Iteration and coiteration schemes for higher-order and nested datatypes

Andreas Abel\textsuperscript{a,}\textsuperscript{*}, Ralph Matthes\textsuperscript{b}, Tarmo Uustalu\textsuperscript{c}

\textsuperscript{a}Department of Computer Science, Chalmers University of Technology, Rännvägen 6, SE-41296 Göteborg, Sweden
\textsuperscript{b}Institut für Informatik der Ludwig-Maximilians-Universität München, Oettingenstraße 67, D-80538 München, Germany
\textsuperscript{c}Institute of Cybernetics, Tallinn University of Technology, 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 $\text{F}^\omega$ 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 $\text{GMIt}^{\omega}$ and $\text{GIt}^{\omega}$ 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 $\text{MIt}^{\omega}$ into System $\text{F}^\omega$.

© 2004 Elsevier B.V. All rights reserved.

Keywords: Higher-order datatypes; Generalized folds; Efficient folds; Iteration; Coiteration; System $\text{F}^\omega$; Higher-order polymorphism; Strong normalization

* Corresponding author.

E-mail addresses: abel@cs.chalmers.se (A. Abel), matthes@informatik.uni-muenchen.de (R. Matthes), tarmo@cs.ioc.ee (T. Uustalu).

0304-3975/ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2004.10.017
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 List $A$ of finite lists of elements taken from $A$, is a separate inductive 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 [23] 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 [44]. Kahrs [30] 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 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 behaviors and, in particular, termination. This is because, in functional programming, the main motivation to program in a disciplined fashion (e.g., 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”, 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 program: 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 this article. In Section 2, we recapitulate System $F^{o}$ of higher-order polymorphism as a type assignment system for the $\lambda$-calculus with pairing (products), tagging (disjoint sums) and packing (existentials). In the following sections, we extend $F^{o}$ 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 $^{o}$ indicates that inductive constructors of all higher kinds are available, which holds for all of the 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 $MIt^{o}$ which is embeddable into $F^{o}$ and generic enough to simulate all other systems. Fig. 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 $MIt^{o}$ to $F^{o}$, 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 Fig. 1 illustrates, all other systems are definable in \( \text{Mit}^{\omega} \). 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 \( \text{GMIt}^{\omega} \) with hard-wired Kan extensions, presented in Section 4. The attribute generalized has been chosen in resemblance of Bird and Paterson’s [11] generalized folds. System \( \text{GMIt}^{\omega} \) has been first described in a previous publication [2] where a direct embedding into \( F^{\omega} \) 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 Sections 5–7.

---

**Fig. 1.**

Legend:
- \( \Rightarrow \) special case of (see Section \( n \))
- \( \rightarrow \) definable in (see Section \( n \))
- \( \rightarrow \) embeddable into (see Section 5)
- \( \rightarrow \) direct definition (Abel and Matthes, 2003)
- \( \rightarrow \) direct definition (Abel, Matthes, and Uustalu, 2003)
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, \( \text{It}^0 \), 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 \( \text{It}^0 \) has been treated before [1]. It is definable in \( \text{F}^0 \), but also in terms of Mendler iteration via a cut-down version of \( \text{GMIt}^0 \), called \( \text{MIt}^0 \), which only uses Kan extensions along the identity. As shown in [2], programming in \( \text{It}^0 \) often requires a second layer of Kan extensions. This flaw is remedied in System \( \text{Glt}^0 \), 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 et al. [35]; a type-theoretic adaptation of their work is shown to be definable in System \( \text{Glt}^0 \). 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 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 [2] which mostly discusses system \( \text{GMIt}^0 \) (called \( \text{MIt}^0 \) in that article). New in this article are the basic systems \( \text{MIt}^0 \) and \( \text{It}^0 \) as well as the discussion of \( \text{Glt}^0 \) and the definability of efficient folds within \( \text{Glt}^0 \). The discussion of \( \text{It}^0 \) (a subsystem of \( \text{Glt}^0 \)) and some examples are taken from Abel and Matthes [1]. 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 \( \text{F}^0 \)**

Our development of higher-order datatypes takes place within the Curry-style version of System \( \text{F}^0 \), extended with binary sums and products, unit type and existential quantification over type constructors. We employ the usual named-variables syntax, but identify \( \pi \)-equivalent expressions that is properly achieved in the nameless version à la de Bruijn.
Table 2
Overview of examples

<table>
<thead>
<tr>
<th>No.</th>
<th>Example</th>
<th>System(s)</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Booleans: true, false, if</td>
<td>F[^]</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Maybe: bind</td>
<td>F[^] + patterns</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Lists: map</td>
<td>Haskell</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Lists: map</td>
<td>Mit[^]</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Streams: head and tail</td>
<td>Mit[^]</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Streams: upfrom</td>
<td>Mit[^]</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>Powerlists: get</td>
<td>Mit[^]</td>
<td>19</td>
</tr>
<tr>
<td>3.6</td>
<td>Powerlists: sum</td>
<td>Mit[^]</td>
<td>20</td>
</tr>
<tr>
<td>3.7</td>
<td>Infinite triangles: decomposition</td>
<td>Mit[^]</td>
<td>22</td>
</tr>
<tr>
<td>3.9</td>
<td>Streams: redecoration</td>
<td>Haskell</td>
<td>22</td>
</tr>
<tr>
<td>3.10</td>
<td>Infinite triangles: redecoration</td>
<td>Mit[^]</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Powerlists: sum</td>
<td>GMit[^]</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Bushes: sum</td>
<td>GMit[^]</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>Infinite triangles: redecoration</td>
<td>GMit[^]</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Powerlists: sum</td>
<td>It[^]</td>
<td>32</td>
</tr>
<tr>
<td>6.4</td>
<td>De Bruijn terms: map</td>
<td>It[^]</td>
<td>40</td>
</tr>
<tr>
<td>6.5</td>
<td>Powerlists: reverse</td>
<td>It[^]</td>
<td>41</td>
</tr>
<tr>
<td>7.1</td>
<td>Powerlists: sum</td>
<td>GIt[^]</td>
<td>43</td>
</tr>
<tr>
<td>7.2</td>
<td>De Bruijn terms: substitution</td>
<td>GIt[^]</td>
<td>43</td>
</tr>
<tr>
<td>8.1</td>
<td>De Bruijn + expl. subst.: map</td>
<td>It[^]</td>
<td>46</td>
</tr>
<tr>
<td>8.2</td>
<td>De Bruijn + expl. subst.: eval</td>
<td>It[^]</td>
<td>47</td>
</tr>
<tr>
<td>8.3</td>
<td>Finite triangles: redecoration</td>
<td>It[^], GMit[^], Mit[^]</td>
<td>48</td>
</tr>
<tr>
<td>9.1</td>
<td>Hofunctors</td>
<td>efold</td>
<td>51</td>
</tr>
<tr>
<td>9.2</td>
<td>Powerlists: efold (typing)</td>
<td>GIt[^]</td>
<td>53</td>
</tr>
<tr>
<td>9.3</td>
<td>Powerlists: sum (typing)</td>
<td>efold</td>
<td>54</td>
</tr>
<tr>
<td>9.4</td>
<td>Powerlists: efold (reduction)</td>
<td>GIt[^]</td>
<td>54</td>
</tr>
<tr>
<td>9.5</td>
<td>Powerlists: sum (reduction)</td>
<td>efold</td>
<td>55</td>
</tr>
<tr>
<td>9.6</td>
<td>De Bruijn terms: efold</td>
<td>GIt[^]</td>
<td>55</td>
</tr>
<tr>
<td>9.7</td>
<td>De Bruijn terms: substitution</td>
<td>efold</td>
<td>56</td>
</tr>
</tbody>
</table>

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[^], 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 $\kappa$:

\[
\begin{align*}
\kappa & ::= * \mid \kappa \to \kappa', \\
\text{rk}(*) & ::= 0, \\
\text{rk}(\kappa \to \kappa') & ::= \max(\text{rk}(\kappa) + 1, \text{rk}(\kappa')).
\end{align*}
\]
The *rank* of kind $\kappa$ is denoted by $\text{rk}(\kappa)$. We introduce abbreviations for some special kinds: $k0 = \ast$, types, $k1 = \ast \to \ast$, unary type transformers and $k2 = (\ast \to \ast) \to (\ast \to \ast)$ unary transformers of type transformers. Then, $\text{rk}(k_i) = i$ for $i \in \{0, 1, 2\}$.

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

**Type constructors.** (Denoted by uppercase latin letters.) Meta-variable $X$ ranges over an infinite set of type constructor variables:

\[
A, B, C, F, G ::= X | \lambda X. F | F \cdot G | \forall X^\kappa. A | \exists X^\kappa. A | A \to B
| A + B | A \times B | 1.
\]

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 $\kappa$. 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., $F \cdot G \cdot H$ stands for $(F \cdot G) \cdot H$. For $\vec{F} = F_1, \ldots, F_n$ a vector of constructors, we abbreviate $F_1 \cdots F_n$ as $\vec{F}$. We write $\text{Id}$ for $\lambda X. X$ and $F \circ G$ for $\lambda X. F(\lambda X. X)$. 

**Objects (terms).** (Denoted by lowercase letters.) The meta-variables $x$ and $y$ range over an infinite set of object variables:

\[
r, s, t ::= x | \lambda x. t | r \cdot s | \text{inl} t | \text{inr} t | \text{case} (r, x, s, y, t)
| \langle \rangle | \langle t_1, t_2 \rangle | \text{fst} r | \text{snd} r | \text{pack} t | \text{open} (r, x, s).
\]

Most term forms are standard; pack introduces and open eliminates existential quantification, see below. The term former $\text{fst}$, whenever it is used without argument, should be understood as the first projection function $\lambda x. \text{fst} x$. This holds analogously for $\text{snd}$, inl, inr and pack. The identity $\lambda x. x$ will be denoted by $\text{id}$ and the function composition $\lambda x. f(g x)$ by $f \circ g$. Application $r s$ associates to the left, hence $r \vec{s} = (\ldots(rs_1) \ldots s_n)$ for $\vec{s} = s_1, \ldots, s_n$.

Note that there is no distinguished form of recursion in the language of System $F^\omega$. 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 well typed. As an auxiliary notion, we need to introduce contexts which record kinds resp. types of free variables.
Contexts. Variables in a context \( \Gamma \) 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.)

\[
\begin{align*}
\Gamma \text{ ext} & , \quad \Gamma \text{ is a wellformed context,} \\
\Gamma \vdash F : \kappa & , \quad F \text{ is a wellformed type constructor of kind } \kappa \text{ in context } \Gamma, \\
\Gamma \vdash t : A & , \quad t \text{ is a wellformed term of type } A \text{ in context } \Gamma.
\end{align*}
\]

Well-formed contexts. \( \Gamma \text{ ext} \)

\[
\dfrac{\cdot \text{ ext}}{\Gamma_1 \cdot \Gamma_2 : \Gamma_1, X^\kappa \cdot \Gamma_2, x : A \text{ ext}}.
\]

Contexts assign kinds to type variables and types (not arbitrary type constructors!) to object variables.

Well-kindred-type constructors. \( \Gamma \text{ ext} \)

\[
\begin{align*}
\Gamma, X^\kappa \in \Gamma & , \quad \Gamma \text{ ext} \\
\Gamma, X^\kappa \vdash F : \kappa & , \quad \Gamma \vdash F : \kappa \rightarrow \kappa' \rightarrow G : \kappa \\
\Gamma, X^\kappa \vdash F \rightarrow \lambda X. F : \kappa & , \quad \Gamma \vdash \lambda X. F : \kappa \rightarrow \kappa' \\
\Gamma, X^\kappa \vdash G \rightarrow F : \kappa & , \quad \Gamma \vdash F \rightarrow G : \kappa' \\
\Gamma, X^\kappa \vdash A : \star & , \quad \Gamma \vdash A : \star \rightarrow B : \star \\
\Gamma, X^\kappa \vdash A : \star & , \quad \Gamma \vdash A : \star \\
\Gamma, X^\kappa \vdash B : \star & , \quad \Gamma \vdash B : \star \\
\Gamma \text{ ext} & , \quad \Gamma \vdash A + B : \star \\
\Gamma \vdash A \times B : \star & , \quad \Gamma \vdash A \times B : \star \\
\Gamma \vdash 1 : \star & , \quad \Gamma \vdash 1 : \star
\end{align*}
\]

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 \( A, B, C, D : \star, G, H, X, Y : \kappa_1 \) and \( F : \kappa_2 \). If the context is clear (by default, we take the empty context "\( \cdot \" "), we write \( F : \kappa \) for \( \Gamma \vdash F : \kappa \). Sums and products can be extended to all kinds: For \( \kappa = \bar{\kappa} \rightarrow \star \) with \(|\bar{X}| = |\bar{X}| = n\), set

\[
\begin{align*}
+^\kappa & := \lambda F \lambda G \lambda \bar{X}. F \bar{X} + G \bar{X} \quad \text{and} \quad \times^\kappa & := \lambda F \lambda G \lambda \bar{X}. F \bar{X} \times G \bar{X}.
\end{align*}
\]

Both these type constructors have kind \( \kappa \rightarrow \kappa \rightarrow \kappa \). If no ambiguity arises, the superscript is omitted.

Equivalence on well-kindred-type constructors. The notion of \( \beta \)-equivalence \( F = F' \) for well-kindred-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 well-kindred-type constructors up to equivalence, which is a decidable relation due to normalization and confluence of simply typed \( \lambda \)-calculus (where our type constructors are the terms and our kinds are the types of that calculus).

As a consequence of this identification, well-kindred-type constructor composition \( \circ \) is associative.
Well-typed terms. \( \Gamma \vdash t : A \). The following chart recapitulates the typing rules for pure Curry-style \( \text{F}^0 \):

\[
\begin{array}{c}
(x : A) \in \Gamma & \frac{\Gamma \vdash x : A}{\Gamma \vdash x : A} \\
\text{ext} & \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \rightarrow B} \\
\text{r} & \frac{\Gamma \vdash r : A \rightarrow B}{\Gamma \vdash r \ s : B} \\
\text{s} & \frac{\Gamma \vdash s : A}{\Gamma \vdash s : A}
\end{array}
\]

\[
\begin{array}{c}
\frac{\Gamma \vdash t : A}{\Gamma, X^\kappa \vdash t : A} \\
\Gamma \vdash t : \forall X^\kappa.A \\
\Gamma \vdash F : \kappa
\end{array}
\]

More rules are needed for introduction and elimination of the System \( \text{F}^0 \) extensions: unit type, binary sum and product types, and existential types:

\[
\begin{array}{c}
\text{ext} & \frac{\Gamma \vdash t : A}{\Gamma \vdash \text{inl } t : A + B} \\
\text{inl} & \frac{\Gamma \vdash t : B}{\Gamma \vdash \text{inr } t : A + B} \\
\text{inr} & \frac{\Gamma \vdash t : A + B}{\Gamma \vdash \text{case } \mathbf{(r,x,s,y,t)} : C} \\
\text{case} & \frac{\Gamma \vdash r : A + B}{\Gamma \vdash \text{inl } t : A + B} \\
\text{fst} & \frac{\Gamma \vdash r : A \times B}{\Gamma \vdash \text{fst } r : A} \\
\text{snd} & \frac{\Gamma \vdash r : A \times B}{\Gamma \vdash \text{snd } r : B} \\
\text{pack} & \frac{\Gamma \vdash t : A[X := F]}{\Gamma \vdash \text{pack } t : \exists X^\kappa.A} \\
\text{open} & \frac{\Gamma \vdash F : \kappa}{\Gamma \vdash \text{open } \mathbf{(r,x,s,y,t)} : C} \\
\end{array}
\]

As for well-kindred-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 \( \Gamma \) iff there are terms \( r, s \) such that \( \Gamma \vdash r : A \rightarrow B \) and \( \Gamma \vdash s : B \rightarrow A \). If the context \( \Gamma \) is clear from the context of discourse, we just write \( A \leftrightarrow B \) for logical equivalence of \( A \) and \( B \).

2.3. Reduction

Terms of System \( \text{F}^0 \) denote functional programs whose operational meaning is given by the following reduction system.

The one-step reduction relation \( t \rightarrow t' \) between terms \( t \) and \( t' \) is defined as the closure of the following axioms under all term formers: \(^1\)

\[
\begin{align*}
(\lambda x.t) \ s & \rightarrow_{\beta} t[x := s], \\
\text{case } \mathbf{(inl \ r,x,s,y,t)} & \rightarrow_{\beta} s[x := r],
\end{align*}
\]

\(^1\)This means, we may apply one of the \( \rightarrow_{\beta} \)-rules to an arbitrary subterm of \( t \) in order to obtain one step of reduction. This especially includes the \( \xi \)-rule which says that \( t \rightarrow t' \) implies \( \lambda x.t \rightarrow \lambda x.t' \). Clearly, this rule is not implemented in the usual functional programming languages. Since we prove strong normalization, we are on the safe side.
We denote the transitive closure of $\rightarrow$ by $\rightarrow^+$ and the reflexive–transitive closure by $\rightarrow^*$.

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

Example 2.1 (Booleans). We can encode the datatype of booleans $\text{Bool} : *$ in System F as $\text{Bool} := 1 + 1$, with data constructors $\text{true} := \text{inl}()$ and $\text{false} := \text{inr}()$. Elimination of booleans is done by if-then-else, which is encoded as $\text{if} := \lambda b \lambda t \lambda e. \text{case } (b, \_t, \_e)$. The reader is invited to check the typings $\text{true} : \text{Bool}$, $\text{false} : \text{Bool}$ and $\text{if} : \text{Bool} \rightarrow \forall A. A \rightarrow A \rightarrow A$ as well as the operational behavior $\text{iftrue } t e \rightarrow^+ t$ and $\text{iffalse } t e \rightarrow^+ e$.

2.4. Syntactic sugar

The term language of System F 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; we formally describe a transformation relation $\Rightarrow$ which eliminates all syntactic sugar.

Non-recursive let bindings. As implemented in some functional programming languages, e.g., Scheme or Ocaml, let $x = r$ in $s$ shall denote the $\beta$-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 from programs by performing the following transformation steps on any part of the program until no let-bindings remain:

\[
\text{let } x = r \text{ in } s \Rightarrow (\lambda x.s) r.
\]

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

\[
p ::= x \mid () \mid (p, p') \mid \text{inl } p \mid \text{inr } p \mid \text{pack } p.
\]

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

Pattern matching is introduced by the notation

\[
\text{match } r \text{ with } p_1 \mapsto s_1 \mid \ldots \mid p_n \mapsto s_n.
\]

The order of the clauses $p_i \mapsto s_i$ is irrelevant. For succinctness, we write $\text{match } r \text{ with } (p_i \mapsto s_i)_{i=1..n}$ or even $\text{match } r \text{ with } p \mapsto s$, for short. The notation $\text{match } r \text{ with } \overline{p} \mapsto \overline{s}$ | $\overline{q} \mapsto \overline{t}$ should also be easily understandable.
Pattern matching is expanded by the following new rules for the transformation relation $\leadsto$. 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:

\[
\text{match } r \text{ with } x \mapsto s \leadsto \text{let } x = r \text{ in } s,
\]
\[
\text{match } r \text{ with } \langle \rangle \mapsto s \leadsto s,
\]
\[
\text{match } r \text{ with } \langle p_i, p_j \rangle \mapsto s_{ij} \leadsto \text{let } x = r \text{ in } \text{match } \text{fst } x \text{ with } \langle p_i \mapsto \text{match } \text{snd } x \text{ with } p_j \mapsto s_{ij} \rangle_{i \in I, j \in J}
\]
\[
\text{match } r \text{ with } \text{inl } \vec{p} \mapsto \vec{s} \leadsto \text{case } (r, x. \text{match } \vec{x} \text{ with } \langle p_i \mapsto \text{match } \vec{y} \text{ with } p_j \mapsto s_{ij} \rangle_{i \in I, j \in J})
\]
\[
\text{match } r \text{ with } \text{inr } \vec{q} \mapsto \vec{t} \leadsto \text{open } (r, x. \text{match } \vec{x} \text{ with } \langle p_i \mapsto \text{match } \vec{y} \text{ with } p_j \mapsto s_{ij} \rangle_{i \in I, j \in J})
\]

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 $r$. Note also that our rule for matching with $\langle \rangle$ just expresses an identification of all terms of type 1 with its canonical inhabitant $\langle \rangle$.

