(Co-)Iteration for Higher-Order Nested Datatypes

Andreas Abel[1*] and Ralph Matthes[2**]

[1] Department of Computer Science,
University of Munich
abel@informatik.uni-muenchen.de
[2] PPS,
CNRS, Université Paris VII (on leave from University of Munich)
matthes@informatik.uni-muenchen.de

Abstract. The problem of defining iteration for higher-order nested datatypes of arbitrary (finite) rank is solved within the framework of System F^ω of higher-order parametric polymorphism. The proposed solution heavily relies on a general notion of monotonicity as opposed to a syntactic criterion on the shape of the type constructors such as positivity or even being polynomial. Its use is demonstrated for some rank-2 heterogeneous/nested datatypes such as powerlists and de Bruijn terms with explicit substitutions. An important feature is the availability of an iterative definition of the mapping operation (the functoriality) for those rank-1 type transformers (i. e., functions from types to types) arising as least fixed-points of monotone rank-2 type transformers. Strong normalization is shown by an embedding into F^ω. The results dualize to greatest fixed-points, hence to coinductive constructors with coiteration.

1 Introduction

What is iteration for nested datatypes? It is a disciplined use of least fixed-points of nested recursive definitions of types and type transformers which rewards the programmer with a guarantee of termination and can usually be expressed in terms of initial algebras. By "nested", different concepts can be understood. The easiest would be to first introduce the type of natural numbers by the equation $\mathsf{Nat} = 1 + \mathsf{Nat}$ and then the lists of naturals by $\mathsf{NatList} = 1 + \mathsf{Nat} \times \mathsf{NatList}$. Here, Nat is nested within $\mathsf{NatList}$. In proof theory, the general principle behind it is called iterated inductive definitions [BFPS81]. More demanding would be nesting by help of parameters: Let $\mathsf{List}(A) = 1 + A \times \mathsf{List}(A)$ be the type of lists over A. Then $\mathsf{FinTree} = \mathsf{List}(\mathsf{FinTree})$ is nested in the sense of an interleaving of the definitions of $\mathsf{FinTree}$ and $\mathsf{List}(\mathsf{FinTree})$. Certainly, this can be represented as a simultaneous definition. Nesting in the sense advocated in [BM98] is different:

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

** The second author has benefitted from financial support by the EC ESPRIT project 21900 "TYPES" and the EU IST working group 29001 "TYPES".

It is a definition of a family of types where the recursive calls may refer to other members of the family, e. g.,

$$\mathsf{Lam}(A) = A + \mathsf{Lam}(A) \times \mathsf{Lam}(A) + \mathsf{Lam}(1 + A)$$

as a representation of untyped lambda terms [BP99b,AR99]. This is just a heterogeneous datatype because Lam itself is not used in determining to which family member recursive calls are made. True nesting, called non-linear [BP99a], occurs in the representation of untyped lambda terms with explicit substitution as follows:

$$\widehat{\mathsf{Lam}}(A) = A + \widehat{\mathsf{Lam}}(A) \times \widehat{\mathsf{Lam}}(A) + \widehat{\mathsf{Lam}}(1 + A) + \widehat{\mathsf{Lam}}(\widehat{\mathsf{Lam}}(A)),$$

see example 4 below.

The aim of the present article is to shed more light on those nested inductive types in terms of type theory, i. e., by a formulation in System F^ω of higher-order parametric polymorphism [Gir72]. We propose a system GMIC of general monotone inductive and coinductive constructors of arbitrary kind of F^ω which hence also covers greatest fixed-points with associated coiteration. The system will be given in Curry-style, hence does not need any type information for the term rewrite rules. The well-known (at least, since [Wra89]) embeddings of inductive and coinductive datatypes into System F can be extended to an embedding of GMIC into F^ω by a syntactic analogue of Kan extensions (mentioned in the context of nested datatypes in [BP99a]).

A major effort has been made to ensure that there are iterative definitions of closed terms of types such as $\forall A \forall B. (A \to B) \to \widehat{\mathsf{Lam}}(A) \to \widehat{\mathsf{Lam}}(B)$ which hence witness monotonicity or "functoriality" of $\widehat{\mathsf{Lam}}$ and the other type transformers that are the solutions to these nested equations.

Unlike previous work of the second author [Mat01], we base our notion of monotonicity on a non-standard definition of inequality. In the case of F, G being definable functions from types to types, it is

$$F \leq G := \forall A \forall B. (A \to B) \to FA \to GB,$$

kindly suggested by Peter Hancock during his visit in Munich in late 2000. The same notion has been used by Altenkirch/McBride [AM02] and Hinze [Hin02] to define map functions generically.

A rough categorical motivation can be given as follows: If F, G were functors, then $\forall A. FA \to GA$ would be the type of natural transformations α from F to G. Then, given some $f : A \to B$, we would have $Gf \circ \alpha_A$ and $\alpha_B \circ Ff$ as equal "morphisms" of type $FA \to GB$. In our definition, we drop functoriality of F and G but directly require the "diagonal" in the associated naturality diagram from FA to GB (see Fig. 1).

The article is organized as follows: The present section is concluded by a short overview of System F^ω (with a more detailed account in appendix A and the Church version in appendix B). Section 2 contains the definition of GMIC

$$
\begin{array}{ccc}
A \xrightarrow{f} B \\
F \xrightarrow{\alpha} G \\
F \xrightarrow[\le]{\ell} G
\end{array}
\qquad
\begin{tikzcd}
F\,A \arrow[rr, "F\,f"] \arrow[d, "\alpha_A"'] \arrow[drr, dotted, "\ell_f"] & & F\,B \arrow[d, "\alpha_B"] \\
G\,A \arrow[rr, "G\,f"] & & G\,B
\end{tikzcd}
$$

Fig. 1. "*map*"-like transformation $\ell : F \le G$

and its specializations to (co-)inductive types and to (co-)inductive functors, including several examples for programming in GMIC. A short argument is given in 2.4 for subject reduction and confluence of GMIC. The syntactic analogue of Kan extensions forms Section 3. They are in close relation with our non-standard definition of $\le$ for type constructors and are needed for our more advanced examples: substitution for de Bruijn terms and resolution of explicit substitutions (Section 3.1). Section 3.2 shows that, logically, the notion of monotonicity in [Mat01] is a special case of the one in GMIC. Section 4 gives the proof of strong normalization by an embedding into System F^ω. As mentioned before, appendices A and B give details of our version of System F^ω.

The Haskell implementation of (co)inductive types and functors plus most of the examples can be obtained from the Web site of the first author [Abe03].

Acknowledgements: Many thanks to Peter Hancock for his suggestion of the unusual notion of the definition of $F \le_{\kappa 1} G$, to Tarmo Uustalu for repeated advice on syntactic Kan extensions and the representation of substitution and to Thorsten Altenkirch for his valuable feedback on early versions of the present article. We also thank the anonymous referees who provided helpful comments.

1.1 System F^ω

Our development of higher-order datatypes takes place within a conservative extension of Curry-style System F^ω by finite sums and products and existential quantification. It contains three syntactic categories:

Kinds. We denote kinds by the letter κ. For *the pure kind of rank* n we introduce the special name κn.

$$
\begin{array}{ll}
\kappa & ::= * \mid \kappa \to \kappa' \\[4pt]
\kappa 0 & := \ * \\
\kappa(n+1) & := \ \kappa n \to \kappa n
\end{array}
$$

Examples for pure kinds are $\kappa 0 = *$, *types*, $\kappa 1 = * \to *$, *type transformers* and $\kappa 2 = (* \to *) \to * \to *$ *transformers of type transformers.* Note that each kind κ' can be written as $\boldsymbol{\kappa} \to *$, where we write $\boldsymbol{\kappa}$ for $\kappa_1, \ldots, \kappa_n$, set

$\kappa_1, \ldots, \kappa_n \to \kappa := \kappa_1 \to \ldots \to \kappa_n \to \kappa$ and assume that $\to$ associates to the right. Also set $|\kappa_1, \ldots, \kappa_n| := n$.

