Towards Generic Programming with Sized Types

Andreas Abel[⋆]

Institut für Informatik
Ludwigs-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, GERMANY
abel@informatik.uni-muenchen.de

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. This problem is tackled by an adaption of type-based termination to generic programming, and a framework for sized polytypic programming is described.

1 Introduction

In the last decade, *polytypic* or *generic* programming has been explored for functional programming languages [34, 7, 25, 28–30]. With polytypic programming, many 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 have often 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 or initial algebras alone. It is the purpose of this article to outline a systematic solution to the termination problem of many generic programs.

As an example, we take Hinze's [24] 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))
```

[⋆] 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).

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 demonstrates 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 [31], and independently by Giménez [20] who advanced the pioneering work of Mendler [37]. Since then, type-based termination has been considered by several authors [1, 2, 8, 9, 14, 15, 18].

In this work, we show that type-based termination can be successfully applied to generic programs. To this end, we have extended the approach to higher-order data types, arriving at System $\mathsf{F}_{\widehat{\omega}}$, which is the object of the author’s thesis [3]. 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. More related work and directions for future research are discussed in Sect. 4.

1.1 Preliminaries

We assume that the reader is firm in the higher-order polymorphic lambda-calculus, System F^{ω} (see Pierce’s text book [46]). Additionally, some familiarity with generic programming would be helpful [29].

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, $\mathsf{fst}\,(r, s) \longrightarrow r$. Sometimes it is convenient to introduce abbreviations for derived data constructors. For instance:

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

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. Similarly, we use a non-recursive $\mathsf{let}\ p = r\ \mathsf{in}\ t$.

2 Sized Types in a Nutshell

We use sized types for type-based termination checking, as described by Hughes, Pareto, and Sabry [31, 44] 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 $\mathsf{F}_\omega^{\widehat{}}$ [3].

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 [37]. 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 $p\kappa \to \kappa'$ 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.

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

$$\mu^a F \le \mu^{a+1} F \le \mu^{a+2} F \le \cdots \le \mu^\infty F.$$

[1] More details can be found in the forthcoming thesis of the author [3, Sect. 3.3.3].

Some examples for sized inductive types are:

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

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, 11–13, 36, 22, 24, 26, 41–43]. 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:

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

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. 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:

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

[2] The constructor underlying Bush fails a purely syntactical covariance test, like the test for *strict positivity* in Coq [32].

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

$$\frac{\imath\!:\!\mathsf{ord},\ f : A\,\imath \vdash t : A\,(\imath+1)}{\mathsf{fix}\,(\lambda f.t) : \forall\imath.\ A\,\imath}$$

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 descent on serveral arguments, and also polymorphic recursion. Hence, we consider types of the shape

$$\forall X_1 \dots \forall X_k.\ \mu^\imath F \to B_1 \to \dots \to B_m \to C,$$

where $\imath$ does not occur in F, index $\imath$ occurs only positively in C, and each of the B_i is either $\imath$-free or of the shape $\mu^\imath F_i$ with F_i $\imath$-free. More valid shapes for the type $A\,\imath$ are described by Hughes, Pareto, and Sabry [31], in Pareto's thesis [44], my thesis [3] and previous work of mine [1].

To obtain a strongly normalizing system, unrolling of fixed-point 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:

$$\begin{array}{ll}
\mathsf{back}_n & := \lambda g \lambda t_1 \dots \lambda t_n \lambda r.\ g\, r\, t_1 \dots t_n \\
\mathsf{front}_n & := \lambda g \lambda r \lambda t_1 \dots \lambda t_n.\ g\, t_1 \dots t_n\, r \\
\mathsf{fix}^\mu_n & := \lambda s.\ \mathsf{back}_n\,(\mathsf{fix}^\mu\,(\lambda f.\,\mathsf{front}_n\,(s\,(\mathsf{back}_n\, f)))).
\end{array}$$

Example 2 (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 $\mathsf{F}^{\widehat{}}_\omega$ as follows:

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

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 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 [25] describes a framework for generic programming which is later extended by Hinze, Jeuring, and Löh [30] and implemented in *Generic Haskell* [29]. 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 [24].

