrightarrow \lambda x.\, t'$. Clearly, this rule is not implemented in the usual functional programming languages. Since we prove *strong* normalization, we are on the safe side.

from programs by performing the following transformation steps on any part of the program until no let-bindings remain.

$$\mathsf{let}\ x = r\ \mathsf{in}\ s \rightsquigarrow (\lambda x.s)\, r$$

Pattern matching. Patterns are terms constructed from variables and introductions, except function type introduction (λ). Formally, they are given by the grammar

$$p ::= x \mid \langle\rangle \mid \langle p,\, p'\rangle \mid \mathsf{inl}\, p \mid \mathsf{inr}\, p \mid \mathsf{pack}\ p.$$

We use shorthand notations for groups of similar patterns. For instance, $\mathsf{inl}\,\vec{p}$ is an abbreviation for the list of patterns $\mathsf{inl}\,p_1, \ldots, \mathsf{inl}\,p_n$ where $n = |\vec{p}|$.

Pattern matching is introduced by the notation

$$\mathsf{match}\ r\ \mathsf{with}\ p_1 \mapsto s_1 \mid \ldots \mid p_n \mapsto s_n.$$

The order of the clauses $p_i \mapsto s_i$ is irrelevant. For succinctness, we write $\mathsf{match}\ r\ \mathsf{with}\ (p_i \mapsto s_i)_{i=1..n}$ or even $\mathsf{match}\ r\ \mathsf{with}\ \vec{p} \mapsto \vec{s}$, for short. The notation $\mathsf{match}\ r\ \mathsf{with}\ \vec{p} \mapsto \vec{s} \mid \vec{q} \mapsto \vec{t}$ should also be easily understandable.

Pattern matching is expanded by the following new rules for the transformation relation $\rightsquigarrow$. Patterns should only be used if they are well-typed, non-overlapping, linear (no variable occurs twice) and exhaustive. We do not present a theory of patterns, but just have them as a meta-syntactic device. Therefore, we restrict the use of pattern matching to the situations where these transformation rules succeed in removing the syntactic sugar.

$$\begin{array}{lcl}
\mathsf{match}\ r\ \mathsf{with}\ x \mapsto s & \rightsquigarrow & \mathsf{let}\ x = r\ \mathsf{in}\ s \\
\mathsf{match}\ r\ \mathsf{with}\ \langle\rangle \mapsto s & \rightsquigarrow & s \\
\mathsf{match}\ r\ \mathsf{with}\ \Big(\langle p_i, p'_j\rangle \mapsto s_{ij}\Big)_{i\in I, j\in J} & \rightsquigarrow & \mathsf{let}\ x = r\ \mathsf{in} \\
 & & \mathsf{match}\ (\mathsf{fst}\, x)\ \mathsf{with} \\
 & & \quad \Big(p_i \mapsto \mathsf{match}\ (\mathsf{snd}\, x)\ \mathsf{with} \\
 & & \qquad\qquad \Big(p'_j \mapsto s_{ij}\Big)_{j\in J}\Big)_{i\in I} \\
\mathsf{match}\ r\ \mathsf{with}\ \mathsf{inl}\,\vec{p} \mapsto \vec{s} & & \\
\qquad\qquad \mid \mathsf{inr}\,\vec{q} \mapsto \vec{t} & \rightsquigarrow & \mathsf{case}\,(r,\ x.\,\mathsf{match}\ x\ \mathsf{with}\ \vec{p} \mapsto \vec{s}, \\
 & & \qquad\quad y.\,\mathsf{match}\ y\ \mathsf{with}\ \vec{q} \mapsto \vec{t}) \\
\mathsf{match}\ r\ \mathsf{with}\ \mathsf{pack}\,\vec{p} \mapsto \vec{s} & \rightsquigarrow & \mathsf{open}\,(r,\ x.\,\mathsf{match}\ x\ \mathsf{with}\ \vec{p} \mapsto \vec{s})
\end{array}$$

In the case of matching against pairs, I and J denote finite index sets. Note that a let expression has been inserted on the right hand side to avoid duplication of term r.

Patterns in lets and abstractions. For a single-pattern matching, which has the form $\mathsf{match}\ r\ \mathsf{with}\ p \mapsto s$ we introduce a more concise notation $\mathsf{let}\ p = r\ \mathsf{in}\ s$, which is common in functional programming. Furthermore, an abstraction of a variable x plus a matching over this variable, $\lambda x.\,\mathsf{let}\ p = x\ \mathsf{in}\ s$, can from now be shortened to $\lambda p.s$. In both cases, in order to avoid clashes with the existing syntax we need to exclude patterns p which consist just of a single variable. Formally, we add two transformation rules:

$$
\begin{array}{lcll}
\mathsf{let}\ p = r\ \mathsf{in}\ s & \rightsquigarrow & \mathsf{match}\ r\ \mathsf{with}\ p \mapsto s & \text{if } p \text{ is not a variable} \\
\lambda p.\, s & \rightsquigarrow & \lambda x.\,\mathsf{match}\ x\ \mathsf{with}\ p \mapsto s & \text{if } p \text{ is not a variable}
\end{array}
$$

Sugar for term abbreviations. In the course of this article, we will often define term abbreviations c of the form $c := \lambda x_1 \ldots \lambda x_n.\, s$ with $n \geq 0$. For such c, we allow the expression $c^{\circ}\vec{t}$ to mean $s[\vec{x} := \vec{t}]$ where $|\vec{t}| = n$. In the special case that s is a pattern p, the sugared expression $c^{\circ}\vec{x}$ is just p, and we can use it in a matching construct to increase readability of code.

In the next section, we will introduce a new term constant in^{κ} and data constructors of the shape $c := \lambda\vec{x}.\,\mathsf{in}^{\kappa} p$. The notation $c^{-}\vec{t}$ shall denote $p[\vec{x} := \vec{t}]$, i.e., instantiation after removal of "in^{κ}". Summarizing, we have two additional transformations:

$$
\begin{array}{ll}
c^{\circ}\vec{t} \rightsquigarrow s[\vec{x} := \vec{t}] & \text{if } c := \lambda\vec{x}.\, s \\
c^{-}\vec{t} \rightsquigarrow p[\vec{x} := \vec{t}] & \text{if } c := \lambda\vec{x}.\,\mathsf{in}^{\kappa} p
\end{array}
$$

Example 2 ("Maybe" Type Transformer) To see the meta syntax in action, consider the option datatype with two data constructors:

$$
\begin{array}{llll}
\mathsf{Maybe} & := & \lambda A.\, 1 + A & : \ \mathsf{k1} \\
\mathsf{nothing} & := & \mathsf{inl}\langle\rangle & : \ \forall A.\ \mathsf{Maybe}\, A \\
\mathsf{just} & := & \lambda a.\ \mathsf{inr}\, a & : \ \forall A.\ A \to \mathsf{Maybe}\, A
\end{array}
$$

The type transformer Maybe is a monad with unit just and the following multiplication operation:

$$
\begin{array}{lll}
\mathsf{bind} & : & \forall A \forall B.\ \mathsf{Maybe}\, A \to (A \to \mathsf{Maybe}\, B) \to \mathsf{Maybe}\, B \\
\mathsf{bind} & := & \lambda m \lambda k.\ \mathsf{match}\ m\ \mathsf{with} \\
& & \qquad\quad \mathsf{nothing}^{\circ} \mapsto \mathsf{nothing} \\
& & \qquad\ \mid \mathsf{just}^{\circ}\, a \ \ \mapsto k\, a
\end{array}
$$

We could have dropped the annotation "$^{\circ}$" in matching against "$\mathsf{nothing}^{\circ}$"—since "$\mathsf{nothing}$" is a data constructor without arguments and the annotation "$^{\circ}$" is changing nothing in this case—, but we included it for reasons of symmetry.

3 System MIt^ω of Basic Mendler Iteration and Coiteration

In this section, we will introduce System MIt^ω, 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 data 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 constructor List is invariant in the type of the data constructors. In our case the argument is always the fixed type A. We will later see examples of heterogeneous types where the argument A to the type constructor, call it T, varies in the different occurrences of T in the type of a data 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 that is motivated from category theory and goes back to Hagino (1987). $\mathsf{List}(A)$ 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:

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

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 map.

Example 3 (Map Function for Lists) It is part of the Haskell standard library and could be defined like this:

```
map :: (a -> b) -> [a] -> [b]
map f = map'
    where map' []       = []
          map' (a : as) = (f a) : (map' as)
```

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

Mendler (1991) first observed that by a certain polymorphic typing of the term 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 `map'` 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 `map'`, and
(2) the function `map'` cannot be called recursively unless applied to "`as`".

Mendler's trick is implemented by requiring s to be of type $(X \to B) \to \mathsf{ListF}(A)X \to B$ for a fresh type variable X. The first parameter of s is the name map' 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 `map'`, 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 `map'`—, 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 map' 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 $\mathsf{MIt}(s)$ for a step term s. It has the following reduction behavior:

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

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 data 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 data 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. $\mu : (* \to *) \to *$

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

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

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

Example 4 (Map Function for Lists) Using the syntax of F^{ω} with the meta-notation for pattern matching described in Section 2, we can encode the Haskell function `map` with Mendler iteration as follows:

$$\begin{aligned}
\mathsf{map} &: \forall A \forall B.\, (A \to B) \to \mathsf{List}(A) \to \mathsf{List}(B) \\
\mathsf{map} &:= \lambda f.\ \mathsf{MIt}\Big(\lambda map' \lambda t.\, \mathsf{match}\ t\ \mathsf{with} \\
&\qquad\qquad \mathsf{nil}^{-} \qquad\quad \mapsto \mathsf{nil} \\
&\qquad\qquad \mid \mathsf{cons}^{-} a\ as \mapsto \mathsf{cons}\ (f\ a)\ (map'\ as)\Big)
\end{aligned}$$

In the following we give an assignment of bound variables to types from which one can infer that map is well-typed.

$$\begin{array}{lcl}
f & : & A \to B \\
map' & : & X \to \mathsf{List}(B) \\
t & : & \mathsf{ListF}(A)\,X \\
a & : & A \\
as & : & X
\end{array}$$

Here, $X : *$ is a fresh type variable introduced by the Mendler iterator. Also note that, according to the conventions introduced in Section 2, $\mathsf{nil}^{-} = \mathsf{inl}\langle\rangle$ and $\mathsf{cons}^{-} a\, as = \mathsf{inr}\langle a, as\rangle$. The $^{-}$-notation discards the general constructor in for inductive types, which is necessary since the Mendler iterator takes an argument t of the unfolded inductive type.

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.

$$
\begin{array}{ll}
\text{Formation.} & \nu \quad : (* \to *) \to * \\[1ex]
\text{Elimination.} & \mathsf{out} : \forall F^{* \to *}.\ \nu F \ \to\ F\,(\nu F) \\[1ex]
\text{Introduction.} & \dfrac{\begin{array}{l}\Gamma \vdash F : * \to * \\ \Gamma \vdash A : * \\ \Gamma \vdash s : \forall X^{*}.\ (A \to X) \to A \to F\,X\end{array}}{\Gamma \vdash \mathsf{MCoit}(s) : A \to \nu F} \\[1ex]
\text{Reduction.} & \mathsf{out}\ (\mathsf{MCoit}(s)\ t) \longrightarrow_{\beta} s\ \mathsf{MCoit}(s)\ t
\end{array}
$$

Dually to the general constructor in for inductive types, coinductive types possess a general destructor out which triggers unrolling of the coiterator 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 5 (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:

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

Example 6 (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:

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

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:

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

```
data 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, 2000a). 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* or *nested* type in contrast to *homogeneous* or *non-nested* 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 just 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 notice that all of the datatypes `PList`, `Bush`, `Lam` are first-order as type constructors: they are of kind $* \to *$. It is possible, of course, also to combine nestedness and higher-orderness, but this combination does not happen in these three examples. Moreover, we do not find this combination very important conceptually, as the challenges are not in the kinds of the parameters of a datatype, but in the kind of the μ-operator employed.

We will encounter all these three datatypes in examples later. For now we are interested in encoding these types in a suitable extension of F^ω. The encoding is possible if a combinator $\mu^{\mathsf{k1}} : (\mathsf{k1} \to \mathsf{k1}) \to \mathsf{k1}$ for least fixed-point types of rank 1 is present. (Recall that $\mathsf{k1} = * \to *$.) In the following we give representations of these three types as least fixed points μF of type transformer transformers $F : \mathsf{k2}$.

$$
\begin{array}{lll}
\mathsf{PListF} & := \lambda X \lambda A.\ A + X\,(A \times A) & : \mathsf{k2} \\
\mathsf{PList} & := \mu^{\mathsf{k1}}\,\mathsf{PListF} & : \mathsf{k1} \\
\mathsf{BushF} & := \lambda X \lambda A.\ 1 + A \times X\,(X\,A) & : \mathsf{k2} \\
\mathsf{Bush} & := \mu^{\mathsf{k1}}\,\mathsf{BushF} & : \mathsf{k1} \\
\mathsf{LamF} & := \lambda X \lambda A.\ A + (X\,A \times X\,A + X\,(1 + A)) & : \mathsf{k2} \\
\mathsf{Lam} & := \mu^{\mathsf{k1}}\,\mathsf{LamF} & : \mathsf{k1}
\end{array}
$$

Similarly to the rank-1 case we just have one general datatype constructor $\mathsf{in}^{\mathsf{k1}}$ which rolls an inhabitant of $F\,(\mu^{\mathsf{k1}} F)$ into the fixed point $\mu^{\mathsf{k1}} F$. Note, however, that $\mu^{\mathsf{k1}} F$ is not a type but a type constructor, hence, we need a polymorphic data constructor $\mathsf{in}^{\mathsf{k1}} : \forall A.\, F\,(\mu^{\mathsf{k1}} F)\,A \to \mu^{\mathsf{k1}} F A$. Now, we are ready to define the usual data constructors for the heterogeneous datatypes we are encoding:

$$
\begin{array}{lll}
\mathsf{zero} & := \lambda a.\,\mathsf{in}^{\mathsf{k1}}\,(\mathsf{inl}\ a) & : \forall A.\, A \to \mathsf{PList}\ A \\
\mathsf{succ} & := \lambda l.\,\mathsf{in}^{\mathsf{k1}}\,(\mathsf{inr}\ l) & : \forall A.\ \mathsf{PList}(A \times A) \to \mathsf{PList}\,A \\
\mathsf{bnil} & := \mathsf{in}^{\mathsf{k1}}\,(\mathsf{inl}\,\langle\rangle) & : \forall A.\ \mathsf{Bush}\ A \\
\mathsf{bcons} & := \lambda a \lambda b.\,\mathsf{in}^{\mathsf{k1}}\,(\mathsf{inr}\,\langle a, b\rangle) & : \forall A.\, A \to \mathsf{Bush}\,(\mathsf{Bush}\ A) \to \mathsf{Bush}\ A \\
\mathsf{var} & := \lambda a.\,\mathsf{in}^{\mathsf{k1}}\,(\mathsf{inl}\ a) & : \forall A.\, A \to \mathsf{Lam}\ A \\
\mathsf{app} & := \lambda t_1 \lambda t_2.\,\mathsf{in}^{\mathsf{k1}}\,(\mathsf{inr}\,(\mathsf{inl}\,\langle t_1, t_2\rangle)) & : \forall A.\ \mathsf{Lam}\ A \to \mathsf{Lam}\ A \to \mathsf{Lam}\ A \\
\mathsf{abs} & := \lambda r.\,\mathsf{in}^{\mathsf{k1}}\,(\mathsf{inr}\,(\mathsf{inr}\ r)) & : \forall A.\ \mathsf{Lam}\,(1 + A) \to \mathsf{Lam}\ A
\end{array}
$$

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, 2000a; Martin, Gibbons, and Bayley, 2003). 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 type 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 be defined as follows:

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

(Here, we have made use of the vector notation $\forall \vec{X}^{\vec{\kappa}}$ as an abbreviation for $\forall X_1^{\kappa_1} \ldots \forall X_n^{\kappa_n}$ where $n = |\vec{X}|$.)

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 $\mathsf{in}^{\mathsf{k1}}$ from the last subsection is a natural transformation of type $F\,(\mu^{\mathsf{k1}} F) \subseteq^{\mathsf{k1}} \mu^{\mathsf{k1}} F$. 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 family of constants, typing and reduction rules, indexed by κ.

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

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

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

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

Notice that for *every* type constructor F of kind $\kappa \to \kappa$, $\mu^\kappa F$ is a type 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), positivity of F is always required, which is a very natural concept in the case $\kappa = *$. (A first-order type constructor $F : * \to *$ is said to be positive iff every occurence of X in $F X$ is positive in the sense of being enclosed in an even number of left-hand sides of $\to$.) For higher kinds, however, there is no such 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 higher-order 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 MIt^κ we have obtained a sensible and usable device for programming.

Example 7 (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 $\mathsf{MIt}^{\mathsf{k1}}$ we can implement a function get such that $\mathsf{get}\,t\,p$ retrieves element a.

$$\begin{array}{lcl}
\mathsf{get} & := & \mathsf{MIt}^{\mathsf{k1}}\Big(\lambda get\,\lambda t\,\lambda p.\ \mathsf{match}\ t\ \mathsf{with} \\
& & \qquad\qquad\qquad \mathsf{zero}^-\ a \mapsto a \\
& & \qquad\qquad\qquad \mid \mathsf{succ}^-\ l \mapsto \mathsf{let}\ \langle a_1, a_2\rangle = get\,l\,(\mathsf{tail}\,p)\ \mathsf{in} \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{if}\ (\mathsf{head}\,p)\ a_1\ a_2\Big) \\
& : & \forall A.\ \mathsf{PList}\,A \to \mathsf{Stream\,Bool} \to A \\
& = & \mathsf{PList} \subseteq^{\mathsf{k1}} (\lambda A.\,\mathsf{Stream\,Bool} \to A)
\end{array}$$

Here we reused the type of streams defined in Example 5 and the booleans defined in Section 2. To verify welltypedness, observe that the bound variables have the following types:

$$\begin{array}{lcl}
X & : & \mathsf{k1} \quad \text{(not visible due to Curry-style)} \\
get & : & \forall A.\,X\,A \to \mathsf{Stream\,Bool} \to A \\
A & : & * \quad \text{(not visible due to Curry-style)} \\
t & : & \mathsf{PListF}\,X\,A = A + X\,(A \times A) \\
p & : & \mathsf{Stream\,Bool} \\
a, a_1, a_2 & : & A \\
l & : & X\,(A \times A)
\end{array}$$

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 $\mathsf{MIt}^{\mathsf{k1}}$ facilitates a kind of polymorphic recursion.

Example 8 (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.

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

$$\begin{array}{l}
\mathsf{sum}'\,(\mathsf{zero}\ a)\,f \ \longrightarrow^+\ f\,a \\
\mathsf{sum}'\,(\mathsf{succ}\ l)\,f \ \longrightarrow^+\ \mathsf{sum}'\ l\ (\lambda\langle a_1, a_2\rangle.\ f\,a_1 + f\,a_2)
\end{array}$$

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 $\mathsf{MIt}^{\mathsf{k1}}$ is a mechanical process which results in the following definition:

$$
\begin{array}{lll}
\mathsf{sum}' & := & \mathsf{MIt}^{\mathsf{k1}}\Big(\lambda sum' \lambda t \lambda f.\, \mathsf{match}\ t\ \mathsf{with} \\
& & \quad \mathsf{zero}^{-}\ a \mapsto f\,a \\
& & \quad \mid \mathsf{succ}^{-}\ l\ \mapsto sum'\,l\ (\lambda\langle a_1, a_2\rangle.\ f\,a_1 + f\,a_2)\Big) \\
& : & \mathsf{PList} \subseteq^{\mathsf{k1}} (\lambda A.\,(A \to \mathsf{Nat}) \to \mathsf{Nat})
\end{array}
$$

The postulated reduction behavior is verified by a simple calculation. From sum', the summation function is obtained by $\mathsf{sum} := \lambda t.\ \mathsf{sum}'\ t\ \mathsf{id}$.

Let us remark here that the result type 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 type constructors $G = H = \lambda B.\,\mathsf{Nat}$, the result type 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 MIt^{κ} 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 MIt^{ω}, which is an extension of Mendler's system (1991) to finite kinds.

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.

$$
\frac{\Gamma \vdash F : \kappa \to \kappa, \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : \forall X^{\kappa}.\ G \subseteq^{\kappa} X \to G \subseteq^{\kappa} F\,X}{\Gamma \vdash \mathsf{MCoit}^{\kappa}(s) : G \subseteq^{\kappa} \nu^{\kappa} F}
$$

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

The reader is invited to check that no problems for subject reduction and confluence arise from these definitions. To demonstrate the usefulness of $\mathsf{MCoit}^{\mathsf{k1}}$ 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:

$$\begin{array}{lll} \mathsf{TriF} & := \lambda X \lambda A.\ A \times X(E \times A) & : \mathsf{k2} \\ \mathsf{Tri} & := \nu^{\mathsf{k1}}\, \mathsf{TriF} & : \mathsf{k1} \end{array}$$

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 9 (Triangle Decomposition) Using the destructor for coinductive types on a triangle $\mathsf{Tri}\ A$, we can obtain the top element of type A and the remainder of type $\mathsf{Tri}\,(E \times A)$ which looks like an infinite trapezium in our visualization.

$$\begin{array}{llll} \mathsf{top} & := \lambda t.\ \mathsf{fst}\,(\mathsf{out}^{\mathsf{k1}}\, t) & : & \forall A.\, \mathsf{Tri}\, A \to A \\ \mathsf{rest} & := \lambda t.\ \mathsf{snd}\,(\mathsf{out}^{\mathsf{k1}}\, t) & : & \forall A.\, \mathsf{Tri}\, A \to \mathsf{Tri}\,(E \times A) \end{array}$$

Cutting off the 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:

$$\begin{array}{ll} \mathsf{cut} := & \mathsf{MCoit}^{\mathsf{k1}} \Big(\lambda cut \lambda t.\ \langle \mathsf{snd}\,(\mathsf{top}\ t),\ \ cut\,(\mathsf{rest}\ t)\rangle\Big) \\ \quad : & (\lambda A.\ \mathsf{Tri}\,(E \times A)) \subseteq^{\mathsf{k1}} \mathsf{Tri} \end{array}$$

Remark 1 (Corrigendum) In Abel, Matthes, and Uustalu (2003) we used $\mathsf{tri}\ \mathsf{snd}$ instead of cut, where tri is the mapping function for Tri and snd the second projection. This does type-check yet not yield the right operational behavior, since it cuts off the side diagonal rather than the top row.

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'. (By an A-decorated tree we mean a tree with

A-labelled branching nodes.) 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

$$\mathsf{redec} : \forall A \forall B. (\mathsf{Stream}\, A \to B) \to \mathsf{Stream}\, A \to \mathsf{Stream}\, B$$

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.

Example 10 (Stream Redecoration) Markus Schnell posted an implementation of stream redecoration to the Haskell Mailing List (2002).

