Type-Based Termination of Generic Programs

Andreas Abel

Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, GERMANY

Abstract

Instances of a polytypic or generic program for a concrete recursive type often exhibit a recursion scheme that is derived from the recursion scheme of the instantiation type. In practice, the programs obtained from a generic program are usually terminating, but the proof of termination cannot be carried out with traditional methods as term orderings alone, since termination often crucially relies on the program type. In this article, it is demonstrated that type-based termination using sized types handles such programs very well. A framework for sized polytypic programming is developed which ensures (type-based) termination of all instances.

1 Introduction

In the last decade, *polytypic* or *generic* programming has been explored for functional programming languages [26,19,22,23]. With polytypic programming, repetitive tasks, like writing a `size`-function for data structures of type A, can be mechanized by writing a generic `size`-function which then can be instantiated to all sorts of types A. Over the years, many useful examples of generic programs have been put forth, like parsing and unparsing, map and zip functions, and even finite maps for key type A. When generic programs are defined by recursion on type A, then the resulting programs often exhibit a recursion structure that corresponds to the recursion structure of type A; and it is the rule that they terminate, if applied to finite input. However, because of the high degree of abstraction that generic programs usually involve, termination cannot be proven with conventional methods like term orderings alone. It is the purpose of this article to outline a systematic solution to the termination problem of many generic programs.

Email address: `abel@informatik.uni-muenchen.de` (Andreas Abel).

As an example, we take Hinze's [18] generic definition of finite maps. If instantiated to key type *list of A*, in Haskell syntax `[a]`, we get the following definition of a finite map:

```
data MapList f v = Leaf
                 | Node (Maybe v) (f (MapList f v))
```

Herein, `v` is the range of the finite map, and `f w` represents the finite maps from `a` to `w`. Instantiating `a` with `Char` and `f w` with `Char→w`, we would get finite maps over strings. Such a finite map is either totally undefined (`Leaf`) or a pair of maybe a piece of data associated with the current key (`Maybe v`) plus a finite map for each extension of the current key by one character (`f (MapList f v)`).

Merging finite maps is a completely generic operation. Again for the key type of lists, we get the following instance. Let

```
comb :: (v -> v -> v) -> Maybe v -> Maybe v -> Maybe v
```

be a conflict resolution function for up to two candidate values of a finite map at a certain key. Then the following Haskell program merges two finite maps over lists:

```
mergeList ::
  (forall w. (w -> w -> w) -> f w -> f w -> f w) ->
  (v -> v -> v) ->
  MapList f v -> MapList f v -> MapList f v
mergeList mergeF c Leaf t = t
mergeList mergeF c t Leaf = t
mergeList mergeF c (Node m1 t1) (Node m2 t2) =
  Node (comb c m1 m2) (mergeF (mergeList mergeF c) t1 t2)
```

This function has an extraordinary recursion behavior: As a recursive "call", the whole function `mergeList mergeF c` is passed to one of its arguments, `mergeF`. It is not immediately obvious that `mergeList` is a total function. Indeed, if we disregard its type, we can create a non-terminating execution: Define

```
mf m t1 t2 = m (Node Nothing t1) (Node Nothing t2)
```

and run:

```
mergeList mf fst (Node Nothing t1) (Node Nothing t2)
```

However, `mf` does not have the right type, and the polymorphic nature of the argument `mergeF` is a critical ingredient for termination.

This example shows that term-based termination arguments do not suffice for generic programs. We need a method for establishing termination which takes the *type* of a program into account. Such a method is *type-based termination*, which has been developed by Hughes, Pareto, and Sabry [24], and independently by Giménez [16] who advanced the pioneering work of Mendler [27]. Since then, type-based termination has been considered by several authors [2,8,9,12].

In this work, we show that type-based termination can be successfully applied to generic programs. We build on previous work, System $\mathsf{F}^{\widehat{}}_{\omega}$ [1], which is an extension of type-based termination to higher-order data types. Our contribution is an adaptation of Hinze, Jeuring, and Löh's generic programming framework [23] to sized types, including a condition on the type of a generic program which ensures that all its instances are admissible recursion types in $\mathsf{F}^{\widehat{}}_{\omega}$.

We will briefly introduce the necessary concepts to the reader in Section 2 and then outline a framework for total generic programming in Sect. 3. In Sect. 4 we prove that all instances of generic programs are well-typed which entails termination. More related work and directions for future research are discussed in Sect. 5.

This article is an extension of my MPC'06 conference paper [3].

1.1 Preliminaries

We assume that the reader is familiar with the higher-order polymorphic lambda-calculus, System F^{ω} (see Pierce's text book [34]), or the functional programming language Haskell. Additionally, some familiarity with generic programming would be helpful [22].

Generic programming takes a minimalistic view on data types: Each ground type can be constructed using the unit type 1, disjoint sum type $A + B$, product type $A \times B$ and recursion. The following terms manipulate these types:

$$\begin{array}{ll}
() & : 1 \\
\mathsf{pair} & : \forall A \forall B.\, A \to B \to A \times B \\
\mathsf{fst} & : \forall A \forall B.\, A \times B \to A \\
\mathsf{snd} & : \forall A \forall B.\, A \times B \to B \\
\mathsf{inl} & : \forall A \forall B.\, A \to A + B \\
\mathsf{inr} & : \forall A \forall B.\, B \to A + B \\
\mathsf{case} & : \forall A \forall B \forall C.\, A + B \to (A \to C) \to (B \to C) \to C
\end{array}$$

Pairs $\mathsf{pair}\, r\, s$ are written (r, s). We assume the usual reduction rules, for instance, $\mathsf{fst}\,(r, s) \longrightarrow r$. Multi-step reduction is denoted by $\longrightarrow^{+}$. Sometimes it is convenient to introduce abbreviations for derived data constructors. For instance:

$$\begin{aligned}\mathsf{Nat} &= 1 + \mathsf{Nat}\\ \mathsf{zero} &= \mathsf{inl}\,()\\ \mathsf{succ} &= \lambda n.\, \mathsf{inr}\, n\end{aligned}$$

To improve readability, we will freely make use of the pattern matching notation

$$\mathsf{match}\ r\ \mathsf{with}\ p_1 \mapsto t_1 \mid \cdots \mid p_n \mapsto t_n$$

for patterns p_i generated from both elementary and derived data constructors. We use a non-recursive $\mathsf{let}\ p{=}r\ \mathsf{in}\ t$ as abbreviation for $\mathsf{match}\ r\ \mathsf{with}\ p \mapsto t$.

2 Sized Types in a Nutshell

We use sized types for type-based termination checking, as described by Hughes, Pareto, and Sabry [24,32] and Barthe, Frade, Giménez, Pinto, and Uustalu [8]. In comparison with the cited works, our system, $\mathsf{F}_{\omega}^{\widehat{}}$, also features higher-order polymorphism and heterogeneous (nested) and higher-order data types. In this section, we quickly introduce the most important features of $\mathsf{F}_{\omega}^{\widehat{}}$ [1], a summary of the rules can be found in the appendix.

Monotone type constructors and polarized kinds. Inductive types are recursively defined types which can only be unfolded finitely many times. The classical example are lists which are given as the least fixed-point of the type constructor $\lambda X.\, 1 + A \times X$, where A is the type of list elements. If the type constructor underlying an inductive type is not covariant (monotone), non-terminating programs can be constructed without explicit recursion [27]. Therefore we restrict inductive types to fixed-points of covariant constructors. We write

$$\begin{array}{llll} * \xrightarrow{+} * & \text{or} & +* \to * & \text{for the kind of covariant,}\\ * \xrightarrow{-} * & \text{or} & -* \to * & \text{for the kind of contravariant, and}\\ * \xrightarrow{\circ} * & \text{or} & \circ* \to * & \text{for the kind of mixed-variant}\end{array}$$

type constructors, the last meaning constructors which are neither co- nor contravariant, or the absence of variance information. For example, $\lambda X.\, X \to 1$ is contravariant, and $\lambda X.\, X \to X$ is mixed-variant. The notion of variance is extended to arbitrary kinds, and p-variant function kinds are written as

$$p\kappa \to \kappa' \text{ or } \kappa \xrightarrow{p} \kappa'.$$

For instance, we have the following kindings for disjoint sum, product, function, and polymorphic type constructor:

$+$	$: * \xrightarrow{+} * \xrightarrow{+} *$	disjoint sum
$\times$	$: * \xrightarrow{+} * \xrightarrow{+} *$	cartesian product
$\to$	$: * \xrightarrow{-} * \xrightarrow{+} *$	function space
$\forall_\kappa$	$: (\kappa \xrightarrow{\circ} *) \xrightarrow{+} *$	quantification

We assume a *signature* Σ that contains the above type constructor constants together with their kinding, plus some base types 1, Char, Int ... The signature Σ is viewed as a function, so $\Sigma(C)$ returns the kind of the constructor constant C. A bit sloppily, we write $C \in \Sigma$ if C is in the domain of this function, $C \in \mathsf{dom}(\Sigma)$. Also, we usually write $\forall X\!:\!\kappa.A$ for $\forall_\kappa \lambda X.A$, or $\forall X A$, if the kind κ is inferable.

Kinding judgement. A kinding context Δ is a finite map from constructor variables X to pairs $p\kappa$ of a polarity p and a kind κ. Kinding $\Delta \vdash F : \kappa$ of constructors is given inductively by the following rules:

$$\frac{C\!:\!\kappa \in \Sigma}{\Delta \vdash C : \kappa} \qquad \frac{X\!:\!p\kappa \in \Delta \qquad p \in \{+, \circ\}}{\Delta \vdash X : \kappa}$$

$$\frac{\Delta, X\!:\!p\kappa \vdash F : \kappa'}{\Delta \vdash \lambda X F : p\kappa \to \kappa'} \qquad \frac{\Delta \vdash F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

The kinding judgement $\Delta \vdash F : \kappa$ expresses that if $\Delta(X) = q\kappa'$ then F is q-variant in X. In particular if $q = +$ then F is monotone in X and if $q = -$ then F is antitone in X. Since X is monotone in X, the judgement $\Delta \vdash X : \kappa$ is valid if and only if $\Delta(X)$ is $+\kappa$ or $\circ\kappa$; this justifies the variable rule. Special attention has to be paid to the application rule: the variance of the application $H = F\,G$ depends not only on the variance of F and G in X, but also on the variance of the function F itself. The hypotheses of the rule are $\Delta \vdash F : p\kappa \to \kappa'$ and $p^{-1}\Delta \vdash G : \kappa$. The operation $p^{-1}\Delta$ modifies the polarities of the free variables in G according to p. If $p = +$, then $+^{-1}\Delta = +\Delta = \Delta$; this is because $F\,G$ is monotone (antitone, resp.) in X if both F and G are. If $p = -$, then $F\,G$ is monotone in X if F is monotone in X and G is antitone in X. This means that the polarities in the kinding context for G have to be reversed. We set $(-^{-1}\Delta)(X) = (-\Delta)(X) = -(\Delta(X))$, where $-+ = -$, $-- = +$, and $-\circ = \circ$. Finally, if $p = \circ$ then $F\,G$ is monotone (antitone, resp.) in X if F is monotone (antitone, resp.) in X and X does not appear in G. Hence $\circ^{-1}\Delta$ needs to erase all variables $X : q\kappa''$ from Δ whose polarity q is $+$ or $-$; only variables of mixed variance are kept. Thus $(\Delta^{-1})(X) = \circ\kappa$ if $\Delta(X) = \circ\kappa$, otherwise $(\Delta^{-1})(X)$ is undefined.

Ordinary F^ω kinding $\Delta \vdash F : \kappa$ is regained if all polarities in Δ and κ are $\circ$.

Further information on kinding can be found in the thesis of the author [1, Ch. 2].

Sized inductive types. We write inductive types as $\mu^a F$, where F is a covariant constructor and a a constructor of special kind ord. This kind models the stage expressions of Barthe et al. [8], which are interpreted as ordinals, and has the following constructors:

s	$: \mathsf{ord} \xrightarrow{+} \mathsf{ord}$	successor of ordinal,
∞	$: \mathsf{ord}$	infinity ordinal.

The *infinity ordinal* is the closure ordinal of all inductive types considered, i. e., an ordinal big enough such that the equation

$$F\,(\mu^\infty F) = \mu^\infty F$$

holds for all type constructors which are allowed as basis for an inductive type. If F is first-order, i. e., does not mention function space, then the smallest infinite ordinal ω is sufficient. However, if we allow higher-order datatypes like the infinitely-branching $\mu^\infty \lambda X.1 + (\mathsf{Nat} \to X)$, higher ordinals are required. [1]

In the following, we will only make use of ordinal constructors that are either ∞ or $\imath + n$, where $\imath$ is a constructor variable of kind ord and n a natural number and $a + n$ is a shorthand for prepending the constructor a with n successor constructors s.