Observe the following typographic conventions:

Capital	$\mathsf{Type}\langle A\rangle$	a type Type indexed by type A
UPPERCASE	$\mathsf{TYPE}\langle\kappa\rangle$	the kind TYPE of type Type indexed by kind κ of type A
lowercase	$\mathsf{poly}\langle A\rangle$	a polytypic program poly instantiated at type A
Capital	$\mathsf{Poly}\langle\kappa\rangle$	the polykinded type Poly of program poly instantiated at kind κ of type A

3.1 Type-indexed Types

In generic programming as proposed by Hinze, Jeuring, and Löh [30], 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. In general, a type-indexed type $\mathsf{Type}\langle A\rangle$ will obey the following laws:

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

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 $\mathsf{Type}\langle F\rangle$. (Actually, the term *constructor-indexed constructor* would be more appropriate, but we stick to the existing terminology.) The kind $\mathsf{TYPE}\langle\kappa\rangle$ depends on the kind κ of constructor F. The given equations for abstraction and application dictate the following laws for function kinds.

$$\mathsf{TYPE}\langle \kappa_1 \xrightarrow{p} \kappa_2\rangle = \mathsf{TYPE}\langle\kappa_1\rangle \xrightarrow{p} \mathsf{TYPE}\langle\kappa_2\rangle$$

The kind $\mathsf{TYPE}\langle *\rangle$ has to be chosen such that $\mathsf{Type}\langle C\rangle : \mathsf{TYPE}\langle \Sigma(C)\rangle$ for all basic type constructors $C \in \Sigma$. (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.

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

Remark 1. Note that the presence of polarities restricts the choices for $\mathsf{Type}\langle C\rangle$. However, if index types are constructed in a signature without polymorphism and function space, as it is usual in the generic programming community, all function kinds are covariant and we do not have to worry about polarities.

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

$$\mathsf{TYPE}\langle\mathsf{ord}\rangle = \mathsf{ord}$$

$$\begin{array}{ll}
\mathsf{Type}\langle\mathsf{s}\rangle & = \mathsf{s} \\
\mathsf{Type}\langle\infty\rangle & = \infty
\end{array}$$

Theorem 1 (Well-kindedness of type-indexed types). *Let Σ be a signature of constructor constants. If* $\mathsf{Type}\langle C\rangle : \mathsf{TYPE}\langle\kappa\rangle$ *for all* $(C\!:\!\kappa) \in \Sigma$*, and* $X_1\!:\!p_1\kappa_1, \dots, X_n : p_n\kappa_n \vdash F : \kappa$*, then* $X_1\!:\!p_1\mathsf{TYPE}\langle\kappa_1\rangle, \dots, X_n\!:\!p_n\mathsf{TYPE}\langle\kappa_n\rangle \vdash \mathsf{Type}\langle F\rangle : \mathsf{TYPE}\langle\kappa\rangle$.

Proof. By induction on the kinding derivation.

Example: finite maps via generalized tries. Hinze [24] 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 [30, page 139].

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

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 [24]. This way we avoid that certain empty tries have an infinite normal form (see [24, page 341]) which requires lazy evaluation. The constructor for finite maps over strings can now be computed as follows:

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

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.

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:

$$\begin{array}{ll}
\mathsf{poly}\langle C \rangle & = \textit{user-defined} \\
\mathsf{poly}\langle X \rangle & = x \\
\mathsf{poly}\langle \lambda X F : \kappa_1 \to \kappa_2 \rangle & = \lambda x.\, \mathsf{poly}\langle F \rangle \\
\mathsf{poly}\langle F\,G \rangle & = \mathsf{poly}\langle F \rangle\, \mathsf{poly}\langle G \rangle \\
\mathsf{poly}\langle \mu_{\kappa} \rangle & = \mathsf{fix}
\end{array}$$

(This definition is sensible if we consider all bound variables in F distinct and require $\mathsf{poly}\langle C\rangle$ to be a closed expression.)