**Patterns in lets and abstractions.** For a single-pattern matching, which has the form $\text{match } r \text{ with } p \mapsto s$—thus excluding matching with $\text{inl}$ and $\text{inr}$ due to the assumption of exhaustive case analysis—we introduce a more concise notation $\text{let } p = r \text{ in } s$, which is common in functional programming. Furthermore, an abstraction of a variable $x$ plus a matching over this variable, $\lambda x. \text{let } p = x \text{ in } s$, can from now be shortened to $\lambda x. s$. 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:

\[
\text{let } p = r \text{ in } s \leadsto \text{match } r \text{ with } p \mapsto s \quad \text{if } p \text{ is not a variable},
\]
\[
\lambda p. s \leadsto \lambda x. \text{match } x \text{ with } p \mapsto s \quad \text{if } p \text{ is not a variable}.
\]

**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 $c^\tau$ to mean $s[\vec{x} := \vec{\tau}]$ where $|\vec{\tau}| = n$. In the special case that $s$ is a pattern $p$, the sugared expression $c^\tau$ 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 $\text{in}^k$ and data constructors of the shape $c := \vec{x}. \text{in}^k p$. The notation $c^\tau$ shall denote $p[\vec{x} := \vec{\tau}]$, i.e., instantiation after removal of $\text{in}^k$. Summarizing, we have two additional transformations:

\[
c^\tau \leadsto s[\vec{x} := \vec{\tau}] \quad \text{if } c := \vec{x}. s,
\]
\[
c^{-\tau} \leadsto p[\vec{x} := \vec{\tau}] \quad \text{if } c := \vec{x}. \text{in}^k p.
\]
Example 2.2 ("maybe" type transformer). To see the meta-syntax in action, consider the option datatype with two data constructors:

\[
\text{Maybe} ::= \lambda A. 1 + A : k1 \\
\text{nothing} ::= \text{inl(}) : \forall A. \text{Maybe A} \\
\text{just} ::= \lambda a. \text{inr} a : \forall A. A \rightarrow \text{Maybe A}
\]

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

\[
\text{bind} : \forall A \forall B. \text{Maybe A} \rightarrow (A \rightarrow \text{Maybe B}) \rightarrow \text{Maybe B} \\
\text{bind} ::= \lambda m \lambda k. \text{match} m \text{ with} \\
\text{nothing}^o \mapsto \text{nothing} \\
\text{just}^o a \mapsto k a
\]

We could have dropped the annotation \( ^o \) in matching against \text{nothing}^o—since nothing is a data constructor without arguments and the annotation \( ^o \) is changing nothing in this case—but we included it for reasons of symmetry.

3. System MIt\( ^o \) of basic Mendler iteration and coiteration

In this section, we will introduce System MIt\( ^o \), a conservative extension of F\( ^o \), 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 \text{List}(A) : * over some element type \( A \), which has two data constructors \text{nil} : \text{List}(A) and \text{cons} : A \rightarrow \text{List}(A) \rightarrow \text{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 [15]. \text{List}(A) is defined as the least fixed point of an operator \text{ListF}(A) ::= \lambda X. 1 + A \times X and we write \text{List}(A) ::= \mu(\text{ListF}(A)). There is just a single constructor \text{in} : \text{ListF}(A)(\text{List}(A)) \rightarrow \text{List}(A) for lists. The functions \text{nil} and \text{cons} can be defined in terms of \text{in}:

\[
\text{nil} ::= \text{in} (\text{inl}()) : \text{List}(A), \\
\text{cons} ::= \lambda a \lambda as. \text{inr} (\text{inr}(a, as)) : A \rightarrow \text{List}(A) \rightarrow \text{List}(A).
\]

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 \texttt{map}.

\textbf{Example 3.1 (map function for lists).} It is part of the Haskell standard library and could be defined like this:

\begin{verbatim}
map :: (a -> b) -> [a] -> [b]
map f = map'
    where map' [] = []
          map' (a : as) = (f a) : (map' as).
\end{verbatim}

The recursive part of this definition is the interior function \texttt{map'} which arises as the fixed point of a functional $s = \lambda \text{map}'. \text{body of \text{map'}}$ such that the equation $\text{map'} = s \text{map'}$ 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 \texttt{map'} occurs only with argument \texttt{as} which is a direct subterm of the input $\langle a : \text{as} \rangle$. This already ensures termination of \texttt{map'}. Furthermore, the sublist \texttt{as} is only used as an argument to \texttt{map'}, hence the function is even \textit{iterative}.

Mendler [40] 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 \texttt{as} and restrict applications of the recursive function \texttt{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 \texttt{as} can neither be further analyzed nor used in any way besides as an argument to \texttt{map'}, and

2. the function \texttt{map'} cannot be called recursively unless applied to \texttt{as}.

Mendler’s trick is implemented by requiring $s$ to be of type $(X \rightarrow B) \rightarrow \text{ListF}(A)X \rightarrow B$ for a fresh type variable $X$. The first parameter of $s$ is the name \texttt{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 $\text{in} t$ of \texttt{map'}, but is now by its type $\text{ListF}(A)X$ restricted to be either the canonical inhabitant $\langle \rangle$ of type $1$—for the case of the empty list \texttt{nil} as input to \texttt{map'}—or a pair of a head element of type $A$ and a tail of type $X$—for the \texttt{cons} case. In the \texttt{nil} case, $s$ somehow has to produce a result of type $B$, in the \texttt{cons} case, $s$ can—among other possibilities—apply \texttt{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 \textit{Mendler iterator}—written \texttt{MIt}(s) for a step term $s$. It has the following reduction behavior:

\[
\text{MIt}(s) \ (\text{in} \ t) \rightarrow s \ \text{MIt}(s) \ t.
\]

During reduction, the type variable $X$ is substituted by the inductive type $\mu F$, in our example $\text{List}(A)$. The fixed-point type $\mu F$ is unrolled into $F(\mu F)$, hence, the data constructor $\text{in}$ 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 $\text{in}$ 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. \( \mu : \ast \to \ast \to \ast \)

Introduction. \( \text{in} : \forall F \to \ast. F(\mu F) \to \mu F \)

Elimination. \( \Gamma \vdash F : \ast \to \ast \)
\( \Gamma \vdash B : \ast \)
\( \Gamma \vdash s : \forall X. (X \to B) \to F X \to B \)
\( \Gamma \vdash \text{Mlt}(s) : \mu F \to B \)

Reduction. \( \text{Mlt}(s)(\text{in } t) \to s \text{ Mlt}(s) t. \)

**Example 3.2** (map function for lists). Using the syntax of \( F^\circ \) with the meta-notation for pattern matching described in Section 2, we can encode the Haskell function map with Mendler iteration as follows:

\[
\text{map} : \forall A \forall B. (A \to B) \to \text{List}(A) \to \text{List}(B),
\]

\[
\text{map} := \lambda f. \text{Mlt}\left(\lambda \text{map}'\lambda t. \text{match } t \text{ with }\right.
\]
\[
\left.\begin{array}{l}
\text{nil}^- \mapsto \text{nil} \\
\text{cons}^- a \ as \mapsto \text{cons}(f \ a) (\text{map}' \ as)
\end{array}\right).
\]

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

\[
f : A \to B
\]
\[
\text{map}' : X \to \text{List}(B)
\]
\[
t : \text{ListF}(A) X
\]
\[
a : A
\]
\[
as : X.
\]

Here, \( X : \ast \) is a fresh type variable introduced by the Mendler iterator. Also note that, according to the conventions introduced in Section 2, \( \text{nil}^- = \text{inl}(\,\) and \( \text{cons}^- a \ as = \text{inr}(a, \,as)\). 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 \( \nu F \) 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:

Formation. \( \nu : \ast \to \ast \to \ast \)

Elimination. \( \text{out} : \forall F \to \ast. \nu F \to F(\nu F) \)
Introduction. \[ \Gamma \vdash F : * \to * \]
\[ \Gamma \vdash A : * \]
\[ \Gamma \vdash s : \forall X^*. (A \to X) \to A \to FX \]
\[ \Gamma \vdash \text{MCoit}(s) : A \to \forall F \]

Reduction. \[ \text{out} (\text{MCoit}(s) t) \to \beta s \text{MCoit}(s) t. \]

Dually to the general constructor in for inductive types, coinductive types possess a general destructor\( \text{out} \) 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 3.3 (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:

\[
\begin{align*}
\text{Stream} &:= \lambda A. \forall (\lambda X. A \times X) : * \to *, \\
\text{head} &:= \lambda r. \text{fst}(\text{out} r) : \forall A. \text{Stream} A \to A, \\
\text{tail} &:= \lambda r. \text{snd}(\text{out} r) : \forall A. \text{Stream} A \to \text{Stream} A.
\end{align*}
\]

**Example 3.4 (sequence of natural numbers).** Assume a type \( \text{Nat} \) 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:

\[
\text{upfrom} := \text{MCoit}(\lambda \text{upfrom}. (n, \text{upfrom}(n + 1)))
\]
\[
: \text{Nat} \to \text{Stream} \text{Nat}.
\]

### 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:

\[
\begin{align*}
\text{data PList} a &= \text{Zero} a & | & \text{Succ} (\text{PList} (a, a)) \\
\text{data Bush} a &= \text{Nil} & | & \text{Cons} a (\text{Bush} (\text{Bush} a)) \\
\text{data Lam} a &= \text{Var} a & | & \text{App} (\text{Lam} a) (\text{Lam} a) \\
&& & \text{Abs} (\text{Lam} (\text{Maybe} a)).
\end{align*}
\]

The first definition, PList, is the type of *powerlists* [9], resp. *perfectly balanced, binary leaf trees* [21]. Note 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” [10] 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 [5] and Bird and Paterson [12]; a precursor of this type has been considered already by Pfennig and Lee [45] and Pierce et al. [46]. 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 + 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 note that all of the datatypes PList, Bush, Lam are first order as type constructors: they are of kind * → *. 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 Fω. The encoding is possible if a combinator μk1 : (k1 → k1) → k1 for least fixed-point types of rank 1 is present. (Recall that k1 = * → *.) In the following we give representations of these three types as least fixed points μF of type transformer transformers F : k2:

\[
\begin{align*}
P\text{List}^F & := \lambda X \lambda A. A + X (A \times A) : k2, \\
P\text{List} & := \mu^k1 P\text{List}^F : k1, \\
B\text{ush}^F & := \lambda X \lambda A. 1 + A \times X (X A) : k2, \\
B\text{ush} & := \mu^k1 B\text{ush}^F : k1, \\
L\text{am}^F & := \lambda X \lambda A. A + (X A \times X A + X (1 + A)) : k2, \\
L\text{am} & := \mu^k1 L\text{am}^F : k1.
\end{align*}
\]

Similar to the rank-1 case we just have one general datatype constructor inl which rolls an inhabitant of F (μk1 F) into the fixed point μk1 F. Note, however, that μk1 F is not a type but a type constructor, hence, we need a polymorphic data constructor inlF : ∀A. F (μk1 F) A → μk1 F A. Now, we are ready to define the usual data constructors for the heterogeneous datatypes we are encoding:

\[
\begin{align*}
\text{zero} & := \lambda a. \text{inl}^k (\text{inl} a) : \forall A. A \rightarrow P\text{List} A, \\
\text{succ} & := \lambda l. \text{inl}^k (\text{inr} l) : \forall A. P\text{List}(A \times A) \rightarrow P\text{List} A, \\
\text{bnil} & := \text{inl}^k (\text{inl} (\text{})) : \forall A. B\text{ush} A, \\
\text{bcons} & := \lambda a \lambda b. \text{inl}^k (\text{inr} (a, b)) : \forall A. A \rightarrow B\text{ush} (B\text{ush} A) \rightarrow B\text{ush} A, \\
\text{var} & := \lambda a. \text{inl}^k (\text{inl} a) : \forall A. A \rightarrow L\text{am} A, \\
\text{app} & := \lambda t_1 \lambda t_2. \text{inl}^k (\text{inr} (\text{inl} (t_1, t_2))) : \forall A. L\text{am} A \rightarrow L\text{am} A \rightarrow L\text{am} A, \\
\text{abs} & := \lambda r. \text{inl}^k (\text{inr} (\text{inr} r)) : \forall A. L\text{am} (1 + A) \rightarrow L\text{am} A.
\end{align*}
\]

Our aim is to define iteration for nested datatypes, a quest which recently has attracted some interest in the functional programming community [11,21,35]. 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 $F \subseteq^κ G$ from type constructor $F : κ$ to $G : κ$. Since every kind $κ$ can be written in the form $\vec{X} \to ^*$, natural transformations for kind $κ$ can simply be defined as follows:

$$F \subseteq^κ G := ∀ \vec{X} F \vec{X} \to G \vec{X}.$$  

(Here, we have made use of the vector notation $∀ \vec{X}$ as an abbreviation for $∀ X_1 \ldots ∀ X_n$, 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 $\text{in}^k_1$ from the last subsection is a natural transformation of type $F(\vec{X}^k_1) \subseteq^k \vec{X}^k_1 F$. The superscript $κ$ in “$\subseteq^κ$” 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. | $μ^κ : (κ \to κ) \to κ$ |
| Introduction. | $\text{in}^κ : ∀ F^{κ\toκ}, F(μ^κ F) \subseteq^κ μ^κ F$ |
| Elimination. | $\Gamma \vdash F : κ \to κ$
| | $\Gamma \vdash G : κ$
| | $\Gamma \vdash s : ∀ X^κ, X \subseteq^κ G \to F X \subseteq^κ G$
| | $\bar{Γ} \vdash \text{Mlt}^κ(s) : μ^κ F \subseteq^h G$
| Reduction. | $\text{Mlt}^κ(s)(\text{in}^κ t) \longrightarrow_β s \text{Mlt}^κ(s) t$. |

Note that for every type constructor $F$ of kind $κ \to κ$, $μ^κ F$ is a type constructor of kind $κ$. In Mendler’s original system [40] as well as its variant for the treatment of primitive (co)recursion [39], positivity of $F$ is always required, which is a very natural concept in the case $κ = ^*$. (A first-order type constructor $F : ^* \to ^*$ is said to be positive iff every occurrence of $X$ in $FX$ is positive in the sense of being enclosed in an even number of left-hand sides of $→$.) For higher kinds, however, there is no such canonical syntactic restriction. Anyway, in Uustalu and Vene [47] 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 [36] 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 $F^0$. In the following we will try to evaluate whether with $\text{Mlt}^κ$ we have obtained a sensible and usable device for programming.

**Example 3.5** (retrieving a leaf of a perfectly balanced tree). Let $t : \text{PList} A$ be a binary leaf-labelled tree and $p : \text{Stream} \text{Bool}$ a bit stream which acts as a path to one of the leaves $a$ of $t$. Using $\text{Mlt}^k_1$ we can implement a function $\text{get}$ such that $\text{get} t p$...
retrieves element \(a\).

\[
\text{get} := \text{Mitk}^1(\lambda \text{get} \lambda \text{t} \lambda \text{p}. \text{match } \text{t} \text{ with } \\
\text{zero} \mapsto a \\
\text{succ} \mapsto \text{let } (a_1, a_2) = \text{get} \text{ (tail } \text{p} \text{) in } \\
\text{if } (\text{head } \text{p}) a_1 a_2 )
\]

\[
: \forall \ A. \ P\text{list } A \rightarrow \text{Stream Bool } \rightarrow A
\]

\[= \ P\text{list } \subseteq^1 (\lambda A. \text{Stream Bool } \rightarrow A).
\]

Here we reused the type of streams defined in Example 3.3 and the booleans defined in Section 2. To verify well-typedness, observe that the bound variables have the following types:

\[
X : k1 \quad \text{(not visible due to Curry style)}
\]

\[
\text{get} : \forall \ A. \ X \ A \rightarrow \text{Stream Bool } \rightarrow A
\]

\[
A : * \quad \text{(not visible due to Curry style)}
\]

\[
t : \ P\text{listF} \ X \ A = A + X (A \times A)
\]

\[
p : \text{Stream Bool}
\]

\[
a, a_1, a_2 : A
\]

\[
l : X (A \times A).
\]

In the recursive calls, the polymorphic type of \(\text{get}\) is instantiated with the product \(A \times A\) which entails the typing \(\text{get} \ (\text{tail } p) : A \times A\). It is now easy to check well-typedness of the whole function body. Note that \(\text{Mitk}^1\) facilitates a kind of polymorphic recursion.

**Example 3.6 (summing up a powerlist).** Next, we want to define a function \(\text{sum} : \ P\text{list Nat } \rightarrow \text{Nat}\) which sums up all elements of a powerlist by iteration over its structure. In the case \(\text{sum} (\text{zero } n)\) we can simply return \(n\). The case \(\text{sum} (\text{succ } t)\), however, imposes some challenge since \(\text{sum}\) cannot be directly applied to \(t : \ P\text{list}(\text{Nat } \times \text{Nat})\). The solution is to define a more general function \(\text{sum}'\) by polymorphic recursion, which has the following behavior:

\[
\text{sum}' : \forall A. \ P\text{list } A \rightarrow (A \rightarrow \text{Nat}) \rightarrow \text{Nat},
\]

\[
\text{sum}' (\text{zero } a) f \rightarrow^+ f a,
\]

\[
\text{sum}' (\text{succ } l) f \rightarrow^+ \text{sum}' l (\lambda (a_1, a_2). f a_1 + f a_2).
\]

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 \(\text{sum}'\), its implementation using \(\text{Mitk}^1\) is a mechanical process which results in the following definition:

\[
\text{sum}' := \text{Mitk}^1(\lambda \text{sum}' \lambda t \lambda f. \text{match } \text{t} \text{ with } \\
\text{zero} \mapsto f a \\
\text{succ} \mapsto \text{sum}' l (\lambda (a_1, a_2). f a_1 + f a_2))
\]

\[= \ P\text{list } \subseteq^1 (\lambda A. (A \rightarrow \text{Nat}) \rightarrow \text{Nat}).
\]

The postulated reduction behavior is verified by a simple calculation. From \(\text{sum}'\), the summation function is obtained by \(\text{sum} := \lambda t. \text{sum}' t \text{ id}\).
Let us remark here that the result type constructor $G' = \lambda A. (A \to \text{Nat}) \to \text{Nat}$ is an instance of a general scheme which is extremely useful for defining functions over heterogeneous datatypes. For the constant type constructors $G = H = \lambda B. \text{Nat}$, 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 hard-wired Kan extensions.

Having completed these two examples we are confident that $\text{MIt}_k$ 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 $\text{Mlt}^\omega$, which is an extension of Mendler’s [40] system to finite kinds:

- **Formation.** $\nu^\kappa : (\kappa \to \kappa) \to \kappa$
- **Elimination.** $\text{out}^\kappa : \forall F^{\kappa \to \kappa}, \nu^\kappa F \subseteq \kappa F (\nu^\kappa F)$
- **Introduction.**
  - $I \vdash F : \kappa \to \kappa$
  - $I \vdash G : \kappa$
  - $I \vdash s : \forall X^\kappa, G \subseteq X \to G \subseteq \kappa F X$
  - $I \vdash \text{MCoit}^\kappa(s) : G \subseteq \kappa \nu^\kappa F$
- **Reduction.** $\text{out}^\kappa (\text{MCoit}^\kappa(s)t) \to \beta s \text{ MCoit}^\kappa(s)t$.

