Generalized Iteration and Coiteration for Higher-Order Nested Datatypes

Andreas Abel[1]*, Ralph Matthes[2] and Tarmo Uustalu[3]**

[1] Department of Computer Science, University of Munich
abel@informatik.uni-muenchen.de
[2] Preuves, Programmes et Systèmes,
CNRS, Université Paris VII (on leave from University of Munich)
matthes@informatik.uni-muenchen.de
[3] Inst. of Cybernetics, Tallinn Technical University
tarmo@cs.ioc.ee

Abstract. We solve the problem of extending Bird and Paterson's generalized folds for nested datatypes and its dual to inductive and coinductive constructors of arbitrarily high ranks by appropriately generalizing Mendler-style (co)iteration. Characteristically to Mendler-style schemes of disciplined (co)recursion, the schemes we propose do not rest on notions like positivity or monotonicity of a constructor and facilitate programming in a natural and elegant style close to programming with the customary `letrec` construct, where the typings of the schemes, however, guarantee termination. For rank 2, a smoothened version of Bird and Paterson's generalized folds and its dual are achieved; for rank 1, the schemes instantiate to Mendler's original (re)formulation of iteration and coiteration. Several examples demonstrate the power of the approach. Strong normalization of our proposed extension of system F^ω of higher-order parametric polymorphism is proven by a reduction-preserving embedding into pure F^ω.

1 Introduction

Within the paradigm of generic programming, Bird and Paterson [8] with colleagues [11, 15] have studied the problem of identifying workable schemes for defining functions for *nested*, non-uniform or heterogeneous *datatypes*, i.e., inductive and coinductive constructors of rank 2 (type transformers), and put forth *generalized folds* as a scheme for defining functions like substitution for the de Bruijn notation of lambda terms in a natural fashion.

In [2], two of the authors of the present article showed that, making good use of right notions of containment and monotonicity of constructors, the schemes of

---

* The first author gratefully acknowledges the support by the PhD Programme *Logic in Computer Science* (GKLI) of the *Deutsche Forschungs-Gemeinschaft*.

** The third author is partially supported by the Estonian Science Foundation (ETF) under grant No. 4155. He is also grateful to the GKLI for two invitations to Munich; the cooperation started during these visits.

iteration and *coiteration* are extensible to monotone (co)inductive constructors of any finite kind. In the present article, we similarly extend the more liberal generalized folds to all finite kinds. We accomplish this thanks to two ideas: a simple, but powerful generalization of the notion of constructor containment, and reformulation of the schemes in the style originated by Mendler [18]. The result is a concise extension of system F^ω of higher-order parametric polymorphism with (co)inductive constructors of any finite kind, equipped with Mendler-style generalized (co)iteration. Switching to Mendler style was not intentional, but in the end turned out rewarding. The reasons are the following.

Firstly, any syntactic positivity requirement can be avoided in the formation rules of (co)inductive types. This is beneficial as positivity and map terms associating to positive constructors would have to be defined by induction outside the system and parametrically polymorphic quantification over all positive constructors is impossible. Moreover, for higher kinds, there is no obvious canonical definition of positivity, although attempts of definition exist [14]. Replacing positivity with monotonicity [17, 2] gives an improvement, but formulations of the systems and especially programming remain clumsy.

Secondly, Mendler style facilitates a programming style very close to programming with general recursion (i.e., the `letrec` construct). The computation rules for Mendler-style disciplined (co)recursion schemes are nearly identical to the rule of `letrec`, the restrictive typings however ensure that all computations terminate.

Thirdly, Mendler-style disciplined (co)recursion schemes tend to be amenable for generalizations whereas conventional ones—making use of map terms or monotonicity witnesses—typically get complicated. Examples are: primitive (co-) recursion [18], course-of-value (co)iteration [21, 22], iteration over multiple inductive types at the same time [22] and—as the examples of this article testify—generalized iteration in the sense of generalized folds.

The article is organized as follows. In Sect. 2, we review our starting point system F^ω of higher-order parametric polymorphism. In Sect. 3, we present our system MIt^ω of (co)inductive constructors of finite ranks with generalized Mendler-style iteration and describe some programming examples. The embedding of MIt^ω into F^ω is presented in Sect. 4. We conclude with a summary and discussion of related work.

Acknowledgements: Many thanks to Peter Hancock for his suggestion in November 2000 of the unusual notion $F \leq_{\kappa 1} G$. It started this whole research project.

2 System F^ω

Our development of higher-order datatypes takes place within a conservative extension of Curry-style system F^ω by binary sums and products, the unit type and existential quantification. It contains three syntactic categories:

Kinds. Kinds are given by the following grammar and denoted by the letter κ.

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

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

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

Constructors. Uppercase latin letters denote constructors, given by the following grammar. The metavariable X ranges over a denumerable set of constructor variables.

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

We identify β-equivalent constructors. A constructor F has kind κ if there is a context Γ such that $\Gamma \vdash F : \kappa$. The kinding rules for constructors appear in Appendix A.

The rank of a constructor is given by the rank of its kind. Preferably we will use letters A, B, C, D for constructors of rank 0 *(types)* and F, G, H for constructors of rank 1. If no kinds are given and cannot be guessed from the context, we assume $A, B, C, D : *$ and $F, G, H : \kappa 1$. Write $\mathsf{Id}_{\kappa} := \lambda X^{\kappa}.X$ for the identity constructor. If the kinding is clear from the context, we just write Id. Constructor application associates to the left, i.e., $F\,G_1 \dots G_n = (\dots(F\,G_1)\dots)\,G_n$. Setting $\boldsymbol{G} := G_1, \dots, G_n$, the constructor $F\,G_1 \dots G_n$ is also written as $F\boldsymbol{G}$. Sums and products can inductively be extended to all kinds: For $F, G : \kappa_1 \to \kappa_2$ set $F + G := \lambda X^{\kappa_1}.FX + GX$ and $F \times G := \lambda X^{\kappa_1}.FX \times GX$.

