Fixed Points of Type Constructors and Primitive Recursion

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

[1] Department of Computer Science, Chalmers University of Technology
abel@cs.chalmers.se
[2] Department of Computer Science, University of Munich
matthes@informatik.uni-muenchen.de

Abstract. For nested or heterogeneous datatypes, terminating recursion schemes considered so far have been instances of iteration, excluding efficient definitions of fixed-point unfolding. Two solutions of this problem are proposed: The first one is a system with equi-recursive non-strictly positive type constructors of arbitrary finite kinds, where fixed-point unfolding is computationally invisible due to its treatment on the level of type equality. Positivity is ensured by a polarized kinding system, and strong normalization is proven by a model construction based on saturated sets. The second solution is a formulation of primitive recursion for arbitrary type constructors of any rank. Although without positivity restriction, the second system embeds—even operationally—into the first one.

1 Introduction

Recently, higher-rank datatypes have drawn interest in the functional programming community [Oka96,Hin01]. Rank-2 non-regular types, so-called *nested datatypes*, have been investigated in the context of the functional programming language Haskell. To define total functions which traverse nested datastructures, Bird et al. [BP99a] have developed *generalized folds* which implement an iteration scheme and are strong enough to encode most of the known algorithms for nested datatypes. In this work, we investigate schemes to overcome some limitations of iteration which we expound in the following.

Since the work of Böhm *et al.* [BB85] it is well-known that iteration for rank-1 datatypes can be simulated in typed lambda calculi. The easiest examples are iterative definitions of addition and multiplication for Church numerals. The iterative definition of the predecessor, however, is inefficient: It traverses the whole numeral in order to remove one constructor. Surely, taking the predecessor should run in constant time.

Primitive recursion is the combination of iteration and efficient predecessor. A typical example for a primitive recursive algorithm is the natural definition

* The first author gratefully acknowledges the support by both the PhD Programme *Logic in Computer Science* (GKLI) of the *Deutsche Forschungs-Gemeinschaft* and the CoVer project by the *Stiftelsen för Strategisk Forskning* (SSF).

of the factorial function. It is common belief that primitive recursion cannot be reduced to iteration in a computationally faithful manner. This is because no encoding of natural numbers in the polymorphic lambda-calculus (System F) seems possible which supports a constant-time predecessor operation (see Spławski and Urzyczyn [SU99]).

In this article, we present two approaches to overcome the predecessor dilemma for higher-rank datatypes. A *first* solution, presented in Section 2, is System Fix^ω of non-strictly positive equi-recursive type constructors, which handles folding and unfolding for fixed points on the level of types, trivially yielding an efficient predecessor. Fix^ω is proven strongly normalizing in Section 3. Even though the system has no native means of recursion, a powerful scheme of primitive recursion is definable in Fix^ω. This schema is embodied in our formulation of a *second* system MRec^ω, given in Section 4. In Section 4.2 we give an extensive example of a function which can most naturally be implemented with primitive recursion—redecoration for triangular matrices. Finally, we give the details of the definition of MRec^ω within Fix^ω, hence establishing strong normalization of the primitive recursion scheme as well (Section 5).

2 System Fix^ω

Since Mendler [Men87], it is known that type equations of the form $X = A$ with X a type variable and A a type expression, can only be added to system F in case X only occurs positively in A. Otherwise, strong normalization of typable terms is lost. In this section, we show that the positive part of Mendler's finding, namely strong normalization in the case where X only occurs positively in A, can be extended to equations for type constructors of arbitrary finite kind, hence within the framework F^ω of higher-order parametric polymorphism.

Equations in solved form, i. e., with a constructor variable on the left-hand side, can equivalently be treated by an explicit type constructor for fixed-points [Urz96]. In the case of fixed-points of types, the purported solution of $X = C$ would be written as the type $\mathsf{fix}\, X.C$, with its characteristic equation being $\mathsf{fix}\, X.C = [\mathsf{fix}\, X.C/X]C$. In the case of nested datatypes, we are interested in equations like $X\,A = A + X(A \times A)$, where X now denotes a *type transformation.* This would be solved by $\mathsf{PList} = \mathsf{fix}\, X.\lambda A.\, A + X(A \times A)$. $\mathsf{PList}\, A$ stands for powerlists over A, i. e., lists with 2^n elements of type A, for some (unspecified) n, which can clearly be seen to be the least fixed-point of the above equation. With function kinds at hand, we can pass from $\mathsf{fix}\, X.C$ to $\mathsf{fix}\, F$ with $F := \lambda X.C$, which is a type transformer.

A manifest idea to isolate positive constructors systematically is to distinguish covariant (monotone), contravariant (antitone) and invariant (no information about monotonicity) constructors through the kinding system. Such systems have been found independently by L. Cardelli, B. C. Pierce, the first author and others, but published only by Steffen [Ste98] and Duggan and Compagnoni [DC98]. In both publications, polarized kinds are used to model subtyping of container types like lists and arrays in object-oriented calculi. We are reusing

Polarities	p	$::=$	$+$	covariant
		$\mid$	$-$	contravariant
		$\mid$	$\circ$	invariant
Kinds	κ	$::=$	$* \mid p\kappa \to \kappa'$	
Constructors	A, B, F, G	$::=$	$X \mid \lambda X^{p\kappa}.\, F \mid F\,G \mid A \to B \mid \forall X^{\kappa}.\, A \mid \mathsf{fix}\, F$	
Objects (terms)	r, s, t	$::=$	$x \mid \lambda x.t \mid r\, s$	
Contexts	Δ	$::=$	$\diamond \mid \Delta, x\!:\!A \mid \Delta, X^{p\kappa}$	

Table 1. Language of Fix^{ω}

their ideas to formulate positive recursive constructors in a strongly normalizing language.

Each function kind $\kappa \to \kappa'$ is decorated with a polarity, yielding $p\kappa \to \kappa'$ in Duggan and Compagnoni's notation. For covariant constructors, $p = +$, for contravariant, $p = -$, and $p = \circ$ if the constructor is neither co- nor contravariant or its variance is unknown. Consequently, abstracted variables now carry kinding and polarity information. For instance, we have

$$\lambda X^{+(+*\to*)} \lambda A^{+*}.\, X\,(X\,A) \;:\; +(+* \to *) \to (+* \to *).$$

The kinding expresses that $X \circ X$ is covariant if X is, and that the "twice" operation $\lambda X.X \circ X$ is itself covariant on covariant arguments, meaning that we may form its fixed point, which would in turn be covariant.[1] We can also classify invariant constructors, e. g., $\lambda A^{\circ *}.\, A \to X\,(A \times A) : \circ* \to *$ for invariant X that occurs covariantly, indicated by $X^{+(\circ*\to*)}$ in the context, and contravariant constructors like $\lambda X^{-*}.\, X \to \bot : -* \to *$. Consequently, $\lambda X^{+*}.\,(X \to \bot) \to \bot : +* \to *$, which hence includes non-strictly positive type transformers.[2]

Language of Fix^{ω}*.* Table 1 shows the syntactic entities of System Fix^{ω}, an extension of F^{ω} by polarized kinds and fixed-points of constructors. Typically, the empty context "$\diamond$" will be suppressed. Furthermore we assume all variables in a context Δ to be pairwise distinct. Capture-avoiding substitution of constructor G for variable X in constructor F is written as $[G/X]F$, likewise substitution in terms is denoted by $[s/x]t$. As usual, it is assumed that constructor application and term application associate to the left, e. g., $F\,G\,X$ denotes $(F\,G)\,X$ and $(\lambda x.r)\,s\,t$ denotes $((\lambda x.r)\,s)\,t$. Iterated applications may be "vectorized", i. e., $r\,t_1 \dots t_n$ will be written as $r\boldsymbol{t}$ with $\boldsymbol{t} := t_1, \dots, t_n$. Then, $|\boldsymbol{t}| := n$.