Constructors. Uppercase latin letters and the greek letters Φ and Ψ denote constructors, given by the following grammar.

$$\begin{aligned} A, B, F, G ::= {} & X \mid \lambda X.F \mid F\,G \mid \forall F^{\kappa}.\, A \mid \exists F^{\kappa}.\, A \mid A \to B \\ & \mid 0 \mid A + B \mid 1 \mid A \times B \end{aligned}$$

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 the constructors can be found in Appendix A. It also contains the typing rules for the terms and the reduction rules.

Preferably we will use letters A, B, C, D for constructors of rank 0 *(types)*, F, G, H for constructors of rank 1 and Φ, Ψ for constructors of rank 2. If no kinds are given and cannot be guessed from the context, we assume $A, B, C, D : *$, $F, G, H : \kappa 1$ and $\Phi, \Psi : \kappa 2$. We write $F \circ G$ for constructor *composition* $\lambda X.F(GX)$.

Objects (Curry terms). Lower case letters denote terms.

$$\begin{aligned} r, s, t ::= {} & x \mid \lambda x.t \mid r\,s \mid \mathsf{abort}\,r \mid \mathsf{inl}\,t \mid \mathsf{inr}\,t \mid \mathsf{case}\,(r,\, x.\,s,\, y.\,t) \\ & \mid \langle\rangle \mid \langle t_1, t_2 \rangle \mid r.0 \mid r.1 \mid \mathsf{pack}\,t \mid \mathsf{open}\,(r,\, x.\,s) \end{aligned}$$

Most term constructors are standard; "pack" introduces and "open" eliminates existential quantification. As for kinds, there is a "vector notation" $\boldsymbol{t}$ for a list $t_1, \ldots, t_n$ of terms. 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 $r\boldsymbol{s}$ associates to the left, hence $r\boldsymbol{s} = (\ldots(rs_1)\ldots s_n)$ for $\boldsymbol{s} = s_1, \ldots, s_n$.

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. As mentioned above, the exact typing and reduction rules can be found in Appendix A.

Church terms. We consider Church-style terms (again, following the distinction between the styles à la Curry and à la Church proposed in [Bar92]) as a *linear notation for typing derivations.* For details see appendix B. Whenever we write a Church term, we mean the typing derivation of the corresponding Curry term.

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

2 Monotone Inductive and Coinductive Constructors

For constructors F and G of kind κ we define the *types* $F \subseteq_{\kappa} G$ and $F \leq_{\kappa} G$ by recursion on κ as follows. Then general monotonicity $\mathsf{mon}_{\kappa}\, F$ for constructor F

can be expressed in terms of $\leq_\kappa$, and will also be a *type*.

$$\begin{aligned}
F \subseteq_* G &:= F \to G \\
F \subseteq_{\kappa\to\kappa'} G &:= \forall X^\kappa.\, F\, X \subseteq_{\kappa'} G\, X \\
F \leq_* G &:= F \to G \\
F \leq_{\kappa\to\kappa'} G &:= \forall X^\kappa \forall Y^\kappa.\, X \leq_\kappa Y \to F\, X \leq_{\kappa'} G\, Y \\
\mathsf{mon}_\kappa\, F &:= F \leq_\kappa F
\end{aligned}$$

(When clear from the context, we will omit subscripts κ from $\subseteq$, $\leq$ and mon. We assume that $\subseteq$ and $\leq$ bind stronger than $\to$ or $\times$.) The proposed (co-)inductive constructors will not rely on some syntactic notion of positivity,[1] but exclusively on the above defined notion of monotonicity, hence giving a logical flavour to the system. Monotonicity as an assertion/proposition is modelled as a type. The propositions-as-types paradigm of type theory, first introduced as the Curry-Howard-isomorphism [How80] for minimal propositional logic, has been an important guide in the development of the system.

Having monotonicity, we can enrich system F^ω with some constructor and term constants to obtain higher-order (co)inductive datatypes. Closed terms receiving a type of the form $\mathsf{mon}_\kappa\, F$ will be called monotonicity witnesses for F.

Inductive constructors.

Formation.	$\mu_\kappa\ :\ (\kappa \to \kappa) \to \kappa$
Introduction.	$\mathsf{in}_\kappa\ :\ \forall F^{\kappa\to\kappa}.\, F\,(\mu_\kappa F) \subseteq \mu_\kappa F$
Elimination.	$\mathsf{It}_\kappa\ :\ \forall F^{\kappa\to\kappa}.\, \mathsf{mon}\, F \to \forall G^\kappa.\, F\, G \subseteq G \to \mu_\kappa F \leq G$
Reduction.	$\mathsf{It}_{\boldsymbol{\kappa}\to *}\, m\, s\, \boldsymbol{f}\, (\mathsf{in}_{\boldsymbol{\kappa}\to *}\, t) \longrightarrow_\beta s\,(m\,(\mathsf{It}_{\boldsymbol{\kappa}\to *}\, m\, s)\, \boldsymbol{f}\, t)$

In the last line we require $|\boldsymbol{f}| = |\boldsymbol{\kappa}|$. As a first example, define

$$\begin{aligned}
\mathsf{M}^\mu_\kappa(m) &:= \mathsf{It}_\kappa\, m\, \mathsf{in}_\kappa \\
\lambda m.\, \mathsf{M}^\mu_\kappa(m) &: \forall F^{\kappa\to\kappa}.\, \mathsf{mon}\, F \to \mathsf{mon}\,(\mu_\kappa F).
\end{aligned}$$

Hence, there is a completely uniform way of inferring monotonicity of $\mu_\kappa F$ from that of F. Moreover, the defined monotonicity witness has the desired reduction behavior: $\mathsf{M}^\mu_\kappa(m)\, \boldsymbol{f}\, (\mathsf{in}_\kappa t) \longrightarrow_\beta \mathsf{in}_\kappa\, (m\ \mathsf{M}^\mu_\kappa(m)\, \boldsymbol{f}\, t)$.

Coinductive constructors.

Formation.	$\nu_\kappa\ :\ (\kappa \to \kappa) \to \kappa$
Introduction.	$\mathsf{Coit}_\kappa\ :\ \forall F^{\kappa\to\kappa}.\, \mathsf{mon}\, F \to \forall G^\kappa.\, G \subseteq F\, G \to G \leq \nu_\kappa F$
Elimination.	$\mathsf{out}_\kappa\ :\ \forall F^{\kappa\to\kappa}.\, \nu_\kappa F \subseteq F\,(\nu_\kappa F)$
Reduction.	$\mathsf{out}_{\boldsymbol{\kappa}\to *}\, (\mathsf{Coit}_{\boldsymbol{\kappa}\to *}\, m\, s\, \boldsymbol{f}\, t) \longrightarrow_\beta m\,(\mathsf{Coit}_{\boldsymbol{\kappa}\to *}\, m\, s)\, \boldsymbol{f}\, (s\, t)$

[1] *Syntactic* in the sense of being a criterion on the shape of the constructor. This criterion is used in most the articles on inductive types [Hag87,Men87,Lei90,Geu92].

Again, we require $|\boldsymbol{f}| = |\boldsymbol{\kappa}|$. Dually to the case of inductive constructors, define

$$\begin{array}{lcl} \mathsf{M}^{\nu}_{\kappa}(m) & := & \mathsf{Coit}_{\kappa}\, m\ \mathsf{out}_{\kappa} \\ \lambda m.\ \mathsf{M}^{\nu}_{\kappa}(m) & : & \forall F^{\kappa\to\kappa}.\, \mathsf{mon}\, F \to \mathsf{mon}\,(\nu_{\kappa} F). \end{array}$$

Hence, also monotonicity of $\nu_{\kappa} F$ follows uniformly from monotonicity of F and has the desired computation rule: $\mathsf{out}_{\kappa}\ (\mathsf{M}^{\nu}_{\kappa}(m)\ \boldsymbol{f}\ t) \longrightarrow_{\beta} m\ \mathsf{M}^{\nu}_{\kappa}(m)\ \boldsymbol{f}\ (\mathsf{out}_{\kappa}\ t)$.