Objects (Curry terms). Lower case letters denote terms. In the grammar below, the metavariable x ranges over a denumerable set of object variables.

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

Most term constructors are standard; "pack" introduces and "open" eliminates existential quantification. The polymorphic identity $\lambda x.x : \forall A.\,A \to A$ will be denoted by id. We write $f \circ g$ for function composition $\lambda x.\,f\,(g\,x)$. Application $r\,s$ associates to the left, hence $r\boldsymbol{s} = (\dots(r\,s_1)\,\dots\,s_n)$ for $\boldsymbol{s} = s_1, \dots, s_n$.

A term t has type A if $\Gamma \vdash t : A$ for some context Γ. The relation $\longrightarrow$ denotes the usual one-step β-reduction which is confluent, type preserving and strongly normalizing. The typing and reduction rules for terms are standard and can be found in Appendix A.

In the following we will refer to the here defined system simply as "F^{ω}".

3 Generalized Mendler-Style Iteration and Coiteration

In this section, we recap and extend the notions of containment and monotonicity presented in [2]. On top of these notions, we define the system MIt^ω of generalized (co)iteration for inductive and coinductive constructors of arbitrary ranks, which we then specialize to rank 1 (types) and rank 2 (type transformers). To give a feel for our system, we spell out some examples involving *nested* or non-uniform *datatypes* [7].

3.1 Containment and Monotonicity of Constructors

Containment. The key to extending Mendler-style iteration and coiteration [19] to finite kinds consists in identifying an appropriate containment relation for constructors of the same kind κ. For types, the canonical choice is implication. For an arbitrary kind $\kappa = \boldsymbol{\kappa} \to *$, the easiest notion is "pointwise implication": The constructor $\subseteq_\kappa \colon \kappa \to \kappa \to *$ is defined by $F \subseteq_\kappa G := \forall \boldsymbol{X}^{\boldsymbol{\kappa}}. F\boldsymbol{X} \to G\boldsymbol{X}$, hence $F \subseteq_\kappa G$ is a type which, as a proposition, states that F is contained in G.

A more refined notion $\leq_\kappa$ has been employed already in previous work [2] which studies (co)iteration for monotone (co)inductive constructors of higher kinds:

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

Monotonicity. Using this notion of containment, we can define monotonicity $\mathsf{mon}_\kappa : \kappa \to *$ for kind κ directly by

$$\mathsf{mon}_\kappa\, F := F \leq_\kappa F.$$

The type $\mathsf{mon}_\kappa\, F$, seen as a proposition, asserts that F is monotone. The same type is used in polytypic programming for generic map functions [13, 3].

This notion does not enter the formulation of system MIt^ω, but many applications. We omit the subscripted kind κ when clear from the context, as in the definition of the following basic *monotonicity witnesses.* These are closed terms whose type is some $\mathsf{mon}\, F$. They will pop up in examples later.

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

Relativized refined containment. In order to be able to extend Mendler (co)iteration to higher kinds so that generalized folds [8] are covered, we have to relativize the notion $\leq_\kappa$, $\kappa = \boldsymbol{\kappa} \to *$, to a vector $\boldsymbol{H}$ of constructors of kinds $\boldsymbol{\kappa} \to \boldsymbol{\kappa}$. For every kind $\kappa = \boldsymbol{\kappa} \to *$, we define a constructor $\leq_\kappa^{(-)} \colon (\boldsymbol{\kappa} \to \boldsymbol{\kappa}) \to \kappa \to \kappa \to *$ by structural recursion on κ as follows:

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

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

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

As an example, for $F, G, H : \kappa1$, one has

$$\begin{array}{lcl} F \;\leq_{\kappa1}^{H}\; G & = & \forall A \forall B.\, (A \to HB) \to FA \to GB, \\ F \;{}^{H}{\leq_{\kappa1}}\; G & = & \forall A \forall B.\, (HA \to B) \to FA \to GB. \end{array}$$

3.2 System MIt^ω

Now we are ready to define generalized Mendler-style iteration and coiteration, which specialize to ordinary Mendler-style iteration and coiteration in the case of (co)inductive types, and to a scheme encompassing generalized folds [8, 11, 15] and the dual scheme for coinductive constructors of rank 2. This gives an extension of Mendler's system [19] to finite kinds. The generalized scheme for coinductive constructors is a new principle of programming with non-wellfounded datatypes.

The system MIt^ω is given as an extension of F^ω by wellkinded constructor constants μ_κ and ν_κ, welltyped term constants $\mathsf{in}_\kappa, \mathsf{GIt}_\kappa, \mathsf{out}_\kappa$ and GCoit_κ for every kind κ, and new term reduction rules.

Inductive constructors. Let $\kappa = \boldsymbol{\kappa} \to *$ and $\boldsymbol{\kappa}' = \boldsymbol{\kappa} \to \boldsymbol{\kappa}$.

Formation. $\mu_\kappa \quad : (\kappa \to \kappa) \to \kappa$

Introduction. $\mathsf{in}_\kappa \quad : \forall F^{\kappa\to\kappa}.\, F\,(\mu_\kappa F) \subseteq_\kappa \mu_\kappa F$
Elimination. $\mathsf{GIt}_\kappa : \forall F^{\kappa\to\kappa} \forall \boldsymbol{H}^{\boldsymbol{\kappa}'} \forall G^\kappa.\, (\forall X^\kappa.\, X \leq^{\boldsymbol{H}} G \to F\,X \leq^{\boldsymbol{H}} G) \to \mu_\kappa F \leq^{\boldsymbol{H}} G$

Reduction. $\mathsf{GIt}_\kappa\, s\, \boldsymbol{f}\, (\mathsf{in}_\kappa\, t) \longrightarrow_\beta s\, (\mathsf{GIt}_\kappa\, s)\, \boldsymbol{f}\, t$

with $|\boldsymbol{f}| = |\boldsymbol{\kappa}|$.