Hinze [27] has observed that type-indexed values $\mathsf{poly}\langle F : \kappa\rangle$ have kind-indexed types $\mathsf{Poly}\langle F, \dots, F : \kappa\rangle : *$ with possibly several copies of F, obeying the following laws:

$$\begin{array}{ll}
\mathsf{Poly}\langle A_1, \dots, A_n : *\rangle & = \textit{user-defined} \\
\mathsf{Poly}\langle F_1, \dots, F_n : \kappa \xrightarrow{p} \kappa'\rangle & = \forall G_1 \mathbin{:} \kappa \dots \forall G_n \mathbin{:} \kappa. \\
& \quad \mathsf{Poly}\langle G_1, \dots, G_n : \kappa\rangle \to \mathsf{Poly}\langle F_1\, G_1, \dots, F_n\, G_n : \kappa'\rangle
\end{array}$$

For example, three copies of F are required for a generic definition of zipping functions [27, 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, resp.):

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

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.

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

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.

In $\mathsf{F}^{\widehat{}}_{\omega}$, there is a second base kind, ord. Since ordinals are only used to increase the static information about programs, not to carry out computations, the occurrence of kind ord in a kind which indexes a type should not alter this type. Thus, the following laws are sensible:

$$\begin{array}{ll}
\mathsf{Poly}\langle A \mathbin{:} *\rangle & = \textit{user-defined} \\
\mathsf{Poly}\langle F \mathbin{:} \mathsf{ord} \xrightarrow{p} \kappa\rangle & = \forall \imath \mathbin{:} \mathsf{ord}.\, \mathsf{Poly}\langle F\, \imath \mathbin{:} \kappa\rangle \\
\mathsf{Poly}\langle F \mathbin{:} \kappa_1 \xrightarrow{p} \kappa_2\rangle & = \forall G \mathbin{:} \kappa_1.\, \mathsf{Poly}\langle G \mathbin{:} \kappa_1\rangle \to \mathsf{Poly}\langle F\, G \mathbin{:} \kappa_2\rangle
\end{array}$$

Kinds suitable as indexes must fit into the grammar: $\kappa ::= * \mid \mathsf{ord} \xrightarrow{p} \kappa \mid \kappa_1 \xrightarrow{p} \kappa_2$. Size expressions appearing in the type A of a generic program $\mathsf{poly}\langle A\rangle$ should not influence the program. We only consider types A which are normalized and contain size expressions only as index to an inductive type. Then we can refine the generation laws for type-indexed programs as follows:

$$\begin{array}{lll}
\mathsf{poly}\langle C\rangle & = \textit{user-defined} & \\
\mathsf{poly}\langle X\rangle & = x & \\
\mathsf{poly}\langle \lambda\imath F : \mathsf{ord} \to \kappa\rangle & = \mathsf{poly}\langle F\rangle & \\
\mathsf{poly}\langle \lambda X F : \kappa_1 \to \kappa_2\rangle & = \lambda x.\, \mathsf{poly}\langle F\rangle & \text{where } \kappa_1 \neq \mathsf{ord} \\
\mathsf{poly}\langle F\, G\rangle & = \mathsf{poly}\langle F\rangle\, \mathsf{poly}\langle G\rangle & \\
\mathsf{poly}\langle \mu^a_\kappa\rangle & = \mathsf{fix}^\mu_n & \text{for some } n
\end{array}$$

In the last equation, 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 $n = 1$.

Example: 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. 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 $(\mathsf{inr}\, v \gg= f) \longrightarrow f\, v$.