```
slide :: ([a] -> b) -> [a] -> [b]
slide f [] = []
slide f xs = f xs : slide f (tail xs)
```

He showed how to encode a lowpass digital filter using `slide`. Here is the implementation of a smoothening filter which replaces each stream element by the average of n adjacent elements.

```
smooth :: Int -> [Float] -> [Float]
smooth n = slide (\ xs -> sum (take n xs) / fromInt n)
```

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 comultiplication (Uustalu and Vene, 2002).

Example 11 (Triangle Redecoration) For triangles, redecoration works as follows: In the triangle

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

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.

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

$$\begin{aligned}
\mathsf{redec}' &:= \ \mathsf{MCoit}^{\mathsf{k1}} \Big(\lambda \mathit{redec}' \lambda(\mathsf{pack}\langle f, t\rangle). \\
&\qquad\qquad \langle f\,t,\ \mathit{redec}'\,(\mathsf{pack}\,\langle \mathsf{lift}\,f,\ \mathsf{rest}\ t\rangle)\rangle\Big) \\
&: \quad (\lambda B \exists A.\,(\mathsf{Tri}\,A \to B) \times \mathsf{Tri}\,A) \subseteq^{\mathsf{k1}} \mathsf{Tri}
\end{aligned}$$

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 $\mathsf{rest}\ t$.

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

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 $\mathsf{Tri}(E \times B)$, which means a pair $\langle e, b\rangle$ of type $E \times B$. Since the elements outside the diagonal do not have to be transformed, the left component e stays fixed. The right component b comes from applying f to the triangle which results from cutting off the top row from t.

For the typing of redec' let $G' := \lambda B \exists A.\,(\mathsf{Tri}\,A \to B) \times \mathsf{Tri}\,A$. If the variable redec' receives the type $G' \subseteq^{\mathsf{k1}} X$ and $\mathsf{pack}\,\langle f, t\rangle$ is matched, then f gets type $\mathsf{Tri}\,A \to B$ and t gets type $\mathsf{Tri}\,A$. Hence $f\,t : B$, 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 instantiated by A and B themselves. It is clear that one gets the following reduction behavior.

$$\begin{aligned}
&\mathsf{out}^{\mathsf{k1}}(\mathsf{redec}'\,(\mathsf{pack}\,\langle f, t\rangle)) &&\longrightarrow^+ \langle f\,t,\ \mathsf{redec}'\,(\mathsf{pack}\,\langle \mathsf{lift}\,f, \mathsf{rest}\ t\rangle)\rangle \\
&\mathsf{top}\,(\mathsf{redec}\,f\,t) &&\longrightarrow^+ f\,t \\
&\mathsf{rest}\,(\mathsf{redec}\,f\,t) &&\longrightarrow^+ \mathsf{redec}^\circ\,(\mathsf{lift}\,f)\ (\mathsf{rest}\ t)
\end{aligned}$$

On the last line we have used the $^\circ$-notation because simply $\mathsf{redec}\,(\mathsf{lift}\,f)\ (\mathsf{rest}\ t)$ is no β-reduct of the left hand side. Without the $^\circ$-notation we could only state that left and right hand side are β-equal, i. e., have a common reduct.

The source type 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'.

In this subsection, we will show strong normalization for System MIt^ω by embedding it into F^ω. Since this shows that MIt^ω 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 MIt^ω in which these concepts are isolated and named.

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

(1) Read off the encoding of (co)inductive type 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 data constructor resp. destructor.

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

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

This equivalent type for $\lambda s.\, \mathsf{MIt}^\kappa(s)$ 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 $\mathsf{MIt}^\kappa(s)$ and in^κ then simply fall into place:

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

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

Proof. By simple calculation. □

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

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

These considerations lead to the following definitions:

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

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

Proof. By simple calculation. □

Theorem 1 (Strong normalization) *System* MIt^ω *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. □

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, Matthes, and Uustalu (2003). Its intension is to ease programming with Mendler iteration in case Kan extensions have to be used (cf. examples 8 and 11 in Section 3). In a sense, we are just hard-wiring the Kan extensions into the (co)iteration scheme of MIt^ω.

The key idea consists in identifying an appropriate containment relation for type 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 MIt^ω.

Refined containment. A more refined notion is $\leq^\kappa$, which first appeared in Hinze's work (1999) as the polykinded type *Map* of generic mapping functions. We learned it first from Peter Hancock in 2000 and employed it already in earlier work (Abel and Matthes, 2003) which studied iteration for monotone inductive type 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 $F, G : \mathsf{k1}$, $F \leq^{\mathsf{k1}} G = \forall A \forall B. (A \to B) \to F A \to G B$. 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 such that generalized and efficient folds (Hinze, 2000a; Martin, Gibbons, and Bayley, 2003) are directly covered, we have to relativize the notion $\leq^\kappa$, for $\kappa = \vec{\kappa} \to *$, to a vector $\vec{H}$ of type 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 type 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 type constructor $\leq^\kappa_{(-)}$: $(\vec{\kappa} \to \vec{\kappa}) \to \kappa \to \kappa \to *$ by structural recursion on κ as follows:

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

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

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

As an example, for $F, G, H : \mathsf{k1}$, one has

$$
\begin{array}{lcl}
F \ \leq^{\mathsf{k1}}_{H} \ G & = & \forall A \forall B.\, (A \to HB) \to FA \to GB, \\
F \ {}_{H}\!\leq^{\mathsf{k1}} \ G & = & \forall A \forall B.\, (HA \to B) \to FA \to GB.
\end{array}
$$

Even more concretely, we have the following types which will be used in the examples 12 and 14:

$$
\begin{array}{lcl}
\mathsf{PList} \leq^{\mathsf{k1}}_{\lambda B.\mathsf{Nat}} \lambda B.\mathsf{Nat} & = & \forall A \forall B.(A \to \mathsf{Nat}) \to \mathsf{PList}\,A \to \mathsf{Nat} \\
\mathsf{Tri} \ {}_{\mathsf{Tri}}\!\leq^{\mathsf{k1}} \ \mathsf{Tri} & = & \forall A \forall B.(\mathsf{Tri}\,A \to B) \to \mathsf{Tri}\,A \to \mathsf{Tri}\,B
\end{array}
$$

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 rank-1 (co)inductive *types*, and to a scheme encompassing generalized folds (Bird and Paterson, 1999a; Hinze, 2000a; Martin, Gibbons, and Bayley, 2003) and the dual scheme for coinductive type constructors of rank 2. The generalized scheme for coinductive type 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 type constructor constants μ^{κ} and ν^{κ}, and welltyped term constants $\mathsf{in}^{\kappa}, \mathsf{out}^{\kappa}$ as for MIt^{ω}, 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 type constructors. Let $\kappa = \vec{\kappa} \to *$.

$$
\begin{array}{ll}
\text{Formation.} & \mu^\kappa \; : (\kappa \to \kappa) \to \kappa \\[1ex]
\text{Introduction.} & \mathsf{in}^\kappa : \forall F^{\kappa \to \kappa}.\, F\,(\mu^\kappa F) \;\subseteq^\kappa\; \mu^\kappa F \\[1ex]
\text{Elimination.} & \dfrac{\begin{array}{l}\Gamma \vdash F : \kappa \to \kappa \\ \Gamma \vdash G : \kappa \\ \Gamma \vdash \vec{H} : \vec{\kappa} \to \vec{\kappa} \\ \Gamma \vdash s : \forall X^\kappa.\ X \leq^\kappa_{\vec{H}} G \to F\,X \leq^\kappa_{\vec{H}} G\end{array}}{\Gamma \vdash \mathsf{GMIt}^\kappa(s) : \mu^\kappa F \leq^\kappa_{\vec{H}} G} \\[1ex]
\text{Reduction.} & \mathsf{GMIt}^\kappa(s)\,\vec{f}\,(\mathsf{in}^\kappa\,t) \longrightarrow_\beta s\;\mathsf{GMIt}^\kappa(s)\,\vec{f}\,t \\[1ex]
 & \text{where } |\vec{f}| = |\vec{\kappa}|.
\end{array}
$$

Example 12 (Summing up a Powerlist, Revisited) The function sum' for powerlists (see Example 8) can be naturally implemented with $\mathsf{GMIt}^{\mathsf{k1}}$. The difference to the original implementation confines itself to swapping the arguments t (powerlist) and f (continuation). The swapping is necessary since for this example $\mathsf{GMIt}^{\mathsf{k1}}$ yields a recursive function of type $\mathsf{PList} \leq^{\mathsf{k1}}_{\lambda B.\mathsf{Nat}} \lambda B.\mathsf{Nat}$, which can be simplified to $\forall A.\,(A \to \mathsf{Nat}) \to \mathsf{PList}\,A \to \mathsf{Nat}$ by removing the void quantification over B.

Example 13 (Summing up a Bush) Recall the nested datatype of "bushy lists" given in Section 3.3 and first considered in Bird and Meertens (1998) within Haskell.

$$
\begin{array}{lll}
\mathsf{BushF} & = \lambda X \lambda A.\, 1 + A \times X\,(X\;A) & : \mathsf{k2} \\
\mathsf{Bush} & = \mu^{\mathsf{k1}}\,\mathsf{BushF} & : \mathsf{k1} \\[1ex]
\mathsf{bnil} & = \mathsf{in}^{\mathsf{k1}}\,(\mathsf{inl}\,\langle\rangle) & : \forall A.\ \mathsf{Bush}\,A \\
\mathsf{bcons} & = \lambda a \lambda b.\ \mathsf{in}^{\mathsf{k1}}\,(\mathsf{inr}\,\langle a,\, b\rangle) & : \forall A.\, A \to \mathsf{Bush}\,(\mathsf{Bush}\;A) \to \mathsf{Bush}\;A
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathsf{bsum}' & := & \mathsf{GMIt}^{\mathsf{k1}}\Big(\lambda bsum' \lambda f \lambda t.\, \mathsf{match}\ t\ \mathsf{with} \\
 & & \qquad \mathsf{bnil}^- \qquad\ \mapsto 0 \\
 & & \qquad \mid \mathsf{bcons}^-\,a\,b \mapsto f\;a + bsum'\;(bsum'\;f)\;b\Big) \\
 & : & \forall A.\,(A \to \mathsf{Nat}) \to \mathsf{Bush}\,A \to \mathsf{Nat}
\end{array}
$$

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

$$\begin{array}{ll}
bsum' & : X \leq^{\mathsf{k1}}_{\lambda B.\mathsf{Nat}} \lambda B.\mathsf{Nat} \\
f & : A \to \mathsf{Nat} \\
t & : 1 + A \times X(XA) \\
a & : A \\
b & : X(XA) \\
bsum' f & : XA \to \mathsf{Nat} \\
bsum'\,(bsum' f) & : X(XA) \to \mathsf{Nat}
\end{array}$$

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:

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

Note that in the outer recursive call the second argument decreases structurally: on the left-hand side, one has $\mathsf{bcons}\,a\,b$, one the right-hand side only b. But, the inner call $\mathsf{bsum}' f$ does not specify a second argument. One could force the second argument by η-expanding it to $\lambda x.\,\mathsf{bsum}'\,f\,x$, which does not help much because the fresh variable x does not stand in any visible relation to the input $\mathsf{bcons}\,a\,b$. Consequently, proving termination of bsum' using a term ordering seems to be problematic, whereas in our system it is just a byproduct of type checking (see Theorem 1).

Coinductive type 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.

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

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 14 (Triangle Redecoration, Revisited) Using $\mathsf{GMCoit}^{\mathsf{k1}}$, triangle redecoration can be implemented much more concisely. In the context of

Example 11 we obtain the function redec directly as follows.

$$\begin{aligned}\mathsf{redec} &:= \mathsf{GMCoit}^{\mathsf{k1}}\Big(\lambda redec\,\lambda f\lambda t.\,\langle f\,t, redec\,(\mathsf{lift}\,f)\,(\mathsf{rest}\,t)\rangle\Big)\\ &:\quad \forall A\forall B.\,(\mathsf{Tri}\,A \to B) \to \mathsf{Tri}\,A \to \mathsf{Tri}\,B \;=\; \mathsf{Tri}\;{}_{\mathsf{Tri}}{\leq^{\mathsf{k1}}}\;\mathsf{Tri}\end{aligned}$$

Thus, one can enforce the desired reduction behavior without any detours. In MIt^ω, where we implemented triangle redecoration in Example 11, we were required to implement an auxiliary function redec' first which used a tagged argument pair $\mathsf{pack}\langle f, t\rangle$. In contrast, the curried version redec above can handle f and t as two separate arguments directly. This leads to a very natural reduction behavior:

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

In Example 11, we had the small spot $^\circ$ in the picture in that only redec° for the function redec we defined there appeared on the right-hand side.

4.3 Embedding GMIt^ω *into* MIt^ω

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

For the sake of the embedding, we use a syntactic version of Kan extensions (see Mac Lane, 1998, Chapter 10). In conjunction with nested datatypes, Kan extensions appear already in Bird and Paterson (1999a, Section 6.2), but not as a tool of programming with nested types, but a means to categorically justify the uniqueness of generalized folds as elimination principles for nested datatypes. In this article, for the examples 8 and 11 within MIt^ω, special Kan extensions have been used already. The same programming tasks have been accomplished directly within GMIt^ω in the examples 12 and 14. In the sequel, this will be clarified: Just by choosing the target type constructor of iteration to be an appropriate Kan extension, one gets the behavior of $\mathsf{GMIt}^\kappa(s)$ within MIt^ω, and similarly for the source type constructor in the coinductive case.

Compared to Abel and Matthes (2003), Kan extensions “along” are now defined for all kinds, not just for rank 1.

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 $|\vec{\kappa}|$:

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

Here, $\kappa_1 \to \tilde{\kappa}$ is the general format for a composed kind κ. Clearly, $\tilde{\kappa} = \kappa_2, \ldots, \kappa_{|\vec{\kappa}|} \to *$.

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 $|\vec{\kappa}|$:

$$\begin{aligned} \mathsf{Lan}^{*}\, F &:= F \\ (\mathsf{Lan}^{\kappa_1 \to \tilde{\kappa}}_{H,\vec{H}}\, F)\, Y \vec{Y} &:= \exists X^{\kappa_1}.\, HX \leq^{\kappa_1} Y \;\times\; (\mathsf{Lan}^{\tilde{\kappa}}_{\vec{H}}\, (FX))\, \vec{Y} \end{aligned}$$

The kind $\tilde{\kappa}$ is to be understood as in the previous definition.

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 $\vec{\kappa}' := \vec{\kappa} \to \vec{\kappa}$, $n := |\vec{\kappa}|$ and define

$$\begin{aligned} \mathsf{leqRan}^{\kappa} &:= \lambda g \lambda t \lambda \vec{f}.\, g\, \vec{f}\, t \\ &: \; \forall F^{\kappa} \forall G^{\kappa} \forall \vec{H}^{\vec{\kappa}'}.\, F \leq^{\kappa}_{\vec{H}} G \to F \subseteq^{\kappa} \mathsf{Ran}^{\kappa}_{\vec{H}}\, G \\[1ex] \mathsf{ranLeq}^{\kappa} &:= \lambda h \lambda \vec{f} \lambda r.\, h\, r\, \vec{f} \\ &: \; \forall F^{\kappa} \forall G^{\kappa} \forall \vec{H}^{\vec{\kappa}'}.\, F \subseteq^{\kappa} \mathsf{Ran}^{\kappa}_{\vec{H}}\, G \to F \leq^{\kappa}_{\vec{H}} G \\[1ex] \mathsf{leqLan}^{\kappa} &:= \lambda g \lambda (\mathsf{pack}\langle f_1, \mathsf{pack}\langle f_2, \ldots \mathsf{pack}\langle f_n, t\rangle \ldots\rangle\rangle).\, g\, \vec{f}\, t \\ &: \; \forall F^{\kappa} \forall G^{\kappa} \forall \vec{H}^{\vec{\kappa}'}.\, F\ {}_{\vec{H}}\!\leq^{\kappa} G \to \mathsf{Lan}^{\kappa}_{\vec{H}}\, F \subseteq^{\kappa} G \\[1ex] \mathsf{lanLeq}^{\kappa} &:= \lambda h \lambda \vec{f} \lambda t.\, h\ \mathsf{pack}\langle f_1, \mathsf{pack}\langle f_2, \ldots \mathsf{pack}\langle f_n, t\rangle \ldots\rangle\rangle \\ &: \; \forall F^{\kappa} \forall G^{\kappa} \forall \vec{H}^{\vec{\kappa}'}.\, \mathsf{Lan}^{\kappa}_{\vec{H}}\, F \subseteq^{\kappa} G \to F\ {}_{\vec{H}}\!\leq^{\kappa} G \end{aligned}$$

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

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

$$\begin{aligned}\mathsf{GMIt}^\kappa(s) \quad &:= \mathsf{ranLeq}^\kappa\,(\mathsf{MIt}^\kappa\,(\mathsf{leqRan}^\kappa \circ s \circ \mathsf{ranLeq}^\kappa))\\ \mathsf{GMCoit}^\kappa(s) &:= \mathsf{lanLeq}^\kappa\,(\mathsf{MCoit}^\kappa\,(\mathsf{leqLan}^\kappa \circ s \circ \mathsf{lanLeq}^\kappa))\end{aligned}$$

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 $\forall X^\kappa.\ X \leq^\kappa_{\vec{H}} G \to F\,X \leq^\kappa_{\vec{H}} G$. Then,

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

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

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

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

With a similar, but notationally more tedious calculation for the coinductive case, we get, with these definitions, in MIt^ω:

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

Since one step of reduction in GMIt^ω is replaced by at least one reduction step of the encoding in MIt^ω, GMIt^ω inherits strong normalization of MIt^ω. Since the number of steps is fixed for every kind κ, in the examples we will just treat GMIt^ω as a subsystem of MIt^ω in the sense that we assume that both iteration and both coiteration schemes are present in MIt^ω together with their reduction rules.

5 Basic Conventional Iteration

We are looking for a system of conventional iteration into which we can embed MIt^ω 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. The fixed-point $\mu^\kappa F$ would be the carrier of an initial F-algebra, and $\mathsf{in}^\kappa : F\,(\mu^\kappa F) \longrightarrow \mu^\kappa F$ be its structure map. 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 data constructor in^κ will remain fixed throughout the article. 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 $F s : F A \longrightarrow F B$. If F is a type constructor, i. e., of kind $\mathsf{k1}$, then this can be represented by the existence of a term m of type