The reader is invited to check that no problems for subject reduction and confluence arise from these definitions. To demonstrate the usefulness of $\text{MCoit}^{k_1}$ as a coiteration scheme, in the following we will develop a redecoration algorithm for infinite triangular matrices, which can be defined as a heterogeneous coinductive type. To this end, we fix a type $E : *$ of matrix elements. The type $\text{Tri} A$ of triangular matrices with diagonal elements in $A$ and ordinary elements $E$ can be obtained as follows:

- $\text{TriF} := \lambda X \lambda A. A \times X (E \times A) : k2$,
- $\text{Tri} := \nu^{k_1} \text{TriF} : k1$.

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 $\text{TriF}$. We can visualize triangles like this:

```
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
A | E | E | E | E | E | E | E | E | E | E | E
```

The vertical lines hint at the decomposition scheme.
Example 3.7 (triangle decomposition). Using the destructor for coinductive types on a triangle \( \text{Tri} \ A \), we can obtain the top element of type \( A \) and the remainder of type \( \text{Tri} (E \times A) \) which looks like an infinite trapezium in our visualization:

\[
top := \lambda t. \ fst (\text{out}^{k_1} t) : \forall A. \ Tri A \rightarrow A,
\]
\[
rest := \lambda t. \ snd (\text{out}^{k_1} t) : \forall A. \ Tri A \rightarrow \text{Tri} (E \times A).
\]

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

\[
cut := \text{MCoit}^{k_1} \left( \lambda \text{cut} \lambda t. \langle \text{snd} (\text{top} t), \text{cut} (\text{rest} t) \rangle \right) :
(\lambda A. \ Tri (E \times A)) \subseteq^{k_1} \text{Tri}.
\]

Remark 3.8 (Corrigendum). In Abel, Matthes, and Uustalu [2] we used \( \text{tri} \ \text{snd} \) instead of \( \text{cut} \), where \( \text{tri} \) is the mapping function for \( \text{Tri} \) and \( \text{snd} \) 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

\[
\text{redec} : \forall A \forall B. (\text{Stream} A \rightarrow B) \rightarrow \text{Stream} A \rightarrow \text{Stream} B,
\]
takes \( f : \text{Stream} A \rightarrow B \) and \( t : \text{Stream} A \) and returns \( \text{redec} \ f \ t \), which is a \( B \)-stream obtained from \( t \) by replacing each of its elements by what \( f \) assigns to the substream this element heads.

Example 3.9 (stream redecoration). Markus Schnell posted an implementation of stream redecoration to the Haskell Mailing List [17]:

\[
\text{slide} :: ([a] \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
\[
\text{slide} f [] = []
\]
\[
\text{slide} f \ xs = f \ xs : \text{slide} f (\text{tail} \ xs).
\]

He showed how to encode a low pass digital filter using \( \text{slide} \). Here is the implementation of a smoothing filter which replaces each stream element by the average of \( n \) adjacent elements:

\[
\text{smooth} :: \text{Int} \rightarrow [\text{Float}] \rightarrow [\text{Float}]
\]
\[
\text{smooth} n = \text{slide} (\ \ xs \rightarrow \text{sum} (\text{take} \ n \ xs) / \text{fromInt} \ n).
\]

Theoretically, redecoration is an operation dual to substitution in trees \( TA \) over some label type \( A \). Viewing \( T \) as a monad, substitution \( (A \rightarrow TB) \rightarrow TA \rightarrow TB \) of \( B \)-labelled trees for \( A \)-labels is the monad multiplication operation. Viewing \( T \) as a comonad, redecoration \( (TA \rightarrow B) \rightarrow TA \rightarrow TB \) becomes the comultiplication [48].
Example 3.10 (triangle redecoration). For triangles, redecoration works as follows: In the triangle

```
  A E E E E ...  
   A E E E ...  
     A E E ...  
       A E ...  
         A ...  
```

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

\[
\text{redec } := \lambda f \lambda t. \text{redec'} \left( \text{pack} \langle f, t \rangle \right) \\
\text{ : } \forall A \forall B. (\text{Tri } A \rightarrow B) \rightarrow \text{Tri } A \rightarrow \text{Tri } B,
\]

\[
\text{redec'} := \text{MCoi}^{k_1} \left( \lambda \text{redec'} \lambda (\text{pack} \langle f, t \rangle) \right). \\
\text{ : } (\lambda B \exists A. (\text{Tri } A \rightarrow B) \times \text{Tri } A) \subseteq^{k_1} \text{Tri}.
\]

Here we make use of a function lift : \( \forall A \forall B. (\text{Tri } A \rightarrow B) \rightarrow \text{Tri}(E \times A) \rightarrow (E \times B) \) which lifts the redecoration rule to trapeziums such that it can be used with the trapezium rest t:

\[
\text{lift } := \lambda f \lambda t. \langle \text{fst} (\text{top } t), f (\text{cut } t) \rangle \\
\text{ : } \forall A \forall B. (\text{Tri } A \rightarrow B) \rightarrow \text{Tri}(E \times A) \rightarrow E \times B.
\]

Hence, if \( f \) is a redecoration rule, the new redecoration rule lift \( f \) for trapeziums takes a trapezium \( t \) of type Tri\((E \times A)\) and yields a diagonal element of a trapezium in 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 \( G' := \lambda B \exists A. (\text{Tri } A \rightarrow B) \times \text{Tri } A \). If the variable redec' receives the type \( G' \subseteq^{k_1} X \) and pack \( \langle f, t \rangle \) is matched, then \( f \) gets type Tri\( A \rightarrow B \) and \( t \) gets type Tri\( A \). Hence \( f i : B \), 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:

\[
\text{out}^{k_1} (\text{redec'} (\text{pack} \langle f, t \rangle)) \rightarrow^+ (f \, t, \text{redec'} (\text{pack} \langle \text{lift } f, \text{rest } t \rangle)),
\]

\[
\text{top} (\text{redec } f \, t) \rightarrow^+ f \, t,
\]

\[
\text{rest} (\text{redec } f \, t) \rightarrow^+ \text{redec}^\circ (\text{lift } f) (\text{rest } t).
\]

On the last line we have used the \( ^\circ \)-notation because simply redec (lift \( f \)) (rest \( t \)) is no \( \beta \)-reduct of the left-hand side. Without the \( ^\circ \)-notation we could only state that left- and right-hand side are \( \beta \)-equal, i.e., have a common reduct.
The source type constructor $G'$ of function $\text{redec}'$ is a left Kan extension of $\text{Tri}$ along $\text{Tri}$, since it is an instance of the general scheme $\lambda B \exists A. (H A \to B) \times G A$ with $G = H = \text{Tri}$. In Section 4 we will introduce a generalized coiteration scheme with hard-wired Kan extensions. This will allow us to define $\text{redec}$ directly and not via an uncurried auxiliary function $\text{redec}'.

3.6. Embedding into $F^{\omega}$

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

Embeddings of (co)inductive type constructors into $F^{\omega}$ 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 $\text{Mlt}^{\omega}$. For the remainder of this section, let kind $\kappa = \vec{r} \to *$:

$$
\forall F^{K} \to \kappa \forall G^{K} (\forall X^{K}, X \subseteq G \to F X \subseteq G) \to \mu^{K} F \subseteq G \\
\equiv \forall F^{K} \to \kappa \forall G^{K} (\forall X^{K}, X \subseteq G \to F X \subseteq G) \to \forall \vec{Y}^{K}, \mu^{K} F \vec{Y} \to G \vec{Y} \\
\leftrightarrow \forall F^{K} \to \kappa \forall \vec{Y}^{K}, \mu^{K} F \vec{Y} \to \forall G^{K}, (\forall X^{K}, X \subseteq G \to F X \subseteq G) \to G \vec{Y}.
$$

This equivalent type for $\text{Mlt}^{\omega}$ states that there is a mapping of $\mu^{K} F \vec{Y}$ into some other type. Now we simply define $\mu^{K} F \vec{Y}$ to be that other type. The definitions of $\text{Mlt}^{K} (s)$ and $\text{in}^{K}$ then simply fall into place:

$$
\mu^{K} : (\kappa \to \kappa) \to \vec{r} \to * \\
\mu^{K} := \lambda F \lambda \vec{Y} \forall G^{K}, (\forall X^{K}, X \subseteq G \to F X \subseteq G) \to G \vec{Y},
$$

$$
\text{Mlt}^{K} (s) := \lambda r. r \text{s},
$$

$$
\text{in}^{K} := \forall F^{K} \to K, F (\mu^{K} F) \subseteq \mu^{K} F
$$

$$
\text{Lemma 3.11. With the definitions above, } \text{Mlt}^{K} (s) (\text{in}^{K} t) \to^{+} s \text{ Mlt}^{K} (s) t \text{ in System } F^{\omega}.
$$

$$
\text{Proof. By simple calculation. } \square
$$
To find an encoding of (co)inductive type constructors, we consider the type of the (universal) coiterator \( \text{MCoit}^\kappa(s) \):

\[
\forall F^{\kappa \to \kappa} \forall G^\kappa. \ (\forall X^\kappa. G \subseteq X \to G \subseteq F X) \to G \subseteq v^\kappa F
\]

\[
\equiv \forall F^{\kappa \to \kappa} \forall G^\kappa. \ (\forall X^\kappa. G \subseteq X \to G \subseteq F X) \to \forall \tilde{Y}^\kappa. G \tilde{Y} \to v^\kappa F \tilde{Y}
\]

\[
\leftrightarrow \forall F^{\kappa \to \kappa} \forall \tilde{Y}^\kappa. \ (\exists G^\kappa. (\forall X^\kappa. G \subseteq X \to G \subseteq F X) \times G \tilde{Y}) \to v^\kappa F \tilde{Y}
\]

These considerations lead to the following definitions:

\[
v^\kappa := (\kappa \to \kappa) \to \kappa \to *
\]

\[
M\text{Coit}^\kappa(s) := \lambda t. \text{pack} \langle s, t \rangle
\]

\[
\text{out}^\kappa := \forall F^{\kappa \to \kappa}, v^\kappa F \subseteq F (v^\kappa F)
\]

\[
\text{out}^\kappa := \lambda (\text{pack}(s, t)). s M\text{Coit}^\kappa(s) t.
\]

**Lemma 3.12.** We have \( \text{out}^\kappa (M\text{Coit}^\kappa(s) t) \rtr {+} s M\text{Coit}^\kappa(s) t \) in System \( F^\omega \), using the definitions above.

**Proof.** By simple calculation. □

**Theorem 3.13 (strong normalization).** System \( \text{Mlt}^\omega \) is strongly normalizing, i.e., for each well-typed term \( t_0 \) there is no infinite reduction sequence \( t_0 \rtr {} \cdot {} T_1 \rtr {} \cdot {} t_2 \rtr {} \cdot {} \cdots \).

**Proof.** By Lemmata 3.11 and 3.12, such a reduction sequence would translate into the infinite sequence \( t_0 \rtr {+} \cdot {} T_1 \rtr {+} \cdot {} t_2 \rtr {+} \cdot {} \cdots \) of well-typed terms of System \( F^\omega \) (here, the above definitions are meant to be unfolded), a contradiction to the strong normalization property of \( F^\omega \). □

4. System \( \text{GMlt}^\omega \): refined and generalized Mendler (co)iteration

The system \( \text{GMlt}^\omega \) of this section is a minor variant of the system \( \text{Mlt}^\omega \) in Abel et al. [2]. Its intension is to ease programming with Mendler iteration in case Kan extensions have to be used (cf. Examples 3.6 and 3.10). In a sense, we are just hard wiring the Kan extensions into the (co)iteration scheme of \( \text{Mlt}^\omega \).

4.1. Containment of type constructors

The key idea consists in identifying an appropriate containment relation for type constructors of the same kind \( \kappa \). For types, the canonical choice is implication. For an arbitrary
kind \( \kappa \), the easiest notion is “pointwise implication” \( \subseteq^K \), used in the previous section for the definition of \( \text{Mit}^d \).

**Refined containment.** A more refined notion is \( \subseteq^K \), which first appeared in Hinze’s [20] work as the polykinded type \( \text{Map} \) of generic mapping functions. We learned it first from Peter Hancock in 2000 and employed it already in earlier work [1] which studied iteration for monotone inductive type constructors of higher kinds:

\[
\begin{align*}
F & \subseteq^* G := F \to G, \\
F & \subseteq^{K\to K'} G := \forall X^K \forall Y^K. X \subseteq^K Y \to FX \subseteq^{K'} GY.
\end{align*}
\]

Hence, for \( F, G : k_1 \), \( F \subseteq^{k_1} G = \forall A \forall B. (A \to B) \to FA \to GB \). 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 [21, 35] are directly covered, we have to relativize the notion \( \subseteq^K \), for \( \kappa = \vec{\kappa} \to \ast \), 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 \( \subseteq^{K\to K'} \), moreover, \( H_i \) is applied to the \( i \)th “target argument” (below, another definition similarly modifies the “source argument”).

For every kind \( \kappa = \vec{\kappa} \to \ast \), define a type constructor \( \subseteq^{K_{\vec{\kappa}}}_{(-)} : (\vec{\kappa} \to \vec{\kappa}) \to \kappa \to \kappa \to \ast \) by structural recursion on \( \kappa \) as follows:

\[
\begin{align*}
F & \subseteq^* G := F \to G, \\
F & \subseteq^{K\to K'}_{H, \vec{H}} G := \forall X^{K_1} \forall Y^{K_1}. X \subseteq^K Y \to FX \subseteq^{K'}_{\vec{H}} GY.
\end{align*}
\]

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

\[
\begin{align*}
F & \subseteq^{k_1} G = \forall A \forall B. (A \to HB) \to FA \to GB, \\
F & \subseteq^{k_1}_{H} G = \forall A \forall B. (HA \to B) \to FA \to GB.
\end{align*}
\]
Even more concretely, we have the following types which will be used in Examples 4.1 and 4.3:

\[
\begin{align*}
\text{PList} & \leq \lambda B. \text{Nat} \lambda B. \text{Nat} = \forall A \forall B. (A \to \text{Nat}) \to \text{PList} A \to \text{Nat}, \\
\text{Tri} & \text{Tri} \leq \lambda B. \text{Nat} = \forall A \forall B. (\text{Tri} A \to B) \to \text{Tri} A \to \text{Tri} B.
\end{align*}
\]

4.2. Definition of GMItκ

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 [11,21,35] 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-well-founded datatypes.

The system GMItκ is given as an extension of MItκ by well-kinded type-constructor constants \(\mu^\kappa\) and \(\nu^\kappa\), well-typed term constants in \(\kappa\), out as for MItκ, the elimination rule GMItκ(s) and the introduction rule GMCoitκ(s) for every kind \(\kappa\), and new term reduction rules.

**Inductive type constructors.** Let \(\kappa = \bar{\kappa} \to \ast\).

- **Formation.** \(\mu^\kappa : (\kappa \to \kappa) \to \kappa\)
- **Introduction.** \(\text{in}^\kappa : \forall F^{\kappa \to \kappa}. F(\mu^\kappa F) \subseteq^{\kappa} \mu^\kappa F\)
- **Elimination.**
  \[\Gamma \vdash F : \kappa \to \kappa\]
  \[\Gamma \vdash G : \kappa\]
  \[\Gamma \vdash H : \bar{\kappa} \to \bar{\kappa}\]
  \[\Gamma \vdash s : \forall X^{\kappa}. X \leq^{\kappa} G \to F X \leq^{\kappa} G\]
  \[\Gamma \vdash \text{GMIt}^\kappa(s) : \mu^\kappa F \leq^{\kappa} \mu^\kappa G\]
- **Reduction.** \(\text{GMIt}^\kappa(s) \overset{t}{\to} \beta s \text{ GMIt}^\kappa(s) \overset{t}{\to} t\) where \(|f| = |\bar{\kappa}|\).

**Example 4.1** (summing up a powerlist, revisited). The function \(\text{sum'}\) for powerlists (see Example 3.6) can be naturally implemented with GMItk1. 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 GMItk1 yields a recursive function of type \(\text{PList} \leq^{k_1} \lambda B. \text{Nat} \lambda B. \text{Nat}\), which can be simplified to \(\forall A. (A \to \text{Nat}) \to \text{PList} A \to \text{Nat}\) by removing the void quantification over \(B\).

**Example 4.2** (summing up a bush). Recall the nested datatype of “bushy lists” given in Section 3.3 and first considered in Bird and Meertens [10] within Haskell:

\[
\begin{align*}
\text{BushF} &= \lambda X \lambda A. 1 + A \times X (X A) : k_2, \\
\text{Bush} &= \mu^{k_1} \text{BushF} : k_1, \\
\text{bn} &= \text{in}^{k_1} (\text{inl} (\langle \rangle)) : \forall A. \text{Bush A}, \\
\text{bcons} &= \lambda a \lambda b. \text{in}^{k_1} (\text{inr} (a, b)) : \forall A. A \to \text{Bush (Bush A)} \to \text{Bush A}.
\end{align*}
\]
Similar to powerlists, we can define a summation function \( \text{sum}' \) for bushes:

\[
\begin{align*}
\text{bsum}' & := \text{GMIt}^k (\lambda \text{bsum}' \lambda t. \text{match } t \text{ with} \\
& \quad \text{bnil} \mapsto 0 \\
& \quad \text{bcons} a b \mapsto f a + \text{bsum}' (\text{bsum}' f) b) \\
& : \forall A. (A \rightarrow \text{Nat}) \rightarrow \text{Bush } A \rightarrow \text{Nat}.
\end{align*}
\]

The 0 in the first case is the supposed zero in \( \text{Nat} \). The types would be assigned as follows:

- \( \text{bsum}' : X \leq_k \text{Nat} \)
- \( f : A \rightarrow \text{Nat} \)
- \( t : 1 + A \times X(XA) \)
- \( a : A \)
- \( b : X(XA) \)
- \( \text{bsum}' f : X(XA) \rightarrow \text{Nat} \)
- \( \text{bsum}' (\text{bsum}' f) : X(XA) \rightarrow \text{Nat} \).

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

- \( \text{bsum}' f \text{ bnil} \rightarrow^+ 0 \),
- \( \text{bsum}' f (\text{bcons } a b) \rightarrow^+ f a + \text{bsum}' (\text{bsum}' f) b \).

Note that in the outer recursive call the second argument decreases structurally: on the left-hand side, one has \( \text{bcons } a b \), one the right-hand side only \( b \). But, the inner call \( \text{bsum}' f \) does not specify a second argument. One could force the second argument by \( \eta \)-expanding it to \( \lambda x. \text{bsum}' f x \), which does not help much because the fresh variable \( x \) does not stand in any visible relation to the input \( \text{bcons } a b \). Consequently, proving termination of \( \text{bsum}' \) using a term ordering seems to be problematic, whereas in our system it is just a byproduct of type checking (see Theorem 3.13).

**Coinductive type constructors.** Let \( \kappa = \vec{\kappa} \rightarrow * \).

**Formation.** \( \nu^\kappa : (\kappa \rightarrow \kappa) \rightarrow \kappa \)

**Elimination.** \( \text{out}^\kappa : \forall F^\kappa. \nu^\kappa F \subseteq^\kappa F (\nu^\kappa F) \)

**Introduction.**

- \( \Gamma \vdash F : \kappa \rightarrow \kappa \)
- \( \Gamma \vdash G : \kappa \)
- \( \Gamma \vdash H : \vec{\kappa} \rightarrow \vec{\kappa} \)
- \( \Gamma \vdash s : \forall X^\kappa. G \vec{H} \leq^\kappa X \rightarrow G \vec{H} \leq^\kappa FX \)

**Reduction.** \( \text{out}^\kappa (\text{GMCoit}^\kappa (s) \vec{f} t) \rightarrow^\beta s \text{ GMCoit}^\kappa (s) \vec{f} t \) where \( |\vec{f}| = |\vec{\kappa}| \).