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

$$\begin{array}{lll}
\mathsf{lookup}\langle 1\rangle & : & \forall V.\, 1 \to 1 + V \to 1 + V \\
\mathsf{lookup}\langle 1\rangle & := & \lambda k \lambda m.\, m
\end{array}$$

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

$$\begin{array}{lll}
\mathsf{lookup}\langle \times\rangle & : & \forall A : *.\, \mathsf{Lookup}\langle A\rangle \to \forall B : *.\, \mathsf{Lookup}\langle B\rangle \to \\
 & & \forall V.\, A \times B \to \mathsf{Map}\langle A\rangle\, (\mathsf{Map}\langle B\rangle\, V) \to 1 + V \\
\mathsf{lookup}\langle \times\rangle & := & \lambda la \lambda lb \lambda (a, b) \lambda tab.\, la\, a\, tab \gg= \lambda tb.\, lb\, b\, tb
\end{array}$$

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.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{lookup}\langle\mathsf{List}\rangle} \\
: & \mathsf{Lookup}\langle\mathsf{List}\rangle \\
: & \forall\imath\forall K\!:\!*.\,\mathsf{Lookup}\langle K\rangle \to \mathsf{Lookup}\langle\mathsf{List}^{\imath}\,K\rangle \\
: & \forall\imath\forall K\!:\!*.\,\mathsf{Lookup}\langle K\rangle \to \forall V.\,\mathsf{List}^{\imath}K \to \mathsf{Map}\langle\mathsf{List}^{\imath}K\rangle \to 1+V \\
: & \forall\imath\forall K\!:\!*.\,\mathsf{Lookup}\langle K\rangle \to \forall V.\,\mathsf{List}^{\imath}K \to (\mu^{\imath}\lambda Y.\,1+(1+V)\times Y) \to 1+V
\end{array}$$

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

Note that the type of $\mathsf{lookup}\langle\mathsf{List}\rangle$ 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. Welltypedness can be ensured on an abstract level:

$$\begin{array}{lll}
\mathit{lookup}_K & & : \mathsf{Lookup}\langle K\rangle \\
\mathit{lookup} & & : \mathsf{Lookup}\langle\mathsf{List}^{\imath}K\rangle \\
\mathsf{lookup}\langle\times\rangle\,\mathit{lookup}_K\,\mathit{lookup} & =: r & : \mathsf{Lookup}\langle K\times\mathsf{List}^{\imath}K\rangle \\
\mathsf{lookup}\langle+\rangle\,\mathsf{lookup}\langle 1\rangle\,r & =: s & : \mathsf{Lookup}\langle 1+K\times\mathsf{List}^{\imath}K\rangle \\
 & & : \mathsf{Lookup}\langle\mathsf{List}^{\imath+1}K\rangle \\
\mathsf{fix}_0^{\mu}\,\lambda\mathit{lookup}.\,s & & : \mathsf{Lookup}\langle\mathsf{List}^{\imath}K\rangle
\end{array}$$

Finally, the type $\mathsf{Lookup}\langle\mathsf{List}^{\imath}K\rangle$ is valid for recursion with fix_0^{μ}, according to criterion given in Sect. 2.

Trie merging. Hinze [24] 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.

First we 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 $1+V$.

$$\begin{array}{ll}
\mathsf{Bin} & :\ \ * \overset{\circ}{\to} * \\
\mathsf{Bin} & := \lambda V.\,V\to V\to V
\end{array}$$

$$\begin{array}{l}
\mathsf{comb} :\ \ \forall V.\,(V\to V\to V)\to(1+V\to 1+V\to 1+V) \\
\mathsf{comb} := \lambda c\lambda m_1\lambda m_2.\,\mathsf{match}\ (m_1,m_2)\ \mathsf{with} \\
\qquad\qquad (\mathsf{inl}(),\_) \qquad\quad \mapsto m_2 \\
\qquad\qquad (\_,\mathsf{inl}()) \qquad\quad \mapsto m_1 \\
\qquad\qquad (\mathsf{inr}\,v_1,\mathsf{inr}\,v_2) \mapsto \mathsf{inr}\,(c\,v_1\,v_2)
\end{array}$$

The following definitions determine a generic merging function.

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

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

The instance for list tries can be computed as follows:

$$\begin{array}{l}
\mathsf{merge}\langle \mathsf{List}\rangle \\
\quad : \mathsf{Merge}\langle \mathsf{List}\rangle \\
\quad : \forall\imath\forall K.\,\mathsf{Merge}\langle K\rangle \to \mathsf{Merge}\langle \mathsf{List}^{\imath} K\rangle \\
\quad : \forall\imath\forall K.\,(\forall V.\,\mathsf{Bin}\,V \to \mathsf{Bin}\,(\mathsf{Map}\langle K\rangle\,V)) \to \\
\qquad\quad \forall W.\,\mathsf{Bin}\,W \to \mathsf{Bin}\,(\mathsf{Map}\langle \mathsf{List}^{\imath} K\rangle\,W)
\end{array}$$

$$\begin{array}{l}
\mathsf{merge}\langle \mathsf{List}\rangle \\
\quad = \mathsf{merge}\langle \lambda\imath\lambda K.\,\mu^{\imath}\lambda X.\,1 + K \times X\rangle \\
\quad = \lambda merge_K.\ \mathsf{fix}^{\mu}_{1}\,\lambda merge.\,\mathsf{merge}\langle +\rangle\ \mathsf{merge}\langle 1\rangle\ (\mathsf{merge}\langle\times\rangle\ merge_K\ merge) \\
\quad = \lambda merge_K.\ \mathsf{fix}^{\mu}_{1}\,\lambda merge \lambda c.\,\mathsf{comb} \\
\qquad\quad \lambda(mv_1, t_1)\lambda(mv_2, t_2).\,(\mathsf{comb}\ c\ mv_1\ mv_2,\ merge_K\ (merge\,c)\,t_1 t_2) \\
\quad [= \lambda merge_K \lambda c.\ \mathsf{fix}^{\mu}_{0}\,\lambda merge.\,\mathsf{comb} \\
\qquad\quad \lambda(mv_1, t_1)\lambda(mv_2, t_2).\,(\mathsf{comb}\ c\ mv_1\ mv_2,\ merge_K\ merge\,t_1\,t_2)]
\end{array}$$

In the last step, 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:

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

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.