This completes the definition of the system GMIC of general monotone inductive and coinductive constructors. To give a feel for the meaning of these dense definitions we will specialize them to kinds $\kappa 0$ and $\kappa 1$ in the following sections.

2.1 (Co)Inductive Types

For kind $\kappa 0$ we obtain monotone (co)inductive types as described in the second author's thesis [Mat98]. These include all interleaved positive datatypes, especially all homogeneous datatypes like natural numbers, lists, trees etc. which are common in functional programming.

Inductive types ($\kappa = *$).

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

For the examples to follow, we will need some basic monotonicity witnesses:

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

Example 1 (Binary trees). As a preparation for Example 6, we show how to encode a type BT of binary trees with constructors $\mathsf{leaf} : \mathsf{BT}$ and $\mathsf{span} : \mathsf{BT} \to \mathsf{BT} \to \mathsf{BT}$ and iterator $\mathsf{ItBT} : \forall A.(1 + A \times A \to A) \to \mathsf{BT} \to A$.

$$\begin{array}{lll} \mathsf{BTF} & : * \to * & := \lambda A.\, 1 + A \times A \\ \mathsf{BT} & : * & := \mu_{*} \mathsf{BTF} \\ \mathsf{leaf} & : \mathsf{BT} & := \mathsf{in}_{*}(\mathsf{inl}\,\langle\rangle) \\ \mathsf{span} & : \mathsf{BT} \to \mathsf{BT} \to \mathsf{BT} & := \lambda t \lambda u.\, \mathsf{in}_{*}(\mathsf{inr}\,\langle t, u\rangle) \\ \mathsf{mon}\,\mathsf{BTF} & : * & = \forall A \forall B.\, (A \to B) \to (1 + A \times A) \to (1 + B \times B) \\ \mathsf{btf} & : \mathsf{mon}\,\mathsf{BTF} & := \mathsf{maybe} \circ \mathsf{fork} \\ \mathsf{ItBT} & \multicolumn{2}{l}{: \forall A.((1 + A \times A) \to A) \to \mathsf{BT} \to A := \mathsf{It}_{*}\ \mathsf{btf}} \end{array}$$

Coinductive types ($\kappa = *$).

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

As for inductive types, these are just the usual definitions, with arbitrary monotonicity witnesses (sometimes also called strength) instead of canonical ones for positive type transformers F. For the positive (covariant) case, their justification from the point of view of category theory has first been given in [Hag87], a very good presentation of the ideas is to be found in [Geu92].

2.2 (Co)Inductive Functors

If we specialize to kind $\kappa 1$, we get heterogeneous (non-regular) and so-called *(non-linear) nested* datatypes. Prominent examples are powerlists [Hin00] and a monadic representation of de Bruijn λ-terms [AR99,BP99b].

Inductive functors ($\kappa = \kappa 1$). Recall that $\kappa 1 = * \to *$ and $\kappa 2 = \kappa 1 \to \kappa 1$.

$$\begin{array}{lll}
\text{Formation.} & \mu_{\kappa 1} & : \kappa 2 \to \kappa 1 \\
\text{Introduction.} & \mathsf{in}_{\kappa 1} & : \forall \Phi^{\kappa 2} \forall A.\, \Phi\, (\mu_{\kappa 1} \Phi)\, A \to \mu_{\kappa 1} \Phi\, A \\
\text{Elimination.} & \mathsf{It}_{\kappa 1} & : \forall \Phi^{\kappa 2}.\, \mathsf{mon}\, \Phi \to \forall G^{\kappa 1}.\, \Phi\, G \subseteq G \to \\
 & & \quad \forall A \forall B.\, (A \to B) \to \mu_{\kappa 1} \Phi\, A \to G\, B \\
\text{Reduction.} & \multicolumn{2}{l}{\mathsf{It}_{\kappa 1}\, m\, s\, f\, (\mathsf{in}_{\kappa 1}\, t) \longrightarrow_\beta s\, (m\, (\mathsf{It}_{\kappa 1}\, m\, s)\, f\, t)}
\end{array}$$

The name "functors" is not by chance. Let Φ be monotone of kind $\kappa 2$ and $m : \mathsf{mon}\, \Phi$ a monotonicity witness. Then $\mathsf{M}^\mu_{\kappa 1}(m)$ is a monotonicity witness for $\mu_{\kappa 1} \Phi$, i. e.,

$$\mathsf{M}^\mu_{\kappa 1}(m) : \mathsf{mon}(\mu_{\kappa 1} \Phi) = \forall A \forall B.\, (A \to B) \to (\mu_{\kappa 1} \Phi) A \to (\mu_{\kappa 1} \Phi) B,$$

hence $\mathsf{M}^\mu_{\kappa 1}(m)$ is the "functorial action" or map function for $\mu_{\kappa 1} \Phi$. Note, however, that the functor laws are beyond our intensional treatment. They could be proven in a theory with extensional equality.

Example 2 (Powerlists). A famous example for a heterogeneous datatype are lists of length 2^n, also called *powerlists* [BGJ00] or perfectly balanced, binary leaf trees [Hin00]. This datatype is present in our system as the least fixed point of the rank-2 constructor $\mathsf{PListF} = \lambda F \lambda A.\, A + F(A \times A)$.

We obtain the type of powerlists with its data constructors and its monotonicity witness in a schematic way.

$$\begin{array}{ll}
\mathsf{PList} : * \to * & := \mu_{\kappa 1}\, \mathsf{PListF} \\
\mathsf{plistf} : \mathsf{mon}\, \mathsf{PListF} & := \lambda s \lambda f.\, \mathsf{either}\, f\, (s\, (\mathsf{fork}\, f)) \\
\mathsf{plist} \;\, : \mathsf{mon}\, \mathsf{PList} & := \mathsf{M}^\mu_{\kappa 1}(\mathsf{plistf}) \\
\mathsf{zero} \;\, : \forall A.\, A \to \mathsf{PList}\, A & := \lambda a.\, \mathsf{in}_{\kappa 1}(\mathsf{inl}\, a) \\
\mathsf{succ} \;\, : \forall A.\, \mathsf{PList}(A \times A) \to \mathsf{PList}\, A & := \lambda l.\, \mathsf{in}_{\kappa 1}(\mathsf{inr}\, l)
\end{array}$$

A reversal algorithm for powerlists is simply derived from a different monotonicity witness for PListF.

$$\begin{aligned}
&\mathsf{swap} &&: \mathsf{mon}(\lambda A.A \times A) := \lambda f \lambda p.\, \langle f\,(p.1),\, f\,(p.0)\rangle\\
&\mathsf{plistfrev} &&: \mathsf{mon}\,\mathsf{PListF} \qquad := \lambda s \lambda f.\, \mathsf{either}\, f\,(s\,(\mathsf{swap}\, f))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{rev}' &&: \mathsf{mon}\,\mathsf{PList} &&:= \mathsf{M}^{\mu}_{\kappa 1}(\mathsf{plistfrev})\\
&\mathsf{rev} &&: \forall A.\ \mathsf{PList}\,A \to \mathsf{PList}\,A &&:= \mathsf{rev}'\ \mathsf{id}
\end{aligned}$$

Note that the freedom in using monotonicity witnesses demonstrated by the previous definition goes beyond the capabilities of Functorial ML [JBM98]. Although that system uses constants witnessing monotonicity to describe iteration, the behavior of those witnesses is fixed by the generic programming approach and consequently only yields the behavior of fork in the case of $\lambda A.A \times A$.

Example 3 (de Bruijn terms). Bird & Paterson [BP99b] and Altenkirch & Reus [AR99] have shown that nameless untyped λ-terms can be represented by a heterogeneous datatype. In our system this type is obtained as the least fixed point of the monotone rank-2 constructor LamF.

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

Now we can define the datatype $\mathsf{Lam}\,A$ of de Bruijn terms with free variables in A, plus its constructors var, app and abs. Furthermore, we give an auxiliary function weak which lifts each variable in a term to provide space for a fresh variable.