Coinductive constructors. Let $\kappa = \boldsymbol{\kappa} \to *$ and $\boldsymbol{\kappa}' = \boldsymbol{\kappa} \to \boldsymbol{\kappa}$.

Formation. $\nu_\kappa \quad : (\kappa \to \kappa) \to \kappa$

Elimination. $\mathsf{out}_\kappa \quad : \forall F^{\kappa\to\kappa}.\, \nu_\kappa F \subseteq_\kappa F\,(\nu_\kappa F)$
Introduction. $\mathsf{GCoit}_\kappa : \forall F^{\kappa\to\kappa} \forall \boldsymbol{H}^{\boldsymbol{\kappa}'} \forall G^\kappa.\, (\forall X^\kappa.\, G \;{}^{\boldsymbol{H}}{\leq}\; X \to G \;{}^{\boldsymbol{H}}{\leq}\; F\,X) \to G \;{}^{\boldsymbol{H}}{\leq}\; \nu_\kappa F$

Reduction. $\mathsf{out}_\kappa\, (\mathsf{GCoit}_\kappa\, s\, \boldsymbol{f}\, t) \longrightarrow_\beta s\, (\mathsf{GCoit}_\kappa\, s)\, \boldsymbol{f}\, t$

with $|\boldsymbol{f}| = |\boldsymbol{\kappa}|$.

Notice that for *every* constructor F of kind $\kappa \to \kappa$, $\mu_\kappa F$ is a constructor of kind κ. In Mendler's original system [19] as well as its variant for the treatment of primitive (co-)recursion [18], always positivity of F is required which is a very natural concept in the case $\kappa = *$. However, for higher kinds, there does not exist such a canonical syntactic restriction. Anyway, in [21] 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 [16] and becomes even more useful for our higher-order nested datatypes.

As for F^ω, denote the term closure of the reduction rules by $\longrightarrow$ and its transitive closure by $\longrightarrow^+$.

3.3 Mendler-style (Co)Iteration for (Co)Inductive Types

In the case $\kappa = *$, the rules for μ_κ and ν_κ match with Mendler's [19], except for our removal of the positivity condition and our choice of Curry-style typing:

Inductive types.

$$\begin{array}{lll}
\text{Formation.} & \mu_* & : (* \to *) \to * \\
\text{Introduction.} & \mathsf{in}_* & : \forall F^{* \to *}.\, F\,(\mu_* F) \to \mu_* F \\
\text{Elimination.} & \mathsf{GIt}_* & : \forall F^{* \to *} \forall Y^*.\ (\forall X^*.\, (X \to Y) \to F\,X \to Y) \to \mu_* F \to Y \\
\text{Reduction.} & \multicolumn{2}{l}{\mathsf{GIt}_*\, s\, (\mathsf{in}_*\, t) \longrightarrow_\beta s\, (\mathsf{GIt}_*\, s)\, t}
\end{array}$$

Coinductive types.

$$\begin{array}{lll}
\text{Formation.} & \nu_* & : (* \to *) \to * \\
\text{Elimination.} & \mathsf{out}_* & : \forall F^{* \to *}.\, \nu_* F \to F\,(\nu_* F) \\
\text{Introduction.} & \mathsf{GCoit}_* & : \forall F^{* \to *} \forall Y^*.\ (\forall X^*.\, (Y \to X) \to Y \to F\,X) \to Y \to \nu_* F \\
\text{Reduction.} & \multicolumn{2}{l}{\mathsf{out}_*\, (\mathsf{GCoit}_*\, s\, t) \longrightarrow_\beta s\, (\mathsf{GCoit}_*\, s)\, t}
\end{array}$$

Relation to general recursion. Typed functional programming languages like ML and Haskell use recursive types instead of inductive and coinductive types and general recursion instead of strongly normalizing restrictions such as Mendler (co)iteration. General recursion can be introduced via a fixed-point combinator

$$\begin{aligned}
&\mathsf{fix} : \forall A.(A \to A) \to A \\
&\mathsf{fix}\ s \longrightarrow s\,(\mathsf{fix}\ s),
\end{aligned}$$

from which the more common `let rec` $f = r$ `in` t can be defined as `let` $f =$ $\mathsf{fix}\ (\lambda f.\, r)$ `in` t. A nice aspect of Mendler (co)iteration is that the reduction behaviour GIt_* and GCoit_* is almost identical to the one of fix. The only difference is that unfolding of GIt resp. GCoit is controlled by a guard "in" resp. "out", which gets removed in the reduction step. Guarded unfolding of recursion is essential to strong normalization; similar setups can be found in other systems which facilitate *type-based termination*, e.g. [10, 1, 5].

In some sense GIt and GCoit are just restricted versions of fix, i. e., each rank-1 MIt^ω program translates (requiring minimal changes) into a Haskell program with the same meaning. For higher kinds κ, GIt_κ and GCoit_κ are not typable in the Hindley-Milner type systems of Haskell 98 and ML, but their reduction behaviour is still included in the one of fix. This suggests that one can code most naturally with GIt and GCoit, which we will demonstrate in the next subsection by some examples involving so-called *nested* or *heterogeneous datatypes*.

3.4 Programming with (Co)Inductive Constructors of Rank 2

Nested or non-uniform datatypes, i.e., inductive and coinductive constructors of rank 2 (more exactly, inductive and coinductive constructors induced by constructors of rank 2), arise in our system as applications of $\mu_{\kappa 1}$ and $\nu_{\kappa 1}$ (recall that $\kappa 1 = * \to *$ and $\kappa 2 = \kappa 1 \to \kappa 1$). We obtain the following instances from the general definitions.