Sized inductive types are explained by the equation $\mu^{a+1} F = F\,(\mu^a F)$. Viewing inductive types as trees and F as the type of the node constructor, it becomes clear that the size index a is an upper bound on the height of trees in $\mu^a F$. Hence, inductive types are covariant in the size index, and their instances stand in the subtyping relation

$$\mu^a F \leq \mu^{a+1} F \leq \mu^{a+2} F \leq \dots \leq \mu^\infty F.$$

Some examples for sized inductive types are:

$$\begin{aligned}
\mathsf{Nat} &: \mathsf{ord} \xrightarrow{+} * \\
\mathsf{Nat} &:= \lambda\imath.\, \mu^\imath \lambda X.\, 1 + X \\[1ex]
\mathsf{List} &: \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{List} &:= \lambda\imath\lambda A.\, \mu^\imath \lambda X.\, 1 + A \times X \\[1ex]
\mathsf{Tree} &: \mathsf{ord} \xrightarrow{+} * \xrightarrow{-} * \xrightarrow{+} * \\
\mathsf{Tree} &:= \lambda\imath\lambda B\lambda A.\, \mu^\imath \lambda X.\, 1 + A \times (B \to X)
\end{aligned}$$

[1] More details can be found in the thesis of the author [1, Sect. 3.3.3].

Nat^a denotes the type of natural numbers $< a$, $\mathsf{List}^a A$ the type of lists of length $< a$, and $\mathsf{Tree}^a B\, A$ the type of B-branching A-labeled trees of height $< a$. For lists, we define the usual constructors:

$$\begin{array}{lll}
\mathsf{nil} & := \mathsf{inl}\,() & : \forall\imath\forall A.\, \mathsf{List}^{\imath+1}\, A \\
\mathsf{cons} & := \lambda a \lambda as.\, \mathsf{inr}\,(a, as) & : \forall\imath\forall A.\, A \to \mathsf{List}^{\imath}\, A \to \mathsf{List}^{\imath+1} A.
\end{array}$$

Heterogeneous data types. Nothing prevents us from considering inductive types of higher kind, i. e., such $\mu^a F$ where F is not of kind $* \xrightarrow{+} *$, but, for instance, of kind $(* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *)$. For such an F we get an inductive *constructor*, or a heterogeneous data type [6], in the literature often called nested type [4,11,18,31]. In general, the least-fixed point constructor μ_κ can be used on any $F : \kappa \xrightarrow{+} \kappa$ where κ must be a pure kind, i. e., must not mention special kind ord. Examples for heterogeneous types are:

$$\begin{array}{ll}
\mathsf{PList} : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{PList} := & \lambda\imath.\, \mu^{\imath}_{+*\to*} \lambda X \lambda A.\, A + X\,(A \times A) \\
\\
\mathsf{Bush} : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Bush} := & \lambda\imath.\, \mu^{\imath}_{+*\to*} \lambda X \lambda A.\, \mathbf{1} + A \times X\,(X\,A)
\end{array}$$

The type $\mathsf{PList}^a\, A$ implements lists with exactly 2^n elements of type A for some $n < a$. The second type, *bushy* lists, is an example of a *truly nested* type since X appears in an argument to X. It is well-defined since we can infer covariance of $X\,(X\,A)$ in X from the assumption that X is covariant itself. [2]

Example 1 (A powerlist) Let $a_0, a_1, a_2, a_3 : A$ and $\imath : \mathsf{ord}$. We can construct the powerlist $\mathsf{PList}^{\imath+3}\, A$ containing these four elements as follows:

$$\begin{array}{rcl}
((a_0, a_1), (a_2, a_3)) & : & ((A \times A) \times (A \times A)) =: A^4 \\ \hline
\mathsf{inl}\,((a_0, a_1), (a_2, a_3)) & : & A^4 + \mathsf{PList}^{\imath}\,(A^4 \times A^4) \\ \hline
\mathsf{inl}\,((a_0, a_1), (a_2, a_3)) & : & \mathsf{PList}^{\imath+1}\; A^4 \\ \hline
\mathsf{inr}\,(\mathsf{inl}\,((a_0, a_1), (a_2, a_3))) & : & A \times A + \mathsf{PList}^{\imath+1}\; A^4 \\ \hline
\mathsf{inr}\,(\mathsf{inl}\,((a_0, a_1), (a_2, a_3))) & : & \mathsf{PList}^{\imath+2}\,(A \times A) \\ \hline
\mathsf{inr}\,(\mathsf{inr}\,(\mathsf{inl}\,((a_0, a_1), (a_2, a_3)))) & : & A + \mathsf{PList}^{\imath+2}\,(A \times A) \\ \hline
\mathsf{inr}\,(\mathsf{inr}\,(\mathsf{inl}\,((a_0, a_1), (a_2, a_3)))) & : & \mathsf{PList}^{\imath+3}\, A
\end{array}$$

Structural recursion. Since we are considering a terminating programming language, recursion cannot be available without restriction. In the following we give

[2] The constructor underlying Bush fails a purely syntactical covariance test, like the test for *strict positivity* in Coq [25].

a typing rule for structurally recursive functions. Herein, we interpret *structurally recursive* in the context of sized types: A function is structurally recursive if the recursive instance is of smaller size than the calling instance. As typing rule, this definition reads:

$$\frac{\Gamma,\ \imath{:}\mathsf{ord},\ f : A\,\imath \vdash t : A\,(\imath + 1)}{\Gamma \vdash \mathsf{fix}^{\mu}(\lambda f.t) : \forall\imath.\ A\,\imath}\ A\ \mathsf{fix}^{\mu}\text{-}\mathsf{adm}$$

Of course, the type $A\,\imath$ must mention the size variable $\imath$ in a sensible way; with the constant type $A\,\imath = \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\infty}$ one immediately allows non-terminating functions. Barthe et al. [8,9] suggest types of the shape $A\,\imath = \mu^{\imath} F \to C$ where $\imath$ does not occur in F and only positively in C. In this article, we want to consider recursive functions that simultaneously descend on several arguments, and also polymorphic recursion. Hence, we consider types of the shape

$$\forall\vec{Y}^0.\ \mu^{\imath} F\,\vec{G} \to \forall\vec{Y}^1.\, B_1 \to \cdots \to \forall\vec{Y}^m.\, B_m \to C,$$

where $\imath$ does not occur in F and $\vec{G}$, index $\imath$ occurs only positively in C, and each of the B_i is either contravariant in $\imath$ or of the shape $\mu^{\imath} F'\,\vec{G}'$ with $F', \vec{G}'$ $\imath$-free. This condition is written $A\ \mathsf{fix}^{\mu}\text{-}\mathsf{adm}$, or $\Gamma\ \vdash A\ \mathsf{fix}^{\mu}\text{-}\mathsf{adm}$ if A has free variables whose types are recorded in Γ. More valid shapes for the type $A\,\imath$ are described by Hughes, Pareto, and Sabry [24], in Pareto's thesis [32] and my thesis [1,2].

To obtain a strongly normalizing system, unrolling of fixed-points has to be restricted to the case

$$\mathsf{fix}^{\mu} s\, v \longrightarrow s\,(\mathsf{fix}^{\mu} s)\, v,$$

where v is a value (an injection, a pair, a λ-abstraction, an under-applied function symbol). For convenience, we define the fixed-point combinator fix^{μ}_n that takes n non-recursive arguments before the first recursive argument:

$$\begin{aligned}
\mathsf{back}_n &:= \lambda g \lambda t_1 \ldots \lambda t_n \lambda r.\ g\, r\, t_1 \ldots t_n \\
\mathsf{front}_n &:= \lambda g \lambda r \lambda t_1 \ldots \lambda t_n.\ g\, t_1 \ldots t_n\, r \\
\\
\mathsf{fix}^{\mu}_n[s] &:= \mathsf{back}_n\,(\mathsf{fix}^{\mu}\,(\lambda f.\, \mathsf{front}_n\,(s\,(\mathsf{back}_n\, f)))). \\
\mathsf{fix}^{\mu}_n &:= \lambda s.\ \mathsf{fix}^{\mu}_n[s]
\end{aligned}$$

Lemma 2 (Reduction of defined fixed-points) *If* $|\vec{t}| = n$*, then*

$$\mathsf{fix}^{\mu}_n[s]\,\vec{t}\,v \longrightarrow^{+} s\,(\mathsf{fix}^{\mu}_n[s])\,\vec{t}\,v.$$

Proof. Easy. Observe, however, that $\mathsf{fix}^{\mu}_n\, s\,\vec{t}\,v \not\longrightarrow^{+} s\,(\mathsf{fix}^{\mu}_n\, s)\,\vec{t}\,v$ (would require one β-expansion step). □

Lemma 3 *If* $\Gamma \vdash s : (\forall \vec{X}^1.\, A_1 \to \cdots \to \forall \vec{X}^n.\, A_n \to \forall \vec{X}.\, B \to C) \to (\forall \vec{Y}^1.\, A'_1 \to \cdots \to \forall \vec{Y}^n.\, A'_n \to \forall \vec{Y}.\, B' \to C')$ *then* $\Gamma \vdash \lambda f.\, \mathsf{front}_n\, (s\, (\mathsf{back}_n\, f)) : (\forall \vec{X}^1 \ldots \vec{X}^n \vec{X}.\, B \to \vec{A} \to C) \to (\forall \vec{Y}^1 \ldots \vec{Y}^n \vec{Y}.\, B' \to \vec{A}' \to C')$.

The admissibility condition is generalized to functions that have n non-recursive arguments before the recursive ones: We write $\Gamma \vdash A\ \mathsf{fix}^\mu_n\text{-}\mathsf{adm}$ if $A\, \imath$ is of the shape

$$\forall \vec{X}^1.\, A_1 \to \cdots \to \forall \vec{X}^n.\, A_n \to \forall \vec{Y}^0.\, \mu^\imath F\, \vec{G} \to \forall \vec{Y}^1.\, B_1 \to \cdots \to \forall \vec{Y}^m.\, B_m \to C,$$

where the $\imath$ may occur only negatively in the A_j, and for the other type expressions the same conditions hold as in the definition of $A\ \mathsf{fix}^\mu\text{-}\mathsf{adm}$.

Lemma 4 (Typing of defined fixed-points) *Let* $\Gamma \vdash A\ \mathsf{fix}^\mu_n\text{-}\mathsf{adm}$. *If* $\Gamma \vdash s : \forall \imath.\, A\, \imath \to A\, (\imath + 1)$, *then* $\Gamma \vdash \mathsf{fix}^\mu_n[s] : \forall \imath.\, A\, \imath$. *Hence,*

$$\Gamma \vdash \mathsf{fix}^\mu_n : (\forall \imath.\, A\, \imath \to A\, (\imath + 1)) \to \forall \imath.\, A\, \imath.$$

Example 5 (Merge sort) Assume a type A with a comparison function $\leq : A \to A \to \mathsf{Bool}$, a function $\mathsf{merge} : \mathsf{List}^\infty A \to \mathsf{List}^\infty A \to \mathsf{List}^\infty A$ which merges two ordered lists into an ordered output list and a function $\mathsf{split} : \forall \imath.\, \mathsf{List}^\imath A \to \mathsf{List}^\imath A \times \mathsf{List}^\imath A$ which splits a list into two parts of roughly the same size. The type of split expresses that none of the output lists is bigger than the input. We can encode merge sort $\mathsf{msort}\, a\, as$ for non-empty lists $\mathsf{cons}\, a\, as$ in $\mathsf{F}^{\frown}_\omega$ as follows:

$$\begin{array}{ll} \mathsf{msort} : & \forall \imath.\, A \to \mathsf{List}^\imath A \to \mathsf{List}^\infty A \\ \mathsf{msort} := & \mathsf{fix}^\mu_1\, \lambda msort \lambda a \lambda xs.\ \mathsf{match}\ xs\ \mathsf{with} \\ & \quad \begin{array}{lll} \mathsf{nil} & \mapsto & \mathsf{cons}\ a\ \mathsf{nil} \\ \mathsf{cons}\, b\, l & \mapsto & \mathsf{let}\ (as, bs) = \mathsf{split}\ l \\ & & \quad \mathsf{in}\ \ \mathsf{merge}\, (msort\, a\, as)\, (msort\, b\, bs) \end{array} \end{array}$$

The recursive calls to $msort$ are legal because of the typing of split. Indeed, we can assign the following types:

$$\begin{array}{ll} msort & : A \to \mathsf{List}^\imath\, A \to \mathsf{List}^\infty\, A \\ a, b & : A \\ xs & : \mathsf{List}^{\imath+1}\, A \\ l & : \mathsf{List}^\imath\, A \\ as, bs & : \mathsf{List}^\imath\, A \end{array}$$

The termination of msort depends on the fact that split is non size-increasing. This information could have been established by other means than typing, e. g., by a term ordering as usual for termination of term rewriting systems. However, for the instances of generic programs we consider in the next section, the typing will be essential for termination checking.