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

Example 4 (de Bruijn terms with explicit substitutions). We consider an extension of the untyped λ-calculus by explicit substitutions $t\{\sigma\}$ where σ provides a term t_i for each of the n free variables x_i of t. We can view $t\{\sigma\}$ as a *term* which has the same structure as t but with variables whose names are taken from the set $\{t_1, \ldots, t_n\}$. This means that the variable names are itself λ-terms. Hence, for the data constructor $\widehat{\mathsf{exs}}$ which makes an explicit substitution a term, the type $\widehat{\mathsf{Lam}}(\widehat{\mathsf{Lam}}\,A) \to \widehat{\mathsf{Lam}}\,A$ is justified. In this case we have to deal with a truly *nested* datatype.

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

The nesting of the type transformers F in $\widehat{\mathsf{Lam}}$ is reflected by the nesting of the transformations s in the respective monotonicity witness. The datatype $\widehat{\mathsf{Lam}}$ with its constructors is now obtained mechanically.

$$
\begin{array}{lll}
\widehat{\mathsf{Lam}} & : \kappa 1 & := \mu_{\kappa 1}\, \widehat{\mathsf{LamF}} \\
\widehat{\mathsf{lam}} & : \mathsf{mon}\, \widehat{\mathsf{Lam}} & := \mathsf{M}^{\mu}_{\kappa 1}(\widehat{\mathsf{lamf}}) \\
\widehat{\mathsf{var}} & : \forall A.\, A \to \widehat{\mathsf{Lam}}\, A & := \lambda a.\, \mathsf{in}_{\kappa 1}(\mathsf{inl}\,(\mathsf{inl}\, a)) \\
\widehat{\mathsf{app}} & : \forall A.\, \widehat{\mathsf{Lam}}\, A \to \widehat{\mathsf{Lam}}\, A \to \widehat{\mathsf{Lam}}\, A & := \lambda t_1 \lambda t_2.\, \mathsf{in}_{\kappa 1}(\mathsf{inl}\,(\mathsf{inr}\,(\mathsf{inl}\,\langle t_1, t_2\rangle))) \\
\widehat{\mathsf{abs}} & : \forall A.\, \widehat{\mathsf{Lam}}(1 + A) \to \widehat{\mathsf{Lam}}\, A & := \lambda r.\, \mathsf{in}_{\kappa 1}(\mathsf{inl}\,(\mathsf{inr}\,(\mathsf{inr}\, r))) \\
\widehat{\mathsf{exs}} & : \forall A.\, \widehat{\mathsf{Lam}}(\widehat{\mathsf{Lam}}\, A) \to \widehat{\mathsf{Lam}}(A) & := \lambda e.\, \mathsf{in}_{\kappa 1}(\mathsf{inr}\, e)
\end{array}
$$

Example 5 (Host). Bird and Paterson [BP99a, Example 3.3] introduce the type transformer Host as an instructive example of true nesting. In GMIC this datatype can be represented as follows.

$$
\begin{array}{llll}
\mathsf{HostF} & : & \kappa 2 & := & \lambda F \lambda A.\, 1 + A \times F\,(A \times F\, A) \\
\mathsf{hostf} & : & \mathsf{mon}\, \mathsf{HostF} & := & \lambda s \lambda f.\, \mathsf{maybe}\,(\mathsf{pair}\ f\,(s\,(\mathsf{pair}\ f\,(s\, f)))) \\
\mathsf{Host} & : & \kappa 1 & := & \mu_{\kappa 1}\, \mathsf{HostF} \\
\mathsf{host} & : & \mathsf{mon}\, \mathsf{Host} & := & \mathsf{M}^{\mu}_{\kappa 1}(\mathsf{hostf})
\end{array}
$$

The mapping function host has the following reduction behavior.

$$
\begin{array}{rl}
\mathsf{host}\, f \circ \mathsf{in}_{\kappa 1} \longrightarrow_{\beta} & \mathsf{in}_{\kappa 1} \circ\ \mathsf{hostf}\ \mathsf{host}\, f \\
\longrightarrow^{+} & \mathsf{in}_{\kappa 1} \circ\ \mathsf{maybe}\ (\mathsf{pair}\, f\,(\mathsf{host}\,(\mathsf{pair}\, f\,(\mathsf{host}\, f))))
\end{array}
$$

Note that the reduct corresponds to the right-hand side of the defining recursive equation of host in the original work [BP99a]. However, their solution was only considered to exist in the semantical model of functor categories. Our system GMIC gives a direct operational justification—host is just an instance of the iterator $\mathsf{It}_{\kappa 1}$. Contrast this with Bird and Paterson's higher-order iterator *hfold*, which is too weak to implement mapping functions like this one.

To overcome the shortcomings of *hfold*, Bird and Paterson introduce *generalized folds (gfold)*, which allow them to define desired operations on nested datatypes like Host; we achieve the same with our notion of iteration and Kan extensions (see Sect. 3.1 below). Existence of *gfold* relies on the existence of mapping functions like host, whose termination is not thoroughly addressed in their work, but justified by our results.

Coinductive functors ($\kappa = \kappa 1$).

$$
\begin{array}{lll}
\text{Formation.} & \nu_{\kappa 1} & : \kappa 2 \to \kappa 1 \\
\text{Introduction.} & \mathsf{Coit}_{\kappa 1} & : \forall \Phi^{\kappa 2}.\, \mathsf{mon}\, \Phi \to \forall G^{\kappa 1}.\, G \subseteq \Phi\, G \to \\
& & \quad \forall A \forall B.\, (A \to B) \to G\, A \to \nu_{\kappa 1} \Phi\, B \\
\text{Elimination.} & \mathsf{out}_{\kappa 1} & : \forall \Phi^{\kappa 2} \forall A.\, \nu_{\kappa 1} \Phi\, A \to \Phi\,(\nu_{\kappa 1} \Phi)\, A \\
\text{Reduction.} & \multicolumn{2}{l}{\mathsf{out}_{\kappa 1}\,(\mathsf{Coit}_{\kappa 1}\, m\, s\, f\, t) \longrightarrow_{\beta} m\,(\mathsf{Coit}_{\kappa 1}\, m\, s)\, f\,(s\, t)}
\end{array}
$$

Similar to the inductive case, functoriality of $\nu_{\kappa 1}\Phi$ is witnessed by the uniform construction $\mathsf{M}^{\nu}_{\kappa 1}(m)$ for m any monotonicity witness for Φ.

Example 6 (Functions over binary trees). Thorsten Altenkirch [Alt01] shows how to encode functions over regular inductive types by elements of nested coinductive types of rank 2. In the following we present his example of functions over unlabelled binary trees ($\lambda A.\, \mathsf{BT} \to A$) as functor ($\mathsf{TFun}$) in our system.

$$\begin{array}{lll} \mathsf{TFunF} & : \kappa 2 & := \lambda F \lambda A.\, A \times F(FA) \\ \mathsf{tfunf} & : \mathsf{mon}\, \mathsf{TFunF} & := \lambda s \lambda f.\, \mathsf{pair}\, f\, (s\, (s\, f)) \end{array}$$

The coinductive type TFun is now obtained as the greatest fixed point of TFunF. We can derive its two destructors head and tail from the general destructor out for coinductive constructors.

$$\begin{array}{lll} \mathsf{TFun} & : \kappa 1 & := \nu_{\kappa 1} \mathsf{TFunF} \\ \mathsf{head} & : \forall A.\, \mathsf{TFun} A \to A & := \lambda b.\, (\mathsf{out}_{\kappa 1} b).0 \\ \mathsf{tail} & : \forall A.\, \mathsf{TFun} A \to \mathsf{TFun}(\mathsf{TFun} A) & := \lambda b.\, (\mathsf{out}_{\kappa 1} b).1 \end{array}$$

Every function $g : \mathsf{BT} \to A$ can be transformed via the function lamBT into an infinite object $\mathsf{lamBT}\, g : \mathsf{TFun} A$. We implement lamBT by coiteration.