While we are using Curry-style objects to express solely the operational behavior, for the type constructors we decided on Church style in order to simplify

[1] The kind of $\lambda X.X \circ X$ obtained here is a syntactic approximation and simplification of the more logic-based concept of rank-2 monotonicity introduced in [Mat01].

[2] Most dependently typed systems such as Coq do not allow non-strict positivity for their native fixed points due to the consistency problem reported in [CP88].

the semantics definition in Section 3. The same decision has been taken by Giannini et al. [GHR93], where equivalence with pure Church typing is also established. Note, however, that impredicative systems with dependent types are richer in this mixed style [vBL+97].

Operations on polarities and contexts. Negation of a polarity $-p$ is given by the three equations $-(+) = -$, $-(-) = +$ and $-(\circ) = \circ$. We define application $p\Delta$ of a polarity p to a polarized context Δ. Positive polarity is neutral and changes nothing: $+\Delta = \Delta$. The operation $-\Delta$ reverses all polarities in Δ. Furthermore $\circ\Delta$ discards all co- and contravariant type variable bindings.

Kinding. We introduce a judgement $\Delta \vdash F : \kappa$ which combines the usual notions of wellkindedness and positive and negative occurrences of type variables. It assures that fixed-points can only be formed over positive type constructors.

$$\frac{X^{p\kappa} \in \Delta \quad p \in \{+, \circ\}}{\Delta \vdash X : \kappa} \qquad \frac{\Delta, X^{p\kappa} \vdash F : \kappa'}{\Delta \vdash \lambda X^{p\kappa}.\, F : p\kappa \to \kappa'} \qquad \frac{\Delta \vdash F : p\kappa \to \kappa' \quad p\Delta \vdash G : \kappa}{\Delta \vdash F\, G : \kappa'}$$

$$\frac{-\Delta \vdash A : * \qquad \Delta \vdash B : *}{\Delta \vdash A \to B : *} \qquad \frac{\Delta, X^{\circ\kappa} \vdash A : *}{\Delta \vdash \forall X^{\kappa}.\, A : *} \qquad \frac{\Delta \vdash F : +\kappa \to \kappa}{\Delta \vdash \mathsf{fix}\, F : \kappa}$$

Kinding is syntax-directed, and, since we are using Church-style constructors, for given Δ and F, the kind κ of F can be computed by structural recursion on F. As a consequence, all rules are invertible in the strong sense that we can recover the applied rule and all the parts of its premises from a given kinding judgement.

The arrow in kinds and in types is assumed to associate to the right, e. g., $A \to B \to C$ stands for $A \to (B \to C)$ and $+* \to -\kappa \to \kappa'$ stands for $+* \to (-\kappa \to \kappa')$.

Example 1. We can define standard type constructors via the usual impredicative encodings and get more informative kinds:

$$\begin{array}{ll}
\times & : \quad +* \to +* \to * \\
\times & := \lambda X^{+*} \lambda Y^{+*} \forall Z^*.\, (X \to Y \to Z) \to Z \\
+ & : \quad +* \to +* \to * \\
+ & := \lambda X^{+*} \lambda Y^{+*} \forall Z^*.\, (X \to Z) \to (Y \to Z) \to Z \\
\exists^{\kappa} & : \quad +(\circ\kappa \to *) \to * \\
\exists^{\kappa} & := \lambda F^{+(\circ\kappa \to *)} \forall Z^*.\, (\forall X^{\kappa}.\, F\, X \to Z) \to Z
\end{array}$$

Notice that all these examples use non-strict positivity. We will use $+$ and $\times$ infix.

Example 2. The reader is invited to check the examples in the introduction, using $\bot := \forall X^*.\, X$ of kind $*$.

Kinding enjoys the usual properties of weakening and strengthening, as well as substitution which respects polarities:

Lemma 1 (Substitution). *If $\Delta, X^{p\kappa} \vdash F : \kappa'$ and $p\Delta \vdash G : \kappa$ then $\Delta \vdash [G/X]F : \kappa'$.*

Proof. By induction on $\Delta, X^{p\kappa} \vdash F : \kappa'$.

Constructor equality. The β-equality $F = F'$ of constructors F, F' is given by the following rules, hence only in the qualified form with contexts:

Computation axioms.

$$\frac{\Delta, X^{p\kappa} \vdash F : \kappa' \qquad p\Delta \vdash G : \kappa}{\Delta \vdash (\lambda X^{p\kappa}. F)\, G = [G/X]F : \kappa'} \qquad \frac{\Delta \vdash F : +\kappa \to \kappa}{\Delta \vdash \mathsf{fix}\ F = F\,(\mathsf{fix}\ F) : \kappa}$$

Congruences.

$$\frac{X^{p\kappa} \in \Delta \qquad p \in \{+, \circ\}}{\Delta \vdash X = X : \kappa} \qquad \frac{\Delta \vdash F = F' : p\kappa \to \kappa' \qquad p\Delta \vdash G = G' : \kappa}{\Delta \vdash F\, G = F'\, G' : \kappa'}$$

$$\frac{\Delta, X^{p\kappa} \vdash F = F' : \kappa'}{\Delta \vdash \lambda X^{p\kappa}. F = \lambda X^{p\kappa}. F' : p\kappa \to \kappa'} \qquad \frac{\Delta, X^{\circ\kappa} \vdash A = A' : *}{\Delta \vdash \forall X^{\kappa}. A = \forall X^{\kappa}. A' : *}$$

$$\frac{-\Delta \vdash A' = A : * \qquad \Delta \vdash B = B' : *}{\Delta \vdash A \to B = A' \to B' : *} \qquad \frac{\Delta \vdash F = F' : +\kappa \to \kappa}{\Delta \vdash \mathsf{fix}\ F = \mathsf{fix}\ F' : \kappa}$$

Symmetry and transitivity.

$$\frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F' = F : \kappa} \qquad \frac{\Delta \vdash F_1 = F_2 : \kappa \qquad \Delta \vdash F_2 = F_3 : \kappa}{\Delta \vdash F_1 = F_3 : \kappa}$$

Lemma 2 (Reflexivity). *If $\Delta \vdash F : \kappa$ then $\Delta \vdash F = F : \kappa$.*

Lemma 3 (Kindedness). *If $\Delta \vdash F = F' : \kappa$ then $\Delta \vdash F : \kappa$ and $\Delta \vdash F' : \kappa$.*

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

$$\frac{}{\diamond\ \mathsf{cxt}} \qquad \frac{\Delta\ \mathsf{cxt}}{\Delta, X^{p\kappa}\ \mathsf{cxt}} \qquad \frac{\Delta\ \mathsf{cxt} \qquad \Delta \vdash A : *}{\Delta, x\!:\!A\ \mathsf{cxt}}$$

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

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

$$\frac{\Delta, X^{\circ\kappa} \vdash t : A}{\Delta \vdash t : \forall X^{\kappa}. A} \qquad \frac{\Delta \vdash t : \forall X^{\kappa}. A \quad \circ\Delta \vdash F : \kappa}{\Delta \vdash t : [F/X]A} \qquad \frac{\Delta \vdash t : A \quad \Delta \vdash A = B : *}{\Delta \vdash t : B}$$

Welltyped terms are closed under substitution (as are constructors, cf. Lemma 1).