**Example 4.3 (triangle redecoration, revisited).** Using \( \text{GMCoit}^\kappa \), triangle redecoration can be implemented much more concisely. In the context of Example 3.10 we obtain
the function \texttt{redec} directly as follows:

\[
\texttt{redec} := \texttt{GMCoit}^{\text{k1}} \left( \lambda \texttt{f} \texttt{t} \cdot (\texttt{f} \texttt{t}, \texttt{redec} (\texttt{lift} \texttt{f}) (\texttt{rest} \texttt{t})) \right)
\]

Thus, one can enforce the desired reduction behavior without any detours. In \texttt{Mlt}^{\text{co}}, where we implemented triangle redecoration in Example 3.10, we were required to implement an auxiliary function \texttt{redec}' first which used a tagged argument pair \texttt{pack}(\texttt{f}, \texttt{t}). In contrast, the curried version \texttt{redec} above can handle \texttt{f} and \texttt{t} as two separate arguments directly. This leads to a very natural reduction behavior:

\[
\begin{align*}
\text{top} (\texttt{redec} \texttt{f} \texttt{t}) & \rightarrow^+ \texttt{f} \texttt{t}, \\
\text{rest} (\texttt{redec} \texttt{f} \texttt{t}) & \rightarrow^+ \texttt{redec} (\texttt{lift} \texttt{f}) (\texttt{rest} \texttt{t}).
\end{align*}
\]

In Example 3.10, we had the small spot ° in the picture: Instead of \texttt{redec} only \texttt{redec}° appeared on the right-hand side.

4.3. Embedding \texttt{GMlt}^{\text{co}} into \texttt{Mlt}^{\text{co}}

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

For the sake of the embedding, we use a syntactic version of Kan extensions (see [34, Chapter 10]). In conjunction with nested datatypes, Kan extensions appear already in ([11, 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 Examples 3.6 and 3.10 within \texttt{Mlt}^{\text{co}}, special Kan extensions have been used already. The same programming tasks have been accomplished directly within \texttt{GMlt}^{\text{co}} in Examples 4.1 and 4.3. 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 \texttt{GMlt}^{\text{co}}(s) within \texttt{Mlt}^{\text{co}}, and similarly for the source type constructor in the coinductive case.

Compared to Abel and Matthes [1], Kan extensions “along” are now defined for all kinds, not just for rank-1.

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

\[
\begin{align*}
\text{Ran}^{\kappa} G & := G, \\
(\text{Ran}^{\kappa_1 \rightarrow \vec{k}}_{\vec{H}, \vec{H}} G) \vec{X} & := \forall Y^{\kappa_1}. X \leq \kappa_1 HY \rightarrow (\text{Ran}^{\vec{k}}_{\vec{H}} (G Y)) \vec{X}.
\end{align*}
\]

Here, \(\kappa_1 \rightarrow \vec{k}\) is the general format for a composed kind \(\kappa\). Clearly, \(\vec{k} = \kappa_2, \ldots, \kappa_{|\vec{k}|} \rightarrow *\).
Left Kan Extension along $\tilde{H}$. Let again $\kappa = \tilde{k} \to *$ and $\tilde{k}' = \tilde{k} \to \tilde{k}$ and define for $F : \kappa, \tilde{H} : \tilde{k}'$ and $Y : \tilde{k}$ the type $(\text{Lan}^K_{\tilde{H}} F) Y$ by iteration on $|\tilde{k}|$:

$$\text{Lan}^\kappa F := F,$$

$$(\text{Lan}^{K_1 \to K}_{H,\tilde{H}} F) \ Y \ Y := \exists X^{K_1}. \ H X \leq K_1 Y \times (\text{Lan}^K_{\tilde{H}} (FX)) \ Y.$$

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

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

**Lemma 4.4.** Let $\kappa = \tilde{k} \to *$, $F, G : \kappa$ and $\tilde{H} : \tilde{k} \to \tilde{k}$. The following types are logically equivalent:

1. $F \leq_K H G$ and $F \subseteq^K \text{Ran}^K_{\tilde{H}} G$.
2. $F \leq_{\tilde{H}} K G$ and $\text{Lan}^K_{\tilde{H}} F \subseteq^K G$.

**Proof.** For $\kappa = *$, all these types are just $F \to G$. Otherwise, let $\tilde{k}' := \tilde{k} \to \tilde{k}, n := |\tilde{k}|$ and define

$$\leq\text{Ran}^K := \lambda \alpha \beta \gamma \delta. \ g \delta t$$

$$\leq \text{ranLeq}^K := \lambda \alpha \beta \gamma \delta. \ h \gamma r \ h \delta$$

$$\leq \text{leqLeq}^K := \lambda \alpha \beta \gamma \delta. \ f \beta \ (\text{pack}(f_1, \text{pack}(f_2, \ldots, \text{pack}(f_n, t), \ldots))). \ g \delta t$$

$$\leq \text{lanLeq}^K := \lambda \alpha \beta \gamma \delta. \ h \alpha \ f \beta \ (\text{pack}(f_1, \text{pack}(f_2, \ldots, \text{pack}(f_n, t), \ldots)))$$

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

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

$$\text{GMT}^\kappa(s) := \text{ranLeq}^K (\text{ML}^\kappa (\text{leqRan}^K \circ s \circ \text{ranLeq}^K)),$$

$$\text{GMCoit}^\kappa(s) := \text{lanLeq}^K (\text{MCoit}^K (\text{leqLa}^\kappa \circ s \circ \text{lanLeq}^K)).$$

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

$$\delta := \text{leqRan}^K \circ s \circ \text{ranLeq}^K : \forall X^K. \ X \leq_K H G \to F X \leq_K H G.$$

Therefore, $\text{ML}^\kappa(\delta) : \mu^K F \leq_K H G$, finally $\text{GMT}^\kappa(s) : \mu^K F \leq_K H G$. 
We calculate, using the above abbreviation \( \hat{s} \),

\[
\text{GMlt}^\kappa(s) \xrightarrow{\vec{f}} (\text{in}^\kappa t) \xrightarrow{+} \text{Mlt}^\kappa(\hat{s})(\text{in}^\kappa t) \xrightarrow{\vec{f}} \hat{s}\text{Mlt}^\kappa(\hat{s})t \xrightarrow{\vec{f}} s(\text{ranLeq}^\kappa(\text{Mlt}^\kappa(\hat{s}))) f t.
\]

With a similar, but notationally more tedious calculation for the coinductive case, we get, with these definitions, in \( \text{Mlt}^\varnothing \):

\[
\text{GMlt}^\kappa(s) \xrightarrow{\vec{f}} (\text{in}^\kappa t) \xrightarrow{+} s\text{GMlt}^\kappa(s) f t,
\]

\[
\text{out}^\kappa(\text{GMCoit}^\kappa(s) f t) \xrightarrow{+} s\text{GMCoit}^\kappa(s) f t.
\]

Since one step of reduction in \( \text{GMlt}^\kappa \) is replaced by at least one reduction step of the encoding in \( \text{Mlt}^\varnothing \), \( \text{GMlt}^\kappa \) inherits strong normalization of \( \text{Mlt}^\varnothing \). Since the number of steps is fixed for every kind \( \kappa \), in the examples we will just treat \( \text{GMlt}^\kappa \) as a subsystem of \( \text{Mlt}^\varnothing \) in the sense that we assume that both iteration and both coiteration schemes are present in \( \text{Mlt}^\varnothing \) together with their reduction rules.

5. Basic conventional iteration

We are looking for a system of conventional iteration into which we can embed \( \text{Mlt}^\varnothing \) 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 \( \kappa \). The fixed point \( \mu^\kappa F \) would be the carrier of an initial \( F \)-algebra, and \( \text{in}^\kappa : F (\mu^\kappa F) \to \mu^\kappa F \) 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 \( \text{in}^\kappa \) 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 \( F^\varnothing \), but have to concentrate on the types: If a functor \( F \) is applied to a morphism \( s : A \to B \), then the result is a morphism \( Fs : FA \to FB \). If \( F \) is a type constructor, i.e., of kind \( k_1 \), then this can be represented by the existence of a term \( m \) of type:

\[
\text{mon}^{k_1} F := \forall A \forall B. (A \to B) \to FA \to FB,
\]

which is nothing but monotonicity of \( F \). Then, \( s : A \to B \) implies \( ms : FA \to FB \).

The notion \( \text{mon}^{k_1} F \) 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
\[ \text{mon}^{\kappa \rightarrow \kappa'} := \lambda F. \forall X^{\kappa} \forall Y^{\kappa}. X \subseteq^{\kappa} Y \rightarrow F X \subseteq^{\kappa'} F Y : (\kappa \rightarrow \kappa') \rightarrow \ast. \]

This will be our notion of monotonicity of type constructors \( F : \kappa \rightarrow \kappa' \), 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 \( \text{mon}^{\kappa \rightarrow \kappa} F \). As has been observed in Matthes [36], 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 [1], it has been shown that strong normalization holds for inductive types. There, only \( \text{mon}^{k_1} \) enters the definitions. Clearly, \( \text{mon}^{k_1} F = \text{mon}^{k_1} F \).

In the next subsection, we will see that a canonical formulation of (co)iteration, based on basic monotonicity \( \text{mon}^{\kappa \rightarrow \kappa} \) for all kinds \( \kappa \), is a subsystem of \( \text{Mit}^{\kappa} \).

### 5.1. Defining basic conventional iteration in terms of Mendler iteration

Define the basic conventional iterators and coiterators by
\[
\begin{align*}
\text{It}^{\kappa}(m, s) & := \text{Mit}^{\kappa}(\lambda t. m \ it \ (m \ it \ t)), \\
\text{Coit}^{\kappa}(m, s) & := \text{MCoit}^{\kappa}(\lambda s \ m \ coit \ (s \ t)).
\end{align*}
\]

Then, one immediately gets the following derived typing rules:
\[
\begin{align*}
\Gamma \vdash F : \kappa \rightarrow \kappa & \quad \Gamma \vdash m : \text{mon}^{\kappa \rightarrow \kappa} F \quad \Gamma \vdash G : \kappa \quad \Gamma \vdash s : F G \subseteq^{\kappa} G \\
& \quad \Gamma \vdash \text{It}^{\kappa}(m, s) : \mu^{\kappa} F \subseteq^{\kappa} G.
\end{align*}
\]

and reduction behavior as follows:
\[
\begin{align*}
\text{It}^{\kappa}(m, s) (\text{in}^{\kappa} t) & \rightarrow^{+} s (m \ \text{It}^{\kappa}(m, s) t), \\
\text{out}^{\kappa} (\text{Coit}^{\kappa}(m, s) t) & \rightarrow^{+} m \ \text{Coit}^{\kappa}(m, s) (s \ t).
\end{align*}
\]

The interpretation of the typing rule for \( \text{It}^{\kappa}(m, s) \) 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 \( \text{It}^{\kappa}(m, s) \) of type \( \mu^{\kappa} F \subseteq^{\kappa} G \).

Also the reduction behavior is as expected (see [37]): If \( \text{It}^{\kappa}(m, s) \) is given the constructor term \( \text{in}^{\kappa} t \), this reduces to the step term (the \( F \)-algebra) \( s \), applied to the recursive call, where \( m \) organizes how the whole function \( \text{It}^{\kappa}(m, s) \) is applied to the term \( t \) which was the argument to \( \text{in}^{\kappa} \).

The rules for \( \text{Coit}^{\kappa} \) are just found by dualization.

**Example 5.1** (summing up a powerlist, conventional style). We redo Example 3.6 with \( \text{It}^{k_1} \). Again, we define a more general function
\[
\text{sum'} : \forall A. \ \text{PList} \ A \rightarrow (A \rightarrow \text{Nat}) \rightarrow \text{Nat} = \text{PList} \subseteq^{k_1} \ G',
\]
with $G' := \lambda A. (A \to \text{Nat}) \to \text{Nat} : k_1$. This is done as follows:

\[
P\text{ListF} := \lambda X. A. X (A \times A)
\]

\[
p\text{listfb} := \lambda s. \lambda t. \text{match } t \text{ with }
\]

\[
| \text{zero } a \mapsto \text{zero } a \\
| \text{succ } l \mapsto \text{succ } (s l)
\]

\[
: \text{mon}^k \text{PListF}
\]

\[
s := \lambda t \lambda f. \text{match } t \text{ with }
\]

\[
| \text{zero } a \mapsto f a \\
| \text{succ } l \mapsto \text{sum } l (\langle a_1, a_2 \rangle. f a_1 + f a_2)
\]

\[
: \text{PList} \subseteq^k G'
\]

\[
\text{sum}' := \text{It}^k (p\text{listfb}, s)
\]

\[
: \text{PList} \subseteq^k G'.
\]

An easy calculation shows that, as in Example 3.6, we get the reduction behavior

\[
\text{sum}' : \forall A. \text{PList} A \to (A \to \text{Nat}) \to \text{Nat},
\]

\[
\text{sum}' (\text{zero } a) f \rightarrow^+ f a,
\]

\[
\text{sum}' (\text{succ } l) f \rightarrow^+ \text{sum } l (\langle a_1, a_2 \rangle. f a_1 + f a_2).
\]

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

5.2. Definition of $\text{It}^\omega$: basic conventional iteration and coiteration

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

\[
\begin{align*}
\text{Formation.} & \quad \mu^\kappa : (\kappa \to \kappa) \to \kappa \\
\text{Introduction.} & \quad \text{in}^\kappa : \forall F^\kappa \to \kappa, F (\mu^\kappa F) \subseteq^k \mu^\kappa F \\
\text{Elimination.} & \quad \Gamma \vdash F : \kappa \to \kappa \\
& \quad \Gamma \vdash m : \text{mon}^\kappa \to \kappa F \\
& \quad \Gamma \vdash G : \kappa \\
& \quad \Gamma \vdash s : FG \subseteq^k G \\
& \quad \Gamma \vdash \text{It}^\kappa (m, s) : \mu^\kappa F \subseteq^k G \\
\text{Reduction.} & \quad \text{It}^\kappa (m, s) (\text{in}^\kappa t) \rightarrow^\beta s (m \text{ It}^\kappa (m, s) t).
\end{align*}
\]

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

Formation. \[ \text{\text{in}^K} : (\kappa \to \kappa) \to \kappa \]

Elimination. \[ \text{out}^K : \forall F^{\text{in}K} \text{ in}^K F \subseteq \text{in}^K F (\text{in}^K F) \]

Introduction. \[ \text{Id}^K : \text{in}^K F \to \kappa \]

Reduction. \[ \text{\text{in}^K} (\text{\text{Coit}^K}(m, s) t) \to \beta m \text{\text{Coit}^K}(m, s) (s t). \]

Using the definitions in the previous subsection, \( \text{It}^{\text{co}} \) embeds into \( \text{Mlt}^{\text{co}} \). The interesting result, however, is the embedding in the converse direction: \( \text{Mlt}^{\text{co}} \) even embeds into \( \text{It}^{\text{co}} \).

5.3. Embedding Mendler iteration into conventional iteration

Here, we present a somewhat surprising embedding of \( \text{Mlt}^{\text{co}} \) into \( \text{It}^{\text{co}} \). Certainly, there is the embedding through \( F^{\text{co}} \) that polymorphically encodes the (co)inductive constructors (see Section 3.6) and ignores the additional capabilities of \( \text{It}^{\text{co}} \). An interesting embedding has to send the Mendler (co)iterators of \( \text{Mlt}^{\text{co}} \) to the conventional (co)iterators of \( \text{It}^{\text{co}} \). Unlike the embedding of \( \text{GmIt}^{\text{co}} \) into \( \text{Mlt}^{\text{co}} \), 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 \( \kappa := \kappa_0 \to \bar{\kappa} \to \bar{*}, F : \kappa, G : \kappa_0 \) and \( G_i : \kappa_i \) for \( 1 \leq i \leq |\bar{\kappa}| \). The types \((\text{Ran}^K F) G \bar{G}\) and \((\text{Lan}^K F) G \bar{G}\) are defined as follows:

\[
(\text{Ran}^K F) G \bar{G} := \forall Y^{\kappa_0}. G \subseteq^{\kappa_0} Y \to F Y \bar{G},
\]

\[
(\text{Lan}^K F) G \bar{G} := \exists X^{\kappa_0}. G \subseteq^{\kappa_0} X \times F X \bar{G}.
\]

Alternatively, \( \text{Ran}^K \) and \( \text{Lan}^K \) can be seen as type constructors of kind \( \kappa \to \kappa \).

Note that; trivially, always \( \text{Ran} F \subseteq F \) and \( F \subseteq \text{Lan} F \). More precisely, we can define for \( \kappa \neq \bar{*} \)

\[
\text{ranId} : \text{Ran}^K \subseteq^K \text{Id} \quad \text{by} \quad \text{ranId} := \hat{x} \cdot \text{id},
\]

\[
\text{lanId} : \text{Id} \subseteq^K \text{Lan}^K \quad \text{by} \quad \text{lanId} := \hat{x} \cdot \text{pack}(\text{id}, x).
\]

Lemma 5.2. For any \( F, F' : \kappa \to \kappa \) and \( G : \kappa \), we have the following logical equivalences (already in \( F^{\text{co}} \)):

\[
(\forall X^{\kappa}. G \subseteq X \to G \subseteq^{\text{Ran}^{\text{co}K} F} X) \iff G \subseteq (\text{Ran}^{\text{co}K} F) G, \quad (1)
\]

\[
(\forall X^{\kappa} \forall Y^{\kappa}. X \subseteq Y \to F X \subseteq^{\text{Ran}^{\text{co}K} F'} Y) \iff F \subseteq (\text{Ran}^{\text{co}K} F), \quad (2)
\]

\[
\text{mon}^{\text{co}K K'} F \iff F \subseteq (\text{Ran}^{\text{co}K} F'), \quad (3)
\]

\[
(\forall X^{\kappa}. X \subseteq G \to F X \subseteq G) \iff (\text{Ran}^{\text{co}K} F) G \subseteq G, \quad (4)
\]
\( (\forall X^\kappa \forall Y^\kappa. X \subseteq Y \rightarrow F X \subseteq F' Y) \leftrightarrow \text{Lan}^{\kappa \rightarrow \kappa'} F \subseteq F', \quad (5) \)

\( \text{mon}^{\kappa \rightarrow \kappa'} F \leftrightarrow \text{Lan}^{\kappa \rightarrow \kappa'} F \subseteq F. \quad (6) \)

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

The terms \( T := \lambda s \lambda t \lambda f. s f t \) and \( U := \lambda s \lambda (\text{pack}(f, t)). s f t \) in the previous proof will be used in the embedding below.