$$\begin{array}{ll} \mathsf{lamBT}' : & (\lambda A.\, \mathsf{BT} \to A) \leq \mathsf{TFun} \\ \mathsf{lamBT}' := & \mathsf{Coit}_{\kappa 1}\, \mathsf{tfunf}\, (\lambda f.\, \langle f\ \mathsf{leaf},\ \lambda l \lambda r.\, f\, (\mathsf{span}\, l\, r) \rangle) \\ \mathsf{lamBT} \ : & \forall A.\, (\mathsf{BT} \to A) \to \mathsf{TFun} A \\ \mathsf{lamBT} \ := & \mathsf{lamBT}'\, \mathsf{id} \end{array}$$

Conversely, an object $b : \mathsf{TFun} A$ can be applied to a binary tree $t : \mathsf{BT}$ to yield an element $\mathsf{appBT}\ t\, b\ : A$. The application function appBT can be encoded by iteration on the tree t.

$$\begin{array}{ll} \mathsf{appBT} : & \mathsf{BT} \to \forall A.\, \mathsf{TFun} A \to A \\ \mathsf{appBT} := & \mathsf{ItBT}(\lambda t \lambda b.\, \mathsf{case}\, (t,\ \_.\ \mathsf{head}\, b,\ p.\ (p.1\, (p.0\, (\mathsf{tail}\, b))))) \end{array}$$

2.3 Beyond Rank 2

To the knowledge of the authors, inductive datatypes having type transformers as arguments are rarely considered. An example would be

$$\lambda A. \mu_{\kappa 1 \to \kappa 1 \to *} \Big(\lambda \Phi^{\kappa 1 \to \kappa 1 \to *} \lambda F \lambda G.\, F(FA) + (\Phi F(G \times G) + \Phi(F \times G)(G \times G)) \Big)$$

with $F \times G := \lambda A. FA \times GA$. It has been studied in [Oka99] for the efficient representation of square matrices. Squareness is ensured by types but without the use of dependent types, by which one would just express that there is an n such that the matrix has n rows and n columns. The argument to $\mu_{\kappa 1 \to \kappa 1 \to *}$ clearly has a monotonicity witness.

As a toy example, we mention $\mu_{\kappa 1 \to *} \Psi$ with $\Psi := \lambda F^{\kappa 1 \to *} \lambda G^{\kappa 1}. G(F(G \circ G))$. $\mathsf{mon}\, \Psi$ is inhabited by $\lambda s \lambda t. t(s(t \circ t))$.

2.4 Subject Reduction and Confluence

The extensions to system F^ω we made to incorporate (co)inductive constructors are *orthogonal* to the existing constructions like function space, products, sums etc. The new reduction rules do not interfere with any of the old ones and there are no critical pairs. Hence, confluence of GMIC immediately follows from standard results. To show subject reduction for GMIC, we only have to make sure that the new reduction rules preserve typing.

Proposition 1 ((Co)iteration is type-preserving). *Let $\kappa = \boldsymbol{\kappa} \to *$ be an arbitrary kind and $F : \kappa \to \kappa$, $G : \kappa$ and $X_i, Y_i : \kappa_i$ (for $1 \le i \le |\boldsymbol{\kappa}|$) be constructors. Furthermore let $f_i : X_i \le_{\kappa_i} Y_i$ be terms for $1 \le i \le |\boldsymbol{\kappa}|$ and $m : \mathsf{mon}\, F$.*

1. *Provided terms $s : FG \subseteq G$ and $t : F(\mu_\kappa F)\boldsymbol{X}$, the following typing derivations are correct.*

$$\begin{array}{rl} & \mathsf{It}_\kappa F\, m\, G\, s\, \boldsymbol{X}\, \boldsymbol{Y}\, \boldsymbol{f}\, (\mathsf{in}_\kappa F\, \boldsymbol{X}\, t) : G\boldsymbol{Y} \\ \longrightarrow_\beta^{Church} & s\, \boldsymbol{Y} \Big(m\, (\mu_\kappa F)\, G\, (\mathsf{It}_\kappa F\, m\, G\, s)\, \boldsymbol{X}\, \boldsymbol{Y}\, \boldsymbol{f}\, t \Big) : G\boldsymbol{Y} \end{array}$$

2. *Provided terms $s : G \subseteq FG$ and $t : G\boldsymbol{X}$ we have the following typings.*

$$\begin{array}{rl} & \mathsf{out}_\kappa F\, \boldsymbol{Y} (\mathsf{Coit}_\kappa F\, m\, G\, s\, \boldsymbol{X}\, \boldsymbol{Y}\, \boldsymbol{f}\, t) : F(\nu_\kappa F)\boldsymbol{Y} \\ \longrightarrow_\beta^{Church} & m\, G\, (\nu_\kappa F)\, (\mathsf{Coit}_\kappa F\, m\, G\, s)\, \boldsymbol{X}\, \boldsymbol{Y}\, \boldsymbol{f}\, (s\, \boldsymbol{X}\, t) : F(\nu_\kappa F)\boldsymbol{Y} \end{array}$$

Proof. By trivial type-checking. Note that the notation is slightly sloppy: instead of $\boldsymbol{X}\, \boldsymbol{Y}\, \boldsymbol{f}$, it should be $X_1 Y_1 f_1 \ldots X_{|\boldsymbol{\kappa}|} Y_{|\boldsymbol{\kappa}|} f_{|\boldsymbol{\kappa}|}$.

Subject reduction would follow trivially for a corresponding formulation of the whole term rewrite system à la Church (where $\longrightarrow_\beta^{Church}$ above would give the reduction rules pertaining to iteration and coiteration), but consequently by standard techniques also for our Curry-style presentation of GMIC. The desired property of strong normalization requires more work; in the following we prepare for an embedding into system F^ω.

3 Kan Extensions

In this section we define Kan extensions for constructors of arbitrary rank, show their most fundamental properties and demonstrate their use in programming with iterators. Finally, we use them to establish logical equivalence between the notion of monotonicity in [Mat01] and rank-2 monotonicity $\mathsf{mon}_{\kappa 2}\, \Phi$ proposed in this article.

Kan extensions along the identity. Let $G : \boldsymbol{\kappa} \to *$ and $Y_i : \kappa_i$ ($0 \le i < |\boldsymbol{\kappa}|$) be constructors. Then the *right Kan extension of $G\boldsymbol{Y}$ along the identity* is a type defined by recursion on the length of $\boldsymbol{Y}$ as follows:

$$\begin{array}{ll} \mathsf{Ran}\, G\ (\cdot) & := G \\ \mathsf{Ran}\, G\ (Y^\kappa, \boldsymbol{Y}) & := \forall Z^\kappa.\ Y \le_\kappa Z \to \mathsf{Ran}\, (GZ)\, (\boldsymbol{Y}) \end{array}$$

Similarily, the *left Kan extension of $G\boldsymbol{Y}$ along the identity* is the following type:

$$\begin{aligned} \mathsf{Lan}\,G\ (\cdot) &:= G \\ \mathsf{Lan}\,G\ (Y^{\kappa}, \boldsymbol{Y}) &:= \exists X^{\kappa}.\ X \leq_{\kappa} Y \times \mathsf{Lan}\,(GX)\,(\boldsymbol{Y}) \end{aligned}$$

These definitions are a syntactic rendering of the Kan extensions in category theory [Lan98, chapter 10] which become especially simple due to the presence of universal and existential types. We will not discuss any of their category-theoretic properties. At this point, let us only note how Kan extensions relate to our notion "$\leq_{\kappa}$".

Proposition 2. *Let $F, G : \boldsymbol{\kappa} \to *$. The following types are logically equivalent:*[2]

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

Proof. By induction on $\boldsymbol{\kappa}$. Part 1. $\iff$ 2. is done by rearranging quantifiers and arguments, Part 2. $\iff$ 3. by currying and uncurrying.

Rank-1 right Kan extension along a functor H. For programming with iterators (see next section), we need the more general right *Kan extension along H* for some type transformer H. We define for constructors G, H of pure kind $\kappa 1$ the type

$$\mathsf{Ran}_H\,G\,(A) := \forall B.\,(A \to HB) \to GB$$

For the special case of the identity functor $H = \lambda A.A$ we obtain the right Kan extension defined above. We will write $\mathsf{Ran}_H\,G$ for $\lambda A.\,\mathsf{Ran}_H\,G\,(A)$ and $\mathsf{Ran}\,G$ for $\lambda A.\,\mathsf{Ran}\,G\,(A)$. The left Kan extension could be modified similarly, but for our purposes the right Kan extension is sufficient.