3 A Framework for Generic Programming with Sized Types

Hinze [19] describes a framework for generic programming which is later extended by Hinze, Jeuring, and Löh [23] and implemented in *Generic Haskell* [22]. In this framework, both types and values can be constructed by recursion on some index type. The behavior is only specified for the type and constructor constants like Int, 1, $+$ and $\times$, and this uniquely defines the constructed type or value. In the following we propose an extension by sized types, *sized polytypic programming*, and demonstrate its strength by giving termination guarantees for Hinze's generalized tries [18].

In the sequel we will employ the following typographic conventions:

Capital	$\mathsf{Type}\langle A\rangle$	generic datatypes	type-indexed types
UPPERCASE	$\mathsf{TYPE}\langle \kappa\rangle$	their kinds	kind-indexed kinds
lowercase	$\mathsf{poly}\langle A\rangle$	generic functions	type-indexed values
Capital	$\mathsf{Poly}\langle \kappa\rangle$	their types	kind-indexed types

The definition of the framework will be grayed, example generic programs will be lightly grayed.

3.1 *Type-indexed Types*

In generic programming as proposed by Hinze, Jeuring, and Löh [23], one can define a family $\mathsf{Type}\langle A\rangle$ indexed by another type A. For instance, one can define the type $\mathsf{Map}\langle A\rangle V$ of finite maps from A to V generically for all index types A, by analyzing the structure of A. To this end, one specifies what $\mathsf{Map}\langle A\rangle$ should be for base types A_0 and for the standard type constructors, e. g., $+$ and $\times$. Then, $\mathsf{Map}\langle A\rangle$ is computed for a specific instance of A, where recursion is interpreted as the infinite unfolding. We differ from this setting in that we deal with inductive types instead of recursive types, thus, in our case, $\mathsf{Map}\langle A\rangle$ for an inductive type A will be itself an inductive type.

Roughly, a type-indexed type $\mathsf{Type}\langle A\rangle$ is A where constructor constants C have been replaced by user-defined constructors $\mathsf{Type}\langle C\rangle$. More precisely, we define $\mathsf{Type}\langle A\rangle$ by recursion on A, using the equations to follow. We generalize this definition to $\mathsf{Type}_{\vec{X}}\langle F\rangle$, where F is a constructor and $\vec{X}$ are the variables under whose binders we have stepped in the course of the definition. So, in fact, we are dealing with *constructor-indexed constructors*, but we stick to the sloppy terminology used by datatype-generic programmers.

$\mathsf{Type}\langle A\rangle$ can be fully computed if A is closed; if A contains a free variable Y, then $\mathsf{Type}\langle A\rangle$ will have occurrences of the subexpression $\mathsf{Type}\langle Y\rangle$ that can be further reduced if we substitute a (closed) type constructor for Y.

$$\begin{array}{lll}
\mathsf{Type}_{\vec{X}}\langle C\rangle & = \mathsf{Type}\langle C\rangle \quad \textit{(user-defined)} & \text{for } C \in \{1, +, \times, \mathsf{Int}, \mathsf{Char}, \dots\} \\
\mathsf{Type}_{\vec{X}}\langle X\rangle & = X & \text{if } X \in \vec{X} \\
\mathsf{Type}_{\vec{X}}\langle \lambda X F\rangle & = \lambda X.\, \mathsf{Type}_{\vec{X},X}\langle F\rangle & \\
\mathsf{Type}_{\vec{X}}\langle F\, G\rangle & = \mathsf{Type}_{\vec{X}}\langle F\rangle\ \mathsf{Type}_{\vec{X}}\langle G\rangle & \\
\mathsf{Type}_{\vec{X}}\langle \mu_\kappa\rangle & = \mu_?
\end{array}$$

What should the kind index to μ be in the last equation? We can answer this question if we look at the kind $\mathsf{TYPE}\langle\kappa\rangle$ of a type-indexed type $\mathsf{Type}\langle F\rangle$. The kind $\mathsf{TYPE}\langle\kappa\rangle$ depends on the kind κ of constructor F. The given equations for abstraction and application dictate the law for function kinds.

$$\begin{array}{ll}
\mathsf{TYPE}\langle *\rangle & = \kappa, \textit{user-defined} \\
\mathsf{TYPE}\langle \kappa_1 \xrightarrow{p} \kappa_2\rangle & = \mathsf{TYPE}\langle\kappa_1\rangle \xrightarrow{p} \mathsf{TYPE}\langle\kappa_2\rangle
\end{array}$$

The kind $\mathsf{TYPE}\langle *\rangle$ has to be chosen such that for the basic type constructors $C \in \Sigma$ the user-defined $\mathsf{Type}\langle C\rangle : \mathsf{TYPE}\langle\Sigma(C)\rangle$. (Of course, $\mathsf{Type}\langle C\rangle$ can be undefined for some C, typically for $C = \to$ and $C = \forall_\kappa$.) For instance, the kind $\mathsf{MAP}\langle\kappa\rangle$ for the type of finite maps $\mathsf{Map}\langle F : \kappa\rangle$ is defined by $\mathsf{MAP}\langle *\rangle = * \xrightarrow{+} *$. We can now complete the construction law for types indexed by inductive types.

$$\mathsf{Type}_{\vec{X}}\langle\mu_\kappa\rangle = \mu_{\mathsf{TYPE}\langle\kappa\rangle}$$

Remark 6 Note that the presence of polarities in $\mathsf{TYPE}\langle\Sigma(C)\rangle$ restricts the choices for $\mathsf{Type}\langle C\rangle$. For instance, $\mathsf{Type}\langle\to\rangle = \lambda A \lambda B. A \times B$ or $\mathsf{Type}\langle\times\rangle = \lambda A \lambda B. A \to B$ are both impossible (and meaningless). However, I do not know any practical, meaningful type-indexed types that are excluded by the polarity restriction.

We extend the framework to sized types by giving homomorphic construction rules for everything that concerns sizes:

$$\begin{array}{ll}
\mathsf{TYPE}\langle\mathsf{ord}\rangle & = \mathsf{ord} \\[1em]
\mathsf{Type}_{\vec{X}}\langle\mathsf{s}\rangle & = \mathsf{s} \\
\mathsf{Type}_{\vec{X}}\langle\infty\rangle & = \infty
\end{array}$$

Since except for constants, the defining clauses for type-indexed types are homomorphic, type-indexed types are compositional.

Lemma 7 (Substitution in type indices)
$\mathsf{Type}_{\vec{X}}\langle [G/X]F \rangle = [\mathsf{Type}_{\vec{X}}\langle G \rangle / X]\mathsf{Type}_{\vec{X},X}\langle F \rangle$.

Proof. By induction on F. □

As a consequence, type-indexed types preserve equality. Kinding and subtyping are also preserved, since $\mathsf{TYPE}\langle \kappa \rangle$ preserves the polarities in κ.

Theorem 8 (Well-kindedness, equality, and subtyping for type-indexed types)
Let Σ be a signature of constructor constants and assume $\mathsf{Type}\langle C \rangle : \mathsf{TYPE}\langle \kappa \rangle$ *for all* $(C\!:\!\kappa) \in \Sigma$.

(1) If $X_1 : p_1\kappa_1, \dots, X_n : p_n\kappa_n \vdash F : \kappa$, *then* $X_1 : p_1\mathsf{TYPE}\langle \kappa_1 \rangle, \dots, X_n : p_n\mathsf{TYPE}\langle \kappa_n \rangle \vdash \mathsf{Type}_{\vec{X}}\langle F \rangle : \mathsf{TYPE}\langle \kappa \rangle$.
(2) If $X_1 : p_1\kappa_1, \dots, X_n : p_n\kappa_n \vdash F = F' : \kappa$, *then* $X_1 : p_1\mathsf{TYPE}\langle \kappa_1 \rangle, \dots, X_n : p_n\mathsf{TYPE}\langle \kappa_n \rangle \vdash \mathsf{Type}_{\vec{X}}\langle F \rangle = \mathsf{Type}_{\vec{X}}\langle F' \rangle : \mathsf{TYPE}\langle \kappa \rangle$.
(3) If $X_1 : p_1\kappa_1, \dots, X_n : p_n\kappa_n \vdash F \leq F' : \kappa$, *then* $X_1 : p_1\mathsf{TYPE}\langle \kappa_1 \rangle, \dots, X_n : p_n\mathsf{TYPE}\langle \kappa_n \rangle \vdash \mathsf{Type}_{\vec{X}}\langle F \rangle \leq \mathsf{Type}_{\vec{X}}\langle F' \rangle : \mathsf{TYPE}\langle \kappa \rangle$.

Proof. Each by induction on the derivation. The rules are given in the appendix. □

Example: finite maps via generalized tries. Hinze [18] defines generalized tries $\mathsf{Map}\langle F \rangle$ by recursion on F. In particular, $\mathsf{Map}\langle K : * \rangle\, V$ is the type of finite maps from domain K to codomain V. The following representation using type-level λ can be found in his article on type-indexed data types [23, page 139].

$$\begin{array}{ll}
\mathsf{MAP}\langle * \rangle & := * \xrightarrow{+} * \\
\\
\mathsf{Map}\langle \mathsf{Int} \rangle & := \lambda V.\ \textit{efficient implementation of}\ \mathsf{Int} \to_{\mathsf{fin}} V \\
\mathsf{Map}\langle \mathsf{Char} \rangle & := \lambda V.\ \textit{efficient implementation of}\ \mathsf{Char} \to_{\mathsf{fin}} V \\
\mathsf{Map}\langle 1 \rangle & := \lambda V.\, 1 + V \\
\mathsf{Map}\langle + \rangle & := \lambda F \lambda G \lambda V.\, 1 + F\,V \times G\,V \\
\mathsf{Map}\langle \times \rangle & := \lambda F \lambda G \lambda V.\, F\,(G\,V)
\end{array}$$

Well-kindedness of these definitions is immediate, except maybe for $\mathsf{Map}\langle \times \rangle$ which must be of kind $(* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *)$. For $\mathsf{Map}\langle + \rangle$ we have used the variant of *spotted products* (or lifted products) which Hinze mentions in section 4.1 of his article [18]. This way we avoid that certain empty tries have an infinite normal form (see [18, page 341]) which requires lazy evaluation. The constructor for finite maps over strings can now be computed as follows:

$$\begin{aligned}
&\mathsf{Map}\langle \lambda\imath.\, \mathsf{List}^{\imath}\, \mathsf{Char}\rangle \\
=\ &\mathsf{Map}\langle \lambda\imath.\, \mu^{\imath}_{*}\, \lambda X.\, 1 + \mathsf{Char} \times X\rangle \\
=\ &\lambda\imath.\, \mu^{\imath}_{*\xrightarrow{+}*}\, \lambda X.\, \mathsf{Map}\langle + \rangle\, \mathsf{Map}\langle 1\rangle\, (\mathsf{Map}\langle \times\rangle\, \mathsf{Map}\langle \mathsf{Char}\rangle\, X) \\
=\ &\lambda\imath.\, \mu^{\imath}_{*\xrightarrow{+}*}\, \lambda X \lambda V.\, 1 + (1 + V) \times \mathsf{Map}\langle \mathsf{Char}\rangle\, (X\, V)
\end{aligned}$$

The matching kind is

$$\mathsf{MAP}\langle \mathsf{ord} \xrightarrow{+} *\rangle = \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} *.$$

Note that the type $\mathsf{Map}\langle \lambda\imath.\, \mathsf{List}^{\imath}\, \mathsf{Char}\rangle$ of sized, string-indexed tries involves a higher-kinded inductive type $\mu_{*\xrightarrow{+}*}$. However, it is not heterogeneous, but homogeneous, meaning that X is always applied to the variable V. Thus, we have the option to simplify it using *λ-dropping* and obtain an ordinary inductive type:

$$\mathsf{Map}\langle \lambda\imath.\, \mathsf{List}^{\imath}\, \mathsf{Char}\rangle = \lambda\imath\lambda V.\, \mu^{\imath}_{*}\, \lambda Y.\, 1 + (1 + V) \times \mathsf{Map}\langle \mathsf{Char}\rangle\, Y$$

It is easy to interpret this type as a trie for strings with prefix p: The trie is either "()" (first 1), meaning that strings with this prefix are undefined in the finite map, or it is a pair of maybe a value v (the value mapped to p) and of one trie for strings with prefix $p \cdot c$ for each $c \in \mathsf{Char}$. A trie for strings with empty prefix is then a finite map over all strings.

Remark 9 (λ-dropping) We use λ-dropping in some places in this article to simplify types or values computed by the generic programming framework. However, it has no official status and is not essential for our results.