As opposed to iso-recursive types with "verbose" folding and unfolding, equi-recursive types yield a leaner term language and hence a more succinct semantics.

Lemma 4. *If $\Delta \vdash t : A$ then $\Delta\ \mathsf{cxt}$ and $\Delta \vdash A : *$.*

Proof. By induction on $\Delta \vdash t : A$.

Reduction. The one-step reduction relation $t \longrightarrow t'$ between terms t and t' is defined as the closure of the β-axiom $(\lambda x.t)\, s \longrightarrow_\beta [s/x]t$ under all term constructors. We denote the transitive closure of $\longrightarrow$ by $\longrightarrow^+$. In the next section, we will see that welltyped terms t_0 admit no infinite reduction $t_0 \longrightarrow t_1 \longrightarrow \ldots$

Constant-time predecessor. For the type $\mathsf{Nat} := \mathsf{fix}\,\lambda A^{+*}.\,1+A$ (using $+$, defined above, and $1 := \forall A^*.\,A \to A$), it is an easy exercise to define closed terms $\mathsf{O} : \mathsf{Nat}$ and $\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}$ (using the injections into sums) that represent the natural numbers, and a closed term $\mathsf{P} : \mathsf{Nat} \to \mathsf{Nat}$ (using the definable case analysis construct) such that $\mathsf{P}\,\mathsf{O} \longrightarrow^+ \mathsf{O}$ and $\mathsf{P}\,(\mathsf{S}\,x) \longrightarrow^+ x$.

3 Strong Normalization of Fix^ω

In this section we prove strong normalization of Fix^ω by a model construction where constructors are interpreted as operators on saturated sets. Due to space constraints, the proof necessarily remains sketchy, but all definitions and facts are given which are required to recover the detailed proof.

As is usual for proving (strong) normalization by a model, only the type system has to be reflected in its construction. In System F^ω, this is just a simply-typed lambda calculus, namely the (simply-)kinded type constructors. Our system Fix^ω additionally has the notions of monotonicity and fixed point. Essentially, we therefore have to give a model of a simply-typed calculus of "syntactically monotone lambda terms". Although the reader will not be surprised by our solution, the authors were surprised that they were not able to find it in the literature.

Following van Raamsdonk and Severi [vRS95,vRS+99] we define the set of strongly normalizing lambda-terms inductively by the following rules (which are implicity also contained in [Gog95]).

$$\frac{t_i \in \mathsf{SN} \text{ for } 1 \leq i \leq |\boldsymbol{t}|}{x\,\boldsymbol{t} \in \mathsf{SN}} \qquad \frac{t \in \mathsf{SN}}{\lambda x.t \in \mathsf{SN}} \qquad \frac{[s/x]t\,\boldsymbol{s} \in \mathsf{SN} \qquad s \in \mathsf{SN}}{(\lambda x.t)\,s\,\boldsymbol{s} \in \mathsf{SN}}$$

This characterization is sound, i.e., if $t_0 \in \mathsf{SN}$ then there is no infinite reduction sequence $t_0 \longrightarrow t_1 \longrightarrow \ldots$, for a proof see *loc. cit.* Our aim is to show $t \in \mathsf{SN}$ for each welltyped term t.

3.1 Lattices of Operators on Saturated Sets

A set of terms $\mathcal{A}$ is called *saturated*, $\mathcal{A} \in \mathsf{SAT}^*$, if it contains only strongly normalizing terms, $\mathcal{A} \subseteq \mathsf{SN}$, and $\mathcal{A}$ is closed under addition of strongly normalizing neutral terms and strongly normalizing weak head expansion:

$$\frac{t_i \in \mathsf{SN} \text{ for } 1 \leq i \leq |\boldsymbol{t}|}{x\,\boldsymbol{t} \in \mathcal{A}} \qquad \frac{[s/x]t\,\boldsymbol{s} \in \mathcal{A} \qquad s \in \mathsf{SN}}{(\lambda x.t)\,s\,\boldsymbol{s} \in \mathcal{A}}$$

For sets of terms $\mathcal{A}$, $\mathcal{B}$ we define the function space $\mathcal{A} \to \mathcal{B} := \{r \in \mathsf{SN} \mid r\,s \in \mathcal{B} \text{ for all } s \in \mathcal{A}\}$. If $\mathcal{A}$ and $\mathcal{B}$ are saturated, so is $\mathcal{A} \to \mathcal{B}$. Furthermore the function space construction is antitone in the domain and monotone in the codomain: if $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{B} \subseteq \mathcal{B}'$ then $\mathcal{A} \to \mathcal{B} \subseteq \mathcal{A}' \to \mathcal{B}'$.

Given an index set I and a family $\mathcal{A}_i$ ($i \in I$) of saturated sets, the infimum $\bigcap_{i\in I} \mathcal{A}_i$ is also saturated. Formation of the infimum is monotone: Given a second family $\mathcal{A}'_i$ of pointwise greater members, $\mathcal{A}_i \subseteq \mathcal{A}'_i$, the infimum is also greater $\bigcap_{i\in I} \mathcal{A}_i \subseteq \bigcap_{i\in I} \mathcal{A}'_i$. Taking set SN as top element, the saturated sets, together with inclusion, $(\mathsf{SAT}^*, \subseteq)$, constitute a complete lattice.

In our model for Fix^ω, types (= constructors of kind "$*$") will be interpreted as saturated sets. To model constructors of higher kinds κ, we need to define a poset $(\mathsf{SAT}^\kappa, \sqsubseteq^\kappa)$ of (higher-order) operators on saturated sets for each kind κ. For the base kind, let $\mathcal{A} \sqsubseteq^* \mathcal{A}' \;:\iff\; \mathcal{A}, \mathcal{A}' \in \mathsf{SAT}^*$ and $\mathcal{A} \subseteq \mathcal{A}'$. To require $\mathcal{A}, \mathcal{A}' \in \mathsf{SAT}^*$ is convenient because the reflexive elements of $\sqsubseteq^*$ are now exactly the saturated sets: $\mathcal{A} \in \mathsf{SAT}^* \iff \mathcal{A} \sqsubseteq^* \mathcal{A}$. The notion of saturated set $\mathsf{SAT}^{p\kappa\to\kappa'}$ and inclusion $\sqsubseteq^{p\kappa\to\kappa'}$ for higher kinds is defined by induction on the kind. Let $\mathcal{F}, \mathcal{F}' \in \mathsf{SAT}^\kappa \to \mathsf{SAT}^{\kappa'}$ be set-theoretic functions.