$$\mathsf{mon}^{\mathsf{k1}}\, F := \forall A \forall B.\ (A \to B) \to F A \to F B,$$

which is nothing but monotonicity of F. Then, $s : A \to B$ implies $m\, s : F A \to F B$.

The notion $\mathsf{mon}^{\mathsf{k1}}\, F$ is the most logically-minded definition of rank-1-functors: 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*, and 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 type 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 data constructor or the iterator. Moreover, it 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 $\mathsf{mon}^{\mathsf{k1}}$ enters the definitions. Clearly, $\underline{\mathsf{mon}}^{\mathsf{k1}}\, F = \mathsf{mon}^{\mathsf{k1}}\, F$.

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 MIt^ω.

Define the basic conventional iterators and coiterators by

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

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

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

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 type constructor G and a term s of type $F\,G \subseteq^{\kappa} G$, "initiality" of $\mu^{\kappa} F$ yields a "morphism" from $\mu^{\kappa} F$ to G. 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 15 (Summing up a Powerlist, Conventional Style) We redo Example 8 with $\underline{\mathsf{It}}^{\mathsf{k1}}$. Again, we define a more general function

$$\mathsf{sum}' : \forall A.\; \mathsf{PList}\, A \to (A \to \mathsf{Nat}) \to \mathsf{Nat} \;=\; \mathsf{PList} \subseteq^{\mathsf{k1}} G'$$

with $G' := \lambda A.\,(A \to \mathsf{Nat}) \to \mathsf{Nat} : \mathsf{k1}$. This is done as follows:

$$
\begin{array}{lcl}
\mathsf{PListF} & := & \lambda X \lambda A.\ A + X\,(A \times A) \\
 & : & \mathsf{k2} \\[1ex]
\mathsf{plistfb} & := & \lambda s \lambda t.\ \mathsf{match}\ t\ \mathsf{with} \\
 & & \qquad\quad \mathsf{zero}^-\ a \mapsto \mathsf{zero}^-\ a \\
 & & \qquad\ \mid \mathsf{succ}^-\ l \ \mapsto \mathsf{succ}^-\ (s\,l) \\
 & : & \underline{\mathsf{mon}}^{\mathsf{k2}}\,\mathsf{PListF} \\[1ex]
s & := & \lambda t \lambda f.\ \mathsf{match}\ t\ \mathsf{with} \\
 & & \qquad\quad \mathsf{zero}^-\ a \quad \mapsto f\ a \\
 & & \qquad\ \mid \mathsf{succ}^-\ \mathit{res} \mapsto \mathit{res}\ (\lambda\langle a_1, a_2\rangle.\ f\,a_1 + f\,a_2) \\
 & : & \mathsf{PListF}\,G' \subseteq^{\mathsf{k1}} G' \\[1ex]
\mathsf{sum}' & := & \underline{\mathsf{It}}^{\mathsf{k1}}(\mathsf{plistfb}, s) \\
 & : & \mathsf{PList} \subseteq^{\mathsf{k1}} G'
\end{array}
$$

An easy calculation shows that, as in Example 8, we get the reduction behavior

$$
\begin{array}{l}
\mathsf{sum}' \ : \ \forall A.\ \mathsf{PList}\,A \to (A \to \mathsf{Nat}) \to \mathsf{Nat} \\
\mathsf{sum}'\,(\mathsf{zero}\ a)\,f \ \longrightarrow^+ \ f\,a \\
\mathsf{sum}'\,(\mathsf{succ}\ l)\,f \ \longrightarrow^+ \ \mathsf{sum}'\ l\ (\lambda\langle a_1, a_2\rangle.\ f\,a_1 + f\,a_2)
\end{array}
$$

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.

5.2 Definition of $\underline{\mathsf{It}}^\omega$: Basic Conventional Iteration and Coiteration

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

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

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

As has been promised, this is nothing more than a system version of the definitions in the previous subsection. Since the embedding of MIt^ω into $\underline{\mathsf{It}}^\omega$ in the next subsection will even change the type 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 general data constructor in^κ, which has been changed to $\underline{\mathsf{in}}^\kappa$. Similar remarks apply to coiteration, given as follows:

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

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

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

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

5.3 Embedding Mendler Iteration into Conventional Iteration

Here, we present a somewhat surprising embedding of MIt^ω 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 MIt^ω to the conventional (co)iterators of $\underline{\mathsf{It}}^\omega$. Unlike the embedding of GMIt^ω into MIt^ω, 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 $1 \le |\vec{\kappa}|$. 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 type 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 $\kappa \neq *$

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

Lemma 4 *For any $F, F' : \kappa \to \kappa$ and $G : \kappa$, we have the following logical equivalences (already in F^ω):*

$$\begin{array}{rcll}
(\forall X^\kappa.\ G \subseteq X \to G \subseteq F\, X) & \leftrightarrow & G \subseteq (\underline{\mathsf{Ran}}^{\kappa\to\kappa}\, F)\, G & (1) \\
(\forall X^\kappa \forall Y^\kappa.\ X \subseteq Y \to F\, X \subseteq F'\, Y) & \leftrightarrow & F \subseteq \underline{\mathsf{Ran}}^{\kappa\to\kappa'}\, F' & (2) \\
\underline{\mathsf{mon}}^{\kappa\to\kappa'}\, F & \leftrightarrow & F \subseteq \underline{\mathsf{Ran}}^{\kappa\to\kappa'}\, F & (3) \\
(\forall X^\kappa.\ X \subseteq G \to F\, X \subseteq G) & \leftrightarrow & (\underline{\mathsf{Lan}}^{\kappa\to\kappa}\, F)\, G \subseteq G & (4) \\
(\forall X^\kappa \forall Y^\kappa.\ X \subseteq Y \to F\, X \subseteq F'\, Y) & \leftrightarrow & \underline{\mathsf{Lan}}^{\kappa\to\kappa'}\, F \subseteq F' & (5) \\
\underline{\mathsf{mon}}^{\kappa\to\kappa'}\, F & \leftrightarrow & \underline{\mathsf{Lan}}^{\kappa\to\kappa'}\, F \subseteq F & (6)
\end{array}$$

Proof. The first two equivalences are proven by Schönfinkel's transposition operator $\mathsf{T} := \lambda s \lambda t \lambda f.\, s\, f\, t$—in either direction. For the fourth and fifth equivalence from left to right, the proof is $\mathsf{U} := \lambda s \lambda(\mathsf{pack}\langle f, t\rangle).\, s\, f\, t$, a kind of uncurrying operator, and $\lambda s \lambda f \lambda t.\, s\, (\mathsf{pack}\langle f, t\rangle)$ for the reverse direction, a currying procedure. Obviously, (3) is an instance of (2), and (6) is an instance of (5). □

The terms $\mathsf{T} := \lambda s \lambda t \lambda f.\, s\, f\, t$ and $\mathsf{U} := \lambda s \lambda(\mathsf{pack}\langle f, t\rangle).\, s\, f\, t$ 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 $\kappa \neq *$.)

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

The embedding $\ulcorner\cdot\urcorner$ of System MIt^ω of Mendler iteration and coiteration into $\underline{\mathsf{It}}^\omega$ is now straightforward. Kinds are left fixed; the translation of most type-constructor and term formers is homomorphic, e.g., a type application $\ulcorner F\,G\urcorner$ is encoded as a type application $\ulcorner F\urcorner\,\ulcorner G\urcorner$. Only syntax related to least and greatest fixed-points has to be translated non-homomorphically:

$$
\begin{array}{llll}
\text{Formation.} & \ulcorner\mu^\kappa\urcorner & : & (\kappa\to\kappa)\to\kappa \\
 & \ulcorner\mu^\kappa\urcorner & := & \underline{\mu}^\kappa\circ\underline{\mathsf{Lan}}^{\kappa\to\kappa} \\[1em]
\text{Introduction.} & \ulcorner\mathsf{in}^\kappa\urcorner & : & \forall F^{\kappa\to\kappa}.\ F\,\ulcorner\mu^\kappa F\urcorner\subseteq\underline{\mu}^\kappa(\underline{\mathsf{Lan}}^{\kappa\to\kappa}\,F) \\
 & \ulcorner\mathsf{in}^\kappa\urcorner & := & \lambda t.\ \underline{\mathsf{in}}^\kappa\,(\mathsf{lanId}\ t)
\end{array}
$$

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

$$
\begin{array}{llll}
\ulcorner\mathsf{MIt}^\kappa(s)\urcorner & := & \lambda x.\ \underline{\mathsf{It}}^\kappa(\underline{\mathsf{M}}^\kappa_{\mathsf{Lan}},\mathsf{U}\,\ulcorner s\urcorner)\ x & \text{where} \\
\mathsf{U}\,\ulcorner s\urcorner & : & \underline{\mathsf{Lan}}^{\kappa\to\kappa}\,\ulcorner F\urcorner\ulcorner G\urcorner\subseteq\ulcorner G\urcorner &
\end{array}
$$

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

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

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

Notice that the embedding employs a left Kan extension for the inductive case while the embedding of GMIt^ω into MIt^ω 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.

$$
\begin{array}{llll}
\text{Formation.} & \ulcorner \nu^\kappa \urcorner & : & (\kappa \to \kappa) \to \kappa \\
 & \ulcorner \nu^\kappa \urcorner & := & \underline{\nu}^\kappa \circ \underline{\mathsf{Ran}}^{\kappa\to\kappa}
\end{array}
$$

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

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

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

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

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

5.4 Remarks on Monotonicity

Although sufficient to cover all of MIt^ω, 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* $\underline{\mathsf{mon}}^{\mathsf{k2}}(\lambda X.\, X \circ X)$ *in* F^ω.

Proof. Assume, to the contrary,

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

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, $s := \lambda t \lambda a.\,t\,a : \forall A.\,XA \to YA$. Therefore, $m\,s : X(XZ) \to Y(YZ) = \neg\neg Z \to (Z \to Z) \to Z$, 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). $\square$

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:

$$\begin{array}{lll} \underline{\mathsf{mon}}^{\kappa\to\kappa'}\, F,\ \underline{\mathsf{mon}}^{\kappa}\, X & \text{imply} & \underline{\mathsf{mon}}^{\kappa'}(F\, X), \\ \underline{\mathsf{mon}}^{\kappa\to\kappa}\, F & \text{implies} & \underline{\mathsf{mon}}^{\kappa}(\underline{\mu}^{\kappa} F). \end{array}$$

This means 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, $\underline{\mathsf{mon}}^{\mathsf{k2}}\, F$ is inhabited by $m_F := \lambda x.\, \mathsf{id}$. Certainly, $F\, X = \lambda A.\, \neg A$ is not monotone: If $m : \mathsf{mon}^{\mathsf{k1}}(\lambda A.\, \neg A)$, 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 $\underline{\mu}^{\mathsf{k1}} F$ is not monotone because this type constructor is logically equivalent with $\lambda A.\, \neg A$: The data constructor $\underline{\mathsf{in}}^{\mathsf{k1}}$ yields one direction, the other comes from

$$\begin{array}{rcl} \underline{\mathsf{It}}^{\mathsf{k1}}(m_F,\, \mathsf{id}) & : & (\underline{\mu}^{\mathsf{k1}} F) \subseteq F\, (\underline{\mu}^{\mathsf{k1}} F) \\ & = & \forall A.\, (\underline{\mu}^{\mathsf{k1}} F)\, A \to \neg A \end{array}$$

Remark 2 (Another notion of monotonicity) Consider a modified notion of monotonicity which excludes some more type 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 MIt^ω. This is because for the appropriate modification of left Kan extension $\underline{\mathsf{Lan}}^\kappa$, (although a term $\underline{\mathsf{M}}_{\mathsf{Lan}}$ exists here as well,) the step term of MIt^κ does not have a type isomorphic to $\underline{\mathsf{Lan}}^{\kappa\to\kappa}\, F\, G \subseteq^\kappa G$.

Although we have shown that monotonicity of $F : \kappa \to \kappa$ fails to entail monotonicity of $\underline{\mu}^\kappa F$ in general, it does work if F additionally preserves monotonicity of its first argument. More precisely, if there is a term p which transforms every monotonicity witness $n : \underline{\mathsf{mon}}^\kappa X$ into a monotonicity witness $p\,n : \underline{\mathsf{mon}}^\kappa(F\,X)$, then $\underline{\mu}^\kappa F : \kappa$ is monotone, canonically. To see this, define

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

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

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

For well-behaved F, p and m (such as the regular rank-2 constructors and their canonical monotonicity-preservation and monotonicity witnesses described in Section 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 $\mathsf{MIt}^\omega_{\leq}$ 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 $\mathsf{MIt}^\kappa_{\leq}$ and GMCoit^κ to $\mathsf{MCoit}^\kappa_{\leq}$. The reduction rules do 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 $\mathsf{MIt}^\omega_{\leq}$. The system will be called $\mathsf{It}^\omega_{\leq}$ 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 in Hinze (2002) as well as in Altenkirch and McBride (2003). Contrast this with $\underline{\mathsf{mon}}^\kappa\,F$ which asserts that F is monotone in its *first* argument position, for *all* argument values.

Notice that for $\mathsf{k1} = * \to *$, the new definition of monotonicity, mon, coincides with the old one, $\underline{\mathsf{mon}}$. For higher ranks, however, the notions differ considerably. For instance, the type constructor $\lambda X.\ X \circ X : (\kappa \to \kappa) \to \kappa \to \kappa$, which for $\kappa = *$ we disproved to be monotonic w. r. t. the old notion in Lemma 5, is monotonic w. r. t. the new notion:

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

Also, the new definition is compatible with application: If $F : \kappa \to \kappa'$ and $X : \kappa$ are monotone, then $F\,X : \kappa'$ is monotone as well:

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

The following are the canonical *monotonicity witnesses* for some closed F which we will need in examples later.

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

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

6.1 Definition of $\mathsf{It}^\omega_=$

The system $\mathsf{It}^\omega_=$ 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.	$\dfrac{\Gamma \vdash F \,:\, \kappa \to \kappa, \quad \Gamma \vdash m \,:\, \mathsf{mon}^{\kappa\to\kappa}\, F \quad \Gamma \vdash G \,:\, \kappa \quad \Gamma \vdash s \;:\, F\,G \subseteq^\kappa G}{\Gamma \vdash \mathsf{It}^\kappa_{=}(m, s) : \mu^\kappa F \leq^\kappa G}$
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 $\lvert\vec{f}\rvert = \lvert\vec{\kappa}\rvert$.

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 $\mathsf{It}^\kappa_{=}$ (the "step term") has the type $F\,G \subseteq^\kappa G$, making (G, s) the syntactic representation of 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:

Lemma 6 (Monotonicity of Least Fixed-Points) *If type constructor $F : \kappa \to \kappa$ is monotone, witnessed by $m : \mathsf{mon}^{\kappa\to\kappa}\, F$, then $\mu^\kappa F$ is again monotone, witnessed by*

$$\mathsf{M}^\kappa_\mu(m) \;:=\; \mathsf{It}^\kappa_{=}(m, \mathsf{in}^\kappa) \;:\; \mathsf{mon}^\kappa(\mu^\kappa F), \quad \textit{where}$$
$$\mathsf{M}^\kappa_\mu(m)\,\vec{f}\,(\mathsf{in}^\kappa\, t) \longrightarrow \mathsf{in}^\kappa\,(m\,\mathsf{M}^\kappa_\mu(m)\,\vec{f}\,t).$$

Proof. Directly by instantiation of the typing and reduction rules for $\mathsf{It}^\kappa_{=}$. □

Remark 3 (Alternative Introduction Rule) To overcome the mismatch between step term and iterator mentioned above, one might accept the type of the iterator term $\mathsf{It}^\kappa_{=}(m, s)$ as it stands, but would use $\leq^\kappa$ instead of $\subseteq^\kappa$ for the types of data constructor in^κ and step term s. In fact, this was the typing rule underlying the original submission of Abel and Matthes (2003) and a similar typing rule was suggested to us also by Peter Aczel in May 2003. However, a data constructor of type $\mathsf{in}^\kappa : F\,(\mu^\kappa F) \leq^\kappa \mu^\kappa F$ would have the drawback that the canonical inhabitants of higher-order inductive types would be of the form $\mathsf{in}^\kappa\,\vec{g}\,t$, where the g_i are functions. As a consequence, a single data object could have several, even infinitely many distinct normal forms. For instance, $\mathsf{in}^{\mathsf{k1}}\ (\lambda n.\, n+5)\ (\mathsf{inl}\,10)$ and $\mathsf{in}^{\mathsf{k1}}\ \mathsf{id}\,(\mathsf{inl}\,15)$ would both denote the powerlist containing solely the number 15. For ground types, i. e., inductive types without embedded function spaces like powerlists, this seems unsatisfactory.

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

$$
\begin{array}{ll}
\text{Formation.} & \nu^\kappa \quad : (\kappa \to \kappa) \to \kappa \\
\text{Elimination.} & \mathsf{out}^\kappa : \forall F^{\kappa\to\kappa}.\, \nu^\kappa F \ \subseteq^\kappa\ F\,(\nu^\kappa F) \\
\text{Introduction.} & \dfrac{\begin{array}{l}\Gamma \vdash F : \kappa \to \kappa, \\ \Gamma \vdash m : \mathsf{mon}^{\kappa\to\kappa}\,F \\ \Gamma \vdash G : \kappa \\ \Gamma \vdash s : G \ \subseteq^\kappa\ F\,G\end{array}}{\Gamma \vdash \mathsf{Coit}^\kappa_=(m,s) : G \ \leq^\kappa\ \nu^\kappa F} \\
\text{Reduction.} & \mathsf{out}^\kappa\,(\mathsf{Coit}^\kappa_=(m,s)\,\vec{f}\,t) \longrightarrow_\beta m\ \mathsf{Coit}^\kappa_=(m,s)\,\vec{f}\,(s\,t) \\
 & \text{where } |\vec{f}| = |\vec{\kappa}|.
\end{array}
$$

As for least fixed-points, monotonicity of greatest fixed-points can be defined canonically.

Lemma 7 (Monotonicity of Greatest Fixed-Points) *If type constructor $F : \kappa \to \kappa$ is monotone, witnessed by $m : \mathsf{mon}^{\kappa\to\kappa}\,F$, then $\nu^\kappa F$ is again monotone, witnessed by*

$$
\begin{array}{l}