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.

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

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

$$\begin{array}{l}
\mathsf{merge}\langle\mathsf{Bush}\rangle \\
\;= \mathsf{merge}\langle\lambda\imath.\, \mu^\imath\, \lambda X \lambda K.\, 1 + K \times X\,(X\,K)\rangle \\
\;= \mathsf{fix}^\mu_2\, \lambda merge \lambda merge_K . \\
\qquad \mathsf{merge}\langle+\rangle\; \mathsf{merge}\langle 1\rangle\; (\mathsf{merge}\langle\times\rangle\; merge_K\; (merge\,(merge\; merge_K))) \\
\;= \mathsf{fix}^\mu_2\, \lambda merge \lambda merge_K \\
\qquad \lambda c.\, \mathsf{comb}\; \lambda(mv_1, t_1)\lambda(mv_2, t_2). \\
\qquad\quad (\mathsf{comb}\; c\; mv_1\; mv_2,\; merge_K\,(merge\,(merge\; merge_K)\,c)\,t_1\,t_2)
\end{array}$$

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 $(merge\; merge_K)$, which involves the recursive instance again! All these complications are coolly handled by type-based termination!

4 Conclusions and Related Work

We have seen a polymorphic λ-calculus with sized higher-order data types, $\mathsf{F}_\omega^{\frown}$, 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 sketched. 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 [28, 29] can be treated in $\mathsf{F}_\omega^{\frown}$, and I am still looking for sensible examples that exceed the capabilities of $\mathsf{F}_\omega^{\frown}$. 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 [44], Barthe, Gregorie, and Pastawski [9], and Blanqui [15] have given constraint-based inference algorithms for sized types.

System $\mathsf{F}_\omega^{\frown}$ is strongly normalizing [3], as is its non-polymorphic predecessor $\lambda^{\frown}$ [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 [31] have presented a similar calculus, with ML-polymorphism, and given it a domain-theoretic semantics. In my view, this semantics has the flaw that it introduces undefinedness ($\perp$), only to show later that no well-typed program is undefined. I would like to find a tailored semantics that can handle infinite objects (coinductive types) but speaks of neither strong normalization nor undefinedness.

Related Work on Termination. The research on *size-change termination* (SCT), which is lead by Neil Jones, has received much attention. Recently, Sereni and Jones have extended this method to higher-order functions [48]. 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 [19], be applied to the generic program, since they disregard typing. (According experiments were carried out by the author 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 [23, 25, 27–30]. Another philosophy of generic programming is rooted in in the initial algebra semantics for data types (see the introductory text by Backhouse, Jansson, Jeuring, and Meertens [7]). Jansson and Jeuring [33–35] 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 [45] 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 *paramorphism*. 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 definition 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 type 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 [39] 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 [40]. Finally, Norell [38] 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 intermediated language of a typed compiler has been studied under the names *intensional polymorphism* and *intensional type analysis* by Harper and Morrisett [21] and Crary, Weirich, and Morrisett [17]. 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 [16] even enrich the kind language by inductive kinds and the constructor language by primitive recursion. Saha, Trifonov, and Shao [47] consider intensional analysis of polymorphism. To this end, they introduce polymorphic kinds. For our purposes, this would be contraproductive 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.