The following is a crucial property of \( \text{Ht}^\alpha \): There is a uniform proof of monotonicity for all Kan extensions. (Recall that \( \kappa \neq \ast \).)

\[
\begin{align*}
\text{M}^{\kappa}_{\text{Lan}} &:= \lambda g \lambda (\text{pack}(f, t)). \text{pack}(g \circ f, t) : \forall G^\kappa. \text{mon}^\kappa (\text{Lan}^\kappa G), \\
\text{M}^{\kappa}_{\text{Ran}} &:= \lambda g \lambda \lambda f. t (f \circ g) : \forall G^\kappa. \text{mon}^\kappa (\text{Ran}^\kappa G).
\end{align*}
\]

The embedding \( \Gamma \cdot \cdot \) of System \( \text{Mlt}^\omega \) of Mendler iteration and coiteration into \( \text{Ht}^\alpha \) is now straightforward. Kinds are left fixed; the translation of most type-constructor and term formers is homomorphic, e.g., a type application \( \Gamma F G \) is encoded as a type application \( \Gamma F \Gamma G \). Only syntax related to least and greatest fixed points has to be translated non-homomorphically:

- **Formation.**
  \( \Gamma \mu^{\kappa} \cdot \cdot : (\kappa \rightarrow \kappa) \rightarrow \kappa \)
  \[ \Gamma \mu^{\kappa} \cdot \cdot := \mu^{\kappa} \circ \text{Lan}^{\kappa \rightarrow \kappa}. \]

- **Introduction.**
  \( \Gamma \text{in}^{\kappa} \cdot \cdot : \forall F^{\kappa \rightarrow \kappa}. F \Gamma \mu^{\kappa} F \subseteq \mu^{\kappa} (\text{Lan}^{\kappa \rightarrow \kappa} F) \)
  \[ \Gamma \text{in}^{\kappa} \cdot \cdot := \lambda t. \text{in}^{\kappa} (\text{lanId} t). \]

- **Elimination.**
  \[ F : \kappa \rightarrow \kappa \]
  \[ G : \kappa \]
  \[ s : \forall X^\kappa. X \subseteq G \rightarrow F X \subseteq G \]
  \[ \Gamma \text{Mlt}^\kappa (s) \Gamma \subseteq \Gamma G \Gamma \]

\[ \Gamma \text{Mlt}^\kappa (s) \cdot \cdot := \lambda x. \text{Ht}^\kappa (\text{M}^{\kappa}_{\text{Lan}}; \text{U} \Gamma s \cdot \cdot ) x \]

where
\[ \text{U} \Gamma s \cdot \cdot := \text{Lan}^{\kappa \rightarrow \kappa} \Gamma F \Gamma G \subseteq \Gamma G \]

- **Reduction.**
  \( \text{Mlt}^\kappa (s) (\text{in}^\kappa t) \rightarrow s \text{Mlt}^\kappa (s) t. \)

This behavior is simulated by finitely many steps inside \( \text{Ht}^\alpha \):

\[ \Gamma \text{Mlt}^\kappa (s) (\text{in}^\kappa t) \rightarrow^+ \text{Ht}^\kappa (\text{M}^{\kappa}_{\text{Lan}}; \text{U} \Gamma s \cdot \cdot ) (\text{in}^{\kappa} (\text{lanId} \Gamma t \cdot \cdot )) \]

\[ \rightarrow \text{U} \Gamma s \cdot \cdot (\text{M}^{\kappa}_{\text{Lan}}; \text{Ht}^\kappa (\text{M}^{\kappa}_{\text{Lan}}; \text{U} \Gamma s \cdot \cdot )) (\text{lanId} (\Gamma t \cdot \cdot )) \]

\[ \rightarrow \text{U} \Gamma s \cdot \cdot (\text{M}^{\kappa}_{\text{Lan}}; \text{Ht}^\kappa (\text{M}^{\kappa}_{\text{Lan}}; \text{U} \Gamma s \cdot \cdot ) (\text{pack} (\text{id}, \Gamma t \cdot \cdot ))) \]
A. Abel et al. / Theoretical Computer Science 333 (2005) 3–66

$$
\rightarrow^+ \ U \rightarrow^\gamma \ \text{pack}((\text{It}^k(\text{M}_{\text{lan}}^\gamma, \ U \rightarrow^\gamma) \circ \text{id}, \ \gamma) \rightarrow^\gamma)
$$

$$
\rightarrow^+ \ \gamma \rightarrow^\gamma (\text{It}^k(\text{M}_{\text{lan}}^\gamma, \ U \rightarrow^\gamma) \circ \text{id}) \rightarrow^\gamma
$$

$$
\rightarrow \ \gamma \rightarrow^\gamma (\xi. \ \text{It}^k(\text{M}_{\text{lan}}^\gamma, \ U \rightarrow^\gamma).X) \rightarrow^\gamma
$$

$$
\gamma \rightarrow^\gamma (s \ \text{Mlt}^k(s) \rightarrow^\gamma).
$$

Note that the embedding employs a left Kan extension for the inductive case while the embedding of $\text{GMlt}^\omega$ into $\text{Mlt}^\omega$ uses a right Kan extension for that purpose. Hence, there is a real need for the existential quantifier also for inductive constructors alone. We come to the coinductive case:

**Formation.**

$$
\gamma v^\kappa : (k \rightarrow k) \rightarrow k
$$

$$
\gamma v^\kappa := v^k \circ \text{Ran}^{k \rightarrow k}
$$

**Elimination.**

$$
\gamma \text{out}^\kappa : \ \forall \ F^{k \rightarrow k}, \ F^k (\text{Ran}^{k \rightarrow k} F) \subseteq F \ \rightarrow^k F
$$

$$
\gamma \text{out}^\kappa := \lambda r. \ \text{ranId} (\text{out}^k r)
$$

**Introduction.**

$$
F : k \rightarrow k \quad G : k \quad s : \forall X^k. G \subseteq X \rightarrow G \subseteq F X
$$

$$
\gamma \text{MCoit}^k(s) \rightarrow^\gamma : G \rightarrow^k F
$$

$$
\gamma \text{MCoit}^k(s) := \xi x. \ \text{Coit}^k (\text{M}_{\text{ran}}^k, T \rightarrow^\gamma s) \rightarrow^\gamma x \quad \text{where}
$$

$$
T \rightarrow^\gamma s := \gamma G \subseteq \text{Ran}^{k \rightarrow k} F \subseteq \gamma^k F
$$

**Reduction.**

$$
\gamma \text{out}^k (\gamma \text{MCoit}^k(s) \rightarrow^\gamma) \rightarrow^\gamma \text{ranId} (\text{out}^k (\text{Coit}^k (\text{M}_{\text{ran}}^k, T \rightarrow^\gamma s) \rightarrow^\gamma t) (s) t)
$$

$$
\rightarrow^\gamma \text{ranId} (\text{Coit}^k (\text{M}_{\text{ran}}^k, T \rightarrow^\gamma s) (T \rightarrow^\gamma s) \rightarrow^\gamma t)
$$

$$
\rightarrow^\gamma \text{ranId} (\lambda f. T \rightarrow^\gamma s \rightarrow^\gamma t (f \circ \text{Coit}^k (\text{M}_{\text{ran}}^k, T \rightarrow^\gamma s))
$$

$$
\rightarrow^\gamma T \rightarrow^\gamma s \rightarrow^\gamma t (\text{id} \circ \text{Coit}^k (\text{M}_{\text{ran}}^k, T \rightarrow^\gamma s))
$$

$$
\rightarrow^\gamma \gamma \rightarrow^\gamma (\lambda x. \ \text{Coit}^k (\text{M}_{\text{ran}}^k, T \rightarrow^\gamma s).x) \rightarrow^\gamma t
$$

$$
\gamma \rightarrow^\gamma (s \ \text{MCoit}^k(s) \rightarrow^\gamma t).
$$

5.4. **Remarks on monotonicity**

Although sufficient to cover all of $\text{Mlt}^\omega$, our basic notion of monotonicity has several defects which will be overcome in Section 6.

**Lemma 5.3.** There is no closed term of type $\text{mon}^k (\lambda X. X \rightarrow \gamma) \in F^\omega$.

**Proof.** Assume, to the contrary,

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

Assume a type variable $Z$ and consider $X := \lambda A. A \rightarrow \bot$ with $\bot := \forall Z. Z$, and $Y := \lambda A. A \rightarrow Z$. Then, $s := \lambda t \lambda a. t a : \forall A. XA \rightarrow YA$. Therefore, $ms : X(XZ) \rightarrow$
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 \( X(XA) \rightarrow Y(YA) \), 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:

\[
\text{mon}^{\kappa \rightarrow \kappa'} F, \text{mon}^{\kappa} X \text{ imply } \text{mon}^{\kappa'}(FX).
\]

This means monotonicity of a least fixed point \( \mu^\kappa F : \kappa \) is not inherited from monotonicity of \( F : \kappa \rightarrow \kappa \). Still, this notion of monotonicity has been shown to be suitable to define iteration and coiteration in the sense of \( \text{It}^\omega \).

Both non-implications are exemplified with the single example \( F := \lambda \_ \lambda A. \neg A \).

Trivially, \( \text{mon}^{\kappa} F \) is inhabited by \( m_F := \lambda x. \text{id} \).

Certainly, \( FX = \lambda A. \neg A \) is not monotone: If \( m : \text{mon}^{\kappa1}(\lambda A. \neg A) \), then \( m : (\bot \rightarrow A) \rightarrow (\bot \rightarrow \bot) \rightarrow \neg A \), hence, \( m \text{id} : \forall A. \neg A \), and logical inconsistency of \( F^\omega \) ensues. Also \( \mu^\kappa F \) is not monotone because this type constructor is logically equivalent with \( \lambda A. \neg A \): The data constructor \( \text{in}^{\kappa1} \) yields one direction, the other comes from

\[
\text{It}^{\kappa1}(m_F, \text{id}) : (\mu^\kappa F) \subseteq F (\mu^\kappa F) = \forall A. (\mu^\kappa F) A \rightarrow \neg A.
\]

**Remark 5.4 (Another notion of monotonicity).** Consider a modified notion of monotonicity which excludes some more type constructors.

\[
\text{vmon}^{\kappa \rightarrow \kappa'} F := \forall X \exists \tilde{X} \forall Y \exists \tilde{Y} \exists \lambda \lambda' \tilde{X} \subseteq \tilde{X} \tilde{Y} \rightarrow FX \rightarrow FY
\]

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

Although we have shown that monotonicity of \( F : \kappa \rightarrow \kappa \) fails to entail monotonicity of \( \mu^\kappa F \) 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 \( n : \text{mon}^\kappa X \) into a monotonicity witness \( pn : \text{mon}^\kappa(FX) \), then \( \mu^\kappa F : \kappa \) is monotone, canonically. To see this, define

\[
\text{M}^\kappa(p, m) := \lambda f \lambda t. \, \text{It}(m, \lambda t' \lambda f', \text{in}^\kappa(m \text{ ranId} (p \text{ M}_{\text{Lan}}^\kappa f', t'))) \, t \, f.
\]
Then $F : \kappa \rightarrow \kappa$, $p : \forall X^\kappa$, $\text{mon}^\kappa X \rightarrow \text{mon}^\kappa(F X)$ and $m : \text{mon}^{\kappa \rightarrow \kappa}$ imply that the step term $s$ has type $F(\text{Ran}^\kappa(\mu F)) \subseteq^\kappa \text{Ran}^\kappa(\mu F)$ and

$$
\begin{align*}
\text{M}_s^\kappa(p, m) : & \text{mon}^\kappa(\mu F), \\
\text{M}_s^\kappa(p, m) f (\text{in}^\kappa t) & \rightarrow^+ \text{in}^\kappa (m \text{ranId}^\kappa (p \text{M}_s^\kappa f (m \text{ran}(m, s) t)))).
\end{align*}
$$

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 extensionally equal to in$^\kappa (p \text{M}_s^\kappa f (m) f t)$. For general $F$ and $m$, however, this relation does not hold without further assumptions (take $F$ to be a constructor variable and $m$ an object variable assumed to inhabit mon$^{\kappa \rightarrow \kappa}$ $F$).

It is also true that if a monotone $F : \kappa \rightarrow \kappa$ preserves monotonicity, then $\text{in}^\kappa F : \kappa$ is canonically monotone.

6. Refined conventional iteration

Let $\text{Mlt}^\omega_1$ be the restriction of $\text{GMlt}^\omega$ where the vector $\vec{H} : \vec{\kappa}$ of $H$’s in the typing of $\text{GMlt}^\kappa$ and $\text{GMCoit}^\kappa$ only consists of identities $\text{Id}$. Consequently, one changes the name of $\text{GMlt}^\kappa$ to $\text{Mlt}^\kappa_1$ and $\text{GMCoit}^\kappa$ to $\text{MCoit}^\kappa_1$. 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 $\text{Mlt}^\omega_1$. The system will be called $\text{lt}^\omega_0$ and is the system discussed in Abel and Matthes [1]. In Section 7, arbitrary vectors $\vec{H}$ will be reintroduced and a system $\text{Glt}^\omega$, corresponding to the full system $\text{GMlt}^\omega$, will be studied.

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

**Refined monotonicity.** We define $\text{mon}^\kappa := \lambda F. F \subseteq^\kappa F$, hence

$$
\text{mon}^{\kappa \rightarrow \kappa'} = \lambda F. F \subseteq^{\kappa \rightarrow \kappa'} F
$$

The type $\text{mon}^\kappa F$, 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 [24] as well as in Altenkirch and McBride [4]. Contrast this with $\text{mon}^\kappa F$ which asserts that $F$ is monotone in its first argument position, for all argument values.

Notice that for $k! = * \rightarrow *$, the new definition of monotonicity, mon, coincides with the old one, mon. For higher ranks, however, the notions differ considerably. For instance, the type constructor $\lambda X. X \circ X : (\kappa \rightarrow \kappa) \rightarrow \kappa \rightarrow \kappa$, which for $\kappa = *$ we disproved to be monotonic w.r.t. the old notion in Lemma 5.3, is monotonic w.r.t. the new notion:

$$
\lambda e \lambda f. e (e f) : \text{mon}^{(\kappa \rightarrow \kappa) \rightarrow (\kappa \rightarrow \kappa)} (\lambda X \lambda Y. X (X Y)).
$$
Also, the new definition is compatible with application: If \( F : \kappa \rightarrow \kappa' \) and \( X : \kappa \) are monotone, then \( F X : \kappa' \) is monotone as well:

\[
m : \text{mon}^{\kappa \rightarrow \kappa'} F \quad \text{and} \quad n : \text{mon}^\kappa X \implies mn : \text{mon}^{\kappa'}(FX).
\]

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

\[
\begin{align*}
\text{pair} & \;::= \;\lambda f.\lambda g.\lambda(a,b).\;\langle f\;a,\;g\;b \rangle \quad : \;\text{mon}_{s \rightarrow s}^{s \rightarrow s}(\lambda A\lambda B. A \times B), \\
\text{fork} & \;::= \;\lambda f.\;\text{pair}\;f\;f \quad : \;\text{mon}_{s \rightarrow s}^{s \rightarrow s}(\lambda A. A \times A), \\
\text{either} & \;:: = \;\lambda f.\lambda g.\lambda x.\;\text{case}(x,\;a.\;\text{inl}(f\;a),\;b.\;\text{inr}(g\;b)) \quad : \;\text{mon}_{s \rightarrow s}^{s \rightarrow s}(\lambda A\lambda B. A + B), \\
\text{maybe} & \;:: = \;\text{either id} \quad : \;\text{mon}_{s \rightarrow s}^{s \rightarrow s}(\lambda A. C + A).
\end{align*}
\]

In the definition of \textit{maybe}, we assume that \( A \) does not occur free in \( C \).

6.1. Definition of \( W^\omega \)

The system \( L^\omega \) is an extension of \( F^\omega \) specified by the following typing and reduction rules.

**Inductive constructors.** Let \( \kappa = \kappa \rightarrow \ast \).

- **Formation.** \( \mu^\kappa : (\kappa \rightarrow \kappa) \rightarrow \kappa \)
- **Introduction.** \( \text{in}^\kappa : \forall F^{\kappa \rightarrow \kappa}. F (\mu^\kappa F) \subseteq^\kappa \mu^\kappa F \)
- **Elimination.** \( \Gamma \vdash F : \kappa \rightarrow \kappa \)
  \( \Gamma \vdash m : \text{mon}^{\kappa \rightarrow \kappa} F \)
  \( \Gamma \vdash G : \kappa \)
  \( \Gamma \vdash s : FG \subseteq^\kappa G \)
  \( \Gamma \vdash \text{lt}^\kappa(m, s) : \mu^\kappa F \subseteq^\kappa G \)

**Reduction.** \( \text{lt}^\kappa(m, s) \rightarrow^\beta s(m \text{lt}^\kappa(m, s) \rightarrow t) \)

where \( |f| = |\kappa| \).

This system is “conventional” in the sense that \( L^\omega \) is conventional: Iteration is only possible in the presence of a monotonicity witness \( m \), being our representation of functoriality. And the argument \( s \) to \( \text{lt}^\kappa \) (the “step term”) has the type \( FG \subseteq^\kappa G \), making \( (G, s) \) the syntactic representation of an \( F \)-algebra. Somewhat surprisingly, the type of \( \text{lt}^\kappa(m, s) \) is not \( \mu^\kappa F \subseteq^\kappa G \), hence there seems to be a mismatch: The type of \( \text{in}^\kappa \) 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 \( F_1 \subseteq^s F_2 \). 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 6.1** (monotonicity of least fixed points). If type constructor \( F : \kappa \rightarrow \kappa \) is monotone, witnessed by \( m : \text{mon}^{\kappa \rightarrow \kappa} F \), then \( \mu^\kappa F \) is again monotone, witnessed by

\[
\begin{align*}
M^\kappa(m) & \;::= \;\text{lt}^\kappa(m, \text{in}^\kappa) \; : \; \text{mon}^{\kappa}(\mu^\kappa F), \quad \text{where} \\
M^\kappa(m) \;\rightarrow^\beta \text{in}^\kappa(m \;M^\kappa(m) \;\rightarrow t).
\end{align*}
\]
Proof. Directly by instantiation of the typing and reduction rules for $\text{It}^\kappa$. □

Remark 6.2 (alternative introduction rule). To overcome the mismatch between step term and iterator mentioned above, one might accept the type of the iterator term $\text{It}^\kappa(m, s)$ as it stands, but would use $\subseteq^\kappa$ instead of $\subseteq^\kappa$ for the types of data constructor $\text{in}^\kappa$ and step term $s$. In fact, this was the typing rule underlying the original submission of Abel and Matthes [1] and a similar typing rule was suggested to us also by Peter Aczel in May 2003. However, a data constructor of type $\text{in}^\kappa : F(\mu^\kappa F) \subseteq^\kappa \mu^\kappa F$ would have the drawback that the canonical inhabitants of higher-order inductive types would be of the form $\text{in}^\kappa \vec{g} t$, where the $g_i$ are functions. As a consequence, a single data object could have several, even infinitely many distinct normal forms. For instance, $\text{in}^{\mu^1} (\lambda n. n + 5) (\text{inl} 10)$ and $\text{in}^{\mu^1} \text{id} (\text{inl} 15)$ 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 $\kappa = \tilde{\kappa} \rightarrow \ast$.

- Formation. $\nu^\kappa : (\kappa \rightarrow \kappa) \rightarrow \kappa$
- Elimination. $\text{out}^\kappa : \forall F : \kappa \rightarrow \kappa. \nu^\kappa F \subseteq^\kappa F (\nu^\kappa F)$
- Introduction. $\Gamma \vdash F : \kappa \rightarrow \kappa$
  $\Gamma \vdash m : \text{mon}^\kappa \rightarrow \kappa F$
  $\Gamma \vdash G : \kappa$
  $\Gamma \vdash s : G \subseteq^\kappa F G$
  $\Gamma \vdash \text{Coit}^\kappa(m, s) : G \subseteq^\kappa \nu^\kappa F$
- Reduction. $\text{out}^\kappa (\text{Coit}^\kappa(m, s) \tilde{f} t) \rightarrow^\beta m \text{Coit}^\kappa(m, s) \tilde{f} (s t)$

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

Lemma 6.3 (monotonicity of greatest fixed points). If type constructor $F : \kappa \rightarrow \kappa$ is monotone, witnessed by $m : \text{mon}^\kappa \rightarrow \kappa F$, then $\nu^\kappa F$ is again monotone, witnessed by

$$M^\kappa_m(m) := \text{Coit}^\kappa(m, \text{out}^\kappa) : \text{mon}^\kappa(\nu^\kappa F), \text{ where}$$
$$\text{out}^\kappa (M^\kappa_m(m) \tilde{f} t) \rightarrow^\beta m M^\kappa_m(m) \tilde{f} (\text{out}^\kappa t).$$

6.2. Examples

The following two developments exemplify the use of $M^\kappa_m$, i.e., the preservation of monotonicity under formation of least fixed points. More examples for programming in $\text{It}^\omega$, also with $\text{Coit}^\kappa$, can be found in Abel and Matthes [1].