\mathsf{M}^\kappa_\nu(m) \ := \ \mathsf{Coit}^\kappa_=(m, \mathsf{out}^\kappa) \ : \ \mathsf{mon}^\kappa(\nu^\kappa F), \quad \textit{where} \\
\mathsf{out}^\kappa\,(\mathsf{M}^\kappa_\nu(m)\,\vec{f}\,t) \longrightarrow m\ \mathsf{M}^\kappa_\nu(m)\,\vec{f}\,(\mathsf{out}^\kappa\,t).
\end{array}
$$

6.2 Examples

The following two developments exemplify the use of $\mathsf{M}^{\mathsf{k1}}_\mu$, i. e., the preservation of monotonicity under formation of least fixed-points. More examples for programming in $\mathsf{It}^\omega_=$, also with $\mathsf{Coit}^{\mathsf{k1}}_=$, can be found in Abel and Matthes (2003).

Example 16 (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.

$$\begin{aligned}
\mathsf{lamf} &:= \lambda e \lambda f.\ \mathsf{either}\ f\ (\mathsf{either}\ (\mathsf{fork}\ (e\,f))\ (e\ (\mathsf{maybe}\ f))) &&: \mathsf{mon}^{\mathsf{k2}}\ \mathsf{LamF} \\
\mathsf{lam} &:= \mathsf{M}_{\mu}^{\mathsf{k1}}(\mathsf{lamf}) &&: \mathsf{mon}^{\mathsf{k1}}\ \mathsf{Lam}
\end{aligned}$$

The reduction behavior shows that we have indeed obtained the mapping function for de Bruijn terms:

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

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 weakening rule of natural deduction.

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

Example 17 (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 witnesses of PListF and PList are

$$\begin{aligned}
\mathsf{plistf} &:= \lambda e \lambda f.\ \mathsf{either}\ f\ (e\ (\mathsf{fork}\ f)) &&: \mathsf{mon}^{\mathsf{k2}}\ \mathsf{PListF} \\
\mathsf{plist} &:= \mathsf{M}_{\mu}^{\mathsf{k1}}(\mathsf{plistf}) &&: \mathsf{mon}^{\mathsf{k1}}\ \mathsf{PList}
\end{aligned}$$

The reversal program, however, does not make use of the canonical monotonicity witnesses. It is manufactured as follows:

$$\begin{aligned}
\mathsf{swap} &:= \lambda f \lambda \langle a_1, a_2\rangle.\ \langle f\ a_2,\ f\ a_1\rangle &&: \mathsf{mon}^{\mathsf{k1}}\ (\lambda A. A \times A) \\
\mathsf{revf}' &:= \lambda e \lambda f.\ \mathsf{either}\, f\ (e\ (\mathsf{swap}\ f)) &&: \mathsf{mon}^{\mathsf{k2}}\ \mathsf{PListF} \\[1ex]
\mathsf{rev}' &:= \mathsf{M}_{\mu}^{\mathsf{k1}}(\mathsf{revf}') &&: \mathsf{mon}^{\mathsf{k1}}\ \mathsf{PList} \\
\mathsf{rev} &:= \mathsf{rev}'\ \mathsf{id} &&: \mathsf{PList} \subseteq^{\mathsf{k1}} \mathsf{PList}
\end{aligned}$$

$$\begin{aligned}
\mathsf{rev}'\ f\ (\mathsf{zero}\ a) &\longrightarrow^{+} \mathsf{zero}^{\circ}\,(f\ a) \\
\mathsf{rev}'\ f\ (\mathsf{succ}\ l) &\longrightarrow^{+} \mathsf{succ}^{\circ}\,(\mathsf{rev}'\ (\mathsf{swap}\ f)\ l)
\end{aligned}$$

Specializing f to id in the reduction rules, it becomes traceable that rev reverses a powerlist.

6.3 *Embedding* $\mathsf{It}^\omega_=$ *into* $\mathsf{MIt}^\omega_=$

The iterator and coiterator of $\mathsf{It}^\omega_=$ are definable within $\mathsf{MIt}^\omega_=$ (see the beginning of this section) so that the typing rules are obeyed and reduction is simulated. We define:

$$\begin{array}{rcl}\mathsf{It}^\kappa_=(m,s) & := & \mathsf{MIt}^\kappa_=(\lambda it\lambda\vec{f}\lambda t.\, s\,(m\; it\; \vec{f}\; t))\\ \mathsf{Coit}^\kappa_=(m,s) & := & \mathsf{MCoit}^\kappa_=(\lambda coit\lambda\vec{f}\lambda t.\, m\; coit\; \vec{f}\,(s\,t))\end{array}$$

Embedding $\mathsf{MIt}^\omega_=$ into $\mathsf{It}^\omega_=$ 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 $\mathsf{MIt}^\omega_=$, 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, Gibbons, and Bayley (2003) are directly definable in this system. This will be shown later on in Section 9.

7.1 *Definition of System* GIt^ω

System GIt^ω recasts the generality of System GMIt^ω following the design of System $\mathsf{It}^\omega_=$. It generalizes $\mathsf{It}^\omega_=$ in two directions: $\leq$ is generalized to $\leq_{\vec{H}}$, and an additional type constructor parameter $F' : \kappa \to \kappa$ appearing in the type of m adds further flexibility. Compared to $\mathsf{It}^\omega_=$, only the typing rules are changed; the reduction rules are the same. Significantly, the term m in $\mathsf{GIt}^\kappa(m,s)$ is no longer a monotonicity witness in general, because of the changed type. Still, we consider it to be a form of conventional iteration as the division of work between the step term s and the pseudo monotonicity witness m is exactly the same as in the case of iteration of system $\mathsf{It}^\omega_=$: The term s handles assembling the result of a call of the iterative function from the results of the recursive calls while m organizes the recursive calls.

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

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

Evidently, $\mathsf{It}^\omega_=$ 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

We demonstrate programming in GIt^ω on two examples.

Example 18 (Summing up a Powerlist, Revisited) In GIt^ω, the implementation of the summation of a powerlist in system GMIt^ω (Example 12) can be closely mimicked. We can define

$$\begin{array}{lcl} \mathsf{sum}' & := & \mathsf{GIt}^{\mathsf{k1}}(m, s) \\ & : & \mathsf{PList} \leq^{\mathsf{k1}}_H G \ = \ \forall A \forall \_. \ (A \to \mathsf{Nat}) \to \mathsf{PList}\ \ A \to \mathsf{Nat} \end{array}$$

where

$$\begin{array}{lcl} G & := & \lambda\_.\ \mathsf{Nat} : \mathsf{k1} \\ H & := & \lambda\_.\ \mathsf{Nat} : \mathsf{k1} \\ Q & := & \text{anything} : \mathsf{k1} \\ F' & := & \lambda X \lambda A.\ \mathsf{Nat} + X\ \ (Q\ \ A) : \mathsf{k2} \\ m & := & \lambda e \lambda f.\ \ \mathsf{either}\ \ f\ \ \Big(e\ \ (\lambda\langle a_1, a_2\rangle.\ f\ a_1 + f\ a_2)\Big) \\ & : & \forall X \forall Y.\ X \leq^{\mathsf{k1}}_{\lambda\_.\ \mathsf{Nat}} Y \to \mathsf{PListF}\, X \leq^{\mathsf{k1}}_{\lambda\_.\ \mathsf{Nat}} F'\ Y \\ & = & \forall X \forall Y.\ \Big(\forall A' \forall B.\ (A' \to \mathsf{Nat}) \to X\ \ A' \to Y\ \ B\Big) \\ & & \quad \to \forall A \forall B'.\ (A \to \mathsf{Nat}) \to A + X\ \ (A \times A) \to \mathsf{Nat} + Y\ \ (Q\ \ B') \\ s & := & \lambda t.\ \mathsf{match}\ t\ \mathsf{with}\ \ \mathsf{inl}\, n \mapsto n \mid \mathsf{inr}\, n \mapsto n \\ & : & F'\ \ (\lambda\_.\ \mathsf{Nat}) \subseteq^{\mathsf{k1}} \lambda\_.\ \mathsf{Nat} = \forall\_.\ \mathsf{Nat} + \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

The reduction behavior is precisely that of sum' in Example 12, but the work accomplished by s in that example is now divided between m and s. Crucially, the addition (the non-polymorphic operation of the recursive definition) takes place in m. The reason why any type constructor of kind $\mathsf{k1}$ can be used as Q is that the type constructors G and H used by the iterator are constant, which is a degenerate situation. (In Example 25, Q will be chosen in a canonical way.)

We see that one should not at all think of real monotonicity witnesses in GIt^ω. The pseudo monotonicity witnesses are meant to do work specific to the programming task at hand. Example 17 of powerlist reversal by means of a noncanonical monotonicity witness demonstrated this idea as well.

The next example shows that the iterator of GIt^ω turns out to be very handy when one wants to move on from variable renaming in de Bruijn terms to substitution.

Example 19 (Substitution for de Bruijn Terms) In GIt^ω, the following smooth definition of substitution for de Bruijn terms as an iteration is possible, where we first define lifting as in the structurally inductive approach in

Altenkirch and Reus (1999).

$$
\begin{array}{lll}
\mathsf{lift} & := & \lambda f \lambda x.\ \mathsf{case}\,(x,\ u.\ \mathsf{var}\ (\mathsf{inl}\ \ u),\ a.\ \mathsf{weak}\ (f\ \ a)) \\
 & : & \forall A \forall B.\ (A \to \mathsf{Lam}\ B) \to 1 + A \to \mathsf{Lam}\,(1 + B) \\
\mathsf{subst} & := & \mathsf{GIt}^{\mathsf{k1}}(m, s) \\
 & : & \mathsf{Lam} \leq^{\mathsf{k1}}_{\mathsf{Lam}} \mathsf{Lam} = \forall A \forall B.\ (A \to \mathsf{Lam}\ B) \to (\mathsf{Lam}\ A \to \mathsf{Lam}\ B)
\end{array}
$$

where

$$
\begin{array}{lll}
F' & := & \lambda X \lambda A.\ \ \mathsf{Lam}\ A + (X\ \ A \times X\ \ A + X\ \ (1 + A)) : \mathsf{k2} \\
m & := & \lambda e \lambda f.\ \ \mathsf{either}\ \ f\ \Big(\mathsf{either}\ (\mathsf{fork}\ (e\ \ f))\ \ (e\ \ (\mathsf{lift}\ \ f))\Big) \\
 & : & \forall X \forall Y.\ X \leq^{\mathsf{k1}}_{\mathsf{Lam}} Y \to \mathsf{LamF}\,X \leq^{\mathsf{k1}}_{\mathsf{Lam}} F'\ \ Y \\
 & = & \forall X \forall Y.\ (\forall A \forall B.\ (A \to \mathsf{Lam}\ \ B) \to X\ \ A \to Y\ \ B) \\
 & & \quad \to \forall A \forall B.\ (A \to \mathsf{Lam}\ \ B) \\
 & & \qquad \to A + (X\ \ A \times X\ \ A + X\ \ (1 + A)) \\
 & & \qquad\quad \to \mathsf{Lam}\ \ B + (Y\ \ B \times Y\ \ B + Y\ \ (1 + B)) \\
s & := & \lambda t.\ \mathsf{match}\ t\ \mathsf{with}\ \ \mathsf{inl}\,u \mapsto u \mid \mathsf{inr}\,t' \mapsto \mathsf{in}^{\mathsf{k1}}(\mathsf{inr}\ \ t') \\
 & : & F'\ \ \mathsf{Lam} \subseteq^{\mathsf{k1}} \mathsf{Lam} \\
 & = & \forall A.\ \ \mathsf{Lam}\ A + (\mathsf{Lam}\ A \times \mathsf{Lam}\ A + \mathsf{Lam}\,(1 + A)) \to \mathsf{Lam}\ A
\end{array}
$$

Notice that we use weakening weak of Example 16 in the definition of the lifting function lift, implicitly embedding $\mathsf{It}^{\omega}_{=}$ into GIt^{ω}. The program subst has exactly the expected reduction behavior in the sense that, if $f : A \to \mathsf{Lam}\,B$ is a substitution rule, then $\mathsf{subst}\ \ f : \mathsf{Lam}\,A \to \mathsf{Lam}\,B$ behaves as the corresponding substitution function replacing the variables of its de Bruijn term argument according to the rule f:

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

Alternatively, one might program substitution within $\mathsf{It}^{\omega}_{=}$, but this would necessitate an explicit use of a right Kan extension—a fact swept under the carpet in Altenkirch and Reus (1999). In GIt^{ω}, this more liberal format is part of the design.

GIt^ω embeds into GMIt^ω much the same way as $\mathsf{It}^\omega_=$ embeds into $\mathsf{MIt}^\omega_=$:

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

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

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

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

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

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

Hence, the reductions are simulated. Type-preservation has not yet been addressed; however, it is a consequence of the following lemma.

Lemma 8 *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 vector $\vec{H}$ enters only the arguments; the Kan extensions are not formed along $\vec{H}$. The premisses $F'Y \subseteq^\kappa F'Y$ and $F'X \subseteq^\kappa F'X$ are there just 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 18 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 MIt^ω, we do the following examples in MIt^ω 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 20 (Explicit Substitutions) Examples 16 and 19 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: [2]

$$\begin{array}{lll}
\widehat{\mathsf{LamF}} & := \lambda X \lambda A.\ \mathsf{LamF}\, X\, A + X\,(X\,A) & : \mathsf{k2} \\
\widehat{\mathsf{Lam}} & := \mu^{\mathsf{k1}}\, \widehat{\mathsf{LamF}} & : \mathsf{k1} \\
 & & \\
\widehat{\mathsf{var}} & := \lambda b.\, \mathsf{in}^{\mathsf{k1}}\,(\mathsf{inl}\,(\mathsf{inl}\, b)) & : \forall A.\ A \to \widehat{\mathsf{Lam}}\ A \\
\widehat{\mathsf{app}} & := \lambda t_1 \lambda t_2.\, \mathsf{in}^{\mathsf{k1}}\,(\mathsf{inl}\,(\mathsf{inr}\,(\mathsf{inl}\,\langle t_1, t_2\rangle))) & : \forall A.\ \widehat{\mathsf{Lam}}\ A \to \widehat{\mathsf{Lam}}\ A \to \widehat{\mathsf{Lam}}\ A \\
\widehat{\mathsf{abs}} & := \lambda r.\, \mathsf{in}^{\mathsf{k1}}\,(\mathsf{inl}\,(\mathsf{inr}\,(\mathsf{inr}\, r))) & : \forall A.\ \widehat{\mathsf{Lam}}\,(1 + A) \to \widehat{\mathsf{Lam}}\ A \\
\widehat{\mathsf{flat}} & := \lambda e.\, \mathsf{in}^{\mathsf{k1}}\,(\mathsf{inr}\, e) & : \forall A.\ \widehat{\mathsf{Lam}}\,(\widehat{\mathsf{Lam}}\ A) \to \widehat{\mathsf{Lam}}\ A
\end{array}$$

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 16:

$$\begin{array}{lll}
\widehat{\mathsf{lamf}} & := \lambda g \lambda f.\ \mathsf{either}\,(\mathsf{lamf}\, g\, f)\,(g\,(g\, f)) & : \mathsf{mon}^{\mathsf{k2}}\ \widehat{\mathsf{LamF}} \\
\widehat{\mathsf{lam}} & := \mathsf{M}^{\mathsf{k1}}_{\mu}(\widehat{\mathsf{lamf}}) = \mathsf{It}^{\mathsf{k1}}_{=}(\widehat{\mathsf{lamf}},\ \mathsf{in}^{\mathsf{k1}}) & : \mathsf{mon}^{\mathsf{k1}}\ \widehat{\mathsf{Lam}}
\end{array}$$

Note that the treatment of explicit flattening in the definition of $\widehat{\mathsf{lamf}}$ would be impossible with basic monotonicity $\underline{\mathsf{mon}}^{\mathsf{k1}}$, see Lemma 5. The following reduction behavior immediately follows:

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

As was the case with the summation for the truly nested datatype Bush in Example 13, 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 MIt^{ω}.

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

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

[2] This presentation should be compared with the slightly unmotivated extension of de Bruijn's notation in Bird and Paterson (1999b, Sect. 5), where $\widehat{\mathsf{abs}}$ and $\widehat{\mathsf{flat}}$ are replaced by just one constructor of type $\forall A.\ \widehat{\mathsf{Lam}}\,(1 + \widehat{\mathsf{Lam}}\ A) \to \widehat{\mathsf{Lam}}\ A$, which again gives rise to true nesting. However, that constructor could easily be defined as $\widehat{\mathsf{flat}} \circ \widehat{\mathsf{abs}}$ in the present system.

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.

Alternatively, one can represent full explicit substitution by way of a data constructor like $\widehat{\mathsf{flat}}$. If we redefine $\widehat{\mathsf{Lam}}$ to be

$$\widehat{\mathsf{Lam}} \;:=\; \mu^{\mathsf{k1}}\Big(\lambda X \lambda A.\; \mathsf{LamF}\, X\, A + \exists B.\,(B \to X\, A) \times X\, B\Big),$$

we may set $\widehat{\mathsf{exs}} \;:=\; \lambda f \lambda t.\,\mathsf{in}^{\mathsf{k1}}\,(\mathsf{inr}\,(\mathsf{pack}\langle f, t\rangle))$, which (after exchanging the bound variables A and B) receives the type we have above for esubst. Notice that $\exists B.\,(B \to X\, A) \times X\, B \;=\; (\underline{\mathsf{Lan}}^{\mathsf{k1}}\, X)\,(X\, A)$, with the naive Kan extension defined in Section 5.3. From Lemma 4, it follows that $(\underline{\mathsf{Lan}}^{\mathsf{k1}}\, X)\,(X\, A)$ and $X\,(X\, A)$ are logically equivalent if X is monotone. Since the fixed point $\widehat{\mathsf{Lam}}$ is monotone, the variant just discussed is logically equivalent with our example above. The formulation with explicit flattening has the advantage of not using quantifiers in the datatype definition. Instead of that, it needs true nesting.

Example 21 (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 $\mathsf{It}^{\mathsf{k1}}_{=}$, this is defined by

$$\begin{aligned} \mathsf{eval}' &:= \mathsf{It}^{\mathsf{k1}}_{=}(\widehat{\mathsf{lamf}}, s) &&: \widehat{\mathsf{Lam}} \leq^{\mathsf{k1}} \mathsf{Lam} \\ \mathsf{eval} &:= \mathsf{eval}'\; \mathsf{id} &&: \widehat{\mathsf{Lam}} \subseteq^{\mathsf{k1}} \mathsf{Lam} \end{aligned}$$

where $s := \lambda t.\; \mathsf{case}\,(t,\, t'.\,\mathsf{in}^{\mathsf{k1}}\; t',\, e.\; \mathsf{flatten}\, e) : \widehat{\mathsf{LamF}}\,\mathsf{Lam} \subseteq^{\mathsf{k1}} \mathsf{Lam}$, with $\mathsf{flatten}$ taken from the previous example. The most interesting case of the reduction behavior is

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

As in the last example, the nesting in the definition of datatype $\widehat{\mathsf{Lam}}$ is reflected in the nested recursion in eval'.

Example 22 (Redecoration of Finite Triangles) In the following, we reimplement the redecoration algorithms for finite triangles, as opposed to infinite ones. Passing from coinductive to inductive types, we need to apply rather different programming methodologies.

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:

$$\begin{aligned} \mathsf{FTriF} &:= \lambda X \lambda A.\; A \times (1 + X\,(E \times A)) &&: \mathsf{k2} \\ \mathsf{FTri} &:= \mu^{\mathsf{k1}}\, \mathsf{FTriF} &&: \mathsf{k1} \end{aligned}$$

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

$$\begin{array}{lcll} \mathsf{sg} & := & \lambda a.\, \mathsf{in}^{\mathsf{k1}}\, \langle a, \mathsf{inl}\, \langle\rangle\rangle & : \ \forall A.\, A \to \mathsf{FTri}\, A, \quad \text{and} \\ \mathsf{cons} & := & \lambda a \lambda r.\, \mathsf{in}^{\mathsf{k1}}\, \langle a, \mathsf{inr}\ r\rangle & : \ \forall A.\, A \to \mathsf{FTri}\,(E \times A) \to \mathsf{FTri}\, A. \end{array}$$

There are two monotonicity witnesses for FTriF, for $\underline{\mathsf{mon}}^{\mathsf{k2}}$ and for $\mathsf{mon}^{\mathsf{k2}}$:

$$\begin{array}{lcll} \underline{\mathsf{ftrif}} & := & \lambda g.\ \mathsf{pair}\ \mathsf{id}\ (\mathsf{maybe}\ g) & : \ \underline{\mathsf{mon}}^{\mathsf{k2}} \\ \mathsf{ftrif} & := & \lambda g \lambda f.\ \mathsf{pair}\ f\,(\mathsf{maybe}\ (g\ (\mathsf{pair}\ \mathsf{id}\ f))) & : \ \mathsf{mon}^{\mathsf{k2}} \\ \mathsf{ftri} & := & \mathsf{M}_{\mu}^{\mathsf{k1}}(\mathsf{ftrif}) & : \ \mathsf{mon}^{\mathsf{k1}}\,\mathsf{FTri} \end{array}$$