References

1. Abel, A.: Termination and guardedness checking with continuous types. In: Hofmann, M., ed., Typed Lambda Calculi and Applications (TLCA 2003), Valencia, Spain, volume 2701 of LNCS. Springer (2003), 1–15
2. Abel, A.: Termination checking with types. RAIRO – Theoretical Informatics and Applications **38** (2004) 277–319. Special Issue: Fixed Points in Computer Science (FICS'03)
3. Abel, A.: A Polymorphic Lambda-Calculus with Sized Higher-Order Types. Ph.D. thesis, Ludwig-Maximilians-Universität München (2006). Draft available under `http://www.tcs.ifi.lmu.de/~abel/diss.pdf`
4. Abel, A., Matthes, R., Uustalu, T.: Iteration schemes for higher-order and nested datatypes. Theoretical Computer Science **333** (2005) 3–66
5. Altenkirch, T., McBride, C.: Generic programming within dependently typed programming. In: Gibbons, J., Jeuring, J., eds., Working Conference on Generic Programming (WCGP'02), Dagstuhl, Germany, volume 243 of IFIP Conference Proceedings. Kluwer (2003), 1–20
6. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: Flum, J., Rodríguez-Artalejo, M., eds., Computer Science Logic, CSL '99, Madrid, Spain, volume 1683 of LNCS. Springer (1999), 453–468
7. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming — an introduction. In: LNCS, volume 1608. Springer (1999), 28–115. Revised version of lecture notes for AFP'98.
8. Barthe, G., Frade, M. J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. Math. Struct. in Comput. Sci. **14** (2004) 1–45
9. Barthe, G., Grégoire, B., Pastawski, F.: Practical inference for type-based termination in a polymorphic setting. In: Urzyczyn, P., ed., Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan, volume 3461 of LNCS. Springer (2005), 71–85
10. Benke, M., Dybjer, P., Jansson, P.: Universes for generic programs and proofs in dependent type theory. Nord. J. Comput. **10** (2003) 265–289
11. Bird, R., Meertens, L.: Nested datatypes. In: Jeuring, J., ed., Mathematics of Program Construction, MPC'98, volume 1422 of LNCS. Springer (1998), 52–67
12. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Asp. Comput. **11** (1999) 200–222
13. Bird, R. S., Paterson, R.: De Bruijn notation as a nested datatype. J. Func. Program. **9** (1999) 77–91

14. Blanqui, F.: A type-based termination criterion for dependently-typed higher-order rewrite systems. In: van Oostrom, V., ed., Rewriting Techniques and Applications (RTA 2004), Aachen, Germany, volume 3091 of LNCS. Springer (2004), 24–39
15. Blanqui, F.: Decidability of type-checking in the Calculus of Algebraic Constructions with size annotations. In: Ong, C.-H. L., ed., Computer Science Logic, CSL 2005, Oxford, UK, volume 3634 of LNCS. Springer (2005), 135–150
16. Crary, K., Weirich, S.: Flexible type analysis. In: International Conference on Functional Programming (ICFP'99), Paris, France, volume 34 of SIGPLAN Notices. ACM (1999), 233–248
17. Crary, K., Weirich, S., Morrisett, J. G.: Intensional polymorphism in type-erasure semantics. In: International Conference on Functional Programming (ICFP'98), Baltimore, Maryland, USA, volume 34 of SIGPLAN Notices. ACM (1999), 301–312
18. Frade, M. J.: Type-Based Termination of Recursive Definitions and Constructor Subtyping in Typed Lambda Calculi. Ph.D. thesis, Universidade do Minho, Departamento de Informática (2003)
19. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: van Oostrom, V., ed., Rewriting Techniques and Applications (RTA 2004), Aachen, Germany, volume 3091 of LNCS. Springer (2004), 210–220
20. Giménez, E.: Structural recursive definitions in type theory. In: International Colloquium on Automata, Languages and Programming (ICALP'98), Aalborg, Denmark, volume 1443 of LNCS. Springer (1998), 397–408
21. Harper, R., Morrisett, J. G.: Compiling polymorphism using intensional type analysis. In: Principles of Programming Languages, POPL'95, San Francisco, California. ACM (1995), 130–141