3.2 Type-indexed Values

The key ingredient to generic programming are type-indexed values, meaning, programs $\mathsf{poly}\langle F\rangle$ which work for different type constructors F but are uniformly (generically) constructed by recursion on F. Again, the user supplies the desired behavior $\mathsf{poly}\langle C\rangle$ on base types and type constructors C, and the polytypic program $\mathsf{poly}\langle F\rangle$ is then constructed by the following laws.

$$\begin{aligned}
\mathsf{poly}\langle C\rangle &= \textit{user-defined} \\
\mathsf{poly}\langle X\rangle &= X \\
\mathsf{poly}\langle \lambda X F\rangle &= \lambda X.\, \mathsf{poly}\langle F\rangle \\
\mathsf{poly}\langle F\, G\rangle &= \mathsf{poly}\langle F\rangle\, \mathsf{poly}\langle G\rangle \\
\mathsf{poly}\langle \mu_{\kappa}\rangle &= \mathsf{fix}
\end{aligned}$$

The handling of variables is still sloppy in this definition; we assume that we can reuse type variables as special term variables. Here we just want to convey the idea, later we will make the variable handling water proof.

Hinze [20] has observed that type-indexed values $\mathsf{poly}\langle F : \kappa\rangle$ have kind-indexed types $\mathsf{Poly}\langle F, \ldots, F : \kappa\rangle : *$ with possibly several copies of the parameter F, obeying the following laws:

$$\begin{aligned}
\mathsf{Poly}\langle A_1, \ldots, A_n : *\rangle &= \textit{user-defined, parametric in } \vec{A} \\
\mathsf{Poly}\langle F_1, \ldots, F_n : \kappa \xrightarrow{p} \kappa'\rangle &= \forall G_1 : \kappa \ldots \forall G_n : \kappa. \\
&\qquad \mathsf{Poly}\langle G_1, \ldots, G_n : \kappa\rangle \to \mathsf{Poly}\langle F_1\, G_1, \ldots, F_n\, G_n : \kappa'\rangle
\end{aligned}$$

For example, three copies of F are required for a generic definition of zipping functions [20, Sect. 7.2].

Hinze works in a framework where only covariant type constructors serve as indices, i. e., $p = +$ in the above equation. However, with polarity information at hand, it is sometimes useful to depart from Hinze's scheme. One example is a generic map function (monotonicity witness, functoriality witness):

$$\begin{aligned}
\mathsf{GMap}\langle A, B : *\rangle &:= A \to B \\
\mathsf{GMap}\langle F, G : \kappa \xrightarrow{-} \kappa'\rangle &:= \forall X \forall Y.\, \mathsf{GMap}\langle Y, X : \kappa\rangle \to \mathsf{GMap}\langle F\,X,\ G\,Y : \kappa'\rangle \\
\mathsf{GMap}\langle F, G : \kappa \xrightarrow{p} \kappa'\rangle &:= \forall X \forall Y.\, \mathsf{GMap}\langle X, Y : \kappa\rangle \to \mathsf{GMap}\langle F\,X,\ G\,Y : \kappa'\rangle \\
&\qquad \text{for } p \in \{+, \circ\}
\end{aligned}$$

With this refined definition of kind-indexed type, a generic map function is definable which also works for data types with embedded function spaces, e. g., Tree.

$$\begin{aligned}
\mathsf{gmap}\langle 1 : *\rangle &:= \lambda u.\, u \\
\mathsf{gmap}\langle + : * \xrightarrow{+} * \xrightarrow{+} *\rangle &:= \lambda f \lambda g \lambda s.\, \mathsf{case}\ s\ (\lambda x.\, \mathsf{inl}\,(f x))\ (\lambda y.\, \mathsf{inr}\,(g\,y)) \\
\mathsf{gmap}\langle \times : * \xrightarrow{+} * \xrightarrow{+} *\rangle &:= \lambda f \lambda g \lambda p.\, (f\,(\mathsf{fst}\,p),\ g\,(\mathsf{snd}\,p)) \\
\mathsf{gmap}\langle \to : * \xrightarrow{-} * \xrightarrow{+} *\rangle &:= \lambda f \lambda g \lambda h \lambda x.\, g\,(h\,(f\,x))
\end{aligned}$$

For the main example we want to consider, generic operations for tries, types $\mathsf{Poly}\langle F : \kappa\rangle$ indexed by a single constructor F are sufficient, hence, we will restrict the following development to this case. Formally, our framework contains rules to compute

$$\mathsf{Poly}\langle \kappa\rangle : \kappa \xrightarrow{\circ} *$$

from the user-defined type constructor $\mathsf{Poly}\langle *\rangle : * \xrightarrow{\circ} *$. The notation $\mathsf{Poly}\langle F : \kappa\rangle$ shall be a shorthand for the β-normal form of $\mathsf{Poly}\langle \kappa\rangle\ F$ (thus, removing administrative redexes). If the kind κ of F is clear from the context of discourse, we abbreviate $\mathsf{Poly}\langle F : \kappa\rangle$ by $\mathsf{Poly}\langle F\rangle$. The computation rules for $\mathsf{Poly}\langle\kappa\rangle$ are the following:

$$\begin{array}{ll}
\mathsf{Poly}\langle *\rangle & = \lambda X A : * \overset{\circ}{\to} *, \textit{user-defined} \\
\mathsf{Poly}\langle \mathsf{ord}\rangle & = \lambda\_1 \\
\mathsf{Poly}\langle \kappa_1 \overset{p}{\to} \kappa_2\rangle & = \lambda F \forall G\!:\!\kappa_1.\, \mathsf{Poly}\langle\kappa_1\rangle\, G \to \mathsf{Poly}\langle\kappa_2\rangle (F\, G)
\end{array}$$

The second line treats base kind ord which is new in $\mathsf{F}^{\widehat{}}_{\omega}$. Since ordinals are only used to increase the static information about programs, they have no computational significance and can be mapped to inhabitants of the unit type.

Theorem 10 (Well-kindedness of kind-indexed types) *If* $\Gamma \vdash \mathsf{Poly}\langle *\rangle : * \overset{\circ}{\to} *$, *then* $\Gamma \vdash \mathsf{Poly}\langle\kappa\rangle : \kappa \overset{\circ}{\to} *$.

Proof. By induction on κ. □

The defined notion $\mathsf{Poly}\langle F : \kappa\rangle$ is trivially substitutive, meaning $\Gamma \vdash \mathsf{Poly}\langle [G/X]F : \kappa\rangle = [G/X]\mathsf{Poly}\langle F : \kappa\rangle : *$ when $\Gamma \vdash G : \kappa'$ and $\Gamma, X\!:\!\kappa' \vdash F : \kappa$. And likewise trivially, it respects equality: $\Gamma \vdash F = F' : \kappa$ implies $\Gamma \vdash \mathsf{Poly}\langle F : \kappa\rangle = \mathsf{Poly}\langle F' : \kappa\rangle : *$.

We can refine the generation laws for type-indexed programs as follows. Formally, we define $\mathsf{poly}_{n,\phi}\langle F\rangle$ where n is a natural number and ϕ maps type variables to term variables.

$$\begin{array}{lll}
\mathsf{poly}\langle C\rangle & = t : \mathsf{Poly}\langle C\rangle, \textit{user-defined} & \\
\mathsf{poly}_{n,\phi}\langle C\rangle & = \mathsf{poly}\langle C\rangle & \\
\mathsf{poly}_{n,\phi}\langle X\rangle & = \phi(X) & \\
\mathsf{poly}_{n,\phi}\langle \lambda X F\rangle & = \lambda x.\, \mathsf{poly}_{n,\phi[X\mapsto x]}\langle F\rangle, & x \text{ fresh} \\
\mathsf{poly}_{n,\phi}\langle F\, G\rangle & = \mathsf{poly}_{n,\phi}\langle F\rangle\, \mathsf{poly}_{n,\phi}\langle G\rangle & \\
\mathsf{poly}_{n,\phi}\langle \mu_{\vec{p}\vec{\kappa}\to *}\rangle & = \lambda\_.\ \mathsf{fix}^{\mu}_{n+|\vec{\kappa}|} &
\end{array}$$

In the last equation there is a void abstraction to take care of the argument of unit type which arises from the ordinal argument a of μ^a_κ. Also, n has to be chosen such that the nth argument to the resulting recursive function is of an inductive type whose size is associated to a. The choice of n depends on the definition of the type $\mathsf{Poly}\langle A\!:\!*\rangle$ of the type-indexed program given by the user. For the example of map lookup functions (see below), the polytypic program is of type

$$\mathsf{Lookup}\langle K\!:\!*\rangle := \forall V.\, K \to \mathsf{Map}\langle K\rangle\, V \to 1 + V.$$

Hence, we set $n = 0$, because the recursive argument of the function that is generated in case $K = \mu^a F$ is the first one, of type K. In the example of finite map merging to follow, we will have the type

$$\mathsf{Merge}\langle K : *\rangle := \forall V.\, \mathsf{Bin}\, V \to \mathsf{Bin}\, (\mathsf{Map}\langle K\rangle\, V)$$

with $\mathsf{Bin}\, V = V \to V \to V$. Since $\mathsf{Map}\langle K\rangle$ is an inductive type for inductive K, the second argument is the recursive one and we have $n = 1$. The choice of n will be formally determined in the next section.

Example: generic finite map lookup. In the following, we implement Hinze's generic lookup function in our framework. The definitions on the program level are unchanged, only the types are now sized, and we give termination guarantees. The lookup function is polymorphic in the value type V, it takes a key of type K and a finite map from K to V and returns maybe a value.

$$\mathsf{Lookup}\langle K : *\rangle := \forall V.\, K \to \mathsf{Map}\langle K\rangle\, V \to 1 + V$$

In the definition of the generic lookup function, we use the bind operation $\gg=$ for the *Maybe* monad $\lambda V.\, 1 + V$. It obeys the laws $(\mathsf{inl}() \gg= f) \longrightarrow \mathsf{inl}()$ and $(\mathsf{inr}\, v \gg= f) \longrightarrow f\, v$.

$$\begin{array}{lll}
\mathsf{lookup}\langle 1\rangle & : & \forall V.\, 1 \to 1 + V \to 1 + V \\
\mathsf{lookup}\langle 1\rangle & := & \lambda k \lambda m.\, m \\
\\
\mathsf{lookup}\langle +\rangle & : & \forall A{:}*.\, \mathsf{Lookup}\langle A\rangle \to \forall B{:}*.\, \mathsf{Lookup}\langle B\rangle \to \\
 & & \forall V.\, A + B \to 1 + (\mathsf{Map}\langle A\rangle\, V) \times (\mathsf{Map}\langle B\rangle\, V) \to 1 + V \\
\mathsf{lookup}\langle +\rangle & := & \lambda la \lambda lb \lambda ab \lambda tab.\, tab \gg= \lambda(ta, tb). \\
 & & \quad \mathsf{match}\ ab\ \mathsf{with} \\
 & & \qquad \mathsf{inl}\, a \mapsto la\, a\, ta \\
 & & \qquad \mathsf{inr}\, b \mapsto lb\, b\, tb \\
\\
\mathsf{lookup}\langle \times\rangle & : & \forall A{:}*.\, \mathsf{Lookup}\langle A\rangle \to \forall B{:}*.\, \mathsf{Lookup}\langle B\rangle \to \\
 & & \forall V.\, A \times B \to \mathsf{Map}\langle A\rangle\, (\mathsf{Map}\langle B\rangle\, V) \to 1 + V \\
\mathsf{lookup}\langle \times\rangle & := & \lambda la \lambda lb \lambda (a, b) \lambda tab.\, la\, a\, tab \gg= \lambda tb.\, lb\, b\, tb
\end{array}$$

All these definitions are well-typed, which is easy to check since there are no references to sizes.