3.1 Programming with Kan extensions

In this section we demonstrate how right Kan extensions provide a general tool to program with rank-2 inductive datatypes. It is known [BP99a,Hin00] that a function $f : \mu_{\kappa 1}\Phi \circ H \subseteq G$ for $G, H : \kappa 1$ cannot be defined by iteration directly if $H \neq \lambda A.A$. The solution to this difficulty proposed in the cited articles is *generalized iteration* (also called "generalized fold"). Another solution (but related to the first one as a method for the justification of generalized iteration [BP99a, section 6.2]), is to define a more general auxiliary function g of type $\mu_{\kappa 1}\Phi \leq \mathsf{Ran}_H\,G$, from which we get f by the trivial specialization $f := \lambda r.\,g\ \mathsf{id}\ r\ \mathsf{id}$. We will demonstrate this technique by continuing our examples from Section 2.

[2] A and B are called logically equivalent if there are closed terms in system F^{ω} of types $A \to B$ and $B \to A$. It is thus equivalence in second-order propositional logic.

Example 7 (Summing up a powerlist). Assume a type Nat of natural numbers with addition "+", written infix. A function sum which adds up all entries in a powerlist of naturals has type $\mathsf{PList}\,\mathsf{Nat} \to \mathsf{Nat}$. This is an instance of the problem described above where $G = H = \lambda A.\mathsf{Nat}$. The right Kan extension of G along H simplifies to $\lambda A.\,(A \to \mathsf{Nat}) \to \mathsf{Nat}$, hence we define the following auxiliary function.

$$\begin{array}{ll}\mathsf{sum}' & :\ \ \mathsf{PList} \leq (\lambda A.\,(A \to \mathsf{Nat}) \to \mathsf{Nat}) \\ \mathsf{sum}' & :=\ \mathsf{It}_{\kappa 1}\ \mathsf{plistf}\ \mathsf{stepSum}\end{array}$$

$$\begin{array}{ll}\mathsf{stepSum} & :\ \ \mathsf{PListF}(\lambda A.\,(A \to \mathsf{Nat}) \to \mathsf{Nat}) \subseteq (\lambda A.\,(A \to \mathsf{Nat}) \to \mathsf{Nat}) \\ \mathsf{stepSum} & :=\ \lambda t \lambda f.\,\mathsf{case}\,(t,\ a.\,f\,a,\ l.\ l\ (\lambda p.\,f\,(p.0) + f\,(p.1)))\end{array}$$

Following our general recipe, the summation function is obtained as $\mathsf{sum} := \lambda l.\ \mathsf{sum}'\,\mathsf{id}\,l\ \mathsf{id}$.

Example 8 (Substitution for de Bruijn terms). De Bruijn terms constitute a Kleisli triple $(\mathsf{Lam}, \mathsf{var}, \mathsf{subst})$ with unit $\mathsf{var} : \forall A.\,A \to \mathsf{Lam}\,A$ and bind operation

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

The reader will note that the type of subst can also be written as $\forall A.\,\mathsf{Lam}\,A \to \mathsf{Ran}_{\mathsf{Lam}}\,\mathsf{Lam}\,(A)$. This suggests that substitution can be defined by iteration (proven by Altenkirch/Reus [AR99]).

$$\begin{array}{ll}\mathsf{subst}' & :\ \ \mathsf{Lam} \leq \mathsf{Ran}_{\mathsf{Lam}}\ \mathsf{Lam} \\ \mathsf{subst}' & :=\ \mathsf{It}_{\kappa 1}\ \mathsf{lamf}\ \mathsf{stepSub}\end{array}$$

$$\begin{array}{ll}\mathsf{stepSub} & :\ \ \mathsf{LamF}\,(\mathsf{Ran}_{\mathsf{Lam}}\ \mathsf{Lam}) \subseteq \mathsf{Ran}_{\mathsf{Lam}}\ \mathsf{Lam} \\ \mathsf{stepSub} & :=\ \lambda t \lambda \sigma.\,\mathsf{case}\,(t,\ a.\,\sigma\,a,\ t'.\,\mathsf{case}\,(t', \\ & \qquad\qquad p.\ \mathsf{app}\,(p.0\,\sigma)\,(p.1\,\sigma), \\ & \qquad\qquad r.\ \mathsf{abs}\,(r\ \lambda x.\,\mathsf{case}\,(x,\ u.\ \mathsf{var}\,(\mathsf{inl}\,u),\ a.\ \mathsf{weak}\,(\sigma\,a)))))\end{array}$$

The substitution function is obtained by specialization: $\mathsf{subst} := \mathsf{subst}'\,\mathsf{id}$.

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

Example 9 (Resolving explicit substitutions). The set of de Bruijn terms Lam can be embedded into the set of de Bruijn terms $\widehat{\mathsf{Lam}}$ with explicit substitution. The embedding function $\mathsf{emb} : \forall A.\ \mathsf{Lam}\,A \to \widehat{\mathsf{Lam}}\,A$ can be defined by iteration in a straightforward manner. The other direction is handled by a function $\mathsf{eval} : \forall A.\ \widehat{\mathsf{Lam}}\,A \to \mathsf{Lam}\,A$ which has to resolve the explicit substitutions.

$$\begin{array}{ll}\mathsf{eval}' & :\ \ \widehat{\mathsf{Lam}} \leq \mathsf{Lam} \\ \mathsf{eval}' & :=\ \mathsf{It}_{\kappa 1}\ \widehat{\mathsf{lamf}}\ \mathsf{stepEv}\end{array}$$

$$\begin{array}{ll}\mathsf{stepEv} & :\ \ \widehat{\mathsf{LamF}}\,\mathsf{Lam} \subseteq \mathsf{Lam} \\ \mathsf{stepEv} & :=\ \lambda t.\ \mathsf{case}\,(t,\ t'.\,\mathsf{in}_{\kappa 1}\,t',\ e.\ \mathsf{join}\ e)\end{array}$$

Note how join is used to carry out the substitutions. The evaluation function is just $\mathsf{eval} := \mathsf{eval}'\ \mathsf{id}$.

3.2 Comparison with Special Monotonicity [Mat01]

In [Mat01], the second author has introduced another notion of monotone inductive and coinductive constructors with μ_κ and ν_κ exclusively for $\kappa \in \{*, * \to *\}$. The associated notions of monotonicity are for constructors of kind $\kappa 1$ and $\kappa 2$. Certainly, for $\kappa 1$, there is only the definition we use in the present article. However, the notion for $\kappa 2$ has been the following:

$$\underline{\mathsf{mon}}\,\Phi := (\forall F.\,\mathsf{mon}\,F \to \mathsf{mon}\,\Phi F) \times (\forall G \forall H. G \subseteq H \to \\ (\mathsf{mon}\,G \to \Phi G \subseteq \Phi H) \times (\mathsf{mon}\,H \to \Phi G \subseteq \Phi H))$$

This "special" notion of monotonicity has been designed so that it is as close as possible to what one expects from category theory, but departing from it as much as needed so that, by help of Kan extensions, inhabitation of $\underline{\mathsf{mon}}\,\Phi \to \mathsf{mon}(\mu_{\kappa 1}\Phi)$ could be shown. Moreover, many closure properties of monotonicity could be established, going far beyond algebraic datatypes such as the ones in our present examples. Unfortunately, that notion does not seem to extend to other kinds of rank 2, not to speak of arbitrary ranks.

Here, we show that, from a logical point of view, there is no difference between special monotonicity $\underline{\mathsf{mon}}\,\Phi$ and general monotonicity $\mathsf{mon}\,\Phi$, i. e., both types are logically equivalent. With respect to the rewrite rules, there are differences which cannot be addressed here for lack of space.