$$
\begin{array}{lll}
\mathcal{F} \sqsubseteq^{p\kappa\to\kappa'} \mathcal{F}' & :\iff & \mathcal{F}(\mathcal{G}) \sqsubseteq^{\kappa'} \mathcal{F}'(\mathcal{G}') \text{ for all } \mathcal{G}, \mathcal{G}' \in \mathsf{SAT}^\kappa \text{ with } \mathcal{G} \sqsubseteq^{p\kappa} \mathcal{G}' \\
\mathcal{F} \in \mathsf{SAT}^{p\kappa\to\kappa'} & :\iff & \mathcal{F} \sqsubseteq^{p\kappa\to\kappa'} \mathcal{F}
\end{array}
$$

Here, we used the abbreviations

$$
\begin{array}{lll}
\mathcal{G} \sqsubseteq^{+\kappa} \mathcal{G}' & :\iff & \mathcal{G} \sqsubseteq^{\kappa} \mathcal{G}', \\
\mathcal{G} \sqsubseteq^{-\kappa} \mathcal{G}' & :\iff & \mathcal{G}' \sqsubseteq^{\kappa} \mathcal{G}, \\
\mathcal{G} \sqsubseteq^{\circ\kappa} \mathcal{G}' & :\iff & \mathcal{G} \sqsubseteq^{\kappa} \mathcal{G}' \text{ and } \mathcal{G}' \sqsubseteq^{\kappa} \mathcal{G}.
\end{array}
$$

(An easy induction on κ shows that $\mathcal{G} \sqsubseteq^{\circ\kappa} \mathcal{G}'$ implies $\mathcal{G} = \mathcal{G}'$, but the present definition is more suitable for a uniform treatment of all variances in the proofs to follow.)

Each SAT^κ has a top element and infima: For the base kind, $\top^* = \mathsf{SN}$ and $\bigsqcap^* = \bigcap$; for higher kinds they are defined pointwise: Let $\mathcal{F}_i \in \mathsf{SAT}^{p\kappa\to\kappa'}$ for each $i \in I$. Then $\top^{p\kappa\to\kappa'} \in \mathsf{SAT}^{p\kappa\to\kappa'}$ with $\top^{p\kappa\to\kappa'}(\mathcal{G}) := \top^{\kappa'}$, and $\bigsqcap_{i\in I}^{p\kappa\to\kappa'} \mathcal{F}_i \in \mathsf{SAT}^{p\kappa\to\kappa'}$ with $(\bigsqcap_{i\in I}^{p\kappa\to\kappa'} \mathcal{F}_i)(\mathcal{G}) := \bigsqcap_{i\in I}^{\kappa'} \mathcal{F}_i(\mathcal{G})$. With these definitions, each poset $(\mathsf{SAT}^\kappa, \sqsubseteq^\kappa)$ forms a complete lattice.

By Tarski's fixed-point theorem, each monotone operator $\mathcal{F}$ on a complete lattice has a least fixed point $\mathsf{lfp}\,\mathcal{F}$. Indeed, given $\mathcal{F} \in \mathsf{SAT}^{+\kappa\to\kappa}$, we can define the least fixed point by $\mathsf{lfp}\,\mathcal{F} := \bigsqcap^\kappa \{\mathcal{G} \in \mathsf{SAT}^\kappa \mid \mathcal{F}(\mathcal{G}) \sqsubseteq^\kappa \mathcal{G}\}$, i.e., as the least pre-fixed point of $\mathcal{F}$, which, by the theorem, is indeed a pre-fixed point of $\mathcal{F}$, and also a post-fixed point: $\mathsf{lfp}\,\mathcal{F} \sqsubseteq^\kappa \mathcal{F}(\mathsf{lfp}\,\mathcal{F})$. We will use lfp to interpret fixed points $\mathsf{fix}\,F$ of wellkinded constructors F.

3.2 Interpretation of Constructors

In the following part we will define an interpretation $[\![F]\!]_\theta \in \mathsf{SAT}^\kappa$ for each constructor of kind κ, where θ is a *valuation* for the free constructor variables in F. For convenience, a valuation θ is a set-theoretical object which maps both constructor variables X to sets $\mathcal{F}$ and term variables x to terms t. Update of a valuation is written as $\theta[X \mapsto \mathcal{F}]$ resp. $\theta[x \mapsto t]$. We extend inclusion and saturatedness to valuations by defining:

$$
\begin{array}{lll}
\theta \sqsubseteq^\Delta \theta' & :\iff & \theta(X) \sqsubseteq^{p\kappa} \theta'(X) \text{ for all } X^{p\kappa} \in \Delta \\
\theta \in \mathsf{SAT}^\Delta & :\iff & \theta \sqsubseteq^\Delta \theta
\end{array}
$$

Lemma 5. *If $\theta \sqsubseteq^{\Delta} \theta'$, then $\theta \sqsubseteq^{+\Delta} \theta'$, $\theta' \sqsubseteq^{-\Delta} \theta$, $\theta \sqsubseteq^{\circ\Delta} \theta'$ and $\theta' \sqsubseteq^{\circ\Delta} \theta$.*

Proof. By induction on the generation of Δ.

For the following definition and lemma which is the crucial part of this normalization proof, let $\Delta \vdash F : \kappa$. For $\theta \in \mathsf{SAT}^{\Delta}$, we define the interpretation $[\![F]\!]_\theta \in \mathsf{SAT}^{\kappa}$ by induction on the structure of F. Simultaneously we need to prove monotonicity of $[\![F]\!]$, the cases for definition and proof are given below.

Lemma 6 (Monotonicity). *If $\theta \sqsubseteq^{\Delta} \theta'$ then $[\![F]\!]_\theta \sqsubseteq^{\kappa} [\![F]\!]_{\theta'}$.*

For $\theta = \theta'$, immediate consequence of monotonicity is welldefinedness of the interpretation, $[\![F]\!]_\theta \in \mathsf{SAT}^{\kappa}$.

Corollary 1 (p-Monotonicity). *Let $p\Delta \vdash F : \kappa$. If $\theta \sqsubseteq^{\Delta} \theta'$ then $[\![F]\!]_\theta \sqsubseteq^{p\kappa} [\![F]\!]_{\theta'}$.*

Proof (of the corollary). In case $p = +$ the corollary just restates monotonicity (Lemma 6). If $p = -$ then $\theta' \sqsubseteq^{-\Delta} \theta$ by Lemma 5. Using monotonicity, $[\![F]\!]_{\theta'} \sqsubseteq^{\kappa} [\![F]\!]_\theta$. This is by definition equivalent to $[\![F]\!]_\theta \sqsubseteq^{-\kappa} [\![F]\!]_{\theta'}$. If otherwise $p = \circ$, then by Lemma 5 both $\theta \sqsubseteq^{\circ\Delta} \theta'$ and $\theta' \sqsubseteq^{\circ\Delta} \theta$. By monotonicity $[\![F]\!]_\theta \sqsubseteq^{\kappa} [\![F]\!]_{\theta'}$ and $[\![F]\!]_{\theta'} \sqsubseteq^{\kappa} [\![F]\!]_\theta$ which entail by definition $[\![F]\!]_\theta \sqsubseteq^{\circ\kappa} [\![F]\!]_{\theta'}$.