Example: lookup for list-shaped keys. The previous definitions determine the instance of the generic lookup function for the type constructor of lists.The kind-indexed type unfolds to:

$$\begin{aligned}
&\mathsf{Lookup}\langle \mathsf{List} : \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} *\rangle \\
&= \forall\imath.\, 1 \to \forall K.\, \mathsf{Lookup}\langle K : *\rangle \to \mathsf{Lookup}\langle \mathsf{List}^{\imath}\, K : *\rangle \\
&= \forall\imath.\, 1 \to \forall K.\, \mathsf{Lookup}\langle K : *\rangle \to \forall V.\, \mathsf{List}^{\imath} K \to \mathsf{Map}\langle \mathsf{List}^{\imath} K\rangle \to 1 + V \\
&= \forall\imath.\, 1 \to \forall K.\, \mathsf{Lookup}\langle K : *\rangle \to \forall V.\, \mathsf{List}^{\imath} K \to \\
&\qquad (\mu^{\imath}_{*} \lambda Y.\, 1 + (1+V) \times Y) \to 1 + V
\end{aligned}$$

Note that the type $\mathsf{Lookup}\langle\mathsf{List}\rangle$ mentions the size variable $\imath$ twice, as index to both inductive arguments. This makes sense, since the length of the search keys determines the depth of the trie. The lookup function is computed as follows:

$$\begin{aligned}
\mathsf{lookup}_0\langle\mathsf{List}\rangle &= \mathsf{lookup}_0\langle \lambda\imath\lambda K.\, \mu^{\imath}_{*}\lambda X.\, 1 + K \times X\rangle \\
&= \lambda i \lambda \mathit{lookup}_K.\, (\lambda\_.\, \mathsf{fix}^{\mu}_{0})\, i\, (\lambda \mathit{lookup}.\, \mathsf{lookup}\langle +\rangle\, \mathsf{lookup}\langle 1\rangle \\
&\qquad\quad (\mathsf{lookup}\langle\times\rangle\, \mathit{lookup}_K\, \mathit{lookup})) \\
&= \lambda i \lambda \mathit{lookup}_K.\, \mathsf{fix}^{\mu}_{0}\, \lambda \mathit{lookup} \lambda l \lambda m.\, m \gg\!= \lambda(n, c). \\
&\qquad \mathsf{match}\ l\ \mathsf{with} \\
&\qquad\quad \mathsf{nil} \qquad\ \ \mapsto n \\
&\qquad\quad \mathsf{cons}\, k\, l' \mapsto \mathit{lookup}_K\, k\, c \gg\!= \lambda m'.\, \mathit{lookup}\, l'\, m'
\end{aligned}$$

Well-typedness follows as the type $\mathsf{Lookup}\langle \mathsf{List}^{\imath} K\rangle$ is valid for recursion with fix^{μ}_{0}, according to criterion given in Sect. 2. We reason on an abstract level:

$$\begin{array}{lll}
\mathit{lookup}_K & & : \mathsf{Lookup}\langle K\rangle \\
\mathit{lookup} & & : \mathsf{Lookup}\langle \mathsf{List}^{\imath} K\rangle \\
\mathsf{lookup}\langle\times\rangle\, \mathit{lookup}_K\, \mathit{lookup} & =: r & : \mathsf{Lookup}\langle K \times \mathsf{List}^{\imath} K\rangle \\
\mathsf{lookup}\langle +\rangle\, \mathsf{lookup}\langle 1\rangle\, r & =: s & : \mathsf{Lookup}\langle 1 + K \times \mathsf{List}^{\imath} K\rangle \\
 & & : \mathsf{Lookup}\langle \mathsf{List}^{\imath+1} K\rangle \\
\mathsf{fix}^{\mu}_{0}\, \lambda \mathit{lookup}.\, s & & : \mathsf{Lookup}\langle \mathsf{List}^{\imath} K\rangle \\
\mathsf{lookup}\langle\mathsf{List}\rangle & & : \mathsf{Lookup}\langle\mathsf{List}\rangle
\end{array}$$

Example: generic trie merging. Hinze [18] presents three elementary operations to construct finite tries: empty, single, and merge. In the following we replay the construction of merge in our framework, since it exhibits a very interesting recursion scheme.

We first define the type $\mathsf{Bin}\, V$ for binary operations on V and a function comb which lifts a merging function for V to a merging function for $1 + V$:

$$\begin{array}{lll}
\mathsf{Bin} & : & * \overset{\circ}{\rightarrow} * \\
\mathsf{Bin} & := & \lambda V.\, V \rightarrow V \rightarrow V \\
\\
\mathsf{comb} & : & \forall V.\, (V \rightarrow V \rightarrow V) \rightarrow (1 + V \rightarrow 1 + V \rightarrow 1 + V) \\
\mathsf{comb} & := & \lambda c \lambda m_1 \lambda m_2.\ \mathsf{match}\ (m_1, m_2)\ \mathsf{with} \\
 & & \qquad (\mathsf{inl}(), \_) \qquad\quad \mapsto m_2 \\
 & & \qquad (\_, \mathsf{inl}()) \qquad\quad \mapsto m_1 \\
 & & \qquad (\mathsf{inr}\, v_1, \mathsf{inr}\, v_2) \mapsto \mathsf{inr}\, (c\, v_1\, v_2)
\end{array}$$

When merging two finite maps of type $\mathsf{Map}\langle K \rangle\, V$, one has to resolve a conflict when both maps assign a value to the same key. This conflict resolution function $c : \mathsf{Bin}\, V$ is abstracted out, leading to the following kind-indexed type:

$$\mathsf{Merge}\langle K : * \rangle := \forall V.\ \mathsf{Bin}\, V \rightarrow \mathsf{Bin}\, (\mathsf{Map}\langle K \rangle\, V)$$

The following definitions determine a generic merging function for finite maps over key whose type is composed from 1, $+$, $\times$, and μ.

$$\begin{array}{lll}
\mathsf{merge}\langle 1 \rangle & : & \mathsf{Merge}\langle 1 \rangle \\
\mathsf{merge}\langle 1 \rangle & := & \mathsf{comb} \\
\\
\mathsf{merge}\langle + \rangle & : & \forall A.\, \mathsf{Merge}\langle A \rangle \rightarrow \forall B.\, \mathsf{Merge}\langle B \rangle \rightarrow \\
 & & \quad \forall V.\, \mathsf{Bin}\, V \rightarrow \mathsf{Bin}\, (1 + \mathsf{Map}\langle A \rangle\, V \times \mathsf{Map}\langle B \rangle\, V) \\
\mathsf{merge}\langle + \rangle & := & \lambda ma \lambda mb \lambda c.\ \mathsf{comb} \\
 & & \quad \lambda(ta_1, tb_1) \lambda(ta_2, tb_2).\ (ma\ c\ ta_1\ ta_2,\ mb\ c\ tb_1\ tb_2) \\
\\
\mathsf{merge}\langle \times \rangle & : & \forall A.\, \mathsf{Merge}\langle A \rangle \rightarrow \forall B.\, \mathsf{Merge}\langle B \rangle \rightarrow \\
 & & \quad \forall V.\, \mathsf{Bin}\, V \rightarrow \mathsf{Bin}\, (\mathsf{Map}\langle A \rangle\, (\mathsf{Map}\langle B \rangle\, V)) \\
\mathsf{merge}\langle \times \rangle & := & \lambda ma \lambda mb \lambda c.\ ma\, (mb\, c)
\end{array}$$

In the last line, $mb\, c : \mathsf{Bin}\, (\mathsf{Map}\langle B \rangle\, V)$ merges finite maps over key type B. This function serves as conflict resolver to merge finite maps from A to $\mathsf{Map}\langle B \rangle\, V$, arriving at the elegant $\lambda c.\ ma\, (mb\, c) : \mathsf{Merge}\langle A \times B : * \rangle$.

Example: list-trie merging. The type of the merge function for finite maps over list-shaped keys is derived as follows:

$$\begin{aligned}\mathsf{Merge}\langle \mathsf{List} : \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} *\rangle &= \forall \imath.\, 1 \to \forall K.\, \mathsf{Merge}\langle K\rangle \to \mathsf{Merge}\langle \mathsf{List}^{\imath} K\rangle \\ &= \forall \imath.\, 1 \to \forall K.\, (\forall V.\, \mathsf{Bin}\, V \to \mathsf{Bin}\, (\mathsf{Map}\langle K\rangle\, V)) \\ &\qquad\qquad \to \forall W.\, \mathsf{Bin}\, W \to \mathsf{Bin}\, (\mathsf{Map}\langle \mathsf{List}^{\imath} K\rangle\, W)\end{aligned}$$

Mechanically, we compute the merge function for list-tries:

$$\begin{aligned}&\mathsf{merge}_1\langle \mathsf{List}\rangle \\ &\quad = \mathsf{merge}_1\langle \lambda\imath\lambda K.\, \mu^{\imath}\lambda X.\, 1 + K \times X\rangle \\ &\quad = \lambda i \lambda merge_K.\, (\lambda\_\, \mathsf{fix}^{\mu}_1)\, i\, \mathsf{merge}_1\langle \lambda X.\, 1 + K \times X\rangle \\ &\quad = \lambda i \lambda merge_K.\, \mathsf{fix}^{\mu}_1(\lambda merge.\, \mathsf{merge}\langle +\rangle\, \mathsf{merge}\langle 1\rangle\, (\mathsf{merge}\langle\times\rangle\, merge_K\, merge)) \\ &\quad = \lambda i \lambda merge_K.\, \mathsf{fix}^{\mu}_1\, \lambda merge \lambda c.\, \mathsf{comb} \\ &\qquad\qquad \lambda(mv_1, t_1)\lambda(mv_2, t_2).\, (\mathsf{comb}\, c\, mv_1\, mv_2,\ merge_K\, (merge\, c)\, t_1 t_2) \\ &\quad [= \lambda i \lambda merge_K \lambda c.\, \mathsf{fix}^{\mu}_0\, \lambda merge.\, \mathsf{comb} \\ &\qquad\qquad \lambda(mv_1, t_1)\lambda(mv_2, t_2).\, (\mathsf{comb}\, c\, mv_1\, mv_2,\ merge_K\, merge\, t_1\, t_2)]\end{aligned}$$

In the last step (enclosed in [brackets]), we have decreased the rank of recursion by λ-dropping.

Surprisingly, recursion happens not by invoking $merge$ on structurally smaller arguments, but by *passing the function itself* to a parameter, $merge_K$. Here, type-based termination reveals its strength; it is not possible to show termination of $\mathsf{merge}\langle\mathsf{List}\rangle$ disregarding its type. With sized types, however, the termination proof is again just a typing derivation, as easy as for $\mathsf{lookup}\langle\mathsf{List}\rangle$. We reason again on the abstract level:

$$\begin{array}{lll} merge_K & & : \mathsf{Merge}\langle K\rangle \\ merge & & : \mathsf{Merge}\langle \mathsf{List}^{\imath} K\rangle \\ \mathsf{merge}\langle\times\rangle\, merge_K\, merge & =: r & : \mathsf{Merge}\langle K \times \mathsf{List}^{\imath} K\rangle \\ \mathsf{merge}\langle +\rangle\, \mathsf{merge}\langle 1\rangle\, r & =: s & : \mathsf{Merge}\langle 1 + K \times \mathsf{List}^{\imath} K\rangle \\ & & : \mathsf{Merge}\langle \mathsf{List}^{\imath+1} K\rangle \\ \mathsf{fix}^{\mu}_1\, \lambda merge.\, s & & : \mathsf{Merge}\langle \mathsf{List}^{\imath} K\rangle \end{array}$$

The type $\mathsf{Merge}\langle \mathsf{List}^{\imath} K\rangle$ is admissible for recursion on the second argument (the first argument is of type $\mathsf{Bin}\, V$): The whole type is of shape $\forall V.\, \mathsf{Bin}\, V \to \mu^{\imath} F \to \mu^{\imath} F \to \mu^{\imath} F$ for some F which does not depend on the size variable $\imath$. Hence, the type has the required shape.

Example: merging bushy tries. An even more dazzling recursion pattern is exhibited by the merge function for "bushy" tries, i. e., finite maps over bushy lists.

$$\begin{array}{ll}
\mathsf{Bush} & : \quad \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Bush} & := \lambda\imath.\, \mu^{\imath}_{* \xrightarrow{+} *}\, \lambda X \lambda K.\, 1 + K \times X\,(X\,K)
\end{array}$$

$$\begin{array}{ll}
\mathsf{Map}\langle\mathsf{Bush}\rangle & : \quad \mathsf{ord} \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *) \\
\mathsf{Map}\langle\mathsf{Bush}\rangle & = \lambda\imath.\, \mu^{\imath}_{(* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *)}\, \lambda X \lambda F \lambda V.\, 1 + (1 + V) \times F\,(X\,(X\,F)\,V)
\end{array}$$

The merge function for bush-indexed tries can be derived routinely:

$$\begin{array}{l}
\mathsf{merge}_1\langle\mathsf{Bush}\rangle \\
\quad = \mathsf{merge}_1\langle\lambda\imath.\, \mu^{\imath}_{* \xrightarrow{+} *} \lambda X \lambda K.\, 1 + K \times X\,(X\,K)\rangle \\
\quad = \lambda i.\, (\lambda\_\, \mathsf{fix}^{\mu}_{2})\; i\; (\lambda \mathit{merge} \lambda \mathit{merge}_K. \\
\qquad\quad \mathsf{merge}\langle+\rangle\; \mathsf{merge}\langle 1\rangle\; (\mathsf{merge}\langle\times\rangle\; \mathit{merge}_K\; (\mathit{merge}\; (\mathit{merge}\; \mathit{merge}_K)))) \\
\quad = \lambda i.\; \mathsf{fix}^{\mu}_{2}\; \lambda \mathit{merge} \lambda \mathit{merge}_K \\
\qquad\quad \lambda c.\, \mathsf{comb}\; \lambda(\mathit{mv}_1, t_1) \lambda(\mathit{mv}_2, t_2). \\
\qquad\qquad (\mathsf{comb}\; c\; \mathit{mv}_1\; \mathit{mv}_2,\;\; \mathit{merge}_K\; (\mathit{merge}\; (\mathit{merge}\; \mathit{merge}_K)\; c)\; t_1\; t_2)
\end{array}$$

The recursion pattern of $\mathsf{merge}\langle\mathsf{Bush}\rangle$ is adventurous. Not only is the recursive instance merge passed to an argument to the function merge_K, but also this function is modified during recursion: it is replaced by $(\mathit{merge}\; \mathit{merge}_K)$, which involves the recursive instance again! All these complications are coolly handled by type-based termination!