Example 6.4 (free variable renaming for de Bruijn terms). The free variables of a de Bruijn term may be renamed by the canonical monotonicity witness of $\text{Lam}$, called $\text{lam}$ below:

$$\text{lam} := \lambda e \lambda \vec{f} e. \text{either } f \text{ (either (fork } (e f)) (e \text{ (maybe } f))) : \text{mon}^{\mu^1} \text{LamF},$$
$$\text{lam} := M^\mu_\beta(\text{lamf}) : \text{mon}^{\mu^1} \text{Lam}.$$
The reduction behavior shows that we have indeed obtained the mapping function for de Bruijn terms:

\[
\begin{align*}
\text{lam } f \ (\text{var } a) & \rightarrow^{+} \text{var}^{\circ} (f \ a), \\
\text{lam } f \ (\text{app } t_1 t_2) & \rightarrow^{+} \text{app}^{\circ} (\text{lam } f \ t_1) (\text{lam } f \ t_2), \\
\text{lam } f \ (\text{abs } r) & \rightarrow^{+} \text{abs}^{\circ} (\text{lam } (\text{maybe } f) \ r).
\end{align*}
\]

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.

\[
\text{weak} := \text{lam } \text{inr} : \forall A. \text{Lam } A \rightarrow \text{Lam } (1 + A).
\]

**Example 6.5 (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

\[
\begin{align*}
\text{plistf} & := \lambda e \lambda f. \text{either } f \ (e \ (\text{fork } f)) : \text{mon}^{k_2} \text{PListF}, \\
\text{plist} & := \text{M}^{k_1}_{\mu} (\text{plistf}) : \text{mon}^{k_1} \text{PList}.
\end{align*}
\]

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

\[
\begin{align*}
\text{swap} & := \lambda f \lambda (a_1, a_2). \ (f \ a_2, \ f \ a_1) : \text{mon}^{k_1} (\lambda A. A \times A), \\
\text{revf} & := \lambda e \lambda f. \text{either } f \ (e \ (\text{swap } f)) : \text{mon}^{k_2} \text{PListF}, \\
\text{rev'} & := \text{M}^{k_1}_{\mu} (\text{revf}) : \text{mon}^{k_1} \text{PList}, \\
\text{rev} & := \text{rev'} \ \text{id} : \text{PList} \subseteq^{k_1} \text{PList},
\end{align*}
\]

Specializing \(f\) to \(\text{id}\) in the reduction rules, it becomes traceable that \(\text{rev}\) reverses a powerlist.

**6.3. Embedding \(\text{lt}^{\omega}_{\mu}\) into \(\text{mit}^{\omega}_{\mu}\)**

The iterator and coiterator of \(\text{lt}^{\omega}_{\mu}\) are definable within \(\text{mit}^{\omega}_{\mu}\) (see the beginning of this section) so that the typing rules are obeyed and reduction is simulated. We define

\[
\begin{align*}
\text{lt}^{\kappa}_{\mu} (m, s) & := \text{Mlt}^{\kappa}_{\mu} (\lambda \text{it} \lambda \tilde{f} \lambda t. s \ (m \ \text{it} \ \tilde{f} \ t)), \\
\text{coit}^{\kappa}_{\mu} (m, s) & := \text{Mcoit}^{\kappa}_{\mu} (\lambda \text{coit} \lambda \tilde{f} \lambda t. m \ \text{coit} \ \tilde{f} \ (s \ t)).
\end{align*}
\]

Embedding \(\text{mit}^{\omega}_{\mu}\) into \(\text{lt}^{\omega}_{\mu}\) in a typing- and reduction-preserving way seems to be impossible, except for the uninformative embedding through \(F^{\omega}\).
7. Generalized refined conventional iteration

Similar to Mit\textsuperscript{\textregistered}\textsuperscript{afii9853}, it is also possible to define a conventional-style counterpart to GMIt\textsuperscript{\textregistered}afii9853. We will now present a system GIt\textsuperscript{\textregistered}afii9853 that accomplishes this. One important aspect is that the efficient folds of Martin et al. [35] are directly definable in this system. This will be shown later on in Section 9.

7.1. Definition of system GIt\textsuperscript{\textregistered}

System GIt\textsuperscript{\textregistered}afii9853 recasts the generality of System GMIt\textsuperscript{\textregistered}afii9853 following the design of System It\textsuperscript{\textregistered}afii9853. It generalizes It\textsuperscript{\textregistered}afii9853 in two directions: \( \leq \) is generalized to \( \leq \vec{H} \), and an additional type constructor parameter \( F' : \kappa \rightarrow \kappa \) appearing in the type of \( m \) adds further flexibility. Compared to It\textsuperscript{\textregistered}afii9853, only the typing rules are changed; the reduction rules are the same. Significantly, the term \( m \) in GIt\textsuperscript{\textregistered}afii9853((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 It\textsuperscript{\textregistered}afii9853: 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\textsuperscript{\textregistered} is specified by the following constants and typing and reduction rules.

**Inductive constructors.** Let \( \kappa = \vec{k} \rightarrow \ast \).

- Formation. \( \mu^\kappa : (\kappa \rightarrow \kappa) \rightarrow \kappa \)
- Introduction. \( \text{in}^\kappa : \forall F^{\kappa \rightarrow \kappa}, F (\mu^\kappa F) \subseteq^\kappa \mu^\kappa F \)

- Elimination. \( \Gamma \vdash F, F' : \kappa \rightarrow \kappa \)
  \( \Gamma \vdash \vec{H} : \vec{k} \rightarrow \vec{k} \)
  \( \Gamma \vdash m : \forall X^\kappa \forall Y^\kappa, X \leq^\vec{H} Y \rightarrow F X \leq^\vec{H} F' Y \)
  \( \Gamma \vdash G : \kappa \)
  \( \Gamma \vdash s : F' G \subseteq^\kappa G \)

\[ \Gamma \vdash \text{GIt}^\kappa(m, s) : \mu^\kappa F \leq^H \vec{k} G \]

- Reduction. \( \text{GIt}^\kappa(m, s) \vec{f} (\text{in}^\kappa t) \rightarrow^\beta s (m \text{GIt}^\kappa(m, s) \vec{f} t) \)
  where \( |\vec{f}| = |\vec{k}| \).

**Coinductive constructors.** Let \( \kappa = \vec{k} \rightarrow \ast \).

- Formation. \( \nu^\kappa : (\kappa \rightarrow \kappa) \rightarrow \kappa \)

- Elimination. \( \text{out}^\kappa : \forall F^{\kappa \rightarrow \kappa}, \nu^\kappa F \subseteq^\kappa F (\nu^\kappa F) \)

- Introduction. \( \Gamma \vdash F' : F : \kappa \rightarrow \kappa \)
  \( \Gamma \vdash \vec{H} : \vec{k} \rightarrow \vec{k} \)
  \( \Gamma \vdash m : \forall X^\kappa \forall Y^\kappa, X \leq^\vec{H} Y \rightarrow F' X \leq^\vec{H} F Y \)
  \( \Gamma \vdash G : \kappa \)
  \( \Gamma \vdash s : G \subseteq^\kappa F' G \)

\[ \Gamma \vdash \text{GCoit}^\kappa(m, s) : G \leq^H \vec{k} \nu^\kappa F \]

- Reduction. \( \text{GCoit}^\kappa(m, s) \vec{f} t \rightarrow^\beta m \text{GCoit}^\kappa(m, s) \vec{f} (s t) \)
  where \( |\vec{f}| = |\vec{k}| \).
Evidently, it\textsuperscript{3}\textsuperscript{4} is (apart from different names for iterators and coiterators) just the special case with $F' = F$ and $H = \text{Id}$.

### 7.2. Examples

We demonstrate programming in GIt\textsuperscript{3}\textsuperscript{4} on two examples.

**Example 7.1** (*summing up a powerlist, revisited*). In GIt\textsuperscript{3}\textsuperscript{4}, the implementation of the summation of a powerlist in system GMIt\textsuperscript{3}\textsuperscript{4} (Example 4.1) can be closely mimicked. We can define

$$\text{sum}' := \text{GIt}^{k_1}(m, s) : \text{PList} \leq_{H}^{k_1} G = \forall A \forall \_ : (A \rightarrow \text{Nat}) \rightarrow \text{PList} A \rightarrow \text{Nat},$$

where

- $G := \lambda \_. \text{Nat} : k_1$
- $H := \lambda \_. \text{Nat} : k_1$
- $Q := \text{anything} : k_1$
- $F' := \lambda X \lambda A. \text{Nat} + X (Q A) : k_2$
- $m := \lambda e \lambda f. \text{either } f \left(e \left(\lambda (a_1, a_2). f a_1 + f a_2\right)\right)$
  $$: \forall X \forall Y. X \leq_{\lambda \_. \text{Nat}}^{k_1} Y \rightarrow \text{PList} F X \leq_{\lambda \_. \text{Nat}}^{k_1} F' Y$$
  $$= \forall X \forall Y. \left(\forall A' \forall B. (A' \rightarrow \text{Nat}) \rightarrow X A' \rightarrow Y B\right)$$
  $$\rightarrow \forall A \forall \_B'. (A \rightarrow \text{Nat}) \rightarrow A + X (A \times A) \rightarrow \text{Nat} + Y (Q B')$$
- $s := \lambda t. \text{match } t \text{ with } \text{inl } n \mapsto n \mid \text{inr } n \mapsto n$
- $F' (\lambda \_. \text{Nat}) \subseteq_{k_1}^{\lambda \_. \text{Nat} = \forall \_. \text{Nat} + \text{Nat} \rightarrow \text{Nat}.$

The reduction behavior is precisely that of $\text{sum}'$ in Example 4.1, 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 $k_1$ 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 9.3, $Q$ will be chosen in a canonical way.)

We see that one should not at all think of real monotonicity witnesses in GIt\textsuperscript{3}\textsuperscript{4}. The pseudo-monotonicity witnesses are meant to do work specific to the programming task at hand. Example 6.5 of powerlist reversal by means of a noncanonical monotonicity witness demonstrated this idea as well.

The next example shows that the iterator of GIt\textsuperscript{3}\textsuperscript{4} turns out to be very handy when one wants to move on from variable renaming in de Bruijn terms to substitution.

**Example 7.2** (*substitution for de Bruijn terms*). In GIt\textsuperscript{3}\textsuperscript{4}, 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 [5]:

\[
\text{lift} \ := \ \lambda f. \lambda x. \ \text{case} (x, \ u. \ \text{var} (\text{inl} \ u), \ a. \ \text{weak} (f \ a)) \\
\text{subst} \ := \ \text{GIt}^{k_1}(m, s) \\
\text{where}
\]

where

\[
F' \ := \ \lambda X \lambda A. \ \text{Lam} \ A + (X A \times X A + X (1 + A)) : k_2 \\
m \ := \ \lambda e \lambda f. \ \text{either} \ f \ \left(\text{either} (\text{fork} (e \ f)) \ (e \ \text{lift} \ f)\right) \\
\text{subst} \ := \ \lambda t. \ \text{match} \ t \ \text{with} \ \text{inl} \ u \mapsto u \ | \ \text{inr} \ t' \mapsto \text{inl}^{k_1} (\text{inr} \ t') \\
s \ := \ \lambda f. \ \text{subst} (\text{lift} \ f) \ r.
\]

Note that we use weakening weak of Example 6.4 in the definition of the lifting function lift, implicitly embedding It\textsuperscript{\omega} into GIt\textsuperscript{\omega}. The program subst has exactly the expected reduction behavior in the sense that, if \( f : A \rightarrow \text{Lam} \ B \) is a substitution rule, then subst \( f : \text{Lam} \ A \rightarrow \text{Lam} \ B \) behaves as the corresponding substitution function replacing the variables of its de Bruijn term argument according to the rule \( f \):

\[
\text{subst} \ f \ (\text{var} \ a) \rightarrow^+ f \ a, \\
\text{subst} \ f \ (\text{app} \ t_1 \ t_2) \rightarrow^+ \text{app}^o (\text{subst} \ f \ t_1) (\text{subst} \ f \ t_2), \\
\text{subst} \ f \ (\text{abs} \ r) \rightarrow^+ \text{abs}^o (\text{subst} (\text{lift} \ f) \ r).
\]

Alternatively, one might program substitution within It\textsuperscript{\omega}, but this would necessitate an explicit use of a right Kan extension—a fact swept under the carpet in Altenkirch and Reus [5]. In GIt\textsuperscript{\omega}, this more liberal format is part of the design.

7.3. Embeddings between GIt\textsuperscript{\omega} and GMIt\textsuperscript{\omega}

GIt\textsuperscript{\omega} embeds into GMIt\textsuperscript{\omega} much the same way as It\textsuperscript{\omega} embeds into MIt\textsuperscript{\omega}:

\[
\text{GIt}^k(m, s) \ := \ \text{GMIt}^k(\lambda \text{it} \lambda \text{f} \lambda \text{t}. \ s \ (\text{m it} \ \text{f} \ \text{t})), \\
\text{GCoit}^k(m, s) \ := \ \text{GMCoit}^k(\lambda \text{coit} \lambda \text{f} \lambda \text{t}. \ m \ \text{coit} \ \text{f} \ (s \ t)).
\]
The embedding of \( \text{GMLt}^\omega \) into \( \text{Glt}^\omega \) is much more interesting and again just a definitional embedding:

\[
\begin{align*}
\text{GMLt}^\omega(s) &:= \text{Glt}^\omega(\text{lanLeq}^{K\to K} \text{id}, \text{leqLan}^{K\to K} s), \\
\text{GMCoi}t^\omega(s) &:= \text{GCoit}^\omega(\text{ranLeq}^{K\to K} \text{id}, \text{leqRan}^{K\to K} s).
\end{align*}
\]

Here, we used the definitions in the proof of Lemma 4.4. It is easy to check that, with these definitions,

\[
\begin{align*}
\text{GMLt}^\omega(s) &\xrightarrow{\tilde{f}} (\text{in}^K t) \quad \rightarrow^+ s \text{GMLt}^\omega(s) \xrightarrow{\tilde{f}} t, \\
\text{out}^K(\text{GMCoi}t^\omega(s) \xrightarrow{\tilde{f}} t) &\rightarrow^+ s \text{GMCoi}t^\omega(s) \xrightarrow{\tilde{f}} t.
\end{align*}
\]

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

**Lemma 7.3.** Assume \( \kappa = \overline{\kappa} \to \ast, n = |\overline{\kappa}|, \overline{\kappa}' = \overline{\kappa} \to \overline{\kappa}, F : \kappa \to \kappa \) and \( \overline{H} : \overline{\kappa}' \). Define the constructor

\[
F' := \lambda Y \lambda \overline{Y} \exists X^K. X \subseteq^K \overline{H} Y \times \text{Lan}^K(F X)(\overline{H} \overline{Y}) : \kappa \to \kappa.
\]

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

\[
\begin{align*}
\text{lanLeq}^{K\to K} &:= \forall X^K \forall Y^K. F' Y \subseteq^K F' Y \to X \subseteq^K \overline{H} Y \to F X \subseteq^K \overline{H} F' Y, \\
\text{leqLan}^{K\to K} &:= \forall G^K. (\forall X^K. X \subseteq^K \overline{H} G \to F X \subseteq^K \overline{H} G) \to F' G \subseteq^K G.
\end{align*}
\]

Redefine the constructor \( F' \) to be

\[
F' := \lambda X \lambda \overline{X} \forall Y^K. X \subseteq^K \overline{H} Y \to \text{Ran}^K(F Y)(\overline{H} \overline{X}).
\]

Then, types can be assigned as follows:

\[
\begin{align*}
\text{ranLeq}^{K\to K} &:= \forall X^K \forall Y^K. F' X \subseteq^K F' X \to X \subseteq^K \overline{H} Y \to F' X \subseteq^K \overline{H} F' Y, \\
\text{leqRan}^{K\to K} &:= \forall G^K. (\forall X^K. G \subseteq^K \overline{H} X \to G \subseteq^K F X) \to G \subseteq^K F' G.
\end{align*}
\]

**Proof.** By simple unfolding of the definitions of \( \text{Lan}^K \) and \( \text{Ran}^K \). Observe that the vector \( \overline{H} \) enters only the arguments; the Kan extensions are not formed along \( \overline{H} \). The premises \( F' Y \subseteq^K F' Y \) and \( F' X \subseteq^K F' X \) are there just for perfect fit with the definitions of \( \text{lanLeq} \) and \( \text{ranLeq} \). They will later always be instantiated by \( \text{id} \). □

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 \( \text{Glt}^\omega \). For the inductive case, it is the definition involving \( \text{Lan}^K \), for the coinductive case, \( F' \) needs \( \text{Ran}^K \).

While all of \( \text{GMLt}^\omega \) can be embedded into \( \text{Glt}^\omega \)—using canonical definitions of \( F' \) and canonical terms \( m \) which do not have an interesting operational meaning—we have seen in the Example 7.1 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 \( \text{Mlt}^{\varnothing} \), we do the following examples in \( \text{Mlt}^{\varnothing} \) and freely use the iteration schemes from everywhere (since, for \( \rightarrow^+ \), there is no difference between the original systems and the embeddings). Therefore, we can also use every definition from the previous examples.

Example 8.1 (explicit substitutions). Examples 6.4 and 7.2 have shown that de Bruijn terms constitute a Kleisli triple \( (\text{Lam}, \text{var}, \text{subst}) \) with unit \( \text{var} : \forall A. A \rightarrow \text{Lam} A \) and bind operation

\[
\text{subst} : \forall A B. (A \rightarrow \text{Lam} B) \rightarrow (\text{Lam} A \rightarrow \text{Lam} B).
\]

From the “Kleisli triple” formulation of \( \text{Lam} \) we mechanically get the “monad” formulation \( (\text{Lam}, \text{var}, \text{flatten}) \) with \( \text{flatten} : \forall A. \text{Lam} (\text{Lam} A) \rightarrow \text{Lam} A \) can be obtained from \( \text{subst} \) as \( \text{flatten} := \text{subst} \text{id} \).

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

\[
\begin{align*}
\hat{\text{LamF}} &:= \lambda X \lambda A. \text{LamF} X A + X (X A) & : k2, \\
\hat{\text{Lam}} &:= \mu^{k1} \hat{\text{LamF}} & : k1, \\
\hat{\text{var}} &:= \hat{\lambda} a. \text{ink}^{k1} (\text{inl} (\text{inl} a)) & : \forall A. A \rightarrow \hat{\text{Lam}} A, \\
\hat{\text{app}} &:= \hat{\lambda} t_1 \hat{\lambda} t_2. \text{ink}^{k1} (\text{inl} (\text{inr} (\text{inl} (t_1, t_2)))) & : \forall A. \hat{\text{Lam}} A \rightarrow \hat{\text{Lam}} A \rightarrow \hat{\text{Lam}} A, \\
\hat{\text{abs}} &:= \hat{\lambda} r. \text{ink}^{k1} (\text{inl} (\text{inr} r)) & : \forall A. \hat{\text{Lam}} (1 + A) \rightarrow \hat{\text{Lam}} A, \\
\hat{\text{flat}} &:= \hat{\lambda} e. \text{ink}^{k1} (\text{inr} e) & : \forall A. \hat{\text{Lam}} (\hat{\text{Lam}} A) \rightarrow \hat{\text{Lam}} A.
\end{align*}
\]

Renaming of free variables in a term is implemented by the canonical monotonicity witness of \( \hat{\text{Lam}} \), derived from the following generic monotonicity witness \( \text{lambf} \) for the datatype functor \( \hat{\text{LamF}} \), using \( \text{lambf} \) from Example 6.4:

\[
\begin{align*}
\hat{\text{lambf}} &:= \hat{\lambda} g \lambda f. \hat{\text{either}} (\text{lambf} g f) (g (g f)) & : \text{mon}^{k2} \hat{\text{LamF}}, \\
\hat{\text{lamlam}} &:= \text{M}^{k1}_\text{lt} (\text{lambf}) = \text{lt}^{k1} (\text{lambf}, \text{ink}^{k1}) & : \text{mon}^{k1} \hat{\text{Lam}}.
\end{align*}
\]

Note that the treatment of explicit flattening in the definition of \( \hat{\text{lambf}} \) would be impossible with basic monotonicity \( \text{mon}^{k1} \), see Lemma 5.3. The following reduction behavior

\[2\] This presentation should be compared with the slightly unmotivated extension of de Bruijn’s notation in Bird and Paterson ([12], Section 5) where \( \text{abs} \) and \( \text{flat} \) are replaced by just one constructor of type \( \forall A. \hat{\text{Lam}} (1 + \hat{\text{Lam}} A) \rightarrow \hat{\text{Lam}} A \), which again gives rise to true nesting. However, that constructor could easily be defined as \( \text{flat} \circ \text{abs} \) in the present system.
immediately follows:

\[
\begin{align*}
\text{\texttt{\textit{lam}}} f (\texttt{\textit{var}}^o a) & \rightarrow^+ \texttt{\textit{var}}^o (f a), \\
\text{\texttt{\textit{lam}}} f (\texttt{\textit{app}}^o t_1 t_2) & \rightarrow^+ \texttt{\textit{app}}^o (\texttt{\textit{lam}} f t_1) (\texttt{\textit{lam}} f t_2), \\
\text{\texttt{\textit{lam}}} f (\texttt{\textit{abs}}^o r) & \rightarrow^+ \texttt{\textit{abs}}^o (\texttt{\textit{lam}} (\texttt{\textit{maybe}} f) r), \\
\text{\texttt{\textit{lam}}} f (\texttt{\textit{flat}}^o e) & \rightarrow^+ \texttt{\textit{flat}}^o (\texttt{\textit{lam}} (\texttt{\textit{lam}} f) e).
\end{align*}
\]

As was the case with the summation for the truly nested datatype Bush in Example 4.2, the termination of \texttt{\textit{lam}} is not obvious at all: In the recursive call in the last line, the parameter \(f\) is changed to \(\texttt{\textit{lam}} f\), hence using the whole iteratively defined function \(\texttt{\textit{lam}} f\). Nevertheless, termination holds by strong normalization of \(\text{MIt}^o\).