Definition of $[\![F]\!]_\theta$ and proof of monotonicity. By induction on the shape of F.

- $\Delta \vdash X : \kappa$. Set $[\![X]\!]_\theta := \theta(X)$. By assumption, $X^{p\kappa} \in \Delta$ with $p \in \{+, \circ\}$. The requirement $\theta \sqsubseteq^{\Delta} \theta'$ implies $\theta(X) \sqsubseteq^{\kappa} \theta'(X)$, hence $[\![X]\!]_\theta \sqsubseteq^{\kappa} [\![X]\!]_{\theta'}$ by definition.
- $\Delta \vdash \lambda X^{p\kappa}. F : p\kappa \to \kappa'$. The interpretation is a set-theoretic function $[\![\lambda X^{p\kappa}. F]\!]_\theta \in \mathsf{SAT}^{\kappa} \to \mathsf{SAT}^{\kappa'}$, $[\![\lambda X^{p\kappa}. F]\!]_\theta(\mathcal{G}) := [\![F]\!]_{\theta[X \mapsto \mathcal{G}]}$. To show monotonicity, assume $\mathcal{G}, \mathcal{G}' \in \mathsf{SAT}^{\kappa}$ with $\mathcal{G} \sqsubseteq^{p\kappa} \mathcal{G}'$. By inversion of the typing derivation, $\Delta, X^{p\kappa} \vdash F : \kappa'$, and, since $\theta[X \mapsto \mathcal{G}] \sqsubseteq^{\Delta, X^{p\kappa}} \theta'[X \mapsto \mathcal{G}']$, by induction hypothesis $[\![F]\!]_{\theta[X \mapsto \mathcal{G}]} \sqsubseteq^{\kappa'} [\![F]\!]_{\theta'[X \mapsto \mathcal{G}']}$. Hence, $[\![\lambda X^{p\kappa}. F]\!]_\theta(\mathcal{G}) \sqsubseteq^{\kappa'} [\![\lambda X^{p\kappa}. F]\!]_{\theta'}(\mathcal{G}')$ by definition. To conclude, $[\![\lambda X^{p\kappa}. F]\!]$ is monotone.
- $\Delta \vdash F\,G : \kappa'$. Set $[\![F\,G]\!]_\theta := [\![F]\!]_\theta([\![G]\!]_\theta)$. Monotonicity and welldefinedness can be seen as follows. By inversion of the kinding derivation, $\Delta \vdash F : p\kappa \to \kappa'$ and $p\Delta \vdash G : \kappa$. Assume $\theta \sqsubseteq^{\Delta} \theta'$. By the first induction hypothesis, $[\![F]\!]_\theta \sqsubseteq^{p\kappa \to \kappa'} [\![F]\!]_{\theta'}$. By the second induction hypothesis, with Corollary 1, $[\![G]\!]_\theta \sqsubseteq^{p\kappa} [\![G]\!]_{\theta'}$. Putting things together, $[\![F]\!]_\theta([\![G]\!]_\theta) \sqsubseteq^{\kappa'} [\![F]\!]_{\theta'}([\![G]\!]_{\theta'})$, which by definition entails our goal.
- $\Delta \vdash A \to B : *$. Set $[\![A \to B]\!]_\theta := [\![A]\!]_\theta \to [\![B]\!]_\theta$. By inversion, $-\Delta \vdash A : *$ and $\Delta \vdash B : *$. By induction hypothesis and Corollary 1, $[\![A]\!]_\theta \sqsubseteq^{-*} [\![A]\!]_{\theta'}$, hence $[\![A]\!]_{\theta'} \subseteq [\![A]\!]_\theta$. Again, by induction hypothesis, $[\![B]\!]_\theta \sqsubseteq^{*} [\![B]\!]_{\theta'}$, hence $[\![B]\!]_\theta \subseteq [\![B]\!]_{\theta'}$ Together, $[\![A \to B]\!]_\theta \subseteq [\![A \to B]\!]_{\theta'}$. Since the functional construction is saturated, we conclude with $[\![A \to B]\!]_\theta \sqsubseteq^{*} [\![A \to B]\!]_{\theta'}$.

– $\Delta \vdash \forall X^\kappa.\, A : *$. Set $[\![\forall X^\kappa.\, A]\!]_\theta := \bigcap_{\mathcal{F} \in \mathsf{SAT}^\kappa} [\![A]\!]_{\theta[X \mapsto \mathcal{F}]}$. By inversion, $\Delta, X^{\circ\kappa} \vdash A : *$. For arbitrary $\mathcal{F} \in \mathsf{SAT}^\kappa$, $\theta[X \mapsto \mathcal{F}] \sqsubseteq^{\Delta, X^{\circ\kappa}} \theta'[X \mapsto \mathcal{F}]$, hence $[\![A]\!]_{\theta[X \mapsto \mathcal{F}]} \sqsubseteq^* [\![A]\!]_{\theta'[X \mapsto \mathcal{F}]}$ by induction hypothesis. This entails $[\![\forall X^\kappa.\, A]\!]_\theta \sqsubseteq^* [\![\forall X^\kappa.\, A]\!]_{\theta'}$ by monotonicity and saturatedness of the infimum.
– $\Delta \vdash \mathsf{fix}\, F : \kappa$. Set $[\![\mathsf{fix}\, F]\!]_\theta := \mathsf{lfp}([\![F]\!]_\theta)$. By inversion, $\Delta \vdash F : +\kappa \to \kappa$, hence, by induction hypothesis, $\mathcal{F} := [\![F]\!]_\theta \in \mathsf{SAT}^{+\kappa \to \kappa}$ is a monotone operator on SAT^κ, and by Tarski's theorem the least fixed-point $\mathsf{lfp}\, \mathcal{F} \in \mathsf{SAT}^\kappa$ exists. To show monotonicity, we assume $\theta \sqsubseteq^\Delta \theta'$ and define $\mathcal{F}' := [\![F]\!]_{\theta'}$. By monotonicity of $[\![F]\!]$, $\mathcal{F} \sqsubseteq^{+\kappa \to \kappa} \mathcal{F}'$. In particular, $\mathcal{F}(\mathcal{G}) \sqsubseteq^\kappa \mathcal{F}'(\mathcal{G})$ for every $\mathcal{G} \in \mathsf{SAT}^\kappa$. Since $\mathsf{lfp}\, \mathcal{F}$ is a monotone function in its argument $\mathcal{F}$, we are done. □

The interpretation is compatible with substitution and constructor equality, as we show in the following lemmata.

Lemma 7 (Soundness of substitution). *If $\Delta, X^{p\kappa} \vdash F : \kappa'$ and $p\Delta \vdash G : \kappa$, then $[\![[G/X]F]\!]_\theta = [\![F]\!]_{\theta[X \mapsto [\![G]\!]_\theta]}$ for all $\theta \in \mathsf{SAT}^\Delta$.*

Proof. By induction on the structure of F.