We have now seen how termination of some *specific* programs, which happened to be instances of generic programs, can be established using types. In the next section we develop a criterion on generic programs which entails termination of *all* of their instances.

4 Termination of Generic Programs

In this section, we establish the soundness of the framework presented in the last section, i. e., we prove that all instances of generic programs are indeed terminating. It suffices to show that they are well-typed in $\mathsf{F}^{\widehat{}}_{\omega}$, since this type system only accepts total programs. Critical is the typing of fixed-points

$$\mathsf{poly}_{n,\phi}\langle\mu_\kappa : \mathsf{ord} \xrightarrow{+} (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa\rangle = \lambda\_.\; \mathsf{fix}^{\mu}_{n+m}, \text{ where } \kappa = \vec{p}\vec{\kappa} \to *, m = |\vec{\kappa}|$$

or more perspicuously, $\mathsf{poly}_{n,\phi}\langle\mu^{\imath}_{\kappa}\rangle = \mathsf{fix}^{\mu}_{n+m} : \forall F.\, \mathsf{Poly}\langle F : \kappa \xrightarrow{+} \kappa\rangle \to \mathsf{Poly}\langle\mu^{\imath}\, F\rangle$. It is easily established that the step term s, the first argument to fix^{μ}_{n+m}, can be instantiated to the correct type.

Lemma 11 *If* $\Gamma \vdash s : \mathsf{Poly}\langle F : \kappa \xrightarrow{+} \kappa\rangle$ *then* $\Gamma \vdash s : \mathsf{Poly}\langle \mu^{\imath} F\rangle \to \mathsf{Poly}\langle \mu^{\imath+1} F\rangle$.

Proof. We have $\mathsf{Poly}\langle F : \kappa \xrightarrow{+} \kappa\rangle = \forall G{:}\kappa. \mathsf{Poly}\langle G : \kappa\rangle \to \mathsf{Poly}\langle F\, G : \kappa\rangle$, hence, instantiating G to $\mu^{\imath} F$ and using the equation $F\ (\mu^{\imath} F) = \mu^{\imath+1} F$, we get the typing $\Gamma \vdash s : \mathsf{Poly}\langle \mu^{\imath} F\rangle \to \mathsf{Poly}\langle \mu^{\imath+1} F\rangle$. □

So $\mathsf{poly}_{n,\phi}\langle F\rangle$ is a candidate for a step term s of a recursive function $\mathsf{fix}^{\mu}_{n+m}\, s$. If we can ensure that $\lambda\imath.\, \mathsf{Poly}\langle \mu^{\imath} F\rangle$ is admissible for recursion, we have ensured termination of all generic programs in our framework. Let us recall admissible recursion types, this time in form of an inductive definition.

Lemma 12 (Inductive definition of fix^{μ}_{n}-adm) *The following rules define* fix^{μ}_{n}-adm *inductively.*

$$\frac{\Gamma, \imath{:}{+}\mathsf{ord} \vdash A : *}{\Gamma \vdash \lambda\imath.\, A\ \mathsf{fix}^{\mu}_{-\infty}\text{-adm}} \qquad (1)$$

$$\frac{\Gamma, Y{:}\circ\kappa \vdash \lambda\imath. A\ \mathsf{fix}^{\mu}_{n}\text{-adm}}{\Gamma \vdash \lambda\imath. \forall Y{:}\kappa.\, A\ \mathsf{fix}^{\mu}_{n}\text{-adm}} \qquad (2)$$

$$\frac{\Gamma \vdash F : (\vec{p\kappa} \to *) \xrightarrow{+} (\vec{p\kappa} \to *) \qquad p_i^{-1}\Gamma \vdash G_i : \kappa_i\ (\mathit{all}\ i) \qquad \Gamma \vdash \lambda\imath. A\ \mathsf{fix}^{\mu}_{n}\text{-adm}}{\Gamma \vdash \lambda\imath. \mu^{\imath} F\, \vec{G} \to A\ \mathsf{fix}^{\mu}_{0}\text{-adm}} \qquad (3)$$

$$\frac{-\Gamma, \imath{:}{-}\mathsf{ord} \vdash A_0 : * \qquad \Gamma \vdash \lambda\imath. A\ \mathsf{fix}^{\mu}_{n}\text{-adm}}{\Gamma \vdash \lambda\imath.\, A_0 \to A\ \mathsf{fix}^{\mu}_{n+1}\text{-adm}} \qquad (4)$$

Each admissible function type $\Gamma \vdash \lambda\imath. A\ \mathsf{fix}^{\mu}_{n}$-adm with $n \geq 0$ must end in a covariant codomain (1), which is preceded by at least one inductive domain (3) and by an arbitrary number of contravariant domains (4). Universal quantification can come in at any point (2). The first inductive domain counts as the recursive argument, and the number of domains preceding it make up n.

With the judgement $\Gamma \vdash^{X}_{n} B$ to follow, we characterize such user-defined types $\mathsf{Poly}\langle *\rangle$ of generic programs that give rise to admissible recursion types. More precisely, if $\diamond \vdash^{X}_{n} \mathsf{Poly}\langle X : *\rangle$ and $\kappa = \vec{p\kappa} \to *$ with $m = |\vec{\kappa}|$, then $\lambda\imath.\, \mathsf{Poly}\langle \mu^{\imath}_{\kappa} F : \kappa\rangle\ \mathsf{fix}^{\mu}_{n+m}$-adm (Lemma 16). Simultaneously, the judgement determines the n in the number $n + m$ of non-recursive arguments in fix^{μ}_{n+m}.

Definition 13 (Admissible types of recursive generic programs) Let Γ be a kinding context, X a variable not assigned in Γ, n a natural number or $-\infty$, and B a type. The judgement $\Gamma \vdash^X_n B$ is inductively given by the following rules.

$$\frac{\Gamma, X\!:\!+* \vdash B : *}{\Gamma \vdash^X_{-\infty} B}\ p \leq + \tag{1}$$

$$\frac{\Gamma, Y\!:\!\circ\kappa \vdash^X_n B}{\Gamma \vdash^X_n \forall Y\!:\!\kappa.\, B} \tag{2}$$

$$\frac{p_i^{-1}\Gamma \vdash H_i : \kappa_i \text{ (for all } i = 1..|\vec{\kappa}|) \qquad \Gamma \vdash^X_n B}{\Gamma \vdash^X_0 \mathsf{Type}\langle X\rangle\, \vec{H} \to B}\ \mathsf{TYPE}\langle *\rangle = \vec{p\kappa} \to *$$

$$\frac{\Gamma \vdash^X_n B}{\Gamma \vdash^X_0 X \to B} \tag{3}$$

$$\frac{-\Gamma, X\!:\!-* \vdash A : * \qquad \Gamma \vdash^X_n B}{\Gamma \vdash^X_{n+1} A \to B}\ p \leq + \tag{4}$$

In $\Gamma \vdash^X_n B$, we assume that $X \notin \Gamma$ and maintain the invariant $\Gamma, X\!:\!\circ* \vdash B : *$. If $n = -\infty$, then even $\Gamma, X : +* \vdash B : *$. We use the subscript $-\infty$ to indicate that an occurrence of X or $\mathsf{Type}\langle X\rangle$ has not yet been encountered, which means that there is no recursive argument yet. (Note that $-\infty + 1 = -\infty$.)

The rules of $\vdash^X_n$ correspond to the ones of fix^μ_n-adm and have been numbered accordingly. The placeholder X will be instantiated to an inductive type $\mu^\imath F\, \vec{G}$. Rule (1): Since B is monotone in X, its instantiation will be monotone in $\imath$. Remember that admissible function types need at least one inductive domain (3). This can either come from X or from a type-indexed type $\mathsf{Type}\langle X\rangle$, since a type indexed by an inductive type is again an inductive type. Finally, leading non-recursive arguments must be in contravariant domains (4).

Example 14 The kind-indexed types of the last section satisfy this judgement.

$$\begin{aligned}
&\vdash^K_0 \mathsf{Lookup}\langle K : *\rangle = \forall V.\, K \to \mathsf{Map}\langle K\rangle\, V \to 1 + V \\
&\vdash^K_1 \mathsf{Merge}\langle K : *\rangle \ \ = \forall V.\, \mathsf{Bin}\, V \to \mathsf{Bin}\, (\mathsf{Map}\langle K\rangle\, V)
\end{aligned}$$

Lemma 15 (Soundness of $\Gamma \vdash^X_n B$) *Let $\Gamma \vdash F : \kappa \xrightarrow{+} \kappa$ for $\kappa = \vec{p\kappa} \to *$ and $p_j^{-1}\Gamma \vdash G_j : \kappa_j$ for $1 \leq j \leq |\vec{\kappa}|$. Let $\imath \notin \mathsf{dom}(\Gamma)$ and $A = \lambda\imath.[\mu^\imath F\vec{G}/X]B$. If $\Gamma \vdash^X_n B$ then $\Gamma \vdash A$ fix^μ_n-adm.*

Proof. By induction on $\Gamma \vdash^X_n B$.

- *Case* (1)

$$\frac{\Gamma, X\!:\!+* \vdash B : *}{\Gamma \vdash^X_{-\infty} B}\ p \leq +$$

Since $\lambda\imath.\, \mu^{\imath} F\, \vec{G} : \mathsf{ord} \xrightarrow{+} *$ and X appears only positively in B, we have $\Gamma \vdash A : \mathsf{ord} \xrightarrow{+} *$.

- *Case* (3)

$$\frac{\Gamma \vdash^X_n B}{\Gamma \vdash^X_0 X \to B}$$

By induction hypothesis, $\Gamma \vdash A\ \mathsf{fix}^{\mu}_n\text{-adm}$. Thus, $\Gamma \vdash \lambda\imath.\, \mu^{\imath} F\, \vec{G} \to A\, \imath\ \mathsf{fix}^{\mu}_0\text{-adm}$.

- *Case* (3)

$$\frac{p_i^{-1}\Gamma \vdash H_i : \kappa_i \text{ (for all } i = 1..|\vec{\kappa}|) \qquad \Gamma \vdash^X_n B}{\Gamma \vdash^X_0 \mathsf{Type}\langle X\rangle \vec{H} \to B}\ \mathsf{TYPE}\langle * \rangle = \vec{p}\vec{\kappa} \to *$$

Let $A := [\mu^{\imath} F\, \vec{G}/X]B$. By induction hypothesis, $\Gamma \vdash \lambda\imath.\, A\ \mathsf{fix}^{\mu}_n\text{-adm}$. Since X not free in $\vec{H}$, we have $[\mu^{\imath} F\, \vec{G}/X](\mathsf{Type}\langle X\rangle \vec{H} \to B) = \mathsf{Type}\langle \mu^{\imath} F\, \vec{G}\rangle\, \vec{H} \to A$. By definition of type-indexed types $\mathsf{Type}\langle \mu^{\imath} F\, \vec{G}\rangle = \mu^{\imath} F'\, \vec{G}'$ for some $\imath$-free $F', \vec{G}'$, hence, $\Gamma \vdash \lambda\imath.\, \mu^{\imath} F'\, \vec{G}'\, \vec{H} \to A\ \mathsf{fix}^{\mu}_0\text{-adm}$.

The remaining two cases for $\Gamma \vdash^X_n B$ follow by induction hypothesis and the matching rule for $\mathsf{fix}^{\mu}_n\text{-adm}$. □

The previous lemma implies that if $\vdash^X_n \mathsf{Poly}\langle X : *\rangle$ then $\mathsf{Poly}\langle \mu^{\imath}_* F : *\rangle\ \mathsf{fix}^{\mu}_n\text{-adm}$. The next lemma extends this result to higher kinds, such that we can treat programs indexed by nested data types.

Lemma 16 *Let $\kappa = p_1\kappa_1 \to \cdots \to p_m\kappa_m \to *$. If $\vdash^X_n \mathsf{Poly}\langle X : *\rangle$ for some $n \geq 0$ and $\Gamma \vdash F : \kappa \xrightarrow{+} \kappa$ then $\Gamma \vdash \mathsf{Poly}\langle \mu^{\imath}_{\kappa} F : \kappa\rangle\ \mathsf{fix}^{\mu}_{m+n}\text{-adm}$.*

Proof. From the previous lemma, by induction on m. □

Now we are able to show that all instances $\mathsf{poly}_{n,\phi}\langle F\rangle$ of a generic program are well-typed, hence, terminating, if their type $\mathsf{Poly}\langle F\rangle$ is of suitable shape.