22. Hinze, R.: Numerical representations as higher-order nested datatypes. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn (1998)
23. Hinze, R.: Polytypic programming with ease (extended abstract). In: Middeldorp, A., Sato, T., eds., Functional and Logic Programming, FLOPS'99, Tsukuba, Japan, volume 1722 of LNCS. Springer (1999), 21–36
24. Hinze, R.: Generalizing generalized tries. J. Func. Program. **10** (2000) 327–351
25. Hinze, R.: A new approach to generic functional programming. In: Principles of Programming Languages, POPL 2000, Boston, Massachusetts, USA. ACM (2000), 119–132
26. Hinze, R.: Manufacturing datatypes. J. Func. Program. **11** (2001) 493–524
27. Hinze, R.: Polytypic values possess polykinded types. MPC Special Issue, Sci. Comput. Program. **43** (2002) 129–159
28. Hinze, R., Jeuring, J.: Generic Haskell: Applications. In: Backhouse, R. C., Gibbons, J., eds., Generic Programming - Advanced Lectures, volume 2793 of LNCS. Springer (2003), 57–96
29. Hinze, R., Jeuring, J.: Generic Haskell: Practice and Theory. In: Backhouse, R. C., Gibbons, J., eds., Generic Programming - Advanced Lectures, volume 2793 of LNCS. Springer (2003), 1–56
30. Hinze, R., Jeuring, J., Löh, A.: Type-indexed data types. MPC Special Issue, Sci. Comput. Program. **51** (2004) 117–151
31. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Principles of Programming Languages, POPL'96 (1996), 410–423
32. INRIA: The Coq Proof Assistant Reference Manual, version 8.0 edition (2004). http://coq.inria.fr/doc/main.html
33. Jansson, P.: Functional Polytypic Programming. Ph.D. thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden (2000)
34. Jansson, P., Jeuring, J.: PolyP—a polytypic programming extension. In: Principles of Programming Languages, POPL'97, Paris, France. ACM (1997), 470–482

35. Jansson, P., Jeuring, J.: Polytypic data conversion programs. Sci. Comput. Program. **43** (2002) 35–75
36. Martin, C., Gibbons, J., Bayley, I.: Disciplined, efficient, generalised folds for nested datatypes. Formal Asp. Comput. **16** (2004) 19–35
37. Mendler, N. P.: Recursive types and type constraints in second-order lambda calculus. In: Logic in Computer Science (LICS'87), Ithaca, N.Y. IEEE Computer Society Press (1987), 30–36
38. Norell, U.: Functional Generic Programming and Type Theory. Master's thesis, Computing Science, Chalmers University of Technology (2002). Available from `http://www.cs.chalmers.se/~ulfn`
39. Norell, U., Jansson, P.: Polytypic programming in Haskell. In: Implementation of Functional Languages (IFL'03). LNCS (2004)
40. Norell, U., Jansson, P.: Prototyping generic programming in Template Haskell. In: Kozen, D., ed., Mathematics of Program Construction, MPC'04, volume 3125 of LNCS. Springer (2004), 314–333
41. Okasaki, C.: Purely Functional Data Structures. Ph.D. thesis, Carnegie Mellon University (1996)
42. Okasaki, C.: From fast exponentiation to square matrices: An adventure in types. In: International Conference on Functional Programming (ICFP'99), Paris, France (1999), 28–35
43. Okasaki, C.: Red-black trees in a functional setting. J. Func. Program. **9** (1999) 471–477
44. Pareto, L.: Types for Crash Prevention. Ph.D. thesis, Chalmers University of Technology (2000)
45. Pfeifer, H., Rueß, H.: Polytypic proof construction. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., , Théry, L., eds., Theorem Proving in Higher Order Logics, TPHOLs '99, Nice, France, volume 1690 of LNCS. Springer (1999), 55–72
46. Pierce, B. C.: Types and Programming Languages. MIT Press (2002)
47. Saha, B., Trifonov, V., Shao, Z.: Intensional analysis of quantified types. ACM Trans. Program. Lang. Syst. **25** (2003) 159–209
48. Sereni, D., Jones, N. D.: Termination analysis of higher-order functional programs. In: Yi, K., ed., Asian Symposium on Programming Languages and Systems (APLAS'05), Tsukuba, Japan, volume 3780 of LNCS. Springer (2005), 281–297