Lemma 8 (Soundness of equality). *If $\Delta \vdash F = F' : \kappa$ then $[\![F]\!]_\theta = [\![F']\!]_\theta$ for all $\theta \in \mathsf{SAT}^\Delta$.*

Proof. By induction on constructor equality, using the previous lemma for the first computation rule.

3.3 Interpretation of Terms

To complete our model, we define an interpretation $(\!|t|\!)_\theta$ of terms and then show $(\!|t|\!)_\theta \in [\![A]\!]_\theta$ for welltyped terms $\Delta \vdash t : A$ and sound valuations θ. For wellformed contexts Δ cxt a valuation is *sound*, $\theta \in [\![\Delta]\!]$, if $\theta \in \mathsf{SAT}^\Delta$ and $\theta(x) \in [\![A]\!]_\theta$ for each $(x\!:\!A) \in \Delta$. The term interpretation $(\!|t|\!)_\theta$ is simply the term t itself where all free variables x have been replaced by their value $\theta(x)$ in valuation θ. Note that theses values are strongly normalizing for sound valuations already; it remains to show that the full term $(\!|t|\!)_\theta$ is strongly normalizing for well-typed θ. This is a consequence of the following theorem.

Theorem 1 (Soundness of typing). *If $\Delta \vdash t : A$ and $\theta \in [\![\Delta]\!]$ then $(\!|t|\!)_\theta \in [\![A]\!]_\theta$.*

Proof. By induction on $\Delta \vdash t : A$. Note that by Lemma 4 the context Δ and the type A are wellformed if the typing judgement is derivable. Since our term language is just pure lambda calculus, the proof is standard, for the rule of type equality use Lemma 8.

Corollary 2. *If $\Delta \vdash t : A$, the term t is strongly normalizing.*

Proof. By Theorem 1, choosing a valuation θ with $\theta(X) = \top^\kappa$ for all $X^{p\kappa} \in \Delta$ and $\theta(x) = x$ for all $(x : B) \in \Delta$. This valuation is sound since the type interpretation $[\![B]\!]_\theta$ is saturated, hence, contains x.

4 Primitive Recursion for Heterogeneous Datatypes

In this section, we propose a second way to equip System F^ω with fixed points of higher rank. Therein, we follow Mendler [Men87] who also—besides considering type equations in System F—gave an extension of F by least and greatest fixed points, together with elimination schemes which we refer to as Mendler (co)recursion. We carry Mendler's schemes to higher ranks and define a system MRec^ω as an extension of F^ω by least fixed points of type constructors, also called higher-order inductive types. In contrast to Fix^ω which possesses *equi*-recursive types, MRec^ω is in the style of *iso*-recursive type systems and has explicit introduction and elimination terms for inductive types. In analogy to Spławski and Urzyczyn [SU99] we conjecture that MRec^ω has no reduction preserving embedding into F^ω. However, it embeds into Fix^ω, as we will show in Section 5.

Our starting point is Curry-style System F^ω, enriched with unit type 1, binary products $A \times B$ and sums $A + B$ and the usual term constructors: $\langle\rangle$ for the inhabitant of the unit type, $\langle t_1, t_2\rangle$, $\mathsf{fst}\, r$ and $\mathsf{snd}\, r$ for pairs and left and right projection, and $\mathsf{inl}\, t$, $\mathsf{inr}\, t$ and $\mathsf{case}\,(r,\, x.\, s,\, y.\, t)$ for left and right injection and case distinction. Note that there are no polarized kinds and no fixed-point constructors. An exposition of the exact rules for typing $\Gamma \vdash t : A$ and reduction $t \longrightarrow t'$ can be found in the appendix of Abel et al. [AMU03]. Since F^ω's notion of constructor equivalence is just plain β-equality, we even identify constructors with their β-normal form on the syntactic level.

4.1 Definition of System MRec^ω

For every kind κ of F^ω, we add the constructor constant μ^κ of kind $(\kappa \to \kappa) \to \kappa$ to the system of constructors of F^ω, denoting least fixed-point formation. The term system of F^ω is extended by two families of constants: in^κ (fixed-point introduction) and MRec^κ (fixed-point elimination) for every kind κ. In order to give their types, we need a notion of constructor containment: Every kind κ can uniquely be written in the form $\kappa_1 \to \dots \kappa_n \to *$, in short $\boldsymbol{\kappa} \to *$. Define

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

for constructors $F, G : \kappa = \boldsymbol{\kappa} \to *$. The typing of the constants can now be given by $\mathsf{in}^\kappa : \forall F^{\kappa\to\kappa}.\, F(\mu^\kappa F) \subseteq^\kappa \mu^\kappa F$ and

$$\mathsf{MRec}^\kappa : \forall F^{\kappa\to\kappa} \forall G^\kappa.\, (\forall X^\kappa.\, X \subseteq^\kappa \mu^\kappa F \to X \subseteq^\kappa G \to F\, X \subseteq^\kappa G) \to \mu^\kappa F \subseteq^\kappa G.$$

The notion $\longrightarrow$ of reduction for untyped terms is extended by the additional basic reduction rule of primitive recursion

$$\mathsf{MRec}^\kappa\, s\, (\mathsf{in}^\kappa\, t) \longrightarrow_\beta s\ \mathsf{id}\ (\mathsf{MRec}^\kappa\, s)\ t,$$

where $\mathsf{id} := \lambda x.x$ is the identity. Intuitively, subject reduction still holds because the type $\forall X^\kappa.\, X \subseteq^\kappa \mu^\kappa F \to X \subseteq^\kappa G \to F\, X \subseteq^\kappa G$ of the term s is instantiated with the fixed-point constructor $\mu^\kappa F$ itself. Therefore, the identity id qualifies as first argument to s. In general, the transformation from the blank type X back into the fixed-point, i. e., of type $X \subseteq^{\kappa} \mu^{\kappa} F$, which is the first formal argument of s, provides access to the predecessor of the recursion argument. This is the feature which distinguishes primitive recursion from iteration.

For $\kappa = *$, we have just restated Mendler's rules for recursive types [Men87]. At this point, let us remark that Mendler-style inductive types $\mu^{\kappa} F$—although not observed by Mendler—do not require positivity for F. This contrasts with the recursive types of Fix^{ω}. It also contrasts with formulations of primitive recursion in conventional style that have to rely on positivity or, less syntactically, on a monotonicity requirement such as that in [Mat01] for $\kappa = *$ or $* \to *$.

4.2 Example: Redecoration of Finite Triangular Matrices

As a non-trivial example of the use of MRec^{ω} for heterogeneous datatypes, we consider a redecoration operation for the diagonal elements of finite triangular matrices. In previous work with Uustalu, we have treated redecoration for *infinite* triangular matrices by higher-order coiteration [AMU03], and the finite ones by a computationally unsatisfactory encoding of recursion within iteration [AMU04].