Using \(\text{\texttt{\textit{lam}}}\), we can represent full explicit substitution

\[
\text{\texttt{\textit{esubst}}} : \forall A \forall B. (A \rightarrow \text{\texttt{\textit{Lam}}} B) \rightarrow (\text{\texttt{\textit{Lam}}} A \rightarrow \text{\texttt{\textit{Lam}}} B),
\]

\[
\text{\texttt{\textit{esubst}}} := \lambda f \lambda t. \texttt{\textit{flat}} (\texttt{\textit{lam}} f t).
\]

\texttt{\textit{esubst}} is explicit substitution in the sense that a term of the form \(\texttt{\textit{flat}} (r)\) is returned for \(\text{\texttt{\textit{esubst}}} ft\), hence only renaming but no substitution is carried out.

Alternatively, one can represent full explicit substitution by way of a data constructor like \(\texttt{\textit{flat}}\). If we redefine \(\text{\texttt{\textit{Lam}}}\) to be

\[
\begin{align*}
\texttt{\textit{Lam}} & := \mu^{k_1} (\lambda X \lambda A. \text{\texttt{\textit{LamF}}} X A + \exists B. (B \rightarrow X A) \times X B),
\end{align*}
\]

we may set \(\texttt{\textit{exs}} := \lambda f \lambda t. \text{\texttt{\textit{in}}}^{k_1} (\\text{\texttt{\textit{inr}}} (\text{\texttt{\textit{pack}}} (f, t)))\), which (after exchanging the bound variables \(A\) and \(B\)) receives the type we have above for \(\texttt{\textit{esubst}}\). Note that \(\exists B. (B \rightarrow X A) \times X B = (\text{\texttt{\textit{Lam}}}^{k_1} X) (X A)\), with the naive Kan extension defined in Section 5.3. From Lemma 5.2, it follows that \((\text{\texttt{\textit{Lam}}}^{k_1} X) (X A)\) and \(X (X A)\) are logically equivalent if \(X\) is monotone. Since the fixed-point \(\text{\texttt{\textit{Lam}}}\) 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 8.2 (resolution of explicit substitutions).** The set of de Bruijn terms \(\text{\texttt{\textit{Lam}}}\) can be embedded into the set of de Bruijn terms \(\texttt{\textit{Lam}}\) with explicit flattening. The embedding function \(\texttt{\textit{emb}} : \forall A. \text{\texttt{\textit{Lam}}} A \rightarrow \text{\texttt{\textit{Lam}}} A\) can be defined by iteration in a straightforward manner. The other direction is handled by a function \(\text{\texttt{eval}} : \forall A. \text{\texttt{\textit{Lam}}} A \rightarrow \text{\texttt{\textit{Lam}}} A\) which has to resolve the explicit flattenings. With the help of \(\texttt{\textit{Itk}}^{k_1}\), this is defined by

\[
\begin{align*}
\text{\texttt{eval}}' & := \text{\texttt{\textit{Itk}}}^{k_1} (\text{\texttt{\textit{lamf}}}, s) : \text{\texttt{\textit{Lam}}} \leq^{k_1} \text{\texttt{\textit{Lam}}}, \\
\text{\texttt{eval}} & := \text{\texttt{\textit{eval}}} \ \text{\texttt{\textit{id}}} : \text{\texttt{\textit{Lam}}} \leq^{k_1} \text{\texttt{\textit{Lam}}},
\end{align*}
\]

where \(s := \lambda t. \text{\texttt{\textit{case}}} (t, t', \text{\texttt{\textit{in}}}^{k_1} t', e. \text{\texttt{\textit{flatten}}} e) : \texttt{\textit{LamF}} \text{\texttt{\textit{Lam}}} \leq^{k_1} \text{\texttt{\textit{Lam}}},\) with flatten taken from the previous example. The most interesting case of the reduction behavior is

\[
\text{\texttt{eval}}' f (\text{\texttt{\textit{flat}}} e) \rightarrow^+ \text{\texttt{\textit{flatten}}} (\text{\texttt{\textit{eval}}} (\text{\texttt{\textit{eval}}} f) e).
\]

As in the last example, the nesting in the definition of datatype \(\texttt{\textit{Lam}}\) is reflected in the nested recursion in \(\text{\texttt{eval}}'\).
Example 8.3 (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 \( E : \ast \) of matrix elements. The type \( \text{FTri} A \) of finite triangular matrices with diagonal elements in \( A \) and ordinary elements \( E \) can be obtained as follows:

\[
\text{FTri} := \lambda X \lambda A. A \times (1 + X (E \times A)) : k2,
\]
\[
\text{FTri} := \mu^{k1} \text{FTri} : k1.
\]

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

\[
\text{sg} := \lambda a. \text{in}^{k1} \langle a, \text{inl} \rangle : \forall A. A \rightarrow \text{FTri} A, \quad \text{and}
\]
\[
\text{cons} := \lambda a \lambda r. \text{in}^{k1} \langle a, \text{inr} r \rangle : \forall A. A \rightarrow \text{FTri} (E \times A) \rightarrow \text{FTri} A.
\]

There are two monotonicity witnesses for \( \text{FTriF} \), for \( \text{mon}^{k2} \) and for \( \text{mon}^{k2} \):

\[
\text{ftrif} := \lambda g. \text{pair id (maybe } g) : \text{mon}^{k2},
\]
\[
\text{ftrif} := \lambda g \lambda f. \text{pair } f \text{(maybe } (g \text{(pair id } f))) : \text{mon}^{k2},
\]
\[
\text{ftri} := \text{M}_{\mu}^{k1} (\text{ftrif}) : \text{mon}^{k1} \text{FTri}.
\]

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

\[
\text{ftop} : \forall A. \text{FTri} A \rightarrow A,
\]
\[
\text{ftop} := \text{It}^{k1} (\text{ftrif}, \text{fst}) id.\]

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

\[
\text{ftop (sg } a) \rightarrow^+ a,
\]
\[
\text{ftop (cons } a r) \rightarrow^+ a.
\]

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

$$\text{fcut'} : \forall A \forall B. (A \to E \times B) \to \text{FTri} A \to \text{FTri} B,$$
$$\text{fcut} := \text{GMIt}_{k1}(\lambda \text{fcut'} \lambda f. \text{pair} (\text{snd} \circ f) (\text{maybe} (\text{fcut'} (\text{pair} \text{id} f)))),$$

$$\text{fcut'} f (\text{sg} a) \to^+ \text{sg}^\circ (\text{snd} (f a)),$$
$$\text{fcut'} f (\text{cons} a r) \to^+ \text{cons}^\circ (\text{snd} (f a)) (\text{fcut'} (\text{pair} \text{id} f) r).$$

The cut function is obtained by specializing $f$ to the identity:

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

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

$$\text{fcut} (\text{sg} \langle e, a \rangle) \to^+ a, \text{ and}$$
$$\text{fcut} (\text{cons} \langle e, a \rangle r) \to^+ \text{cons} a (\text{fcut'} (\text{pair} \text{id} id) r), \text{ but}$$
$$\text{fcut} (\text{cons} \langle e, a \rangle r) \not\to^+ \text{cons} a (\text{fcut} r).$$

However, if one added a tiny bit of extensionality, one would have extensional equality of $\text{pair} \text{id} \text{id}$ and $\text{id}$, 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 3.10, but with the new auxiliary functions.

$$\text{flift} : \forall A \forall B. (\text{FTri} A \to B) \to \text{FTri} (E \times A) \to E \times B,$$
$$\text{flift} := \lambda f \lambda t. (\text{fst} (\text{ftop} t), f (\text{fcut} t)).$$

Finally, we can define redecoration $\text{fredec}$. Its description is the same as that for $\text{redec}$ that works on infinite triangles in Example 3.10. The only difference is that we swapped its arguments such that its type now is

$$\forall A. \text{FTri} A \to \forall B. (\text{FTri} A \to B) \to \text{FTri} B = \text{FTri} \subseteq_{k1} G,$$
where $\forall B. (\text{FTri} A \to B) \to \text{FTri} B =: G$.

Unfortunately, $G$ is not a right Kan extension (which might have allowed a direct definition of $\text{fredec}$ inside $\text{GMIt}^{(\omega)}$), but $G = (\text{ran}_{k1} \text{FTri}) \circ \text{FTri}$. Moreover, $\text{fredec}$ 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 $\text{fredec'} : \text{FTri} \subseteq_{k1} \text{FTri} \times_{k1} G$. It seems that any hard-wired Kan extensions in the system would only complicate the following definition which is done
in plain \texttt{Mlt}.

\[
\begin{align*}
\text{fredec'} : & \forall A. \text{FTri} A \to (\text{FTri} A \times (\forall B. (\text{FTri} A \to B) \to \text{FTri} B)), \\
\text{fredec'} & : \text{Mlt}^1(\lambda t. \text{fredec'} t).
\end{align*}
\]

\[
\begin{align*}
\text{let } fid & = \text{fst} \circ \text{fredec'} \text{ in} \\
\text{let } \text{fredec} & = \text{snd} \circ \text{fredec'} \text{ in} \\
\text{let } r & = \text{in}^1(\text{ftrif} \text{ fid} t) \text{ in} \\
\{ r, \lambda f. \text{match } t \text{ with} \\
\ & \text{sg}_- x \mapsto \text{sg} (f r) \\
\ & \text{cons}_- x \mapsto \text{cons} (f r) (\text{fredec} x (\text{flift} f)) \}.
\end{align*}
\]

The function \text{fredec'} actually defines two functions simultaneously:

\[
\begin{align*}
\text{fid} & : \text{FTri} \subseteq^1 \text{FTri}, \\
\text{an iterative identity on finite triangles, and} \\
\text{fredec} & : \text{FTri} \subseteq^1 G,
\end{align*}
\]

the actual redecoration function. The iterative identity is needed to reconstruct the current function argument \(r\) from its unfolded version \(t\). Why a simple "\(r = \text{in}^1 t\)" does not do the job can be seen from the types of the bound variables:

\[
\begin{align*}
\text{fredec'} : & X \subseteq^1 \text{FTri} \times^1 G \\
\text{t} : & \text{FTriF} X A = A \times (1 + X (E \times A)) \\
\text{fid} : & X \subseteq^1 \text{FTri} \\
\text{r} : & \text{FTri} A \\
\text{x} : & X (E \times A) \\
\text{fredec} : & X (E \times A) \to (\text{FTri} (E \times A) \to (E \times B)) \to \text{FTri} (E \times B).
\end{align*}
\]

Since in the Mendler discipline \(t\) is \textit{not} of type \text{FTriF} \text{FTri} \(A\), we cannot apply \text{in}^1 \text{t} \text{ directly}, but need a conversion function of type \(X \subseteq^1 \text{FTri}\). The functionality of \text{fredec'} can be understood through its reduction behavior:

\[
\begin{align*}
\text{fredec'} (\text{sg} a) & \longrightarrow^+ [\text{sg}^\circ a, \lambda f. \text{sg} (f (\text{sg}^\circ a))], \\
\text{fredec'} (\text{cons} a x) & \longrightarrow^+ [\text{cons}^\circ a (\text{fid} x), \\
\ & \lambda f. \text{cons} (f (\text{cons}^\circ a (\text{fid} x))) (\text{fredec} x (\text{flift} f))].
\end{align*}
\]

Untangling the two intertwined functions within \text{fredec'}, we get the following reduction relations from which correctness of the implementation becomes apparent:

\[
\begin{align*}
\text{fid} (\text{sg} a) & \longrightarrow^+ \text{sg}^\circ a, \\
\text{fid} (\text{cons} a x) & \longrightarrow^+ \text{cons}^\circ a (\text{fid} x), \\
\text{fredec} (\text{sg} a) f & \longrightarrow^+ \text{sg} (f (\text{sg}^\circ a)), \\
\text{fredec} (\text{cons} a x) f & \longrightarrow^+ \text{cons} (f (\text{cons}^\circ a (\text{fid} x))) (\text{fredec} x (\text{flift} f)).
\end{align*}
\]
Since \( \text{fid} \) is an iteratively defined identity, \( \text{fid} \ x \) with \( x \) a variable does not reduce to \( x \). Apart from this deficiency, which could be overcome if a scheme of \emph{primitive recursion} was available, \text{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 [10] were the first to publish iteration schemes for nested datatypes, called simple folds, which correspond to our System \text{It}^{09}. To overcome their limited usability, Bird and Paterson [11] formulated \emph{generalized folds}. Their proposal inspired our work on System \text{GMI}^{09}, 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 [21] proposed an alternative system of folds for nested datatypes. Inspired from that, Martin et al. [35] presented their \emph{efficient folds}, or \text{efolds} for short, which are closest to Bird and Paterson’s generalized folds.

All of the above-mentioned approaches only deal with least fixed points \( \mu F \) of special type constructors \( F : k2 \) of rank 2, which are called \emph{hofunctors} (short for \emph{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 et al. [35]. Their \text{efolds} can be expressed in System \text{GMI}^{09}.

\textbf{Hofunctors.} Following Martin et al. [35], hofunctors are type constructors \( F : k2 \) of one of the following shapes:

\begin{itemize}
  \item[(a)] \( \lambda_. \; Q \) with \( mQ : \text{mon}^{k1} \) (note \( Q : k1 \)) \quad \text{constant}
  \item[(b)] \( \lambda X. X \) \quad \text{identity}
  \item[(c)] \( F0 +^{k2} F1 \) with \( F0, F1 \) hofunctors \quad \text{disjoint sum}
  \item[(d)] \( F0 \times^{k2} F1 \) with \( F0, F1 \) hofunctors \quad \text{product}
  \item[(e)] \( \lambda X. Q \circ (F0 \; X) \) with \( mQ : \text{mon}^{k1} \) and \( F0 \) hofunctor \quad \text{composition}
  \item[(f)] \( \lambda X. X \circ (F0 \; X) \) with \( F0 \) hofunctor \quad \text{nesting}.
\end{itemize}

Note that this inductive characterization is not deterministic, e.g., one can always apply rule (e) with \( Q = \text{Id} \) without modifying the hofunctor extensionally. Even more, case (b) is a special case of (f) with \( F0 = \lambda_. \; \text{Id} \). Probably, case (b) is present in Martin et al. [35] in order to characterize non-nested hofunctors by rules (a)–(e).

**Example 9.1 (hofunctors).** All of the type constructors \( F : k2 \) whose fixed points we considered in the previous examples are hofunctors. For instance,

\[
\text{PListF} := \lambda X\lambda A. \; A + X \; (A \times A) \\
= (\lambda_. \; \text{Id}) +^{k2} \lambda X. \; X \circ (((\lambda_. \; \text{Id}) \times^{k2} (\lambda_. \; \text{Id})) \; X),
\]
BushF := \lambda X . A . 1 + X . (X A) \\
= (\lambda _1 . X) + ^{k_2} \lambda X . X \circ ((\lambda X . X) X),

LamF := \lambda X . A . A + ((X A \times X A) + X . (1 + A)) \\
= (\lambda Id . X) + ^{k_2} ((Id \times ^{k_2} Id) + ^{k_2} \lambda X . X \circ ((\lambda . A . 1 + A) X)).

Efficient folds are another means to construct functions of type $\mu^{k_1} F \subseteq^{k_1} G$ which eliminate inhabitants of the nested datatype $\mu^{k_1} F$. As in the previous sections, $F : k_2$ and $G, H : k_1$, but now $F$ additionally needs to be a hofunctor. By induction on the generation of hofunctor $F$ we define a type constructor $F^F_H : k_2$ which is parametric in $H$, a sequence of types $\vec{D}_F^F_H$ all of which are parametric in $H$, and a term

$$M^F(\vec{d}) : \forall X . \forall Y . X \subseteq^{k_1}_H Y \rightarrow F X \subseteq^{k_1}_H F^F_H Y,$$

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

**Elimination.**

\[ \Gamma \vdash F : k_2 \text{ hofunctor} \]
\[ \Gamma \vdash H : k_1 \]
\[ \Gamma \vdash \vec{d} : \vec{D}_F^F_H \]
\[ \Gamma \vdash G : k_1 \]
\[ \Gamma \vdash s : F^F_H G \subseteq^{k_1}_H G \]
\[ \frac{}{\Gamma \vdash \text{efold}^F(\vec{d}, s) : \mu^{k_1} F \subseteq^{k_1}_H G} \]

**Reduction.**

\[ \text{efold}^F(\vec{d}, s) f (\text{in}^{k_1} t) \longrightarrow^\beta s (M^F(\vec{d}) \text{ efold}^F(\vec{d}, s) f t). \]

**Embedding into GIt$^{k_0}$.** Efficient folds are simply an instance of generalized conventional iteration for kind $k_1$:

$$\text{efold}^F(\vec{d}, s) := \text{GIt}^{k_1}(M^F(\vec{d}), s).$$

To see that this definition preserves typing and reduction, recall the $k_1$ elimination and computation rule for generalized conventional iteration:

**Elimination.**

\[ \Gamma \vdash m : \forall X . \forall Y . X \subseteq^{k_1}_H Y \rightarrow F X \subseteq^{k_1}_H F' Y \]
\[ \Gamma \vdash s : F' G \subseteq^{k_1}_H G \]
\[ \frac{}{\Gamma \vdash \text{GIt}^{k_1}(m, s) : \mu^{k_1} F \subseteq^{k_1}_H G} \]

**Reduction.**

\[ \text{GIt}^{k_1}(m, s) f (\text{in}^{k_1} t) \longrightarrow^\beta s (m \text{ GIt}^{k_1}(m, s) f t). \]

The free parameter $F' : k_2$ in the elimination rule is instantiated by the type constructor $F^F_H$ generated from $F$, and $m$ is replaced by $M^F(\vec{d})$, which assembles the simpler terms $\vec{d}$ into a pseudo-monotonicity witness. Hence, efficient folds can be viewed as a user interface for GIt$^{k_1}$, 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 $\vec{D}^F_H$ 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. Fig. 2 lists Martin et al.’s [35] choices of $F^F_H$, $\vec{D}^F_H$ 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 $F^F_0$, not recursively via $F^F_{H}$. This is due to the emission of a new term $d$.