In this subsection, F, G, H always have kind $\kappa 1$ and Φ has kind $\kappa 2$. The direction from general monotonicity to special monotonicity does not require Kan extensions and can be programmed as follows:

$$\begin{array}{ll}
\mathsf{c00} & : \ \forall \Phi.\ \mathsf{mon}\,\Phi \to \forall F.\ \mathsf{mon}\,F \to \mathsf{mon}\,\Phi F \\
\mathsf{c00} & := \lambda m \lambda n.\, m\, n \\
\mathsf{monSubLeq} & : \ \forall G \forall H.\ \mathsf{mon}\,G \to G \subseteq H \to G \le H \\
\mathsf{monSubLeq} & := \lambda m \lambda g \lambda f \lambda x.\, g\,(m\, f\, x) \\
\mathsf{subMonLeq} & : \ \forall G \forall H. G \subseteq H \to \mathsf{mon}\,H \to G \le H \\
\mathsf{subMonLeq} & := \lambda g \lambda m \lambda f \lambda x.\, m\, f\,(g\, x) \\
\mathsf{c10} & : \ \forall \Phi.\ \mathsf{mon}\,\Phi \to \forall G \forall H. G \subseteq H \to \mathsf{mon}\,G \to \Phi G \subseteq \Phi H \\
\mathsf{c10} & := \lambda m \lambda \ell \lambda n \lambda t.\, m\,(\mathsf{monSubLeq}\, n\, \ell)\ \mathsf{id}\ t \\
\mathsf{c11} & : \ \forall \Phi.\ \mathsf{mon}\,\Phi \to \forall G \forall H. G \subseteq H \to \mathsf{mon}\,H \to \Phi G \subseteq \Phi H \\
\mathsf{c11} & := \lambda m \lambda \ell \lambda n \lambda t.\, m\,(\mathsf{subMonLeq}\, \ell\, n)\ \mathsf{id}\ t \\
\mathsf{gmsm} & : \ \forall \Phi.\ \mathsf{mon}\,\Phi \to \underline{\mathsf{mon}}\,\Phi \\
\mathsf{gmsm} & := \lambda m.\, \langle \mathsf{c00}\, m, \lambda \ell.\, \langle \mathsf{c10}\ m\, \ell,\ \mathsf{c11}\ m\, \ell \rangle \rangle
\end{array}$$

We come to the interesting direction from special monotonicity to general monotonicity: Under the assumptions $\underline{\mathsf{mon}}\,\Phi$, $G \le H$ and $A \to B$, we have to show $\Phi G A \to \Phi H B$. This is done via two intermediate types: We show $\Phi G A \to \Phi(\mathsf{Ran}\, H) A \to \Phi(\mathsf{Ran}\, H) B \to \Phi H B$. The first step is an instance of $\Phi G \subseteq$

$\Phi(\mathsf{Ran}\,H)$ which follows from $\underline{\mathsf{mon}}\,\Phi$, $G \subseteq \mathsf{Ran}\,H$ and $\mathsf{mon}(\mathsf{Ran}\,H)$.

$$\begin{array}{ll}
\mathsf{monRan} : & \forall H.\ \mathsf{mon}(\mathsf{Ran}\,H) \\
\mathsf{monRan} := & \lambda f \lambda k \lambda g.\, k\,(g \circ f)
\end{array}$$

$$\begin{array}{lll}
\mathsf{leqRan} & : & \forall G \forall H.\, G \leq H \to G \subseteq \mathsf{Ran}\,H \ \text{(as an instance of proposition 2)} \\
\mathsf{leqRan} & := & \lambda s \lambda x \lambda f.\, s f x
\end{array}$$

$$\begin{array}{lll}
\mathsf{t1} & : & \forall \Phi.\ \underline{\mathsf{mon}}\,\Phi \to \forall G \forall H.\, G \leq H \to \Phi G \subseteq \Phi(\mathsf{Ran}\,H) \\
\mathsf{t1} & := & \lambda m \lambda s.\, (m.1\,(\mathsf{leqRan}\,s)).1\ \mathsf{monRan}
\end{array}$$

The second step is just monotonicity of $\Phi(\mathsf{Ran}\,H)$, following from $\underline{\mathsf{mon}}\,\Phi$ and $\mathsf{mon}(\mathsf{Ran}\,H)$.

$$\begin{array}{ll}
\mathsf{t2} : & \forall \Phi.\ \underline{\mathsf{mon}}\,\Phi \to \forall H.\ \mathsf{mon}(\Phi(\mathsf{Ran}\,H)) \\
\mathsf{t2} := & \lambda m.\, (m.0)\ \mathsf{monRan}
\end{array}$$

The third step comes from $\Phi(\mathsf{Ran}\,H) \subseteq \Phi H$ which in turn uses $\underline{\mathsf{mon}}\,\Phi$, $\mathsf{Ran}\,H \subseteq H$ and, once again, $\mathsf{mon}(\mathsf{Ran}\,H)$.

$$\begin{array}{ll}
\mathsf{subRan} : & \forall H.\ \mathsf{Ran}\,H \subseteq H \\
\mathsf{subRan} := & \lambda k.\, k\,\mathsf{id}
\end{array}$$

$$\begin{array}{lll}
\mathsf{t3} & : & \forall \Phi.\ \underline{\mathsf{mon}}\,\Phi \to \forall H.\, \Phi(\mathsf{Ran}\,H) \subseteq \Phi H \\
\mathsf{t3} & := & \lambda m.\, (m.1\ \mathsf{subRan}).0\ \mathsf{monRan}
\end{array}$$

$$\begin{array}{lll}
\mathsf{smgm} & : & \forall \Phi.\ \underline{\mathsf{mon}}\,\Phi \to \mathsf{mon}\,\Phi \\
\mathsf{smgm} & := & \lambda m \lambda s \lambda f \lambda x.\ \mathsf{t3}\,m\,(\mathsf{t2}\,m\,f\,(\mathsf{t1}\,m\,s\,x))
\end{array}$$

Apparently, the crucial idea is the formation of the monotone (even non-strictly positive) constructor $\mathsf{Ran}\,H$ from an arbitrary type transformer H.

4 Embedding into System F^ω

The purpose of this section is a perspicuous proof of strong normalization of GMIC. In fact, we show that the new constructor and term constants can already be defined in System F^ω so that typing is preserved, and that for the defined terms, one has in F^ω that

$$\begin{array}{ll}
\mathsf{It}_{\kappa\to *}\,m\,s\,\boldsymbol{f}\,(\mathsf{in}_{\kappa\to *}\,t) & \longrightarrow^+ s\,(m\,(\mathsf{It}_{\kappa\to *}\,m\,s)\,\boldsymbol{f}\,t) \\
\mathsf{out}_{\kappa\to *}\,(\mathsf{Coit}_{\kappa\to *}\,m\,s\,\boldsymbol{f}\,t) & \longrightarrow^+ m\,(\mathsf{Coit}_{\kappa\to *}\,m\,s)\,\boldsymbol{f}\,(s\,t).
\end{array}$$

Therefore, if there is no typable term in F^ω with an infinite reduction sequence, there is neither a typable term in GMIC allowing an infinite sequence of reduction steps. In other words, strong normalization is inherited from that of F^ω which in turn is a well-known fact [Gir72].[3]

[3] In that original work, only weak normalization has been proven but an extension to strong normalization is standard technology since [Tai75].