Inductive constructors of rank 2.

$$\begin{array}{lll}
\text{Formation.} & \mu_{\kappa 1} & : \kappa 2 \to \kappa 1 \\[4pt]
\text{Introduction.} & \mathsf{in}_{\kappa 1} & : \forall F^{\kappa 2} \forall A.\, F\,(\mu_{\kappa 1} F)\, A \to\ \mu_{\kappa 1} F A \\
\text{Elimination.} & \mathsf{GIt}_{\kappa 1} & : \forall F^{\kappa 2} \forall H^{\kappa 1} \forall G^{\kappa 1}.\ (\forall X^{\kappa 1}.\, X \leq^H G \to F\,X \leq^H G) \to \mu_{\kappa 1} F \leq^H G \\[4pt]
\text{Reduction.} & \multicolumn{2}{l}{\mathsf{GIt}_{\kappa 1}\, s\, f\, (\mathsf{in}_{\kappa 1}\, t) \longrightarrow_\beta s\, (\mathsf{GIt}_{\kappa 1}\, s)\, f\, t}
\end{array}$$

Coinductive constructors of rank 2.

$$\begin{array}{lll}
\text{Formation.} & \nu_{\kappa 1} & : \kappa 2 \to \kappa 1 \\[4pt]
\text{Elimination.} & \mathsf{out}_{\kappa 1} & : \forall F^{\kappa 2} \forall A.\, \nu_{\kappa 1} F A\ \to\ F\,(\nu_{\kappa 1} F) A \\
\text{Introduction.} & \mathsf{GCoit}_{\kappa} & : \forall F^{\kappa 2} \forall H^{\kappa 1} \forall G^{\kappa 1}.\ (\forall X^{\kappa 1}.\, G\ {}^H\!\!\leq X \to G\ {}^H\!\!\leq F\,X) \to G\ {}^H\!\!\leq \nu_{\kappa 1} F \\[4pt]
\text{Reduction.} & \multicolumn{2}{l}{\mathsf{out}_{\kappa 1}\, (\mathsf{GCoit}_{\kappa 1}\, s\, f\, t) \longrightarrow_\beta s\, (\mathsf{GCoit}_{\kappa 1}\, s)\, f\, t}
\end{array}$$

An example of a structure which can be modeled by a nested datatype is lists of length 2^n, which are called *powerlists* [6] or *perfectly balanced, binary leaf trees* [11]. In our system, they are represented by the type transformer $\mathsf{PList} := \mu_{\kappa 1}\, \mathsf{PListF}$ where $\mathsf{PListF} : \kappa 2 := \lambda F \lambda A.\, A + F(A \times A)$. The data constructors are given by

$$\begin{array}{ll}
\mathsf{zero} : \forall A.\, A \to \mathsf{PList}\, A & := \lambda a.\, \mathsf{in}_{\kappa 1}(\mathsf{inl}\, a) \\
\mathsf{succ} : \forall A.\ \mathsf{PList}(A \times A) \to \mathsf{PList}\, A & := \lambda l.\, \mathsf{in}_{\kappa 1}(\mathsf{inr}\, l)
\end{array}$$

Assume a type Nat of natural numbers with addition "+" and multiplication "×", both written infix. Suppose we want to define a function $\mathsf{sum} : \mathsf{PList}\,\mathsf{Nat} \to \mathsf{Nat}$ which sums up all elements of a powerlist by iteration over its structure. The case $\mathsf{sum}\,(\mathsf{succ}\ t)$ imposes some challenge, since sum cannot be directly applied to $t : \mathsf{PList}(\mathsf{Nat} \times \mathsf{Nat})$. The solution is to define a more general function sum' by polymorphic recursion, which has the following behaviour.

$$\begin{array}{l}
\mathsf{sum}' : \forall A.\, (A \to \mathsf{Nat}) \to \mathsf{PList}\, A \to \mathsf{Nat} \\[4pt]
\mathsf{sum}'\ f\, (\mathsf{zero}\ a) \longrightarrow^+ f\, a \\
\mathsf{sum}'\ f\, (\mathsf{succ}\ l) \ \longrightarrow^+ \mathsf{sum}'\, (\lambda p.\, f\, (p.0) + f\, (p.1))\, l
\end{array}$$

Here, the iteration process builds up a continuation f which in the end sums up the contents packed into a. From sum', the summation function is obtained by $\mathsf{sum} := \mathsf{sum}'\ \mathsf{id}$.

The system MIt^ω has been designed so that functions like sum' can be defined directly via generalized iteration. In our case, use the instantiations $F := \mathsf{PListF}$ and $G := H := \lambda\_.\,\mathsf{Nat}$ and define:

$$\begin{aligned}
\mathsf{sum}' &: \quad \mu_{\kappa 1} F \leq^H G \\
\mathsf{sum}' &:= \quad \mathsf{GIt}_{\kappa 1}\ \lambda sum' \lambda f \lambda x.\,\mathsf{case}\,(x,\ a.\,f\,a, \\
&\qquad\qquad\qquad\qquad\qquad l.\,sum'\,(\lambda p.\,f\,(p.0) + f\,(p.1))\,l)
\end{aligned}$$

The postulated reduction behaviour is verified by a simple calculation.

For another example consider the non-wellfounded version of perfectly balanced, binary (node-labelled) trees. They are represented by the type transformer $\mathsf{BTree} := \nu_{\kappa 1}\,\mathsf{BTreeF}$ where $\mathsf{BTreeF} : \kappa 2 := \lambda F \lambda A.\,A \times F(A \times A)$. The data destructors are

$$\begin{aligned}
&\mathsf{root} : \forall A.\ \mathsf{BTree}\,A \to A && := \lambda t.\,(\mathsf{out}_{\kappa 1} t).0 \\
&\mathsf{subs} : \forall A.\ \mathsf{BTree}\,A \to \mathsf{BTree}(A \times A) && := \lambda t.\,(\mathsf{out}_{\kappa 1} t).1
\end{aligned}$$