Note that the last definition uses means of $\mathsf{It}^{\omega}_{=}$. For the definition of redecoration we need methods to decompose triangles. The following function ftop returns the first column of a triangle, which happens to be just the topmost diagonal element. Later we will define another function fcut which takes the remaining trapezium and removes its top row.

$$\begin{array}{lcl} \mathsf{ftop} & : & \forall A.\ \mathsf{FTri}\ A \to A \\ \mathsf{ftop} & := & \underline{\mathsf{It}}^{\mathsf{k1}}(\underline{\mathsf{ftrif}}, \mathsf{fst}) \end{array}$$

This function uses the iterator from $\underline{\mathsf{It}}^{\omega}$. It could be also be implemented in $\mathsf{It}^{\omega}_{=}$ as $\mathsf{It}^{\mathsf{k1}}_{=}(\mathsf{ftrif}, \mathsf{fst})\ \mathsf{id}$. Either way, reduction is as expected:

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

As announced above, 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 $\mathsf{FTri}\ A$. Since in the domain type of this function, the argument to FTri is not a type variable, it does not fit directly into any of our iteration schemes. Aiming at using GMIt^{ω}, we need to define a more general function $\mathsf{fcut}' : \mathsf{FTri} \leq^{\mathsf{k1}}_{H} \mathsf{FTri}$ with $H := \lambda A.\, E \times A$. Note that this is a rare instance of the scheme $\mu F \leq^{\kappa_1}_{H} G$ with $G \neq H \neq \mathsf{Id}$.

$$\begin{array}{lcl} \mathsf{fcut}' & : & \forall A \forall B.\ (A \to E \times B) \to \mathsf{FTri}\ A \to \mathsf{FTri}\ B \\ \mathsf{fcut}' & := & \mathsf{GMIt}^{\mathsf{k1}}\Big(\lambda \mathit{fcut}' \lambda f.\ \mathsf{pair}\ (\mathsf{snd} \circ f)\ (\mathsf{maybe}\ (\mathit{fcut}'\ (\mathsf{pair}\ \mathsf{id}\ f)))\Big) \end{array}$$

$$\begin{array}{ll} \mathsf{fcut}'\ f\ (\mathsf{sg}\ a) & \longrightarrow^{+}\ \mathsf{sg}^{\circ}\,(\mathsf{snd}\,(f\ a)) \\ \mathsf{fcut}'\ f\ (\mathsf{cons}\, a\, r) & \longrightarrow^{+}\ \mathsf{cons}^{\circ}\,(\mathsf{snd}\,(f\ a))\ (\mathsf{fcut}'\,(\mathsf{pair}\ \mathsf{id}\ f)\, r) \end{array}$$

The cut function is obtained by specializing f to the identity.

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

Unfortunately, in the β-theory alone we do not get the desired reduction behavior. It holds that

$$\begin{array}{lll} \mathsf{fcut}\ (\mathsf{sg}\,\langle e, a\rangle) & \longrightarrow^+ a, & \text{and} \\ \mathsf{fcut}\ (\mathsf{cons}\ \langle e, a\rangle\ r) & \longrightarrow^+ \mathsf{cons}\ a\ (\mathsf{fcut}'\ (\mathsf{pair}\ \mathsf{id}\ \mathsf{id})\ r), & \text{but} \\ \mathsf{fcut}\ (\mathsf{cons}\ \langle e, a\rangle\ r) & \not\longrightarrow^+ \mathsf{cons}\ a\ (\mathsf{fcut}\ r). & \end{array}$$

However, if one added a tiny bit of extensionality, one would have extensional equality of $\mathsf{pair}\ \mathsf{id}\ \mathsf{id}$ and id, which would imply extensional equality of left and right hand side of the last relation.

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 in Example 11, but with the new auxiliary functions.

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

Finally, we can define redecoration fredec. Its description is the same as that for redec that works on infinite triangles in Example 11. The only difference is that we swapped its arguments such that its type now is

$$\begin{array}{lrcl} \forall A.\ \mathsf{FTri}\,A \to & \forall B.\ (\mathsf{FTri}\,A \to B) \to \mathsf{FTri}\,B & = & \mathsf{FTri} \subseteq^{\mathsf{k1}} G, \\ \text{where} & \forall B.\ (\mathsf{FTri}\,A \to B) \to \mathsf{FTri}\,B & =: & G. \end{array}$$

Unfortunately, G is not a right Kan extension (which might have allowed a direct definition of fredec inside GMIt^ω), but $G = (\mathsf{Ran}^{\mathsf{k1}}_{\mathsf{Id}}\,\mathsf{FTri}) \circ \mathsf{FTri}$. 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 $\mathsf{fredec}' : \mathsf{FTri} \subseteq^{\mathsf{k1}} \mathsf{FTri} \times^{\mathsf{k1}} G$. It seems that any hardwired Kan extensions in the system would only complicate the following definition which is done in plain MIt^ω.

$$
\begin{array}{lll}
\mathsf{fredec}' & : & \forall A.\ \mathsf{FTri}\,A \to (\mathsf{FTri}\,A \times (\forall B.\ (\mathsf{FTri}\,A \to B) \to \mathsf{FTri}\,B)) \\
\mathsf{fredec}' & := & \mathsf{MIt}^{\mathsf{k1}}\Big(\lambda \mathit{fredec}' \lambda t. \\
& & \qquad\qquad \begin{array}{lll} \mathsf{let}\ \mathit{fid} & = \mathsf{fst} \circ \mathit{fredec}' & \mathsf{in} \\ \mathsf{let}\ \mathit{fredec} & = \mathsf{snd} \circ \mathit{fredec}' & \mathsf{in} \\ \mathsf{let}\ r & = \mathsf{in}^{\mathsf{k1}}(\underline{\mathsf{ftrif}}\ \mathit{fid}\, t) & \mathsf{in} \end{array} \\
& & \qquad\qquad\quad \Big\langle r,\ \lambda f.\ \mathsf{match}\ t\ \mathsf{with} \\
& & \qquad\qquad\qquad\qquad \begin{array}{ll} \mathsf{sg}^-\ \_ & \mapsto \mathsf{sg}\ (f\,r) \\ \mathsf{cons}^-\ \_\ x & \mapsto \mathsf{cons}\ (f\,r)\ (\mathit{fredec}\ x\ (\mathsf{flift}\,f))\Big\rangle \end{array}
\end{array}
$$

The function fredec' actually defines two functions simultaneously: $\mathsf{fid} := \mathsf{fst} \circ \mathsf{fredec}' : \mathsf{FTri} \subseteq^{\mathsf{k1}} \mathsf{FTri}$, an iterative identity on finite triangles, and $\mathsf{fredec} := \mathsf{snd} \circ \mathsf{fredec}' : \mathsf{FTri} \subseteq^{\mathsf{k1}} G$, the actual redecoration function. The iterative identity is needed to reconstruct the current function argument r from its unfolded version t. Why a simple "$r = \mathsf{in}^{\mathsf{k1}} t$" does not do the job can be seen from the types of the bound variables:

$$
\begin{array}{lll}
\mathit{fredec}' & : & X \subseteq^{\mathsf{k1}} \mathsf{FTri} \times^{\mathsf{k1}} G \\
t & : & \mathsf{FTriF}\ X\ A = A \times (1 + X\,(E \times A)) \\
\mathit{fid} & : & X \subseteq^{\mathsf{k1}} \mathsf{FTri} \\
r & : & \mathsf{FTri}\ A \\
f & : & \mathsf{FTri}\ A \to B \\
x & : & X\,(E \times A) \\
\mathit{fredec} & : & X\,(E \times A) \to (\mathsf{FTri}\,(E \times A) \to (E \times B)) \to \mathsf{FTri}\,(E \times B)
\end{array}
$$

Since in the Mendler discipline t is *not* of type $\mathsf{FTriF}\ \mathsf{FTri}\ A$, we cannot apply $\mathsf{in}^{\mathsf{k1}}$ to t directly, but need a conversion function of type $X \subseteq^{\mathsf{k1}} \mathsf{FTri}$. The functionality of fredec' can be understood through its reduction behavior:

$$
\begin{array}{lll}
\mathsf{fredec}'\ (\mathsf{sg}\ a) & \longrightarrow^+ & \Big\langle \mathsf{sg}^\circ\, a,\ \lambda f.\ \mathsf{sg}\ (f\ (\mathsf{sg}^\circ\, a))\Big\rangle \\
\mathsf{fredec}'\ (\mathsf{cons}\ a\, x) & \longrightarrow^+ & \Big\langle \mathsf{cons}^\circ\ a\ (\mathsf{fid}\, x), \\
& & \quad \lambda f.\ \mathsf{cons}\ \Big(f\ (\mathsf{cons}^\circ\, a\ (\mathsf{fid}\, x))\Big)\ \Big(\mathsf{fredec}\ x\,(\mathsf{flift}\, f)\Big)\Big\rangle
\end{array}
$$

Untangling the two intertwined functions within fredec', we get the following reduction relations from which correctness of the implementation becomes apparent.

$$
\begin{array}{lll}
\mathsf{fid}\ (\mathsf{sg}\ a) & \longrightarrow^{+} & \mathsf{sg}^{\circ}\ a \\
\mathsf{fid}\ (\mathsf{cons}\ a\, x) & \longrightarrow^{+} & \mathsf{cons}^{\circ}\ a\ (\mathsf{fid}\, x) \\[1ex]
\mathsf{fredec}\ (\mathsf{sg}\ a)\ f & \longrightarrow^{+} & \mathsf{sg}\ (f\ (\mathsf{sg}^{\circ}\ a)) \\
\mathsf{fredec}\ (\mathsf{cons}\ a\, x)\ f & \longrightarrow^{+} & \mathsf{cons}\ (f\ (\mathsf{cons}^{\circ}\ a\ (\mathsf{fid}\, x)))\ (\mathsf{fredec}\ x\, (\mathsf{flift}\, f))
\end{array}
$$

Since fid is an iteratively defined identity, $\mathsf{fid}\, x$ with x a variable does not reduce to x. Apart from this deficiency, which could be overcome if a scheme of *primitive recursion* was available, fredec behaves as specified.

9 Efficient Folds

In this section, we relate our iteration schemes to other approaches found in the literature. Bird and Meertens (1998) were the first to publish iteration schemes for nested datatypes, called simple folds, which correspond to our System $\underline{\mathsf{It}}^{\omega}$. To overcome their limited usability, Bird and Paterson (1999a) formulated *generalized folds*. Their proposal inspired our work on System GMIt^{ω}, but our attempts to establish a clear relationship between their and our approach failed, for reasons we can explain better at the end of this section.

The generalized folds of Bird and Paterson exhibit an inefficiency in their computational behavior. To mend this flaw, Hinze (2000a) proposed an alternative system of folds for nested datatypes. Inspired from that, Martin, Gibbons and Bayley (2003) presented their *efficient folds*, or *efolds* for short, which are closest to Bird and Paterson's generalized folds.

All of the abovementioned approaches only deal with least fixed points $\mu^{\mathsf{k1}} F$ of special type constructors $F : \mathsf{k2}$ of rank 2, which are called *hofunctors* (short for *higher-order functors*). Since our systems have no such restrictions, it may well be possible that the other approaches can be simulated in our systems. In the following, we will demonstrate this for the proposal of Martin, Gibbons, and Bayley (2003). Their efolds can be expressed in System GIt^{ω}.

Hofunctors. Following Martin, Gibbons, and Bayley (2003), hofunctors are type constructors $F : \mathsf{k2}$ of one of the following shapes:

(a)	$\lambda\_.Q$	with $m_Q : \mathsf{mon}^{\mathsf{k1}}$ (note $Q : \mathsf{k1}$)	constant
(b)	$\lambda X.X$		identity
(c)	$F_0 +^{\mathsf{k2}} F_1$	with F_0, F_1 hofunctors	disjoint sum
(d)	$F_0 \times^{\mathsf{k2}} F_1$	with F_0, F_1 hofunctors	product
(e)	$\lambda X.\, Q \circ (F_0\ X)$	with $m_Q : \mathsf{mon}^{\mathsf{k1}}$ and F_0 hofunctor	composition
(f)	$\lambda X.\, X \circ (F_0\ X)$	with F_0 hofunctor	nesting

Note that this inductive characterization is not deterministic, e. g., one can always apply rule (e) with $Q = \mathsf{Id}$ without modifying the hofunctor extensionally. Even more, case (b) is a special case of (f) with $F_0 = \lambda\_.\,\mathsf{Id}$. Probably, case (b) is present in Martin, Gibbons, and Bayley (2003) in order to characterize non-nested hofunctors by rules (a) – (e).

Example 23 (Hofunctors) All of the type constructors $F : \mathsf{k2}$ whose fixed-points we considered in the previous examples are hofunctors. For instance,

$$
\begin{aligned}
\mathsf{PListF} &:= \lambda X \lambda A.\, A + X\,(A \times A) \\
&= (\lambda\_.\,\mathsf{Id}) +^{\mathsf{k2}} \lambda X.\ X \circ \Big(((\lambda\_.\,\mathsf{Id}) \times^{\mathsf{k2}} (\lambda\_.\,\mathsf{Id}))\ X\Big) \\[1ex]
\mathsf{BushF} &:= \lambda X \lambda A.\, 1 + X\,(X\ A) \\
&= (\lambda\_\lambda\_.1) +^{\mathsf{k2}} \lambda X.\ X \circ ((\lambda X.\ X)\ X) \\[1ex]
\mathsf{LamF} &:= \lambda X \lambda A.\, A + ((X\ A \times X\ A) + X\,(1 + A)) \\
&= (\lambda\_.\,\mathsf{Id}) +^{\mathsf{k2}} \Big((\mathsf{Id} \times^{\mathsf{k2}} \mathsf{Id}) +^{\mathsf{k2}} \lambda X.\ X \circ ((\lambda\_\lambda A.\, 1 + A)\ X)\Big)
\end{aligned}
$$

Efficient folds are another means to construct functions of type $\mu^{\mathsf{k1}} F \leq^{\mathsf{k1}}_H G$ which eliminate inhabitants of the nested datatype $\mu^{\mathsf{k1}} F$. As in the previous sections, $F : \mathsf{k2}$ and $G, H : \mathsf{k1}$, but now F additionally needs to be a hofunctor. By induction on the generation of hofunctor F we define a type constructor $\mathsf{F}^F_H : \mathsf{k2}$ which is parametric in H, a sequence of types $\vec{\mathsf{D}}^F_H$ all of which are parametric in H, and a term

$$\mathsf{M}^F(\vec{d}) \ :\ \forall X \forall Y.\ X \leq^{\mathsf{k1}}_H Y \to F\,X \leq^{\mathsf{k1}}_H \mathsf{F}^F_H\,Y$$

which is dependent on a sequence of terms $\vec{d} : \vec{\mathsf{D}}^F_H$. How exactly we obtain F^F_H, $\vec{\mathsf{D}}^F_H$ and $\mathsf{M}^F(\cdot)$ will be explained later. With these definition, we can introduce typing and reduction for efolds.

$$\text{Elimination.}\quad \frac{\begin{array}{l}\Gamma \vdash F\ :\mathsf{k2}\ \text{hofunctor}\\ \Gamma \vdash H\ :\mathsf{k1}\\ \Gamma \vdash \vec{d}\ :\vec{\mathsf{D}}^F_H\\ \Gamma \vdash G\ :\mathsf{k1}\\ \Gamma \vdash s\ \ :\mathsf{F}^F_H\,G \subseteq^{\mathsf{k1}} G\end{array}}{\Gamma \vdash \mathsf{efold}^F(\vec{d}, s) : \mu^{\mathsf{k1}} F \leq^{\mathsf{k1}}_H G}$$

$$\text{Reduction.}\quad \mathsf{efold}^F(\vec{d}, s)\ f\ (\mathsf{in}^{\mathsf{k1}}\, t) \longrightarrow_\beta s\ (\mathsf{M}^F(\vec{d})\ \mathsf{efold}^F(\vec{d}, s)\ f\ t)$$

Embedding into GIt^ω. Efficient folds are simply an instance of generalized conventional iteration for kind $\mathsf{k1}$.

$$\mathsf{efold}^F(\vec{d}, s) \quad := \quad \mathsf{GIt}^{\mathsf{k1}}(\mathsf{M}^F(\vec{d}),\ s)$$

To see that this definition preserves typing and reduction, recall the $\mathsf{k1}$ elimination and computation rule for generalized conventional iteration:

$$\text{Elimination.}\quad \frac{\begin{array}{l}\Gamma \vdash m : \forall X \forall Y.\ X \leq^{\mathsf{k1}}_H Y \to F\,X \leq^{\mathsf{k1}}_H F'\,Y\\ \Gamma \vdash s : F'\,G \subseteq^{\mathsf{k1}} G\end{array}}{\Gamma \vdash \mathsf{GIt}^{\mathsf{k1}}(m, s) : \mu^{\mathsf{k1}} F \leq^{\mathsf{k1}}_H\ G}$$

$$\text{Reduction.}\quad \mathsf{GIt}^{\mathsf{k1}}(m, s)\ f\ (\mathsf{in}^{\mathsf{k1}}\, t) \longrightarrow_\beta s\ (m\ \mathsf{GIt}^{\mathsf{k1}}(m, s)\ f\ t)$$

The free parameter $F' : \mathsf{k2}$ in the elimination rule is instantiated by the type constructer F^F_H generated from F, and m is replaced by $\mathsf{M}^F(\vec{d})$, which assembles the simpler terms $\vec{d}$ into a pseudo monotonicity witness. Hence, efficient folds can be viewed as a user interface for $\mathsf{GIt}^{\mathsf{k1}}$, which takes on the difficult task of choosing an appropriate F'.

Definition of efficient folds. To complete the description of efolds, the hofunctor F^F_H, the types $\vec{\mathsf{D}}^F_H$ and the term M^F are defined inductively by the hofunctoriality of F. In principle, any consistent definition gives rise to a class of efficient folds, the question is only how useful they will be, i. e., which functions can be programmed as instances of these efolds. Figure 2 lists Martin, Gibbon and Bayley's (2003) choices of F^F_H, $\vec{\mathsf{D}}^F_H$ and M^F for each rule (a – f) how to generate a hofunctor. Especially interesting are cases (b), (e) and (f) where a new term d is assumed. We will comment on the role of these terms later. Also observe, that in the last two cases F^F_H is defined via F_0, not recursively via $\mathsf{F}^{F_0}_H$. This is due to the emission of a new term d.

	F	F^F_H	$\vec{d} : \vec{\mathsf{D}}^F_H$	$\mathsf{M}^F(\vec{d})\ e\ f$
(a)	$\lambda\_.Q$	$\lambda\_.\ Q \circ H$		$m_Q\ f$
(b)	$\lambda X.X$	$\lambda X.X$	$d\ : H \subseteq^{\mathsf{k1}} H$	$e\ (d \circ f)$
(c)	$F_0 +^{\mathsf{k2}} F_1$	$\mathsf{F}^{F_0}_H +^{\mathsf{k2}} \mathsf{F}^{F_1}_H$	$\vec{d_0} : \vec{\mathsf{D}}^{F_0}_H$, $\vec{d_1} : \vec{\mathsf{D}}^{F_1}_H$	$\mathsf{either}\ (\mathsf{M}^{F_0}(\vec{d_0})\ e\ f)\ (\mathsf{M}^{F_1}(\vec{d_1})\ e\ f)$
(d)	$F_0 \times^{\mathsf{k2}} F_1$	$\mathsf{F}^{F_0}_H \times^{\mathsf{k2}} \mathsf{F}^{F_1}_H$	$\vec{d_0} : \vec{\mathsf{D}}^{F_0}_H$, $\vec{d_1} : \vec{\mathsf{D}}^{F_1}_H$	$\mathsf{pair}\ (\mathsf{M}^{F_0}(\vec{d_0})\ e\ f)\ (\mathsf{M}^{F_1}(\vec{d_1})\ e\ f)$
(e)	$\lambda X.\ Q \circ (F_0\ X)$	$\lambda X.\ Q \circ H \circ (F_0\ X)$	$d\ : \forall X.\ \mathsf{F}^{F_0}_H\ X \subseteq^{\mathsf{k1}} H \circ (F_0\ X)$, $\vec{d_0} : \vec{\mathsf{D}}^{F_0}_H$	$m_Q\ (d \circ (\mathsf{M}^{F_0}(\vec{d_0})\ e\ f))$
(f)	$\lambda X.\ X \circ (F_0\ X)$	$\lambda X.\ X \circ (F_0\ X)$	$d\ : \forall X.\ \mathsf{F}^{F_0}_H\ X \subseteq^{\mathsf{k1}} H \circ (F_0\ X)$, $\vec{d_0} : \vec{\mathsf{D}}^{F_0}_H$	$e\ (d \circ (\mathsf{M}^{F_0}(\vec{d_0})\ e\ f))$

Figure 2. Definition of efficient folds

Example 24 (Efolds for Powerlists, Typing) Recall that the general typing rule for efold was

$$\frac{\Gamma \vdash \vec{d} : \vec{\mathsf{D}}^F_H \qquad \Gamma \vdash s : \mathsf{F}^F_H\, G \subseteq^{\mathsf{k1}} G}{\Gamma \vdash \mathsf{efold}^F(\vec{d}, s) : \mu^{\mathsf{k1}} F \leq^{\mathsf{k1}}_H G}$$

where we took the freedom to omit the kinding judgements of F, G and H for conciseness. The *typing* of an efficient fold for a concrete hofunctor F requires only the recursively computed F^F_H and $\vec{\mathsf{D}}^F_H$. For powerlists, we obtain the following instances.

$$\begin{aligned}
\mathsf{PListF} &= \lambda X \lambda A.\ A + X\,(A \times A) \\
\mathsf{F}_H^{\mathsf{PListF}} &= \lambda X \lambda A.\ H\,A + X\,(A \times A) \\
\vec{\mathsf{D}}_H^{\mathsf{PListF}} &= (\forall X \forall A.\ H\,A \times H\,A \to H\,(A \times A))
\end{aligned}$$

Instantiating the general rule for efficient folds and expanding the definitions of $\subseteq^{\mathsf{k1}}$ and $\leq_H^{\mathsf{k1}}$, we obtain efficient folds for powerlists.

$$\frac{\begin{array}{l}\Gamma \vdash d : \forall A.\ H\,A \times H\,A \to H\,(A \times A) \\ \Gamma \vdash s : \forall A.\ H\ A + G\ (A \times A) \to G\ A\end{array}}{\Gamma \vdash \mathsf{efold}^{\mathsf{PListF}}(d, s)\ :\ \forall A \forall B.\ (A \to H\,B) \to \mathsf{PList}\ A \to G\,B}$$