Let $\kappa = \boldsymbol{\kappa} \to *$, set $n := |\boldsymbol{\kappa}|$, and define for $|\boldsymbol{f}| = n$

$$
\begin{array}{ll}
\mu_\kappa & := \lambda F^{\kappa \to \kappa} \lambda \boldsymbol{Y}^{\boldsymbol{\kappa}}.\, \mathsf{mon}\, F \to \forall G^\kappa. (F\, G \subseteq G) \to (\mathsf{Ran}\, G\, (\boldsymbol{Y})) \\
\mathsf{It}_\kappa & := \lambda m \lambda s \lambda \boldsymbol{f} \lambda r.\, r m s \boldsymbol{f} \\
\mathsf{in}_\kappa & := \lambda t \lambda m \lambda s \lambda \boldsymbol{f}.\, s \Big(m (\mathsf{It}_\kappa m s) \boldsymbol{f} t \Big) \\
\nu_\kappa & := \lambda F^{\kappa \to \kappa} \lambda \boldsymbol{Y}^{\boldsymbol{\kappa}}.\, \mathsf{mon}\, F \times \exists G^\kappa. (G \subseteq F\, G) \times (\mathsf{Lan}\, G\, (\boldsymbol{Y})) \\
\mathsf{Coit}_\kappa & := \lambda m \lambda s \lambda \boldsymbol{f} \lambda t.\, \langle m, \mathsf{pack} \langle s, \mathsf{pack} \langle f_1, \ldots \mathsf{pack} \langle f_n, t \rangle \ldots \rangle \rangle \rangle \\
\mathsf{out}_\kappa & := \lambda r.\, \mathsf{open}\, (r.1,\, p_0.\, \mathsf{open}\, (p_0.1,\, p_1.\, \ldots \mathsf{open}\, (p_{n-1}.1,\, p_n.\, v) \ldots)) \\
 & \qquad \text{with } v := r.0 \Big(\mathsf{Coit}_\kappa (r.0)(p_0.0) \Big) (p_1.0) \ldots (p_n.0) \Big(p_0.0\, (p_n.1) \Big)
\end{array}
$$

Compared with the classical encoding of (co-)inductive datatypes [Wra89,Geu92], the new ingredients are the relativization to monotone F and the use the Kan extensions. Welltypedness and the purported reduction behaviour are a matter of trivial calculation.

By the reasoning above, this yields the following theorem. (We now use "$\longrightarrow$" to denote the one-step reduction relation of the full system GMIC.)

Theorem 1 (Strong Normalization). *Whenever $\Gamma \vdash r : A$ in* GMIC, *then r admits no infinite reduction sequence $r \longrightarrow r_1 \longrightarrow r_2 \longrightarrow \ldots$* □

5 Conclusions and Further Work

The System GMIC presented in this article is an idealized programming language with support for arbitrarily nested datatypes of arbitrary kind (i. e., higher order type transformers). The key ingredient is a notion of monotonicity which is wider than any notion of positivity and goes far beyond polynomial or algebraic higher-order functors. We have shown that typical examples can be treated easily.

We would hope for many more examples that exploit the capabilities of GMIC—examples that use

- nesting in the second sense of our introduction, i. e., simultaneous inductive and coinductive constructors, combined with heterogeneity,
- non-strict positivity, i. e., arguments that occur an even time to the left of an arrow $\to$,
- monotonicity of $\mu_{\kappa 1} \Phi$ for the formation of new datatypes such as $\mu_*(\mu_{\kappa 1} \Phi)$,
- μ_κ and ν_κ for rank $\mathsf{rk}(\kappa) > 1$.

References

[Abe03] Andreas Abel. Haskell examples for iteration and coiteration on higher-order datatypes. Haskell code accompanying this article, available on the author's homepage, 2003.

[Alt01] Thorsten Altenkirch. Representations of first order function types as terminal coalgebras. In Samson Abramsky, editor, *Fifth International Conference on Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 8–21. Springer, 2001.

[AM02] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. To appear in the proceedings of WCGP'02, 2002.

[AR99] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Tom S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 117–309. 1992.

[BFPS81] Wilfried Buchholz, Solomon Feferman, Wolfram Pohlers, and Wilfried Sieg. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies*, volume 897 of *Lecture Notes in Mathematics*. Springer Verlag, 1981.

[BGJ00] Richard Bird, Jeremy Gibbons, and Geraint Jones. Program optimisation, naturally. In *Millenial Perspectives in Computer Science*, Palgrave, 2000.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer Verlag, 1998.

[BP99a] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[BP99b] Richard S. Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[Geu92] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In Bengt Nordström, Kent Pettersson, and Gordon Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, pages 193–217, 1992. Only published via `ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.dvi.Z`.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII, 1972.

[Hag87] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *Lecture Notes in Computer Science*, pages 140–157. Springer Verlag, 1987.

[Hin00] Ralf Hinze. Efficient generalized folds. In Johan Jeuring, editor, *Proceedings of the Second Workshop on Generic Programming, WGP 2000*, Ponte de Lima, Portugal, July 2000.

[Hin02] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming, MPC Special Issue*, 43:129–159, 2002.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[JBM98] C. Barry Jay, Gianna Bellè, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[Lan98] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, second edition, 1998.

[Lei90] Daniel Leivant. Contracting proofs to programs. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, volume 31 of *APIC Studies in Data Processing*, pages 279–327. Academic Press, 1990.

[Mat98] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types.* PhD thesis, Ludwig-Maximilians-University, May 1998.

[Mat01] Ralph Matthes. Monotone inductive and coinductive constructors of rank 2. In Laurent Fribourg, editor, *Proceedings of CSL 2001*, volume 2142 of *Lecture Notes in Computer Science*, pages 600–614. Springer Verlag, 2001.

[Men87] Nax P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, N.Y.*, pages 30–36. IEEE Computer Society Press, 1987.

[Oka99] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*, volume 34 of *SIGPLAN Notices*, pages 28–35. ACM, 1999.

[Tai75] William W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Logic Colloquium Boston 1971/72*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer Verlag, 1975.

[Wra89] G. C. Wraith. A note on categorical datatypes. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 118–127. Springer Verlag, 1989.

A System F^ω

In the following we present Curry-style System F^ω enriched with binary sums and products, empty and 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 and rank.

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

Constructors. (Denoted by uppercase letters)

$$\begin{array}{rl} A, B, F, G ::= & X \mid \lambda X.F \mid F\,G \mid \forall F^\kappa.\,A \mid \exists F^\kappa.\,A \mid A \to B \\ & \mid 0 \mid A + B \mid 1 \mid A \times B \end{array}$$

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)

$$\begin{aligned} r, s, t ::= {} & x \mid \lambda x.t \mid r\, s \mid \mathsf{abort}\, r \mid \mathsf{inl}\, t \mid \mathsf{inr}\, t \mid \mathsf{case}\,(r,\, x.\, s,\, y.\, t) \\ & \mid \langle\rangle \mid \langle t_1, t_2\rangle \mid r.0 \mid r.1 \mid \mathsf{pack}\, t \mid \mathsf{open}\,(r,\, x.\, s) \end{aligned}$$

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

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

Judgments. (Simultaneously defined)

Γ 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. Γ 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. $\Gamma \vdash F : \kappa$

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

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

$$\frac{\Gamma\ \mathsf{cxt}}{\Gamma \vdash 0 : *} \qquad \frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A + B : *} \qquad \frac{\Gamma\ \mathsf{cxt}}{\Gamma \vdash 1 : *} \qquad \frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A \times B : *}$$

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

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

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

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

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

$$\frac{\Gamma \vdash r : A + B \qquad \Gamma, x\!:\!A \vdash s : C \qquad \Gamma, y\!:\!B \vdash t : C}{\Gamma \vdash \mathsf{case}\,(r,\, x.\, s,\, y.\, t) : C} \qquad \frac{\Gamma \vdash r : 0 \qquad \Gamma \vdash C : *}{\Gamma \vdash \mathsf{abort}\, r : C}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \langle s, t\rangle : A \times B} \qquad \frac{\Gamma \vdash r : A_0 \times A_1 \quad i \in \{0,1\}}{\Gamma \vdash r.i : A_i}$$

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.

B Notation for Typing Derivations

As is usual, typing derivations for F^ω will be communicated in a short-hand form, namely by the raw terms of a Church-style variant for which, given a context, typing and type-checking is a trivial matter. Here, we introduce those raw terms simultaneously with their typing rules.

The rules for $x, rs, \langle\rangle, \langle s,t\rangle$ and $r.i$ are the same as for F^ω. The others are:

$$\frac{\Gamma, x\!:\!A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B} \qquad \frac{\Gamma \vdash r : 0 \qquad \Gamma \vdash C : *}{\Gamma \vdash \mathsf{abort}_C\, r : C}$$

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

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

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : *}{\Gamma \vdash \mathsf{inl}_B\, t : A + B} \qquad \frac{\Gamma \vdash t : B \qquad \Gamma \vdash A : *}{\Gamma \vdash \mathsf{inr}_A\, t : A + B}$$

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