We want to define the tree $\mathsf{nats} : \mathsf{BTree}\,\mathsf{Nat}$ filled with natural numbers starting with 1 breadth-first left-first. A more general function $\mathsf{nats}' : \forall A.\ (\mathsf{Nat} \to A) \to (\mathsf{Nat} \to \mathsf{BTree}\,A)$ with the reduction behaviour

$$\begin{aligned}
&\mathsf{root}(\mathsf{nats}'\ f\,n) \longrightarrow^+ f\,n \\
&\mathsf{subs}(\mathsf{nats}'\ f\,n) \longrightarrow^+ \mathsf{nats}'\,(\lambda m.\,\langle f\,(2 \times m), f\,(2 \times m + 1)\rangle)\,n
\end{aligned}$$

is definable as a generalized coiteration by

$$\mathsf{nats}' := \mathsf{GCoit}_{\kappa 1}\ \lambda nats' \lambda f \lambda n.\ \langle f\,n,\ nats'\,(\lambda m.\,\langle f\,(2 \times m), f\,(2 \times m + 1)\rangle)\,n\rangle$$

choosing $F := \mathsf{BTreeF}$, $G := H := \lambda\_.\,\mathsf{Nat}$. To obtain nats, one sets $\mathsf{nats} := \mathsf{nats}'\ \mathsf{id}\ 1$.

Higher-order representation of de Bruijn terms. Bird & Paterson [9] and Altenkirch & Reus [4] have shown that nameless untyped λ-terms can be represented by a heterogeneous datatype. As in the system GMIC of [2], this type is obtained in MIt^ω as the least fixed point of the monotone rank-2 constructor LamF.

$$\begin{aligned}
&\mathsf{LamF} : \kappa 2 && := \lambda F \lambda A.\,A + (F A \times F A + F(1 + A)) \\
&\mathsf{lamf}\ \ : \mathsf{mon}\,\mathsf{LamF} && := \lambda s \lambda f.\ \mathsf{either}\,f\,\Big(\mathsf{either}\,(\mathsf{fork}\,(s\,f))\,(s\,(\mathsf{maybe}\ f))\Big)
\end{aligned}$$

The type $\mathsf{Lam}\,A$ again represents all de Bruijn terms with free variables in A, the constructors var, app and abs are simplified w. r. t. [2]. Again, we provide an auxiliary function weak which lifts each variable in a term to provide space for a fresh variable.

$$\begin{array}{lll}
\mathsf{Lam} & : \kappa 1 & := \mu_{\kappa 1}\,\mathsf{LamF} \\
\mathsf{lam} & : \mathsf{mon}\,\mathsf{Lam} & := \mathsf{GIt}_{\kappa 1}\ \lambda map \lambda f \lambda x.\,\mathsf{in}_{\kappa 1}\,(\mathsf{lamf}\ map\, f\, x) \\
\mathsf{var} & : \forall A.\, A \to \mathsf{Lam}\,A & := \lambda a.\,\mathsf{in}_{\kappa 1}\,(\mathsf{inl}\,a) \\
\mathsf{app} & : \forall A.\ \mathsf{Lam}\,A \to \mathsf{Lam}\,A \to \mathsf{Lam}\,A & := \lambda t_1 \lambda t_2.\,\mathsf{in}_{\kappa 1}\,(\mathsf{inr}\,(\mathsf{inl}\,\langle t_1, t_2\rangle)) \\
\mathsf{abs} & : \forall A.\ \mathsf{Lam}(1 + A) \to \mathsf{Lam}\,A & := \lambda r.\,\mathsf{in}_{\kappa 1}\,(\mathsf{inr}\,(\mathsf{inr}\,r)) \\
\mathsf{weak} & : \forall A.\ \mathsf{Lam}\,A \to \mathsf{Lam}(1 + A) & := \mathsf{lam}\,(\lambda a.\ \mathsf{inr}\,a)
\end{array}$$

The most natural question on this representation of untyped λ-calculus is the representability of substitution. With generalized iteration, it is possible to give a direct definition of substitution (the bind or extension operation of the lambda terms monad):

$$\begin{array}{ll}
\mathsf{subst} : & \forall A \forall B.\,(A \to \mathsf{Lam}\,B) \to \mathsf{Lam}\,A \to \mathsf{Lam}\,B \ \equiv\ \mathsf{Lam} \leq^{\mathsf{Lam}} \mathsf{Lam} \\
\mathsf{subst} := & \mathsf{GIt}_{\kappa 1}\ \lambda subst \lambda f \lambda t.\ \mathsf{case}\,(t, \\
& \quad a.\ f\,a,\ t'.\ \mathsf{case}\,(t', \\
& \quad p.\ \mathsf{app}\,(subst\,f\,(p.0))\,(subst\,f\,(p.1)), \\
& \quad r.\ \mathsf{abs}\,(subst\,(\mathsf{lift}\,f)\,r)), \\
\mathsf{lift} \ \ : & \forall A \forall B.\,(A \to \mathsf{Lam}\,B) \to (1 + A) \to \mathsf{Lam}\,(1 + B) \\
\mathsf{lift} \ := & \lambda f \lambda x.\ \mathsf{case}\,(x,\ u.\ \mathsf{var}\,(\mathsf{inl}\ u),\ a.\ \mathsf{weak}\,(f\,a))
\end{array}$$

Note that the formulation of generalized folds in [8] would yield the flattening function (the join or multiplication operation of the monad)

$$\mathsf{flatten} : \forall A.\ \mathsf{Lam}(\mathsf{Lam}\,A) \to \mathsf{Lam}\,A.$$

We obtain flattening as special case of substitution by $\mathsf{flatten} := \mathsf{subst}\ \mathsf{id}$.