Theorem 17 (Termination of generic programs) *Let $\Gamma \vdash \mathsf{poly}\langle C\rangle : \mathsf{Poly}\langle C : \kappa\rangle$ for all constants $(C : \kappa) \in \Sigma$ and $\vdash^X_n \mathsf{Poly}\langle X : *\rangle$ for some $n \geq 0$. Let $\Delta := (X_i : p_i\kappa_i)_{i=1..l}$ and $\Delta' := (X_i : p_i\kappa_i, x_i : \mathsf{Poly}\langle X_i : \kappa_i\rangle)_{i=1..l}$. If $\Delta \vdash F : \kappa$ and $\phi(X_i) = x_i$ for $i = 1..l$, then $\Gamma, \Delta' \vdash \mathsf{poly}_{n,\phi}\langle F\rangle : \mathsf{Poly}\langle F : \kappa\rangle$.*

Proof. By induction on $\Delta \vdash F : \kappa$. The interesting case is $\Delta \vdash \mu_{\kappa} : \mathsf{ord} \xrightarrow{+} (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa$ with $\kappa = p'_1\kappa'_1 \to \ldots p'_m\kappa'_m \to *$.

$$\begin{aligned} \mathsf{poly}_{n,\phi}\langle \mu_{\kappa}\rangle &= \lambda\_.\, \mathsf{fix}^{\mu}_{m+n} = \lambda\_\lambda s.\ \mathsf{fix}^{\mu}_{m+n}[s] \\ \mathsf{Poly}\langle \mu_{\kappa}\rangle &= \forall\imath.\, 1 \to \forall F.\, \mathsf{Poly}\langle F : \kappa \xrightarrow{+} \kappa\rangle \to \mathsf{Poly}\langle \mu^{\imath}_{\kappa} F\rangle \end{aligned}$$

Let $A := \lambda\imath.\, \mathsf{Poly}\langle\mu^{\imath}_{\kappa} F\rangle$. By Lemma 16, $\Delta \vdash A\ \mathsf{fix}^{\mu}_{m+n}\text{-}\mathsf{adm}$. By Lemma 11, $\Delta, \imath : \circ\mathsf{ord}, \_ : 1, F : \circ(\kappa \xrightarrow{+} \kappa), s : \mathsf{Poly}\langle F\rangle \vdash s : A\,\imath \to A\,(\imath + 1)$, hence, $\Delta, \_ : 1, F : \circ(\kappa \xrightarrow{+} \kappa), s : \mathsf{Poly}\langle F\rangle \vdash s : \forall\imath.\, A\,\imath \to A\,(\imath + 1)$. By Lemma 4, $\Delta, \imath : \circ\mathsf{ord}, \_ : 1, F : \circ(\kappa \xrightarrow{+} \kappa), s : \mathsf{Poly}\langle F\rangle \vdash \mathsf{fix}^{\mu}_{m+n}[s] : A\,\imath$. Thus, by abstraction, generalization, and weakening, we conclude $\Gamma, \Delta' \vdash \mathsf{poly}_{n,\phi}\langle\mu_\kappa\rangle : \mathsf{Poly}\langle\mu_\kappa\rangle$. $\square$

5 Conclusions and Related Work

We have seen a polymorphic λ-calculus with sized higher-order data types, $\mathsf{F}^{\widehat{}}_{\omega}$, in which all programs are terminating. This calculus is strong enough to certify termination of arbitrary instances of generic programs, provided the generic programs themselves do not use unrestricted recursion. A systematic method to certify termination using the framework of sized polytypic programming has been presented. The approach of type-based termination we have seen can handle convoluted recursion patterns that go far beyond schemes of iteration and primitive recursion stemming from the initial algebra semantics of data types. The recursion patterns of many examples for generic programming [21,22] can be treated in $\mathsf{F}^{\widehat{}}_{\omega}$, and I am still looking for sensible examples that exceed the capabilities of $\mathsf{F}^{\widehat{}}_{\omega}$. It seems promising to pursue this approach further.

In this article, we have not addressed the problem of type-checking sized types. However, some solutions exist in the literature: Pareto [32], Barthe, Grégoire, and Pastawski [9], and Blanqui [12] have given constraint-based inference algorithms for sized types.

System $\mathsf{F}^{\widehat{}}_{\omega}$ is strongly normalizing [1], as is its non-polymorphic predecessor $\lambda^{\widehat{}}$ [8]. More suitable for functional programming seems an interpretation of types as sets of closed values or finite observations—this, however, is future work. Hughes, Pareto, and Sabry [24] have presented a similar calculus, with ML-polymorphism, and given it a domain-theoretic semantics. What is not so pretty in this approach is that one constructs the semantics using undefinedness ($\perp$), but then later shows that each well-typed program is totally defined. One wonders why one has to speak about $\perp$ in the first place.

Related Work on Termination. The research on *size-change termination* (SCT), which is led by Neil Jones, has received much attention. Recently, Sereni and Jones have extended this method to higher-order functions [36]. Is SCT able to check termination of the generic programs presented in this work? No, because SCT analyses only the *untyped* program, and without typing information termination of, e. g., `mergeList` cannot be established, as explained in the introduction (`mergeList` diverges on ill-typed arguments). Neither can the methods developed for higher-order term rewriting systems, as for instance bundled in the tool AProve [15], be applied to the generic program, since they disregard typing. (I conducted my experiments in Fall 2005.)

Related Work on Generic Programming. We have considered generic programming in the style of *Generic Haskell* which has been formulated by Hinze, Jeuring, and Löh [19–23]. Another philosophy of generic programming is rooted in the initial algebra semantics for data types (see the introductory text by Backhouse, Jansson, Jeuring, and Meertens [7]). Jansson and Jeuring [26] present *PolyP*, a polytypic extension for Haskell which gives more control in defining polytypic functions, for instance, "recursion" is a type constructor one can treat in a clause of the polytypic program, whereas in *Generic Haskell* and our extension to sized types, recursion on types is always mapped to a recursive program.

Pfeifer and Rueß [33] study polytypic definitions in dependent type theory where all expressions are required to terminate. Termination is achieved by limiting recursion to the elimination combinators for inductive types, which correspond to the scheme of primitive recursion or *paramorphisms*. This excludes many interesting generic programs we can treat, like merging of tries, that do not fit into this scheme. Benke, Dybjer, and Jansson [10] extend the approach of Pfeifer and Rueß to generic definitions over inductive families. They also restrict recursion to iteration and primitive recursion. Altenkirch and McBride [5] pursue a similar direction; they show that generic programming is dependently typed programming with tailored type universes. They construct a generic fold for members of the universe of Haskell types, which allows to define generic *iterative* functions (catamorphisms).

Norell and Jansson [29] exploit the type class mechanism to enable polytypic programming in Haskell without language extensions. They also present an approach to generic programming using template Haskell [30]. Finally, Norell [28] describes an encoding of generic programs in dependent type theory. None of these works considers the problem of termination of the generated programs.

Generic programming within an intermediate language of a typed compiler has been studied under the names *intensional polymorphism* and *intensional type analysis* by Harper and Morrisett [17] and Crary, Weirich, and Morrisett [14]. The gist of this approach is to have a *type case* construct on the level of programs, in later developments even also on the level of types. This way, certain compiler optimizations such as untagging and unboxing can be performed in a type-safe way. Crary and Weirich [13] even enrich the kind language by inductive kinds and the constructor language by primitive recursion. Saha, Trifonov, and Shao [35] consider intensional analysis of polymorphism. To this end, they introduce polymorphic kinds. For our purposes, this would be counterproductive since a language with two impredicative universes on top of each other is non-normalizing (Girard's paradox).

Acknowledgments. The idea to apply type-based termination to generic programming was born in discussions with Ralph Matthes. Thanks to the anonymous referees for helpful comments which helped to improve the presentation.

Research supported by the coordination action *TYPES* (510996) and thematic network *Applied Semantics II* (IST-2001-38957) of the European Union and the project *Cover* of the Swedish Foundation of Strategic Research (SSF).

A System $\mathsf{F}^{\widehat{}}_{\omega}$

This section gives a short overview over system $\mathsf{F}^{\widehat{}}_{\omega}$ and lists typing, subtyping, and kinding rules. In comparison to the author's thesis [1], the version of $\mathsf{F}^{\widehat{}}_{\omega}$ presented here has no coinductive types, but only inductive types.

A.1 Kinds and Constructors

Polarities. To track the monotonicity of type constructors, we introduce polarities p, which range over the three values $+$ (covariant), $-$ (contravariant), and $\circ$ (mixed-variant). They are partially ordered; the order is given by $\circ \leq p$ and $p \leq p$.

Kinds κ are the "types" of type constructors. In $\mathsf{F}^{\widehat{}}_{\omega}$, size expressions belong to the type language and have kind ord.

$\kappa ::= *$	types
$\mid \mathsf{ord}$	ordinals
$\mid p\kappa_1 \to \kappa_2$	co-/contra-/mixed-variant constructor transformers

Notation: $\kappa \xrightarrow{p} \kappa'$ for $p\kappa \to \kappa'$. Pure kinds κ_* are kinds that do not mention ord.

Constructors are λ-terms over some constants C.

$$a, b, A, B, F, G ::= C \mid X \mid \lambda X F \mid F\, G$$

The constructor constants C are taken from a fixed signature Σ_0 which contains at least the following constants together with their kinding.

1	$: *$	unit type
$+$	$: * \xrightarrow{+} * \xrightarrow{+} *$	disjoint sum
$\times$	$: * \xrightarrow{+} * \xrightarrow{+} *$	cartesian product
$\to$	$: * \xrightarrow{-} * \xrightarrow{+} *$	function space
$\forall_\kappa$	$: (\kappa \xrightarrow{\circ} *) \xrightarrow{+} *$	quantification
μ_{κ_*}	$: \mathsf{ord} \xrightarrow{+} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_*$	inductive constructors
s	$: \mathsf{ord} \xrightarrow{+} \mathsf{ord}$	successor of ordinal
∞	$: \mathsf{ord}$	infinity ordinal

We sometimes write the size index superscript, e. g., $\mu^{\imath}$ instead of $\mu\,\imath$.

Polarized contexts. A kinding context usually records the kinds of the free variables of a type constructor; a polarized context additionally records variance information for each variable to distinguish whether this free variable occurs only positively ($p = +$), only negatively ($p = -$), or arbitrarily ($p = \circ$) in the type constructor under consideration.

$$\Delta ::= \diamond \mid \Delta,\ X\!:\!p\kappa$$

Negation of a polarity $-p$ is given by the three equations $-(+) = -$, $-(-) = +$ and $-(\circ) = \circ$. We define inverse application $p^{-1}\Delta$ of a polarity p to a polarized context Δ.

$$\begin{array}{ll@{\qquad}ll}
+^{-1}\Delta & = \Delta & \circ^{-1}(\diamond) & = \diamond \\
 & & \circ^{-1}(\Delta, X\!:\!\circ\kappa) & = \circ^{-1}\Delta, X\!:\!\circ\kappa \\
-^{-1}(\diamond) & = \diamond & \circ^{-1}(\Delta, X\!:\!+\kappa) & = \circ^{-1}\Delta \\
-^{-1}(\Delta, X\!:\!p\kappa) & = -^{-1}\Delta, X\!:\!(-p)\kappa & \circ^{-1}(\Delta, X\!:\!-\kappa) & = \circ^{-1}\Delta
\end{array}$$

Inverse application is used in the kinding judgement.

Kinding. $\Delta \vdash F : \kappa$

$$\frac{C\!:\!\kappa \in \Sigma_0}{\Delta \vdash C : \kappa} \qquad \frac{X\!:\!p\kappa \in \Delta \qquad p \le +}{\Delta \vdash X : \kappa}$$

$$\frac{\Delta, X\!:\!p\kappa \vdash F : \kappa'}{\Delta \vdash \lambda X F : p\kappa \to \kappa'} \qquad \frac{\Delta \vdash F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

Constructor equality. Constructors are identified modulo β and η. Furthermore, there are two axioms for sizes and inductive types.