We will refer to d as *distributivity term* for reasons its type makes apparent: d witnesses that the product constructor $\times$ distributes over constructor H.

Example 25 (Summing up a Powerlist, Typing) We can define function sum' of Example 18 using efficient folds for powerlists. We set $G := H := \lambda\_.\mathsf{Nat}$, as in the previous implementations of sum', and

$$\begin{array}{lll}
\mathsf{sum}' & := \mathsf{efold}^{\mathsf{PListF}}(d, s) & : \forall A.\ (A \to \mathsf{Nat}) \to \mathsf{PList}\ \ A \to \mathsf{Nat}, \quad \text{where} \\
d & := \lambda\langle n, m\rangle.\ n + m & : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}, \quad \text{and} \\
s & := \lambda x.\, \mathsf{case}\,(x,\ n.\,n,\ n.\,n) & : \mathsf{Nat} + \mathsf{Nat} \to \mathsf{Nat}.
\end{array}$$

The given implementation is type-correct, we will verify the reduction behavior later.

As in the implementation using $\mathsf{GIt}^{\mathsf{k1}}$ in Example 18, the task of the step term s is trivial whereas the addition happens in the other term. Back then, we could use $F' := \lambda X \lambda A.\ \mathsf{Nat} + X\ (Q\ A)$ with Q any constructor in $\mathsf{k1}$. With the special choice $F' := \mathsf{F}^{\mathsf{PListF}}_{\lambda\_.\mathsf{Nat}}$, which the given definition of efficient folds takes, Q is fixed to $\lambda A.\ A \times A$.

Example 26 (Efolds for Powerlists, Reduction) Setting $m_{\mathsf{Id}} := \lambda f \lambda x.\, f\, x$ for the canonical monotonicity witness of $\mathsf{Id} : \mathsf{k1}$, we can compute the pseudo monotonicity witness for efficient powerlist folds according to Figure 2.

$$
\begin{array}{lcl}
d & : & \forall A.\ H\,A \times H\,A \to H\,(A \times A) \\
\mathsf{M}^{\mathsf{PListF}}(d) & : & \forall X \forall Y.\ X \leq_H^{\mathsf{k1}} Y \to \mathsf{PListF}\ X \leq_H^{\mathsf{k1}} \mathsf{F}_H^{\mathsf{PListF}}\,Y \\
& = & \forall X \forall Y.\ X \leq_H^{\mathsf{k1}} Y \to \forall A \forall B.\ (A \to H\,B) \to \\
& & \qquad A + X\,(A \times A) \to H\,B + Y\,(B \times B) \\
\mathsf{M}^{\mathsf{PListF}}(d)\ e\ f & := & \mathsf{either}\ (m_{\mathsf{Id}}\,f)\ (e\ (d \circ (\mathsf{pair}\ (m_{\mathsf{Id}}\,f)\ (m_{\mathsf{Id}}\,f)))) \\
& \longrightarrow^+ & \mathsf{either}\ (\lambda x.\, f\,x)\ (e\ (d \circ (\mathsf{fork}^\circ\ f)))
\end{array}
$$

Note that fork is defined in terms of pair° on page 45. Using this pseudo monotonicity witness in the general reduction rule for efolds, which was

$$
\mathsf{efold}^F(\vec{d}, s)\ f\ (\mathsf{in}^{\mathsf{k1}}\ t) \longrightarrow_\beta s\ (\mathsf{M}^F(\vec{d})\ \mathsf{efold}^F(\vec{d}, s)\ f\ t),
$$

we get the following reduction behavior for powerlist efolds:

$$
\begin{array}{rcl}
\mathsf{efold}^{\mathsf{PListF}}(d, s)\ f\ (\mathsf{zero}\ \ a) & \longrightarrow^+ & s\ (\mathsf{inl}\ (f\ \ a)) \\
& = & s\ (\mathsf{zero}^-\ (f\ \ a)) \\
\mathsf{efold}^{\mathsf{PListF}}(d, s)\ f\ (\mathsf{succ}\ \ l) & \longrightarrow^+ & s\ (\mathsf{inr}\ (\mathsf{efold}^{\mathsf{PListF}}(d, s)\ (d \circ (\mathsf{fork}^\circ\ f))\ l)) \\
& = & s\ (\mathsf{succ}^-\ (\mathsf{efold}^{\mathsf{PListF}}(d, s)\ (d \circ (\mathsf{fork}^\circ\ f))\ l))
\end{array}
$$

Example 27 (Summing up a Powerlist, Reduction) Instantiating the above scheme for

$$
\begin{array}{lll}
\mathsf{sum}' := \mathsf{efold}^{\mathsf{PListF}}(d, s), & & \text{where} \\
d \quad := \lambda \langle n, m \rangle.\ n + m, & & \text{and} \\
s \quad := \lambda x.\, \mathsf{case}\,(x,\ n.\, n,\ n.\, n), & &
\end{array}
$$

we obtain the precise reduction behavior of Example 18.

Example 28 (Efolds for de Bruijn Terms) As shown in Example 23, the generating type constructor LamF for de Bruijn terms is a hofunctor. Hence, we can calculate $\mathsf{F}_H^{\mathsf{LamF}}$ and $\vec{\mathsf{D}}_H^{\mathsf{LamF}}$ according to Figure 2.

$$
\begin{array}{lcl}
\mathsf{LamF} & = & \lambda X \lambda A.\ A + (X\ A \times X\ A + X\,(1 + A)) \\
\mathsf{F}_H^{\mathsf{LamF}} & = & \lambda X \lambda A.\ H\,A + (X\ A \times X\ A + X\,(1 + A)) \\
\vec{\mathsf{D}}_H^{\mathsf{LamF}} & = & (\forall A.\ H\,A \to H\,A, \\
& & \ \forall A.\ H\,A \to H\,A, \\
& & \ \forall X \forall A.\ 1 + H\,A \to H\,(1 + A))
\end{array}
$$

The first two components of $\vec{\mathsf{D}}_H^{\mathsf{LamF}}$ arise from the two homogeneous applications $X\ A$ in LamF, the third from the heterogeneous application $X\,(1 + A)$.

The efficient fold for de Bruijn terms is typed as follows:

$$\frac{\begin{array}{l}\Gamma \vdash d_1 : \forall A.\ H\,A \to H\,A \\ \Gamma \vdash d_2 : \forall A.\ H\,A \to H\,A \\ \Gamma \vdash d_3 : \forall A.\ 1 + H\,A \to H\,(1 + A) \\ \Gamma \vdash s\ \ : \forall A.\ H\,A + (G\,A \times G\,A + G\,(1 + A)) \to G\,A\end{array}}{\Gamma \vdash \mathsf{efold}^{\mathsf{LamF}}(d_1, d_2, d_3, s)\ :\ \forall A \forall B.\ (A \to H\,B) \to \mathsf{Lam}\ A \to G\,B}$$

Recall that maybe is the canonical monotonicity witness for $\lambda A.\ 1 + A$. The pseudo monotonicity witness $\mathsf{M}^{\mathsf{LamF}}$ is computed as

$$\begin{aligned}\mathsf{M}^{\mathsf{LamF}}(d_1, d_2, d_3)\ e\ f = {} & \mathsf{either}\ (m_{\mathsf{Id}}\ f) \\ & \quad (\mathsf{either}\ (\mathsf{pair}\ (e\ (d_1 \circ f))\ (e\ (d_2 \circ f))) \\ & \qquad (e\ (d_3 \circ (\mathsf{maybe}\ f)))).\end{aligned}$$

Setting $e := \mathsf{efold}^{\mathsf{LamF}}(d_1, d_2, d_3, s)$ this yields the following reduction behavior for this special efficient fold.

$$\begin{aligned}e\ f\ (\mathsf{var}\ \ a) & \longrightarrow^+ s\ (\mathsf{inl}\ (f\,a)) \\ & = s\ (\mathsf{var}^-\ (f\,a)) \\ e\ f\ (\mathsf{app}\ \ t_1\ t_2) & \longrightarrow^+ s\ (\mathsf{inr}\ (\mathsf{inl}\ \langle e\ (d_1 \circ f)\ t_1,\ e\ (d_2 \circ f)\ t_2\rangle)) \\ & = s\ (\mathsf{app}^-\ (e\ (d_1 \circ f)\ t_1)\ (e\ (d_2 \circ f)\ t_2)) \\ e\ f\ (\mathsf{abs}\ \ r) & \longrightarrow^+ s\ (\mathsf{inr}\ (\mathsf{inr}\ (e\ (d_3 \circ (\mathsf{maybe}\,f))\ r))) \\ & = s\ (\mathsf{abs}^-\ (e\ (d_3 \circ (\mathsf{maybe}\,f))\ r))\end{aligned}$$

Example 29 (Renaming and Substitution for de Bruijn Terms)
The functions lam of Example 16 and subst of Example 19 can be expressed with efficient folds as

$$\begin{aligned}\mathsf{lam} & := \mathsf{efold}^{\mathsf{LamF}}(\mathsf{id}, \mathsf{id}, \mathsf{id}, \mathsf{in}^{\mathsf{k1}})\ \ : \mathsf{Lam} \leq^{\mathsf{k1}}_{\mathsf{Id}} \mathsf{Lam} \\ \mathsf{subst} & := \mathsf{efold}^{\mathsf{LamF}}(\mathsf{id}, \mathsf{id}, \mathsf{lift}\ \mathsf{id}, s) : \mathsf{Lam} \leq^{\mathsf{k1}}_{\mathsf{Lam}} \mathsf{Lam},\end{aligned}$$

where we use lift and s from Example 19.

Since composition with the identity η-expands terms, the reduction behavior of lam is not exactly as in Example 16. 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.

A second look at efficient folds. As we pointed out before, Figure 2 describes just one possible definition of efficient folds. One might wonder whether it could not be simplified at bit. The first case worth a discussion is (b) identity: Is it really necessary to emit a distributivity witness $d : H \subseteq^{\mathsf{k1}} H$ here? This question has been raised already by Bird and Paterson (1999a, Sec. 4.1) for their version of generalized folds. In the last example, these terms are just instantiated with the identity id. So supposedly, they could be dropped, leading to the simpler definition

$$\mathsf{M}^{\lambda X.X}()\ e\ f := e\ f\ .$$

Another questionable clause is (e) composition. As mentioned in the beginning of this section, clause (e) can be iterated with $Q := \mathsf{Id}$ in the proof of hofunctoriality for a type constructor F. This means that one can also obtain an arbitrary number of different definitions of an efficient fold for such an F. Each iteration of the rule would emit another distributivity term d. We therefore suggest different definition clauses for case (e):

$$\begin{array}{lcl}F & = & \lambda X.\ Q \circ (F_0\,X) \\ \mathsf{F}^F_H & := & \lambda X.\ Q \circ (\mathsf{F}^{F_0}_H\ X) \\ \vec{\mathsf{D}}^F_H & := & \vec{\mathsf{D}}^{F_0}_H \\ \mathsf{M}^F(\vec{d})\ e\ f & := & m_Q\ (\mathsf{M}^{F_0}(\vec{d})\ e\ f)\end{array}$$

In contrast to the original definition, this variant defines F^F_H recursively through $\mathsf{F}^{F_0}_H$. Iteration of these clauses with $Q = \mathsf{Id}$ do now neither change the typing rule for the efficient fold nor the reduction behavior of the pseudo monotonicity witness M^F. Now, the only case where the need of a distributivity term d arises is (f) nesting. This means that for a homogeneous hofunctor F and $H = \mathsf{Id}$, it holds that $\mathsf{F}^F_H = F$, $\vec{\mathsf{D}}^F_H$ is empty, M^F is the canonical monotonicity witness of F and the eliminator efold^F is identical to $\mathsf{It}^{\mathsf{k1}}_{=}$.

Comparison with Bird and Paterson (1999a). Whilst for a fixed ho-functor $F : \mathsf{k2}$, the efficient fold "e" is of type $\mu^{\mathsf{k1}} F \leq^{\mathsf{k1}}_{H} G$, Bird and Paterson define a generalized fold "g" of type $(\mu^{\mathsf{k1}} F) \circ H \subseteq^{\mathsf{k1}} G$. As observed by Martin, Gibbons, and Bayley (2003), both kinds of folds are interdefinable, extensionally:

$$g = e\ \mathsf{id}$$
$$e = \lambda f.\ g \circ (m_{\mu F}\ f)$$

where $m_{\mu F}$ is the canonical monotonicity witness of $\mu^{\mathsf{k1}} F$. These equations explain why e is called "efficient": it combines two traversals of a datastructure, a fold and a map, into a single traversal.

One might wonder whether generalized folds can also be expressed in System GMIt^{ω}. Recall the reduction rule for efficient folds:

$$e\ f\ (\mathsf{in}^{\mathsf{k1}}\ t) \ \longrightarrow_{\beta}\ s\ (\mathsf{M}^{F}(\vec{d})\ e\ f\ t)$$

If we could alter the definition of M^{F} in such a way that in the resulting term $\mathsf{M}^{F}(\vec{d})\ e\ f\ t$ the variable e occurred only in the form $e\ \mathsf{id}$, then by setting $f = \mathsf{id}$ we would obtain a reduction rule for g which is simulated in System GMIt^{ω}. The necessary changes affect certainly clause (f) with $F = \mathsf{F}^{F}_{H} = \lambda X.\ X \circ (F_0\, X)$ of the definition of M^{F}, which we recall in a somewhat sketchy form as follows:

$$\begin{array}{lrcl}
\text{Provided} & e & : & \forall A \forall B.\ (A \to H\, B) \to X\, A \to Y\, B \\
\text{and} & f & : & A \to H\, B, \\
\mathsf{M}^{F}(\ldots)\ e\ f & & : & F\, X\, A \to \mathsf{F}^{F}_{H}\, Y\, B \\
 & & = & X\, (F_0\, X\, A) \to Y\, (F_0\, Y\, B) \\
\mathsf{M}^{F}(\ldots)\ e\ f & & := & e\, f', \\
\text{where} & f' & : & F_0\, X\, A \to H\, (F_0\, Y\, B) \\
 & f' & := & \ldots
\end{array}$$

In order to obtain the reduction behavior of Bird and Paterson (1999a) we need to change the definition of $\mathsf{M}^{F}(\ldots)\, e\, f$ to $e\ \mathsf{id} \circ (m_{\mu F}\ f')$. But this is not well-typed. Typing requires monotonicity m_X for the abstract type constructor X instead of $m_{\mu F}$.

To summarize this discussion, we might say that Bird and Paterson's generalized folds are not an instance of GMIt^{ω} due to their inefficient reduction behavior. Whether they can be simulated in System F^{ω} in a different way, remains an open question.

10 Related and Future Work

A discussion of related work on iteration schemes for higher-order datatypes can be found in the previous section. This section tries to develop a broader perspective—especially with respect to possible applications in the field of generic programming.

Generic Programming (also called polytypic programming) aims at programming functions operating "canonically" on all the datatypes associated with a class of admissible F's, which are typically the regular datatypes. For an extensive overview of generic programming, see Backhouse, Jansson, Jeuring, and Meertens (1999). The tutorial also includes a description of the generic programming language PolyP (Jansson and Jeuring, 1997). Typically for generic programming, as well as for Jay's Constructor Calculus (Jay, 2001), admissible F's are built in a combinatorial, i. e., λ-free calculus.[3] Polytypic functions are then constructed by recursion on the generation of their type parameter F. In contrast, our constructions of fixed points and the associated schemes of iteration and coiteration just assume some *arbitrary* type constructor $F : \kappa \to \kappa$. In this respect, we follow the approach in category theory where an arbitrary endofunctor on some category would be given (for the definition of initial algebras and final coalgebras—not for existence theorems). There is no analysis of the form of F, and thus, our constructions have to work uniformly in F. Unlike the category-theoretic situation, we do not even impose any equational laws on F. In the conventional-style systems, the *usage*—as opposed to the existence—of the schemes rests on terms inhabiting one of our notions of monotonicity. In $\mathsf{It}^{\omega}_{=}$, there would be a canonical choice of a witness of monotonicity for a wide range of type constructors F, including, for instance, the hofunctors of Section 9. The canonical monotonicity witness could be computed by recursion on the structure of all these admissible F's. Note, however, that "ungeneric" powerlist reversal (Example 17) uses some monotonicity witness that cannot be found generically.

Type Classes. Norell and Jansson (2003) describe an implementation of the polytypic programming language PolyP within the Haskell programming language, using the type-class mechanism (Wadler and Blott, 1989). The latter is a form of ad-hoc polymorphism where a class name is associated with a number of functions, called the dictionary, whose types may involve the type parameter of the class. A type becomes a member of the class by providing

[3] This even holds for the related work on nested datatypes we mentioned in Section 9.

implementations of the dictionary functions. Most importantly, the type system allows to provide an implementation for type $H\,A$ under the assumption that already A belongs to the class, and hence by using the assumed implementations of the dictionary functions for A.

Summing Up with Type Classes. Our running example of powerlist summation may be recast in the framework of type classes by defining a type class $\mathsf{Summable}$ so that type A belongs to it iff there is an implementation of the function $\mathsf{sum} : A \to \mathsf{Nat}$. (For a Haskell implementation, see below.) Trivially, Nat is summable, and if A is summable then so is $A \times A$. The crucial step is to show that summability of A entails that of $\mathsf{PList}\,A$. The argument $f : A \to \mathsf{Nat}$ to sum' in Example 8 is no longer needed because one can just take sum for type A. On the other hand, the freedom to manipulate f is also lost, and no function $\lambda\langle a_1, a_2\rangle.\ f\,a_1 + f\,a_2$ can be given as an additional argument. Fortunately, $\lambda\langle a_1, a_2\rangle.\ \mathsf{sum}\ a_1 + \mathsf{sum}\ a_2$ is precisely the function sum for type $A \times A$. Finally, sum at type $\mathsf{PList}\,\mathsf{Nat}$ is the function we were after in the first place. Certainly, its termination is not guaranteed by this construction, but intuitively holds, anyway. This is more delicate with summation for bushes (Example 13). In terms of type class $\mathsf{Summable}$ it just requires that for summable A, also $\mathsf{Bush}\,A$ is summable. The crucial definition clause is then $\mathsf{sum}\,(\mathsf{bcons}\,a\,b) := \mathsf{sum}\,a + \mathsf{sum}\,b$. The first summand uses the assumed function sum for type A (which used to be f in that example), the second one uses polymorphic recursion: the term b is of type $\mathsf{Bush}\,(\mathsf{Bush}\,A)$, hence the same definition of sum is invoked with $\mathsf{Bush}\,A$ in place of A. Its hidden argument f is therefore $\mathsf{sum}\,f$, in accordance with the reduction behavior shown in Example 13. Again, no termination guarantee is provided by this implementation. Moreover, bsum' in the example works for arbitrary types A as soon as a function $f : A \to \mathsf{Nat}$ is provided. This includes different functions for the very same type A, not just the one derived by the type class instantiation mechanism. For instance, the first argument—$\mathsf{bsum}' f$—in the recursive call may be modified to, e. g., $\mathsf{bsum}'\,(\lambda x.\, f\,x + 1)$, keeping typability and thus termination. Using type class $\mathsf{Summable}$, there is just no room for such an "ungeneric" modification, as is clear from the explanation above.

The following Haskell code corresponds to the above discussion and can be executed with current extensions to the Haskell 98 standard. These extensions are only needed because we instantiate `Summable(a,a)` with two occurrences of `a`.

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

class Summable a where
  sum:: a -> Integer
```

```
instance Summable Integer where
  sum = id

instance Summable a => Summable (a,a) where
  sum (a1,a2) = sum a1 + sum a2

instance Summable a => Summable (PList a) where
  sum (Zero a) = sum a
  sum (Succ l) = sum l

instance Summable a => Summable (Bush a) where
  sum Nil = 0
  sum (Cons a b) = sum a + sum b
```