Triangles. The dual of substitution for variables in a term or non-wellfounded term is redecoration of a non-wellfounded or wellfounded decorated tree, cf. [23]. An interesting and intuitive example of decorated tree types arising from a rank-2 coinductive constructor are triangles. Define

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

Then $\mathsf{Tri}\,EA$ is the type of triangular tables of the sort

$$\begin{array}{r|l}
A & E\ E\ E\ E \ldots \\
 & A\ E\ E\ E \ldots \\
 & \quad\ A\ E\ E \ldots \\
 & \qquad\ \ A\ E \ldots \\
 & \qquad\qquad A \ldots
\end{array}$$

decomposing into a scalar (an element of A) and a trapezium (an element of $\mathsf{Tri}\,E(E \times A)$). The destructors and the monotonicity witness are

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

$$\mathsf{tri}\ := \mathsf{GCoit}_{\kappa 1}\ \lambda map \lambda f \lambda x.\,\langle f\,(\mathsf{top}\ x),\ map\,(\mathsf{pair}\ \mathsf{id}\ f)\,(\mathsf{rest}\ x)\rangle : \forall E.\,\mathsf{mon}_{\kappa 1}(\mathsf{Tri}\,E)$$

Redecoration is an operation dual to substitution that takes a redecoration rule f (an assignment of B-decorations to A-decorated trees) and an A-decorated tree t, and returns a B-decorated tree t'. The return tree t' is obtained from t by B-redecorating every node based on the A-decorated subtree it roots, as instructed by the redecoration rule. For streams, for instance $\mathsf{redec} : \forall A \forall B.\, (\mathsf{Str}\, A \to B) \to \mathsf{Str}\, A \to \mathsf{Str}\, B)$ takes $f : \mathsf{Str}\, A \to B$ and $t : \mathsf{Str}\, A$ and returns $\mathsf{redec}\ f\ t$, which is a B-stream obtained from t by replacing each of its elements by what f assigns to the substream this element heads. Triangles are a generalization of streams much in the same way as de Bruijn notations for lambda terms differ from terms in the universal algebra style signature with one binary and one unary operator. For triangles, redecoration works as follows: In the triangle

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

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. This is straightforward to define using GCoit:

$$\begin{array}{lll} \mathsf{lift} & : & \forall E \forall A \forall B.\, (\mathsf{Tri}\, E A \to B) \to \mathsf{Tri}\, E (E \times A) \to E \times B \\ \mathsf{lift} & := & \lambda f \lambda y.\, \langle (\mathsf{top}\ y).0,\ f\, (\mathsf{tri}\, (\lambda p.\ p.1)\, y) \rangle \\ \mathsf{redec} & : & \forall E \forall A \forall B.\, (\mathsf{Tri}\, E A \to B) \to \mathsf{Tri}\, E A \to \mathsf{Tri}\, E B \\ \mathsf{redec} & := & \mathsf{GCoit}_{\kappa 1}\ \lambda \mathit{redec} \lambda f \lambda x.\, \langle f\, x, \mathit{redec}\, (\mathsf{lift}\, f)\, (\mathsf{rest}\, x) \rangle \end{array}$$

4 Embedding into System F^ω

In this section, we show how to embed MIt^ω into F^ω. The embedding establishes strong normalization for MIt^ω.

4.1 Kan extensions

For the sake of the embedding of MIt^ω into its subsystem F^ω, we use a syntactic version of Kan extensions, see [20, chapter 10]. Compared with [2], Kan extensions "along" are now defined for all kinds, not just for rank 1.

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

$$\begin{array}{rl} \mathsf{Ran}\ G & := G \\ (\mathsf{Ran}_{H,\boldsymbol{H}}\, G)\, X \boldsymbol{X} & := \forall Y^{\kappa_1}.\ X \leq H Y\ \to\ (\mathsf{Ran}_{\boldsymbol{H}}\, (G\, Y))\, \boldsymbol{X} \end{array}$$

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

$$\begin{array}{rl} \mathsf{Lan}\, F :=& F \\ (\mathsf{Lan}_{H,\boldsymbol{H}}\, F)\, Y\, \boldsymbol{Y} :=& \exists X^{\kappa_1}.\; HX \leq Y \;\times\; (\mathsf{Lan}_{\boldsymbol{H}}\, (FX))\, \boldsymbol{Y} \end{array}$$

Proposition 1. *Let $F, G : \boldsymbol{\kappa} \to *$ and $\boldsymbol{H} : \boldsymbol{\kappa} \to \boldsymbol{\kappa}$. The following pairs of types are logically equivalent:*

1. $F \leq^{\boldsymbol{H}} G$ *and* $\forall \boldsymbol{X}^{\boldsymbol{\kappa}}.\; F\boldsymbol{X} \to (\mathsf{Ran}_{\boldsymbol{H}}\, G)\boldsymbol{X}$.
2. $F \;{}^{\boldsymbol{H}}\!\!\leq G$ *and* $\forall \boldsymbol{Y}^{\boldsymbol{\kappa}}.\; (\mathsf{Lan}_{\boldsymbol{H}}\, F)\boldsymbol{Y} \to G\,\boldsymbol{Y}$.

Proof. Part 1 requires just a close look at the definition of $\leq^{\boldsymbol{H}}$. Part 2 is only slightly more complicated. □

4.2 Embedding

We can simply define the new constants of MIt^{ω} in F^{ω}. Let $\kappa = \boldsymbol{\kappa} \to *$ and $n := |\boldsymbol{\kappa}|$.