**Example 9.2 (efolds for powerlists, typing).** Recall that the general typing rule for efold was

$$
\begin{align*}
\Gamma \vdash d : \vec{D}^F_H \\
\Gamma \vdash s : F^F_H G \subseteq^k_1 G \\
\Gamma \vdash \text{efold}^F(d, s) : \mu^k_1 F \subseteq^k_1 H G
\end{align*}
$$
where we took the freedom to omit the kinding judgments 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 $D^F_H$. For powerlists, we obtain the following instances:

$$\begin{align*}
\text{PList}_F &= \lambda X. A + X (A \times A), \\
\text{FPList}_F &= \lambda X. H A + X (A \times A), \\
\overrightarrow{D}\text{PList}_F &= (\forall X \forall. H A \times H A \to H (A \times A)).
\end{align*}$$

Instantiating the general rule for efficient folds and expanding the definitions of $\subseteq^{k_1}$ and $\leq^{k_1}_H$, we obtain efficient folds for powerlists:

$$\begin{align*}
\Gamma \vdash d : \forall A. H A \times H A \to H (A \times A) \\
\Gamma \vdash s : \forall A. H A + G (A \times A) \to G A
\end{align*}$$

$$\Gamma \vdash \text{efold}_{\text{PList}}(d, s) : \forall A \forall B. (A \to H B) \to \text{PList} A \to G B.$$  

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

**Example 9.3** (summing up a powerlist, typing). We can define function $\text{sum}'$ of Example 7.1 using efficient folds for powerlists. We set $G := H := \lambda_.\text{Nat}$, as in the previous implementations of $\text{sum}'$, and

$$\begin{align*}
\text{sum}' &:= \text{efold}_{\text{PList}}(d, s) : \forall A. (A \to \text{Nat}) \to \text{PList} A \to \text{Nat}, \\
\; d &:= \lambda(n, m). n + m : \text{Nat} \times \text{Nat} \to \text{Nat}, \quad \text{and} \\
\; s &:= \lambda x. \text{case} (x, n, n, n) : \text{Nat} + \text{Nat} \to \text{Nat}.
\end{align*}$$

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

As in the implementation using $G\text{It}_{k_1}$ in Example 7.1, the task of the step term $s$ is trivial whereas the addition happens in the other term. Back then, we could use $F' := \lambda X. A. \text{Nat} + X (Q A)$ with $Q$ any constructor in $k_1$. With the special choice $F' := \text{FPList}_F$, which the given definition of efficient folds takes, $Q$ is fixed to $\lambda A. A \times A$.

**Example 9.4** (efolds for powerlists, reduction). Let $m_{\text{Id}} := \lambda f \lambda x. f x$ be the canonical monotonicity witness of $\text{Id} : k_1$. We can compute the pseudo-monotonicity witness for efficient powerlist folds according to Fig. 2:

$$\begin{align*}
\text{M}_{\text{PList}}(d) : \forall X \forall Y. X \leq^{k_1} Y \to \text{PList} X \leq^{k_1} \text{PList} Y \\
= \forall X \forall Y. X \leq^{k_1} Y \to \forall A \forall B. (A \to H B) \to A + X (A \times A) \to H B + Y (B \times B)
\end{align*}$$

$$\begin{align*}
\text{M}_{\text{PList}}(d) e f &:= \text{either} (m_{\text{Id}} f) (e (d \circ \text{pair} (m_{\text{Id}} f) (m_{\text{Id}} f))) \\
&\quad \to^+ \text{either} (\lambda x. f x) (e (d \circ \text{fork} f)).
\end{align*}$$
Note that fork is defined in terms of pair\(^\circ\) on page 39. Using this pseudo-monotonicity witness in the general reduction rule for efolds, which was

\[
efold^F (\vec{d}, s) \cdot f \cdot (\text{in}^{kl} t) \rightarrow_{\beta} s \cdot (M^F (\vec{d}) \cdot \efold^F (\vec{d}, s) \cdot f \cdot t),
\]

we get the following reduction behavior for powerlist efolds:

\[
\efold^{\text{PList}}^F (\vec{d}, s) \cdot f \cdot (\text{zero} \cdot a) \rightarrow^+ s \cdot (\text{inl} \cdot (f \cdot a)) = s \cdot (\text{zero}^- \cdot (f \cdot a)),
\]

\[
\efold^{\text{PList}}^F (\vec{d}, s) \cdot f \cdot (\text{succ} \cdot l) \rightarrow^+ s \cdot (\text{inr} \cdot (\efold^{\text{PList}}^F (\vec{d}, s) \cdot (d \circ (\text{fork}^\circ \cdot f)) \cdot l)) = s \cdot (\text{succ}^- \cdot (\efold^{\text{PList}}^F (\vec{d}, s) \cdot (d \circ (\text{fork}^\circ \cdot f)) \cdot l)).
\]

**Example 9.5** *(summing up a powerlist, reduction).* Instantiating the above scheme for \(\text{sum}' := \efold^{\text{PList}}^F (d, s)\), where

\[
d := \lambda (n, m). n + m,
\]

\[
s := \lambda x. \text{case} \cdot (x, n, n, n),
\]

we obtain the precise reduction behavior of Example 7.1.

**Example 9.6** *(efolds for de Bruijn terms).* As shown in Example 9.1, the generating type constructor \(\text{LamF}\) for de Bruijn terms is a hofunctor. Hence, we can calculate \(F^\text{LamF}_H\) and \(\vec{D}^\text{LamF}_H\) according to Fig. 2:

\[
\begin{align*}
\text{LamF} &= \lambda X \lambda A. A + (X \cdot A \times X \cdot A + X \cdot (1 + A)), \\
F^\text{LamF}_H &= \lambda X \lambda A. H \cdot A + (X \cdot A \times X \cdot A + X \cdot (1 + A)), \\
\vec{D}^\text{LamF}_H &= (\forall A. H \cdot A \rightarrow H \cdot A, \\
&\quad \forall A. H \cdot A \rightarrow H \cdot A, \\
&\quad \forall X \forall A. 1 + H \cdot A \rightarrow H \cdot (1 + A)).
\end{align*}
\]

The first two components of \(\vec{D}^\text{LamF}_H\) arise from the two homogeneous applications \(X \cdot A\) in \(\text{LamF}\), the third from the heterogeneous application \(X \cdot (1 + A)\). The efficient fold for de Bruijn terms is typed as follows:

\[
\begin{align*}
\Gamma \vdash d_1 & : \forall A. H \cdot A \rightarrow H \cdot A \\
\Gamma \vdash d_2 & : \forall A. H \cdot A \rightarrow H \cdot A \\
\Gamma \vdash d_3 & : \forall A. 1 + H \cdot A \rightarrow H \cdot (1 + A) \\
\Gamma \vdash s & : \forall A. H \cdot A + (G \cdot A \times G \cdot A + G \cdot (1 + A)) \rightarrow G \cdot A
\end{align*}
\]

\[
\Gamma \vdash \text{efold}^{\text{LamF}} (d_1, d_2, d_3, s) : \forall A \forall B. (A \rightarrow H \cdot B) \rightarrow \text{Lam} \cdot A \rightarrow G \cdot B.
\]

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

\[
M^{\text{LamF}} (d_1, d_2, d_3) \cdot e \cdot f = \text{either} \cdot (\text{mId} \cdot f) \\
\quad \quad \text{either} \cdot (\text{pair} \cdot (e \cdot (d_1 \circ f)) \cdot (e \cdot (d_2 \circ f))))
\]
A. Abel et al. / Theoretical Computer Science 333 (2005) 3–66

\begin{equation}
(e \circ (\text{maybe } f))).
\end{equation}

Setting \( e := \text{efoldLamF}(d_1, d_2, d_3, s) \) this yields the following reduction behavior for this special efficient fold.

\begin{align*}
\text{ef} & (\text{var } a) \rightarrow^+ s \left( \text{inl} (f a) \right) \\
& = s \left( \text{var}^- (f a) \right), \\
\text{ef} & (\text{app } t_1 t_2) \\
& \\
& \\
\text{ef} & (\text{abs } r) \rightarrow^+ s \left( \text{inr} \left( \text{inr} \left( e (d_3 \circ (\text{maybe } f)) r \right) \right) \right) \\
& = s \left( \text{abs}^- (e (d_3 \circ (\text{maybe } f)) r) \right).
\end{align*}

Example 9.7 (renaming and substitution for de Bruijn terms). The functions \text{lam} of Example 6.4 and \text{subst} of Example 7.2 can be expressed with efficient folds as

\begin{align*}
\text{lam} & := \text{efoldLamF}(\text{id}, \text{id}, \text{id}, \text{ink}^\text{1}) : \text{Lam} \leqslant \text{id} \text{Lam}, \\
\text{subst} & := \text{efoldLamF}(\text{id}, \text{id}, \text{lift id}, s) : \text{Lam} \leqslant \text{Lam} \text{Lam},
\end{align*}

where we use \text{lift} and \text{s} from Example 7.2.

Since composition with the identity \( \eta \)-expands terms, the reduction behavior of \text{lam} is not exactly as in Example 6.4. This problem is even more visible for \text{subst}:

\begin{align*}
\text{subst } f & (\text{var } a) \rightarrow^+ f a, \\
\text{subst } f & (\text{app } t_1 t_2) \rightarrow^+ \text{app} \left( \text{subst} \left( \lambda x. f x \right) t_1 \right) \left( \text{subst} \left( \lambda x. f x \right) t_2 \right), \\
\text{subst } f & (\text{abs } r) \rightarrow^+ \text{abs} \left( \text{subst} \left( \lambda x. \text{lift id} \left( \text{maybe } f x \right) \right) r \right).
\end{align*}

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

A second look at efficient folds. As we pointed out before, Fig. 2 describes just one possible definition of efficient folds. One might wonder whether it could not be simplified at all. The first case worth a discussion is (b) identity: Is it really necessary to emit a distributivity witness \( d : H \subseteq^\text{kl} H \) here? This question has been raised already by Bird and Paterson [11, Section 4.1] for their version of generalized folds. In the last example, these terms are just instantiated with the identity \text{id}. So supposedly, they could be dropped, leading to the simpler definition

\begin{equation}
M^{\lambda x.x} e f := e f.
\end{equation}

Another questionable clause is (e) composition. As mentioned in the beginning of this section, clause (e) can be iterated with \( Q := \text{id} \) in the proof of hofunctional 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):

\[
\begin{align*}
F &= \lambda X. Q \circ (F_0 X), \\
F^F_H &= \lambda X. Q \circ (F^F_0 X), \\
D^F_H &= \overline{D^F_0}, \\
M^F(\overline{d}) e f &= m_Q (M^F_0(\overline{d}) e f).
\end{align*}
\]

In contrast to the original definition, this variant defines $F^F_H$ recursively through $F^F_0 H X$. Iteration of these clauses with $Q = \text{id}$ 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 $H = \text{id}$, it holds that $F^F_H = F$, $D^F_H$ is empty, $M^F$ is the canonical monotonicity witness of $F$ and the eliminator $\text{efold}^F$ is identical to $\text{It}^{k_1}$.

**Comparison with Bird and Paterson** [11]. Whilst for a fixed hofunctor $F : k_2$, the efficient fold “e” is of type $\mu^{kl} F \leq_{k_1} H G$, Bird and Paterson define a generalized fold “g” of type $(\mu^{kl} F) \circ H \leq_{k_1} G$. As observed by Martin et al. [35], both kinds of folds are interdefinable, extensionally:

\[
\begin{align*}
g &= e \text{id}, \\
e &= \lambda f. g \circ (m_{\mu F} f),
\end{align*}
\]

where $m_{\mu F}$ is the canonical monotonicity witness of $\mu^{kl} F$. 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 $\text{GMLt}^{\omega}$. Recall the reduction rule for efficient folds:

\[
\beta \text{- red: } e f (\text{ink}^{k_1} t) \rightarrow s (M^F(\overline{d}) e f t).
\]

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

Provided $e : \forall A Y B. (A \rightarrow H B) \rightarrow X A \rightarrow Y B$
and $f : A \rightarrow H B$,

\[
\begin{align*}
M^F(...) e f &= F X A \rightarrow F^F_H Y B \\
&= X (F_0 X A) \rightarrow Y (F_0 Y B), \\
M^F(...) e f' &= e f',
\end{align*}
\]

where $f' : F_0 X A \rightarrow H (F_0 Y B)$
f' := \ldots
In order to obtain the reduction behavior of Bird and Paterson [11] we need to change the definition of $M^n(\ldots)e\ f\ \text{id} \circ (m_{\mu F}\ f')$. 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 $GM^n$ due to their inefficient reduction behavior. Whether they can be simulated in System $F^\omega$ 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 et al. [7]. The tutorial also includes a description of the generic programming language PolyP [28]. Typically for generic programming, as well as for Jay’s Constructor Calculus [29], admissible $F$’s are built in a combinatorial, i.e., $\lambda$-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 $F : \kappa \rightarrow \kappa$. 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 $\text{It}^\omega$, 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 non-generic powerlist reversal (Example 6.5) uses some monotonicity witness that cannot be found generically.

**Type classes.** Norell and Jansson [42] describe an implementation of the polytypic programming language PolyP within the Haskell programming language, using the type-class mechanism [51]. 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 implementations of the dictionary functions. Most importantly, the type system allows to provide an

³ This even holds for the related work on nested datatypes we mentioned in Section 9.
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 `Summable` so that type $A$ belongs to it if there is an implementation of the function $\text{sum} : A \rightarrow \text{Nat}$. (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 $\text{PList} A$. The argument $f : A \rightarrow \text{Nat}$ to $\text{sum'}$ in Example 3.6 is no longer needed because one can just take $\text{sum}$ for type $A$. On the other hand, the freedom to manipulate $f$ is also lost, and no function $\lambda(a_1, a_2). f a_1 + f a_2$ can be given as an additional argument. Fortunately, $\text{sum} a_1 + \text{sum} a_2$ is precisely the function $\text{sum}$ for type $A \times A$. Finally, $\text{sum}$ at type $\text{PList Nat}$ 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 4.2). In terms of type class `Summable` it just requires that for summable $A$, also $\text{Bush } A$ is summable. The crucial definition clause is then $\text{sum} (\text{beons } a b) := \text{sum } a + \text{sum } b$. The first summand uses the assumed function $\text{sum}$ for type $A$ (which used to be $f$ in that example), the second one uses polymorphic recursion: the term $b$ is of type $\text{Bush } (\text{Bush } A)$, hence the same definition of $\text{sum}$ is invoked with $\text{Bush } A$ in place of $A$. Its hidden argument $f$ is therefore $\text{sum } f$, in accordance with the reduction behavior shown in Example 4.2. Again, no termination guarantee is provided by this implementation. Moreover, $\text{bsum'}$ in the example works for arbitrary types $A$ as soon as a function $f : A \rightarrow \text{Nat}$ 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—$\text{bsum'} f$—in the recursive call may be modified to, e.g., $\text{bsum'} (\lambda x. f x + 1)$, keeping typability and thus termination. Using type class `Summable`, there is just no room for such a non-generic 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 $a$:

```haskell
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 [14] 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 [24]. The idea of this calculation roughly follows the idea of the type-class mechanism, e.g., the function associated with \( \text{PList} : * \to * \) takes any function for any type \( A \) and yields a function for the type \( \text{PList} A \), i.e., \( \tau(\text{PList}) = \forall A. \tau(A) \to \tau(\text{PList} A) \). 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 \( \mu^{\kappa_2}(FG) \) 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 [27], using a model of the kinded type constructor system based on applicative structures. The most recent presentation of Generic Haskell is given by Hinze and Jeuring [26]; there is also a collection containing the three most interesting examples [25], among which the generalization of the trie data structure to datatypes of all kinds [22]. 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 [25, Section 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 \( \text{F}^0 \). 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 instantiatied 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 \( \text{GIt}^0 \), hence with a generic definition for every specific task at hand; recall that real work is also delegated

---

4 Intensional type analysis [16] 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 [52] 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.
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 $\text{F}_\omega$, 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 [4]. The operational behavior of the datatypes thus obtained has to be studied further. Interestingly, a non-trivial part of programming with dependent types can be simulated within Haskell [38,13].

**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 $\text{F}_\omega$ is undecidable [53]. Nevertheless, we have chosen the annotation-free Curry style due to its succinctness. 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, 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 [18,31,32]. Type abstraction in Haskell is only partly solved in Neubauer and Thiemann [41] 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” [33]. 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 8.1, where the least fixed point of $\lambda X \lambda A. \ldots + \ldots$ 

\[ F_{\omega 0.10 \alpha}, \text{by Abel, available on his homepage.} \]
$X(XA)$ is considered. This datatype would be called “doubly nested” in Bird and Paterson [12], and in general, true nesting is called “non-linear nests” in Hinze [23]. Okasaki [43] considers the fixed point (called \textit{square}) of

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

with $V, W : \text{kl}$ and $A$ some fixed type. 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 [19], 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 [23] contains plenty of examples where higher-order parameters are varied in the recursive calls to the datatype being defined, but nowhere “true nesting”. Truly nested datatypes may seem to be esoteric, but they occur naturally in the representation of explicit substitution (see Example 8.1), 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 [22] serve this purpose.

\textbf{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” [8]. 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 [49], more details are to be found in Wadler [50], and interleaved positive (co)inductive types are treated in Altenkirch [3]. An interesting new field is the connection between parametric equality and generic programming: Bookhouse and Hoogendijk [6] show that, under reasonable naturality and functoriality assumptions on the family of \textit{zip} functions, exactly one such \textit{zip} exists for all “regular relators”.

\section{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 $\text{It}_{\alpha}^\omega$, $\text{It}_{=\omega}^\omega$ and $\text{GIt}_{=\omega}^\omega$, 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 $\text{MIt}_{\alpha}^\omega$, $\text{MIt}_{=\omega}^\omega$ and $\text{GMIIt}_{=\omega}^\omega$.

Of the systems considered here, $\text{GMIIt}_{\alpha}^\omega$ and $\text{GIt}_{\alpha}^\omega$ are clearly the most advanced in terms of direct expressive power. In particular, this is witnessed by the fact that the efolds of Martin et al. [35] are very straightforwardly defined in $\text{GIt}_{\alpha}^\omega$. But for many applications, the more basic $\text{It}_{\alpha}^\omega$ and $\text{MIIt}_{\alpha}^\omega$ are perfectly sufficient and invoking $\text{GMIIt}_{\alpha}^\omega$ or $\text{GIt}_{\alpha}^\omega$ is simply not necessary. There are many interesting cases where the more basic systems $\text{It}_{\alpha}^\omega$ and $\text{MIIt}_{\alpha}^\omega$ do not suffice to express the algorithmic idea appropriately, but where the freedom in choosing the additional parameters $\vec{H}$ in both $\text{GMIIt}_{\alpha}^\omega$ and $\text{GIt}_{\alpha}^\omega$, and parameter $F'$ in $\text{GIt}_{\alpha}^\omega$, is not needed. These more rigid typings are embodied in the intermediary systems $\text{MIIt}_{=\omega}^\omega$ and $\text{It}_{=\omega}^\omega$, which have exactly the reduction behavior of their “generalized” versions $\text{GMIIt}_{\alpha}^\omega$ and $\text{GIt}_{\alpha}^\omega$ but are easier to typecheck in practice.

In $\text{GMIIt}_{\alpha}^\omega$ and $\text{MIIt}_{\alpha}^\omega$, where the iterator and coiterator are Mendler style, their computational behavior is very close to \texttt{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 3.8, 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 $\text{MIIt}_{\alpha}^\omega$, but constantly use the expressive capabilities of the other systems, viewed as macro definitions on top of $\text{MIIt}_{\alpha}^\omega$.

Acknowledgements

Peter Hancock deserves thanks for suggesting to us the definition $\text{mon } F := F \leq F$. 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.

Abel acknowledges the support by both the Ph.D. Program Logic in Computer Science (GKLI) of the Deutsche Forschungs-Gemeinschaft and the project CoVer of the Swedish Foundation of Strategic Research. Uustalu 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