Generic Haskell (Clarke, Jeuring, and Löh, 2002) is a system of generic programming in all kinds: A family of functions may be programmed where the indices range over all type constructors F *of all kinds*. The type $\tau(F)$ of the function indexed by F is calculated from the kind of F, hence has a polykinded type (Hinze, 2002). The idea of this calculation roughly follows the idea of the type-class mechanism, e. g., the function associated with $\mathsf{PList} : * \to *$ takes any function for any type A and yields a function for the type $\mathsf{PList}\,A$, i. e., $\tau(\mathsf{PList}) = \forall A.\, \tau(A) \to \tau(\mathsf{PList}\,A)$. Therefore, only the types $\tau(A)$ for A a veritable type (type constructor of kind $*$, also called a manifest type) can be freely chosen.

Clearly, the iteration schemes of this article do not follow that discipline: By no means is an iterator for $\mu^{\kappa_1 \to \kappa_2} F$ explained in terms of an iterator for $\mu^{\kappa_2}(F\,G)$ for some or any $G : \kappa_1$. However, programming *iterators* inside Generic Haskell would counteract its philosophy. In fact, Generic Haskell leaves the programmer from the burden to consider the fixed points that come from the definitions of datatypes. The associated instances are automatically generated by the compiler—without any clause in the generic program referring to them. Likewise, type abstraction and type instantiation are dealt with automatically (Hinze, Jeuring, and Löh, 2003), using a model of the kinded type constructor system based on applicative structures. [4] The most recent