$$\begin{array}{lll} \mu_\kappa & : & (\kappa \to \kappa) \to \boldsymbol{\kappa} \to * \\ \mu_\kappa & := & \lambda F^{\kappa\to\kappa} \lambda \boldsymbol{X}^{\boldsymbol{\kappa}} \forall \boldsymbol{H}^{\boldsymbol{\kappa}\to\boldsymbol{\kappa}} \forall G^{\kappa}.\; (\forall X^{\kappa}. X \leq^{\boldsymbol{H}} G \to FX \leq^{\boldsymbol{H}} G) \to (\mathsf{Ran}_{\boldsymbol{H}}\, G)\boldsymbol{X} \end{array}$$

$$\begin{array}{lll} \mathsf{GIt}_\kappa & : & \forall F^{\kappa\to\kappa} \forall \boldsymbol{H}^{\boldsymbol{\kappa}\to\boldsymbol{\kappa}} \forall G^{\kappa}.\; (\forall X^{\kappa}. X \leq^{\boldsymbol{H}} G \to F\, X \leq^{\boldsymbol{H}} G) \to \mu_\kappa F \leq^{\boldsymbol{H}} G \\ \mathsf{GIt}_\kappa & := & \lambda s \lambda \boldsymbol{f} \lambda r.\, r\; s\; \boldsymbol{f} \end{array}$$

$$\begin{array}{lll} \mathsf{in}_\kappa & : & \forall F^{\kappa\to\kappa} \forall \boldsymbol{X}^{\boldsymbol{\kappa}}.\; F\,(\mu_\kappa F)\,\boldsymbol{X} \to \mu_\kappa F\, \boldsymbol{X} \\ \mathsf{in}_\kappa & := & \lambda t \lambda s \lambda \boldsymbol{f}.\, s\,(\mathsf{GIt}_\kappa\, s)\, \boldsymbol{f}\, t \end{array}$$

$$\begin{array}{lll} \nu_\kappa & : & (\kappa \to \kappa) \to \boldsymbol{\kappa} \to * \\ \nu_\kappa & := & \lambda F^{\kappa\to\kappa} \lambda \boldsymbol{Y}^{\boldsymbol{\kappa}} \exists \boldsymbol{H}^{\boldsymbol{\kappa}\to\boldsymbol{\kappa}} \exists G^{\kappa}.\; (\forall X^{\kappa}. G \;{}^{\boldsymbol{H}}\!\!\leq X \to G \;{}^{\boldsymbol{H}}\!\!\leq FX) \times (\mathsf{Lan}_{\boldsymbol{H}}\, G)\boldsymbol{Y} \end{array}$$

$$\begin{array}{lll} \mathsf{GCoit}_\kappa & : & \forall F^{\kappa\to\kappa} \forall \boldsymbol{H}^{\boldsymbol{\kappa}\to\boldsymbol{\kappa}} \forall G^{\kappa}.\; (\forall X^{\kappa}. G \;{}^{\boldsymbol{H}}\!\!\leq X \to G \;{}^{\boldsymbol{H}}\!\!\leq F\, X) \to G \;{}^{\boldsymbol{H}}\!\!\leq \nu_\kappa F \\ \mathsf{GCoit}_\kappa & := & \lambda s \lambda \boldsymbol{f} \lambda t.\; \mathsf{pack}^{n+1}\, \langle s,\, \mathsf{pack}\langle f_1,\, \ldots,\, \mathsf{pack}\langle f_n,\, t\rangle \ldots\rangle\rangle \end{array}$$

$$\begin{array}{lll} \mathsf{out}_\kappa & : & \forall F^{\kappa\to\kappa} \forall \boldsymbol{Y}^{\boldsymbol{\kappa}}.\; \nu_\kappa F\, \boldsymbol{Y} \to F\,(\nu_\kappa F)\,\boldsymbol{Y} \\ \mathsf{out}_\kappa & := & \lambda r.\; \mathsf{open}\,(r,\, r_1.\, \mathsf{open}\,(r_1,\, r_2.\, \ldots \mathsf{open}\,(r_{n-1},\, r_n.\, \mathsf{open}\,(r_n,\, ft_0. \\ & & \quad \mathsf{open}\,(ft_0.1,\, ft_1.\, \mathsf{open}\,(ft_1.1,\, ft_2.\, \ldots \mathsf{open}\,(ft_{n-1}.1,\, ft_n. \\ & & \quad ft_0.0\; (\mathsf{GCoit}_\kappa\; ft_0.0)\; ft_1.0\; \ldots ft_n.0\; ft_n.1) \ldots)))) \ldots)) \end{array}$$

Theorem 1 (Simulation). *With the definitions above, the following reductions take place in F^{ω}:*

$$\begin{array}{ll} \mathsf{GIt}_\kappa\, s\, \boldsymbol{f}\, (\mathsf{in}_\kappa\, t) & \longrightarrow^{+} s\,(\mathsf{GIt}_\kappa\, s)\, \boldsymbol{f}\, t \\ \mathsf{out}_\kappa\, (\mathsf{GCoit}_\kappa\, s\, \boldsymbol{f}\, t) & \longrightarrow^{+} s\,(\mathsf{GCoit}_\kappa\, s)\, \boldsymbol{f}\, t \end{array}$$

Proof. By easy computation.

Corollary 1 (Strong Normalization). *System MIt^{ω} is strongly normalizing, i. e., there is no infinite reduction sequence $r_0 \longrightarrow r_1 \longrightarrow r_2 \longrightarrow \ldots$ for any typable term r_0.*

Proof. Use strong normalization of F^ω and simulation.

Since there are no critical pairs in MIt^ω, reduction is locally confluent; by strong normalization and Newman's Lemma, it is confluent on well-typed terms.

5 Conclusion and Related Work

We have proposed MIt^ω, a system of generalized (co)iteration for arbitrary ranks, which turned out to be a definitional extension of Girard's system F^ω and hence enjoys its good meta-theoretic properties, most notably strong normalization. It combines the ideas of Mendler for (co)inductive types with the notion of generalized folds for inductive constructors of rank 2 invented by Bird and Paterson.

MIt^ω has been carefully set up to come with a perspicuous computational behavior, which is very close to general recursion à la `letrec`—a distinctive feature of Mendler-style recursion schemes. For higher ranks, i.e., for the treatment of fixed-points which are themselves type transformers, we described a modified containment relation (via the index $\boldsymbol{H}$) in order to encompass generalized folds, proposed by Bird and Paterson as a means of more elegant definitions of functions operating on nested datatypes.

Therefore, MIt^ω might serve as a basis of a total programming language for nested datatypes. Alternatively, it can be seen as a discipline of programming in existing languages like Haskell which gives termination guarantees for free.