Fix a type $E : *$ of matrix elements. The type $\mathsf{Tri}\,A$ of finite triangular matrices with diagonal elements in A and ordinary elements E can be obtained as follows, with $\kappa 1 := * \to *$:

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

We think of these triangles decomposed columnwise: The first column is a singleton of type A, the second a pair of type $E \times A$, the third a triple of type $E \times (E \times A)$, the fourth a quadruple of type $E \times (E \times (E \times A))$ etc. Hence, if some column has some type A' we obtain the type of the next column as $E \times A'$. By taking the left injection into the sum $1 + \ldots$, one can construct an element without further recurrence, the last column. We can visualize triangles like this:

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

The vertical lines hint at the decomposition scheme. In general, elements of type $\mathsf{Tri}\,A$ are constructed by means of

$$\begin{array}{lll} \mathsf{sg} & := \lambda a.\, \mathsf{in}^{\kappa 1} \, \langle a, \mathsf{inl}\,\langle\rangle\rangle & : \forall A^{*}.\, A \to \mathsf{Tri}\,A \\ \mathsf{cons} & := \lambda a \lambda t.\, \mathsf{in}^{\kappa 1} \, \langle a, \mathsf{inr}\,t\rangle & : \forall A^{*}.\, A \to \mathsf{Tri}\,(E \times A) \to \mathsf{Tri}\,A \end{array}$$

The function $\mathsf{top} : \forall A^{*}.\, \mathsf{Tri}\,A \to A \ = \ \mathsf{Tri} \subseteq^{\kappa 1} \lambda A^{*}.\, A$ that yields the topmost diagonal element, is defined as $\mathsf{top} := \mathsf{MRec}^{\kappa 1}(\lambda i \lambda top \lambda p.\, \mathsf{fst}\, p)$. As reduction behavior, we get

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

If we remove the first column of a triangle $\mathsf{Tri}\,A$, we obtain a trapezium $\mathsf{Tri}\,(E \times A)$. We can get back a (smaller) triangle if we cut off the top row of the trapezium using the function $\mathsf{cut} : \forall A^*.\, \mathsf{Tri}\,(E \times A) \to \mathsf{Tri}\,A$. The exact definition of this function, which is like fcut in [AMU04, Example 34], has to be omitted due to lack of space.

Let $T\,A$ denote some sort of A-decorated (or A-labelled) trees. *Redecoration* [UV02] is an operation that takes an A-decorated tree $t : T\,A$ and a redecoration rule $f : T\,A \to B$ and returns a B-decorated tree $t' : T\,B$. For triangles, redecoration works as follows: In the triangle

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

the underlined A (as an example) gets replaced by the B assigned by the redecoration rule to the sub triangle cut out by the horizontal line; similarly, every other A is replaced by a B.

For the definition of redecoration, we will need a means of lifting a redecoration rule on triangles to one on trapeziums.

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

For a detailed explanation in which sense this is a lifting, see [AMU04]. Finally, we can define redecoration

$$\mathsf{redec} : \forall A^* \forall B^*.\, \mathsf{Tri}\,A\ \to (\mathsf{Tri}\,A \to B) \to \mathsf{Tri}\,B\ =\ \mathsf{Tri} \subseteq^{\kappa 1} G$$

with $G := \lambda A^* \forall B^*.\, (\mathsf{Tri}\,A \to B) \to \mathsf{Tri}\,B$. The definition makes essential use of primitive recursion in that it also uses the variable $i : X \subseteq^{\kappa 1} \mathsf{Tri}$ in the body of argument to $\mathsf{MRec}^{\kappa 1}$:

$$\begin{array}{l} \mathsf{redec} := \mathsf{MRec}^{\kappa 1}\Big(\lambda i \lambda \mathit{redec} \lambda t \lambda f.\ \mathsf{case}\,\big(\mathsf{snd}\,t, \\ \qquad\qquad u.\,\mathsf{sg}\,(f\,(\mathsf{sg}\ (\mathsf{fst}\,t))), \\ \qquad\qquad r.\,\mathsf{cons}\ (f\,(\mathsf{cons}\ (\mathsf{fst}\,t)\ (i\,r)))\ (\mathit{redec}\,r\,(\mathsf{lift}\,f))\big)\Big) \end{array}$$

Its reduction behavior is easy to calculate:

$$\begin{array}{lll} \mathsf{redec}\,(\mathsf{sg}\,a)\,f & \longrightarrow^+ & \mathsf{sg}\,(f\,(\mathsf{sg}\,a)) \\ \mathsf{redec}\,(\mathsf{cons}\,a\,r)\,f & \longrightarrow^+ & \mathsf{cons}\,(f\,(\mathsf{cons}\ a\,r))\,(\mathsf{redec}\ r\,(\mathsf{lift}\,f)) \end{array}$$

The reader is invited to compare this concise behaviour with the one obtained in [AMU04] within definitional extensions of system F^ω that therefore can only provide iteration schemes and no primitive recursion. Notice that the number of reduction steps does *not* depend on the terms a, r and f since these may just be variables. By a modification of the definition of redec above, it is easy to define a constant-time predecessor operation on triangles (a left inverse of $\mathsf{in}^{\kappa 1}$ for Tri even with respect to reductions of open terms): The access to r in the cons case of the reduction will be type-correct by using $(i\,r)$ instead of r, just as for redec.

5 Embedding of Mendler-Style Recursion into Fix^ω

In this section, we prove—via an embedding into Fix^ω—that Mendler recursion for higher ranks is strongly normalizing. The proof proceeds in two steps: First, we show that all constructions of MRec^ω can be defined in Fix^ω such that reduction is simulated. Then we map each welltyped term of MRec^ω onto a still welltyped term of Fix^ω of exactly the same shape (the translation is purely homomorphic). Thus, each infinite reduction sequence of MRec^ω would map onto an infinite sequence of Fix^ω, which is a contradiction to the result of Section 3.

Products and sums can be defined in Fix^ω via the standard impredicative encoding (see Example 1). The interesting part is the definition of least fixed-points $\mu^\kappa F$ within Fix^ω. We give their definition only for kinds carrying no polarity information, i.e., kinds of the form $\kappa = \circ\boldsymbol{\kappa} \to *$. This suffices because their purpose is just to serve as images in the translation of the least fixed points in the polarity-free system MRec^ω. We define $\mu^\kappa := \lambda F^{\circ(\circ\kappa\to\kappa)}.\ \mathsf{fix}\,\Phi_F$ with

$$\Phi_F := \lambda Y^{+\kappa}\lambda \boldsymbol{X}^{\circ\boldsymbol{\kappa}}\forall G^\kappa.\,(\forall X^\kappa.\,X \subseteq^\kappa Y \to X \subseteq^\kappa G \to F\,X \subseteq^\kappa G) \to G\,\boldsymbol{X}.$$

It is not hard to see that $F^{\circ(\circ\kappa\to\kappa)} \vdash \Phi_F : +\kappa \to \circ\boldsymbol{\kappa} \to *$, since the variable Y occurs twice to the left of an arrow in the body of the definition of Φ_F. Thus, for any $F : \circ\kappa \to \kappa$ we have $\Phi_F : +\kappa \to \kappa$, and $\mu^\kappa : \circ(\circ\kappa \to \kappa) \to \kappa$ as required.