Computation axioms.

$$\frac{\Delta, X\!:\!p\kappa \vdash F : \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash (\lambda X F)\,G = [G/X]F : \kappa'} \qquad \frac{\Delta \vdash F : p\kappa \to \kappa'}{\Delta \vdash (\lambda X.\,F\,X) = F : p\kappa \to \kappa'}$$

$$\frac{}{\Delta \vdash \mathsf{s}\,\infty = \infty : \mathsf{ord}} \qquad \frac{\Delta \vdash a : \mathsf{ord}}{\Delta \vdash \mu_\kappa^{\mathsf{s}\,a} = \lambda F.\,F\,(\mu_\kappa^{a}\,F) : (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa}$$

Congruences.

$$\frac{X\!:\!p\kappa \in \Delta \qquad p \leq +}{\Delta \vdash X = X : \kappa} \qquad \frac{\Delta, X\!:\!p\kappa \vdash F = F' : \kappa'}{\Delta \vdash \lambda X F = \lambda X F' : p\kappa \to \kappa'}$$

$$\frac{C\!:\!\kappa \in \Sigma_0}{\Delta \vdash C = C : \kappa} \qquad \frac{\Delta \vdash F = F' : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G = G' : \kappa}{\Delta \vdash F\,G = F'\,G' : \kappa'}$$

Symmetry and transitivity.

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

Higher-Order Subtyping. Subtyping captures the natural inclusion order $\mu^a F \leq \mu^{\mathsf{s}\,a} F$ between sized inductive types.

Reflexivity, transitivity, antisymmetry.

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

$$\frac{\Delta \vdash F \leq F' : \kappa \qquad \Delta \vdash F' \leq F : \kappa}{\Delta \vdash F = F' : \kappa}$$

Abstraction and application.

$$\frac{\Delta, X\!:\!p\kappa \vdash F \leq F' : \kappa'}{\Delta \vdash \lambda X F \leq \lambda X F' : p\kappa \to \kappa'} \qquad \frac{\Delta \vdash F \leq F' : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G \leq F'\,G : \kappa'}$$

$$\frac{\Delta \vdash F : +\kappa \to \kappa' \quad \Delta \vdash G \leq G' : \kappa}{\Delta \vdash F\,G \leq F\,G' : \kappa'} \qquad \frac{\Delta \vdash F : -\kappa \to \kappa' \quad -\Delta \vdash G' \leq G : \kappa}{\Delta \vdash F\,G \leq F\,G' : \kappa'}$$

Successor and infinity.

$$\frac{\Delta \vdash a : \mathsf{ord}}{\Delta \vdash a \leq \mathsf{s}\,a : \mathsf{ord}} \qquad \frac{\Delta \vdash a : \mathsf{ord}}{\Delta \vdash a \leq \infty : \mathsf{ord}}$$

A.2 Terms and Typing

Terms.

$$\begin{array}{lll} \mathsf{Tm} & \ni r, s, t ::= & c \mid x \mid \lambda x t \mid r\,s \mid \mathsf{fix}^{\mu} \\ \mathsf{Const} & \ni c ::= & () \mid \mathsf{pair} \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{case} \end{array}$$

The signature Σ_0 contains the types for the constants in Const.

Typing contexts.

$$\Gamma ::= \diamond \mid \Gamma, x\!:\!A \mid \Gamma, X\!:\!p\kappa$$

Wellformed contexts.

$$\frac{}{\diamond\ \mathsf{cxt}} \qquad \frac{\Gamma\ \mathsf{cxt}}{\Gamma, X\!:\!\circ\kappa\ \mathsf{cxt}} \qquad \frac{\Gamma\ \mathsf{cxt} \qquad \Gamma \vdash A : *}{\Gamma, x\!:\!A\ \mathsf{cxt}}$$

Typing. $\Gamma \vdash t : A$
Lambda-calculus.

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

Quantification.

$$\frac{\Gamma, X\!:\!\circ\kappa \vdash t : F\,X}{\Gamma \vdash t : \forall_\kappa F}\ X \notin \mathsf{FV}(F) \qquad \frac{\Gamma \vdash t\ :\ \forall_\kappa F \qquad \Gamma \vdash G : \kappa}{\Gamma \vdash t : F\,G}$$

Subsumption.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leq B : *}{\Gamma \vdash t : B}$$

Recursion and further constants.

$$\frac{\Gamma \vdash A\ \mathsf{fix}^\mu\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}^\mu : (\forall\imath\!:\!\mathsf{ord}.\, A\,\imath \to A\,(\imath+1)) \to A\,a} \qquad \frac{(c\!:\!A) \in \Sigma_0}{\Gamma \vdash c : A}$$

$\Gamma \vdash A\ \mathsf{fix}^\mu\text{-}\mathsf{adm}$ means that $\Gamma \vdash A = \lambda\imath.\forall\vec{X}^0.\mu^\imath F\,\vec{G} \to \forall\vec{X}^1.B_1 \cdots \to \forall\vec{X}^m.B_m \to C : \mathsf{ord} \xrightarrow{\circ} *$ where the $\imath$ does not occur in F and $\vec{G}$ and only positively in C. Each B_j either is of the form $\mu^\imath F'\,\vec{G}'$ where $\imath$ does not occur in $F', \vec{G}'$ or $\imath$ occurs only negatively in B_j.

A.3 Values and Reduction

Values are weak-head normal forms.

$v ::= \lambda x t \mid \mathsf{pair}\, t_1\, t_2 \mid \mathsf{inl}\, t \mid \mathsf{inr}\, t$	proper values
$\mid \mathsf{fix}^\mu \mid \mathsf{fix}^\mu\, s \mid \mathsf{pair} \mid \mathsf{pair}\, t \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{case}$	under-applied functions

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

$$\begin{array}{lll}
(\lambda x t)\, s & \longrightarrow & [s/x]t \\
\mathsf{fix}^{\mu}_{n}\, s\, v & \longrightarrow & s\, (\mathsf{fix}^{\mu}_{n}\, s)\, v \\
\mathsf{fst}\, (r, s) & \longrightarrow & r \\
\mathsf{snd}\, (r, s) & \longrightarrow & s \\
\mathsf{case}\, (\mathsf{inl}\, r) & \longrightarrow & \lambda x \lambda y.\, x\, r \\
\mathsf{case}\, (\mathsf{inr}\, r) & \longrightarrow & \lambda x \lambda y.\, y\, r
\end{array}$$

Well-typed terms are strongly normalizing under this reduction.

References

[1] A. Abel, A polymorphic lambda-calculus with sized higher-order types, Ph.D. thesis, Ludwig-Maximilians-Universität München (2006).

[2] A. Abel, Semi-continuous sized types and termination, in: Z. Ésik (ed.), Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 21-24, 2006, Proceedings, vol. 4207 of Lecture Notes in Computer Science, Springer-Verlag, 2006.

[3] A. Abel, Towards generic programming with sized types, in: T. Uustalu (ed.), Mathematics of Program Construction: 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006. Proceedings, vol. 4014 of Lecture Notes in Computer Science, Springer-Verlag, 2006.

[4] A. Abel, R. Matthes, T. Uustalu, Iteration schemes for higher-order and nested datatypes, Theoretical Computer Science 333 (1–2) (2005) 3–66.

[5] T. Altenkirch, C. McBride, Generic programming within dependently typed programming., in: J. Gibbons, J. Jeuring (eds.), Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11-12, 2002, Dagstuhl, Germany, vol. 243 of IFIP Conference Proceedings, Kluwer, 2003.

[6] T. Altenkirch, B. Reus, Monadic presentations of lambda terms using generalized inductive types, in: J. Flum, M. Rodríguez-Artalejo (eds.), Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings, vol. 1683 of Lecture Notes in Computer Science, Springer-Verlag, 1999.

[7] R. Backhouse, P. Jansson, J. Jeuring, L. Meertens, Generic programming — an introduction, in: S. D. Swierstra, P. R. Henriques, J. N. Oliveira (eds.), Revised Lectures from the Third International School on Advanced Functional Programming, AFP '98, vol. 1608 of Lecture Notes in Computer Science, Springer-Verlag, 1999.

[8] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, T. Uustalu, Type-based termination of recursive definitions, Mathematical Structures in Computer Science 14 (1) (2004) 1–45.

[9] G. Barthe, B. Grégoire, F. Pastawski, Practical inference for type-based termination in a polymorphic setting, in: P. Urzyczyn (ed.), Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan, vol. 3461 of Lecture Notes in Computer Science, Springer-Verlag, 2005.

[10] M. Benke, P. Dybjer, P. Jansson, Universes for generic programs and proofs in dependent type theory, Nordic Journal of Computing 10 (4) (2003) 265–289.

[11] R. Bird, L. Meertens, Nested datatypes, in: J. Jeuring (ed.), Mathematics of Program Construction, MPC'98, Proceedings, vol. 1422 of Lecture Notes in Computer Science, Springer-Verlag, 1998.

[12] F. Blanqui, Decidability of type-checking in the Calculus of Algebraic Constructions with size annotations., in: C.-H. L. Ong (ed.), Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings, vol. 3634 of Lecture Notes in Computer Science, Springer-Verlag, 2005.

[13] K. Crary, S. Weirich, Flexible type analysis, in: Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, vol. 34 of SIGPLAN Notices, ACM Press, 1999.

[14] K. Crary, S. Weirich, J. G. Morrisett, Intensional polymorphism in type-erasure semantics, in: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998, vol. 34 of SIGPLAN Notices, ACM Press, 1999.

[15] J. Giesl, R. Thiemann, P. Schneider-Kamp, S. Falke, Automated termination proofs with AProVE, in: V. van Oostrom (ed.), Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3 – 5, 2004, Proceedings, vol. 3091 of Lecture Notes in Computer Science, Springer-Verlag, 2004.

[16] E. Giménez, Structural recursive definitions in type theory, in: K. G. Larsen, S. Skyum, G. Winskel (eds.), Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings, vol. 1443 of Lecture Notes in Computer Science, Springer-Verlag, 1998.

[17] R. Harper, J. G. Morrisett, Compiling polymorphism using intensional type analysis, in: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95, San Francisco, California, January 23-25, 1995, ACM Press, 1995.

[18] R. Hinze, Generalizing generalized tries, Journal of Functional Programming 10 (4) (2000) 327–351.

[19] R. Hinze, A new approach to generic functional programming, in: Proceedings of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages, POPL 2000, Boston, Massachusetts, USA, January 19-21, 2000, ACM Press, 2000.

[20] R. Hinze, Polytypic values possess polykinded types, MPC Special Issue, Sci. Comput. Programming 43 (2002) 129–159.

[21] R. Hinze, J. Jeuring, Generic Haskell: Applications, in: R. C. Backhouse, J. Gibbons (eds.), Generic Programming - Advanced Lectures, vol. 2793 of Lecture Notes in Computer Science, Springer-Verlag, 2003.

[22] R. Hinze, J. Jeuring, Generic Haskell: Practice and Theory, in: R. C. Backhouse, J. Gibbons (eds.), Generic Programming - Advanced Lectures, vol. 2793 of Lecture Notes in Computer Science, Springer-Verlag, 2003.

[23] R. Hinze, J. Jeuring, A. Löh, Type-indexed data types, MPC Special Issue, Sci. Comput. Programming 51 (2004) 117–151.

[24] J. Hughes, L. Pareto, A. Sabry, Proving the correctness of reactive systems using sized types, in: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96, 1996.

[25] INRIA, The Coq Proof Assistant, Version 8.1, http://coq.inria.fr/ (2007).

[26] P. Jansson, J. Jeuring, PolyP—a polytypic programming extension, in: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, Paris, France, ACM Press, 1997.

[27] N. 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., IEEE Computer Society Press, 1987.

[28] U. Norell, Functional generic programming and type theory, Master's thesis, Computing Science, Chalmers University of Technology, available from `http://www.cs.chalmers.se/~ulfn` (2002).

[29] U. Norell, P. Jansson, Polytypic programming in Haskell, in: P. W. Trinder, G. Michaelson, R. Pena (eds.), Implementation of Functional Languages (IFL'03), Lecture Notes in Computer Science, 2004.

[30] U. Norell, P. Jansson, Prototyping generic programming in Template Haskell, in: D. Kozen, C. Shankland (eds.), Proceedings of the Seventh International Conference on Mathematics of Program Construction, MPC '04, vol. 3125 of Lecture Notes in Computer Science, Springer-Verlag, 2004.

[31] C. 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, 1999.

[32] L. Pareto, Types for crash prevention, Ph.D. thesis, Chalmers University of Technology (2000).

[33] H. Pfeifer, H. Rueß, Polytypic proof construction, in: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, , L. Théry (eds.), Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99, Nice, France, vol. 1690 of Lecture Notes in Computer Science, Springer-Verlag, 1999.

[34] B. C. Pierce, Types and Programming Languages, MIT Press, 2002.

[35] B. Saha, V. Trifonov, Z. Shao, Intensional analysis of quantified types, ACM Transactions on Programming Languages and Systems 25 (2) (2003) 159–209.

[36] D. Sereni, N. D. Jones, Termination analysis of higher-order functional programs, in: K. Yi (ed.), Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings, vol. 3780 of Lecture Notes in Computer Science, Springer-Verlag, 2005.