Some related work. The generalized iteration scheme of the present article is a reformulation working in all finite ranks of generalized folds in the liberal sense of Sec. 4.1 and 6 of [8]. More exactly, it extends the "efficient" [11, 15] version of that scheme. The efficient generalized folds differ from the original generalized folds in the target type which is constructed with $\leq^{\boldsymbol{H}}$ rather than $\subseteq^{\boldsymbol{H}}$. Here, $\subseteq^{\boldsymbol{H}}$ is the appropriate relativized version of $\subseteq$, defined by

$$F \subseteq^{\boldsymbol{H}} G := \forall \boldsymbol{X}^{\boldsymbol{\kappa}}. F(\boldsymbol{H}\boldsymbol{X}) \to G\boldsymbol{X}.$$

Future work. The realm of higher-rank datatypes seems hardly explored. We certainly wish to try out our schemes on the examples of inductive constructors of rank 3 from [12]. Further, we seek to extend MIt^ω to cover other recursion schemes on nested datatypes like a form of "generalized primitive recursion". This scheme, and others, would no longer have an operationally faithful embedding into System F^ω.

References

1. A. Abel. Termination checking with types. Technical Report 0201, Inst. für Informatik, Ludwigs-Maximilians-Univ. München, 2002.
2. A. Abel and R. Matthes. (Co-)iteration for higher-order nested datatypes. To appear in H. Geuvers, F. Wiedijk, eds., *Post-Conf. Proc. of IST WG TYPES 2nd Ann. Meeting, TYPES'02*, *Lect. Notes in Comput. Sci.*, Springer-Verlag.

3. T. Altenkirch and C. McBride. Generic programming within dependently typed programming. To appear in J. Gibbons and J. Jeuring, *Proc. of IFIP TC2 WC on Generic Programming, WCGP 2002*, Kluwer Acad. Publishers.
4. T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In J. Flum and M. Rodríguez-Artalejo, eds., *Proc. of 13th Int. Wksh. on Computer Science Logic, CSL'99*, vol. 1683 of *Lect. Notes in Comput. Sci.*, pp.53–468. Springer-Verlag, 1999.
5. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Math. Struct. in Comput. Sci.*, to appear.
6. R. Bird, J. Gibbons, and G. Jones. Program optimisation, naturally. In J. Davies, B. Roscoe, J. Woodcock, eds., *Millenial Perspectives in Computer Science*. Palgrave, 2000.
7. R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, ed., *Proc. of 4th Int. Conf. on Mathematics of Program Construction, MPC'98*, vol. 1422 of *Lect. Notes in Comput. Sci.*, pp. 52–67. Springer-Verlag, 1998.
8. R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Comput.*, 11(2):200–222, 1999.
9. R. Bird and R. Paterson. De Bruijn notation as a nested datatype. *J. of Funct. Program.*, 9(1):77–91, 1999.
10. E. Giménez. Structural recursive definitions in type theory. In *Proc. of 25th Int. Coll. on Automata, Languages and Programming, ICALP'98*, vol. 1443 of *Lect. Notes in Comput. Sci.*, pp. 397–408. Springer-Verlag, 1998.
11. R. Hinze. Efficient generalized folds. In J. Jeuring, ed., *Proc. of 2nd Wksh. on Generic Programming, WGP 2000*, Tech. Report UU-CS-2000-19, Dept. of Comput. Sci., Utrecht Univ., pp. 1–16. 2000.
12. R. Hinze. Manufacturing datatypes. *J. of Funct. Program.*, 11(5): 493–524, 2001.
13. R. Hinze. Polytypic values possess polykinded types. *Sci. of Comput. Program.*, 43(2–3):129–159, 2002.
14. C. B. Jay. Distinguishing data structures and functions: The constructor calculus and functorial types. In S. Abramsky, ed., *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Appl., TLCA'01*, vol. 2044 of *Lect. Notes in Comput. Sci.*, pp. 217–239. Berlin, 2001.
15. C. Martin, J. Gibbons and I. Bayley. Disciplined, efficient, generalised folds for nested datatypes. Submitted.
16. R. Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians-Univ. München, 1998.
17. R. Matthes. Monotone inductive and coinductive constructors of rank 2. In L. Fribourg, ed., *Proc. of 15th Int. Wksh. on Computer Science Logic, CSL 2001*, vol. 2142 of *Lect. Notes in Comput. Sci.*, pp. 600–614. Springer-Verlag, 2001.
18. N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proc. of 2nd Ann. IEEE Symp. on Logic in Computer Science, LICS'87*, pp. 30–36. IEEE CS Press, 1987.
19. N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. of Pure and Appl. Logic*, 51(1–2):159–172, 1991.
20. S. Mac Lane. *Categories for the Working Mathematician*, vol. 5 of *Graduate Texts in Mathematics*, 2nd ed. Springer-Verlag, 1998.
21. T. Uustalu and V. Vene. A cube of proof systems for the intuitionistic predicate μ-, ν-logic. In M. Haveraaen and O. Owe, eds., *Selected Papers from the 8th Nordic Wksh. on Programming Theory, NWPT '96*, Res. Rep. 248, Dept. of Informatics, Univ. of Oslo, pp. 237–246, 1997.

22. T. Uustalu and V. Vene. Coding recursion à la Mendler (extended abstract). In J. Jeuring, ed., *Proc. of 2nd Wksh. on Generic Programming, WGP 2000*, Tech. Rep. UU-CS-2000-19, Dept. of Comput. Sci., Utrecht Univ., pp. 69–85. 2000.
23. T. Uustalu and V. Vene. The dual of substitution is redecoration. In K. Hammond and S. Curtis, eds., *Trends in Funct. Programming 3*, pp. 99–110. Intellect, 2002.

A System F^ω

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

Kinds are generated from the kind $*$ for types by the binary function kind constructor $\to$:

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

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

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

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

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

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

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

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

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

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

Judgments. (Simultaneously defined)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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