Once we have found a suitable representation of μ^κ in Fix^ω, the definition of elimination and introduction falls into place:

$$\begin{array}{ll} \mathsf{MRec}^\kappa & := \lambda s\lambda r.\,r\,s \\ \mathsf{in}^\kappa & := \lambda t\lambda s.\ s\ \mathsf{id}\ (\mathsf{MRec}^\kappa s)\ t \end{array}$$

Note that the right-hand sides do not depend on κ. These definitions yield simulation of primitive recursion within Fix^ω, as we can confirm by performing four β-reduction steps: $\mathsf{MRec}^\kappa\,s\,(\mathsf{in}^\kappa\,t) \longrightarrow^+ s\,\mathsf{id}\,(\mathsf{MRec}^\kappa\,s)\,t$.

Now, System MRec^ω can be translated into Fix^ω by replacing each arrow kind $\kappa \to \kappa'$ by $\circ\kappa \to \kappa'$, and each annotated abstraction λX^κ by $\lambda X^{\circ\kappa}$. All other syntactical constructions remain unchanged. Certainly, we can only map the constants in^κ and MRec^κ of MRec^ω onto their defined counterparts in Fix^ω, if the types of source and target match. This can been seen by type-checking, for which the following chart might be an aid.

$$\begin{array}{lll} \Delta & := & F^{\circ(\circ\kappa\to\kappa)},\ G^{\circ\kappa},\ X^{\circ\boldsymbol{\kappa}}, \\ & & s : \forall X^\kappa.\,X \subseteq^\kappa \mu^\kappa F \to X \subseteq^\kappa G \to F\,X \subseteq^\kappa G, \\ & & r : \mu^\kappa F\,\boldsymbol{X}\ =\ \Phi_F(\mu^\kappa F)\,\boldsymbol{X}, \\ & & t : F\,(\mu^\kappa F)\,\boldsymbol{X} \\ \\ \Delta & \vdash & r\,s\ :\ G\,\boldsymbol{X} \\ \Delta & \vdash & \lambda s.\ s\ \mathsf{id}\ (\mathsf{MRec}^\kappa s)\ t\ :\ \Phi_F(\mu^\kappa F)\,\boldsymbol{X}\ =\ \mu^\kappa F\,\boldsymbol{X} \end{array}$$

Theorem 2. *In System* MRec^ω *of Mendler recursion for arbitrary kinds there exists no infinite reduction sequence* $t_0 \longrightarrow t_1 \longrightarrow \ldots$ *starting with a welltyped term* t_0.

Proof. By Corollary 2, using the abovementioned translation into Fix^ω.

6 Conclusion and Future Work

We have presented two systems for total functions over higher-order and nested datatypes where the predecessor runs in constant time. The first system, Fix^ω, supports positive equi-recursive types of higher order. No primitive combinator for recursive functions is built in, but due to the strength of equi-recursive types in combination with impredicativity, customary recursion schemes can be defined. One instance is Mendler-style primitive recursion MRec, which for example can be used to define a redecoration algorithm for triangular matrices. We have shown that Mendler-style primitive recursion can be simulated in Fix^ω.

This simulation could have been extended to also account for coinductive type constructors, by defining Mendler-style corecursion for higher ranks in Fix^ω. A naturally corecursive program is substitution for the infinite version of de Bruijn terms coded as a nested datatype [AR99,BP99b]. Due to space restrictions we have to leave this direction to future work.

The systematic use of nested datatypes to represent datastructures with invariants is rather new [Hin98] (but also see [Oka96, Sections 10, 11] for earlier work). As an example, Hinze [Hin01] implemented Okasaki's functional version of red-black trees [Oka99] by help of a nested datatype to actually ensure the balancing properties of red-black trees by the type system. Most algorithms for nested datatypes published so far require just *iteration*, hence can be implemented in the framework of generalized folds [BP99a] or efficient folds [MGB04] or Mendler iteration [AMU04]. As more classical algorithms will find functional implementations using nested datatypes, we imagine many more examples requiring primitive recursion for higher-rank datatypes, and thus may infer termination of the respective algorithms.

References

[AMU03] A. Abel, R. Matthes, and T. Uustalu. Generalized iteration and coiteration for higher-order nested datatypes. In A. Gordon, ed., *Proc. of FoSSaCS 2003*, vol. 2620 of *LNCS*, pp. 54–69. 2003.

[AMU04] A. Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 2004. 79 pages, accepted for publication.

[AR99] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Proceedings of CSL '99*, vol. 1683 of *LNCS*, pp. 453–468. 1999.

[BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

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

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

[CP88] T. Coquand and C. Paulin. Inductively defined types—preliminary version. In P. Martin-Löf and G. Mints, eds., *Proceedings of COLOG '88*, vol. 417 of *LNCS*, pp. 50–66. 1988.

[DC98] D. Duggan and A. Compagnoni. Subtyping for object type constructors, 1998. Presented at FOOL 6.

[GHR93] P. Giannini, F. Honsell, and S. Ronchi Della Rocca. Type inference: some results, some problems. *Fundamenta Informaticae*, 19(1-2):87 – 125, 1993.

[Gog95] H. Goguen. Typed operational semantics. In M. Dezani-Ciancaglini and G. Plotkin, eds., *Proc. of TLCA '95*, vol. 902 of *LNCS*, pp. 186–200. 1995.

[Hin98] R. Hinze. Numerical representations as higher-order nested datatypes. Tech. Rep. IAI-TR-98-12, Institut für Informatik III, Universität Bonn, 1998.

[Hin01] R. Hinze. Manufacturing datatypes. *Journal of Functional Programming*, 11(5):493–524, 2001.

[Mat01] R. Matthes. Monotone inductive and coinductive constructors of rank 2. In L. Fribourg, ed., *Proc. of CSL 2001*, vol. 2142 of *LNCS*, pp. 600–614. 2001.

[Men87] N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of LICS '87*, pp. 30–36. 1987.

[MGB04] C. Martin, J. Gibbons, and I. Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, 16(1):19–35, 2004.

[Oka96] C. Okasaki. *Purely Functional Data Structures*. Ph.D. thesis, Carnegie Mellon University, 1996.

[Oka99] C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

[Ste98] M. Steffen. *Polarized Higher-Order Subtyping*. Ph.D. thesis, Technische Fakultät, Universität Erlangen, 1998.

[SU99] Z. Spławski and P. Urzyczyn. Type fixpoints: Iteration vs. recursion. In *Proceedings of ICFP'99*, pp. 102–113. SIGPLAN Notices, 1999.

[Urz96] P. Urzyczyn. Positive recursive type assignment. *Fundamenta Informaticae*, 28(1–2):197–209, 1996.

[UV02] T. Uustalu and V. Vene. The dual of substitution is redecoration. In K. Hammond and S. Curtis, eds., *Trends in Functional Programming 3*, pp. 99–110. Intellect, Bristol, Portland, OR, 2002.

[vBL+97] S. van Bakel, L. Liquori, S. Ronchi Della Rocca, and P. Urzyczyn. Comparing cubes of typed and type assignment systems. *Annals of Pure and Applied Logic*, 86(3):267–303, 1997.

[vRS95] F. van Raamsdonk and P. Severi. On normalisation. Tech. Rep. CS-R9545, CWI, 1995.

[vRS+99] F. van Raamsdonk, P. Severi, M. H. Sørensen, and H. Xi. Perpetual reductions in lambda calculus. *Inf. and Comp.*, 149(2):173–225, 1999.