[4] Intensional type analysis (Harper and Morrisett, 1995) is a compilation technology which uses an intermediate language with "intensional polymorphism", i. e., with structural recursion over types. In the extension to all kinds that has been motivated by Generic Haskell, Weirich (2002) directly encodes the notion of set-theoretic model for the lambda calculus, including environments, in order to describe the instantiation mechanism. This might also help in understanding the output of the Generic Haskell compiler.

presentation of Generic Haskell is given by Hinze and Jeuring (2003b); there is also a collection containing the three most interesting examples (Hinze and Jeuring, 2003a), among which the generalization of the trie data structure to datatypes of all kinds (Hinze, 2000b). The tries over a regular datatype are often already truly nested datatypes, hence the latter ones arise naturally also in this context. Since the merging functions for tries are recursive in two arguments (Hinze and Jeuring, 2003a, Sect. 2.7), we would need to extend our iteration schemes in order to cover them, too.

As becomes clear from all the examples cited above, there is usually no need to have any kind of recursive calls inside the programs in Generic Haskell. Therefore, a useful version of Generic Haskell could well be imagined that only uses program clauses taken from System F^ω. We would hope to extend our Mendler-style iteration schemes in order to be able to provide a syntactic analysis of those restricted Generic Haskell programs that allows to conclude that any instantiated function (as being generated by the Generic Haskell compiler) terminates as well, as long as only fixed points of monotone type constructors are formed. Here, we would use Mendler-style systems since they directly allow to express the algorithms. The step term s of the iterator would be provided by a modification of the compiler for generic programs. In the other direction, we might get help from generic programming in building libraries of pseudo monotonicity witnesses for System GIt^ω, hence with a generic definition for every specific task at hand; recall that real work is also delegated to the witnesses in that system. In general, our conventional schemes would typically be used with some generically found monotonicity witness.

Dependent Types. Let us comment on systems with dependent types (i. e., with types that depend on terms) and universes. These are very rich systems in which higher-order datatypes can easily be expressed. Impredicative dependent type theories, like the Calculus of Constructions, encompass System F^ω, hence our schemes. Interestingly, the native fixed-point constructions of all of these theories, especially the systems Coq and LEGO, exclude non-strictly positive type constructors. But non-strictly positive dependencies immediately arise with Kan extensions. For our intended extensions to systems with primitive recursion, one would have to require non-strictly positive "native" fixed points in the system. On the other hand, there is plenty of work on predicative systems of dependent types. One would use small universes as the system of admissible type indices of the families in question. Hence, one gives up the uniform treatment of all possible indices, see Altenkirch and McBride (2003). The operational behavior of the datatypes thus obtained has to be studied further. Interestingly, a nontrivial part of programming with dependent types can be simulated within Haskell (McBride, 2002; Chen et al., 2003).

Type Checking and Type Inference. This article deliberately neglects the important practical problem of finding the types which are given to the example programs throughout the article. It is well-known that already type-checking for Curry-style System F is undecidable (Wells, 1999). Nevertheless, we have chosen the unannotated Curry-style due to its succintness. The type annotations are only given on the informal level of presentation. With these, the terms in Church-style formulation, hence with explicit type abstraction, type instantiation and type annotations for every bound variable have successfully been reconstructed with a prototype implementation[5], at least for the examples that have been tried, and these were the majority of the programming examples in the article. Note that, in systems of dependent types, termination of well-typed programs would be a necessity for type checking, since the types may depend on the terms. Our systems are layered, hence these problems are not intertwined.

The problem of type inference is deeper than that of type checking—already polymorphic recursion, i. e., recursion where different instances of a universally quantified target type have to be inferred, makes type inference undecidable (Henglein, 1993; Kfoury, Tiuryn, and Urzyczyn, 1993a,b). Type abstraction in Haskell is only partly solved in Neubauer and Thiemann (2002) by providing a restricted amount of lambda expressions. The problem is also known for the programming language family ML as the "quest for type inference with first-class polymorphic types" (Le Botlan and Rémy, 2003). A practical system would certainly allow the user to communicate her typing intuitions. In this respect, Haskell is half way: help with types is accepted, but not with kinds.

On "higher-order nested". In this article, "higher-order nested" means that fixed-points of higher ranks are formed and that recursive calls are heterogeneous. "True nesting" means nested calls to the datatype, as in Example 20, where the least fixed-point of $\lambda X \lambda A. \ldots + X\,(X\,A)$ is considered. This datatype would be called "doubly nested" in Bird and Paterson (1999b), and in general, true nesting is called "non-linear nests" in Hinze (2001). Okasaki (1999a) considers the fixed-point (called `square_`) of

$$\lambda F \lambda V \lambda W.\, V\,(V\,A) + F\,V\,(W \times^{\mathsf{k1}} W) + F\,(V \times^{\mathsf{k1}} W)\,(W \times^{\mathsf{k1}} W),$$

with $V, W : \mathsf{k1}$. The type constructor V is even nested, but it is just a parameter which is used heterogeneously. Nevertheless, this would be called a higher-order nest by Hinze (1998), regardless of the component $V\,(V\,A)$, but because the higher-order parameters V and W (which are not types but type transformers) are given as arguments to the variable F, representing the fixed point. Hinze (2001) contains plenty of examples where higher-order parameters are varied in the recursive calls to the datatype being defined, but nowhere

[5] `Fomega 0.10 alpha`, by the first author, available on his homepage

"true nesting". Truly nested datatypes may seem to be esoteric, but they occur naturally in the representation of explicit substitution (see Example 20), a fact which might explain why termination questions in connection with explicit substitutions are notoriously difficult. Certainly, we would like to see more natural examples of true nesting. As indicated above, trie data structures (Hinze, 2000b) serve this purpose.

Extensional Equality. Finally, an important subject for future research would be the study of the equational laws for our proposed iteration and coiteration schemes in order to use the mathematics of program calculation for the verification of programs expressed by them. After all, this is seen as the major benefit of a programming discipline with iterators in a setting of partial functions, e. g., the "algebra of programming" (Bird and de Moor, 1997). Note that these calculations would always be carried out within an extensional framework, such as parametric equality theory. The goal would be to demonstrate that a given program denotes some specified element in some semantics which, e. g., could be total functions. This article views a program as an algorithm which is explored in its behavior, e.g., whether it is strongly normalizing as a term rewrite system. Parametricity would, e. g., be used for establishing that our syntactic natural transformations would also be natural in the category-theoretic sense. For an introduction to these ideas, see Wadler (1989), more details are to be found in Wadler (2003), and interleaved positive (co)inductive types are treated in Altenkirch (1999). An interesting new field is the connection between parametric equality and generic programming: Backhouse and Hoogendijk (2003) show that, under reasonable naturality and functoriality assumptions on the family of zip functions, exactly one such zip exists for all "regular relators".

11 Conclusion

We have put forth and compared the expressive power of several possible formulations of iteration and coiteration for (co)inductive constructors of higher kinds. All of them have a clear logical underpinning (exploiting the well-known Curry-Howard isomorphism) and uniformly extend from rank-2 type constructors to rank-n type constructors, for arbitrary n.

The main technical problem we faced with the formulation of (co)iteration schemes, is the absence of a canonical definition of admissible constructor for forming the least/greatest fixed-points. In our approach, *every* constructor is allowed for the formation of the fixed-points. For conventional-style schemes (inspired from initial algebras in category theory), the search for an optimal type-based approach led to several plausible notions of monotonicity, which is required when applying the (co)iteration scheme in the systems $\underline{\mathsf{It}}^\omega$, $\mathsf{It}^\omega_=$ and GIt^ω, respectively. In the even more radical line of thought (inspired by Mendler's work, but not derived from it), *every* constructor is allowed for the (co)iteration, but the typing requirements for its step term nevertheless guarantee termination for our Mendler-style systems MIt^ω, $\mathsf{MIt}^\omega_=$ and GMIt^ω.

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, Gibbons, and Bayley (2003) are very straightforwardly defined in GIt^ω. But for many applications, the more basic $\underline{\mathsf{It}}^\omega$ and MIt^ω are perfectly sufficient and invoking GMIt^ω or GIt^ω is simply not necessary. There are many interesting cases where the more basic systems $\underline{\mathsf{It}}^\omega$ and MIt^ω 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 $\mathsf{MIt}^\omega_=$ and $\mathsf{It}^\omega_=$, 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 MIt^ω, where the iterator and coiterator are Mendler-style, their computational behavior is very close to `letrec`, except that termination of computations is guaranteed. Thus, through type checking, these systems provide termination checking for given algorithmic ideas. The conventional-style systems aid more in *finding* algorithms: According to the type of the function to be programmed, one chooses the generic (pseudo) monotonicity witness m and tries to find a term s of the right type. Certainly, this "type-directed programming" might fail, see Remark 1, but has proven its usefulness in many cases.

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

Acknowledgments Peter Hancock deserves thanks for suggesting to us the definition $\mathsf{mon}\ F := F \leq F$. We also highly appreciate the feedback we got from Peter Aczel at TYPES'03 in Turin, April/May 2003. One quarter of this article can be traced back to the stimulating advice by our anonymous referees whom we warmly thank.

The first author acknowledges the support by both the PhD Programme Logic in Computer Science (GKLI) of the *Deutsche Forschungs-Gemeinschaft* and the project CoVer of the Swedish Foundation of Strategic Research. 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.

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., 1999. Logical relations and inductive/coinductive types. In: Gottlob, G., Grandjean, E., Seyr, K. (Eds.), Computer Science Logic, 12th International Workshop, Brno, Czech Republic, August 24–28, 1998, Proceedings. Vol. 1584 of Lecture Notes in Computer Science. pp. 343–354.

Altenkirch, T., McBride, C., 2003. Generic programming within dependently typed programming. In: Gibbons, J., Jeuring, J. (Eds.), Proceedings of the IFIP TC2 Working Conference on Generic Programming, Dagstuhl, Germany, July 2002. Kluwer Academic Publishers, pp. 1–20.

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.

Backhouse, R., Hoogendijk, P., 2003. Generic properties of datatypes. In: Backhouse, R., Gibbons, J. (Eds.), Generic Programming. Vol. 2793 of Lecture Notes in Computer Science. pp. 97–132.

Backhouse, R., Jansson, P., Jeuring, J., Meertens, L., 1999. Generic programming—an introduction. In: Swierstra, S. D., Henriques, P. R., Oliveira, J. N. (Eds.), Advanced Functional Programming. Vol. 1608 of Lecture Notes in Computer Science. pp. 28–115.

Bird, R., de Moor, O., 1997. Algebra of Programming. Vol. 100 of International Series in Computer Science. Prentice Hall.

Bird, R., Gibbons, J., Jones, G., 2000. Program optimisation, naturally. In: Millenial Perspectives in Computer Science. Palgrave.

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.

Bird, R. S., Paterson, R., 1999b. De Bruijn notation as a nested datatype. Journal of Functional Programming 9 (1), 77–91.

Chen, C., Zhu, D., Xi, H., 2003. Implementing cut elimination: A case study of simulating dependent types in Haskell, submitted.

Clarke, D., Jeuring, J., Löh, A., 2002. The Generic Haskell user's guide—version 1.23. Technical Report UU-CS-2002-047, Institute of Information and Computing Sciences, Utrecht University.

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.

Harper, R., Morrisett, G., January 1995. Compiling polymorphism using intensional type analysis. In: Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California. ACM Press, pp. 130–141.

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

Henglein, F., 1993. Type inference with polymorphic recursion. ACM Transactions on Programming Languages and Systems (TOPLAS) 15 (2), 253–289.

Hinze, R., December 1998. Numerical representations as higher-order nested datatypes. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn.

Hinze, R., December 1999. Polytypic values possess polykinded types. Technical Report IAI-TR-99-15, Institut für Informatik III, Universität Bonn.

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

Hinze, R., July 2000b. Generalizing generalized tries. Journal of Functional Programming 10 (4), 327–351.

Hinze, R., September 2001. Manufacturing datatypes. Journal of Functional Programming 11 (5), 493–524.

Hinze, R., 2002. Polytypic values possess polykinded types. Science of Computer Programming, MPC Special Issue 43, 129–159.

Hinze, R., Jeuring, J., 2003a. Generic Haskell: applications. In: Backhouse, R., Gibbons, J. (Eds.), Generic Programming. Vol. 2793 of Lecture Notes in Computer Science. Springer, pp. 57–97.

Hinze, R., Jeuring, J., 2003b. Generic Haskell: practice and theory. In: Backhouse, R., Gibbons, J. (Eds.), Generic Programming. Vol. 2793 of Lecture Notes in Computer Science. pp. 1–56.

Hinze, R., Jeuring, J., Löh, A., 2003. Type-indexed data types. To appear in Science of Computer Programming .

Jansson, P., Jeuring, J., 1997. PolyP—a polytypic programming language ex-

tension. In: Conference Record of POPL'97. ACM Press, pp. 470–482.
Jay, C. B., 2001. Distinguishing data structures and functions: The constructor calculus and functorial types. In: Abramsky, S. (Ed.), Typed Lambda-Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings. Vol. 2044 of Lecture Notes in Computer Science. Springer, pp. 217–239.
Kahrs, S., July 2001. Red-black trees with types. Journal of Functional Programming 11 (4), 425–432.
Kfoury, A., Tiuryn, J., Urzyczyn, P., 1993a. Type reconstruction in the presence of polymorphic recursion. ACM Transactions on Programming Languages and Systems (TOPLAS) 15 (2), 290–311.
Kfoury, A., Tiuryn, J., Urzyczyn, P., 1993b. The undecidability of the semi-unification problem. Information and Computation 102 (1), 83–101.
Le Botlan, D., Rémy, D., 2003. MLF: raising ML to the power of system F. In: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming, Uppsala, Sweden: 25-29 August 2003. ACM Press, pp. 27–38.
Mac Lane, S., 1998. Categories for the Working Mathematician, 2nd Edition. Vol. 5 of Graduate Texts in Mathematics.
Martin, C., Gibbons, J., Bayley, I., 2003. Disciplined, efficient, generalised folds for nested datatypes. Accepted for publication in Formal Aspects of Computing .
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.
McBride, C., 2002. Faking it (simulating dependent types in Haskell). Journal of Functional Programming 12 (4+5), 375–392.
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. 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.
Neubauer, M., Thiemann, P., 2002. Type classes with more higher-order polymorphism. In: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. ACM Press, pp. 179–190.
Norell, U., Jansson, P., 2003. Polytypic programming in Haskell. In: Proceedings of the 15th International Workshop on the Implementation of Functional Languages, Edinburgh, Scotland September 8th - 10th, 2003. Lecture Notes in Computer Science. Accepted for publication.
Okasaki, C., 1999a. From fast exponentiation to square matrices: An adventure in types. In: Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September

27-29, 1999. Vol. 34 of SIGPLAN Notices. ACM, pp. 28–35.

Okasaki, C., 1999b. Red-black trees in a functional setting. Journal of Functional Programming 9 (4), 471–477.

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.

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.

Wadler, P., 1989. Theorems for free! In: Proceedings of the fourth international conference on functional programming languages and computer architecture, Imperial College, London, England, September 1989. ACM Press, pp. 347–359.

Wadler, P., 2003. The Girard-Reynolds isomorphism. Information and Computation 186 (2), 260–284.

Wadler, P., Blott, S., January 1989. How to make ad-hoc polymorphism less ad hoc. In: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas. ACM Press, pp. 60–76.

Weirich, S., 2002. Higher-order intensional type analysis. In: Métayer, D. L. (Ed.), Proceedings of 11th European Symposium on Programming, ESOP 2002. Vol. 2305 of Lecture Notes in Computer Science. pp. 98–114.

Wells, J. B., 1999. Typability and type checking in system F are equivalent and undecidable. Annals of Pure and Applied Logic 98 (1–3), 111–156.