Implementing a Normalizer Using Sized Heterogeneous Types

Andreas Abel[1]
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, GERMANY
abel@tcs.ifi.lmu.de

Abstract

In the simply-typed lambda-calculus, a hereditary substitution replaces a free variable in a normal form r by another normal form s of type a, removing freshly created redexes on the fly. It can be defined by lexicographic induction on a and r, thus, giving rise to a structurally recursive normalizer for the simply-typed lambda-calculus. We generalize this scheme to simultaneous substitutions, preserving its simple termination argument. We further implement hereditary simultaneous substitutions in a functional programming language with sized heterogeneous inductive types, $\mathsf{F}_{\widehat{\omega}}$, arriving at an interpreter whose termination can be tracked by the type system of its host programming language.

Keywords: Sized Types, Heterogeneous Types, Nested Types, Interpreter, Normalization, De Bruijn Terms

1. INTRODUCTION

Inductive types T can be expressed as the least solution of the recursive equation $F\,X = X$ for some suitable monotone type function F; we write $T = \mu F$ and say that T is the least fixed-point of F. The least fixed point can be obtained in two ways: *from above*, using the theorem of Knaster and Tarski, and *from below* using transfinite iteration. Defining

$$\begin{array}{lll} \mu^0 \;\; F &=& \text{empty} \\ \mu^{\alpha+1}\; F &=& F\,(\mu^\alpha F) \\ \mu^\lambda \;\; F &=& \bigcup_{\alpha<\lambda} \mu^\alpha F, \end{array}$$

the least fixed-point of F is reached for some ordinal γ and we have $F\,(\mu^\gamma F) = \mu^\gamma F$. The construction of an inductive type from below is convenient if we want to define a function $f : T \to C$ over an inductive type; we can reason by transfinite induction that f is well-defined. This is especially the case if f is *structurally recursive*, i. e., refers in recursive calls only to smaller elements of the inductive type. Consider $f = \mathsf{fix}\,s$ being the fixed point of a functional s, i. e., $\mathsf{fix}\,s = s\,(\mathsf{fix}\,s)$. Such functions can be introduced by the rule

$$\frac{s \in (\mu^\alpha F \to C) \to (\mu^{\alpha+1} F \to C) \text{ for all } \alpha < \beta}{\mathsf{fix}\,s \in \mu^\beta F \to C}.$$

The rule can be justified by transfinite induction up to β. Base case: Since $\mu^0 F$ is empty, $\mathsf{fix}\,s \in \mu^0 F \to C$ trivially. For the step case, assume $\mathsf{fix}\,s \in \mu^\alpha F \to C$. By the premise of the rule, $s\,(\mathsf{fix}\,s) \in \mu^{\alpha+1} F \to C$, and by the fix-point equation, $\mathsf{fix}\,s \in \mu^{\alpha+1} F \to C$. Finally, for the limit case, assume $\mathsf{fix}\,s \in \mu^\alpha F \to C$ for all $\alpha < \lambda$ and assume $t \in \mu^\lambda F$. By definition of iteration at a limit, $t \in \mu^\alpha F$ for some $\alpha < \lambda$, hence $\mathsf{fix}\,s\,t \in C$. Since t was arbitrary, $\mathsf{fix}\,s \in \mu^\lambda F \to C$.

Up to now, we have considered the *semantics* of inductive types and the justification of *semantical* functions. Mendler [22] first observed that by turning these semantical concepts into *syntax*,[2] one

[1]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).

[2]Another example where semantics has been successfully turned into syntax is the *monad*. Invented by Moggi as a tool to reason about impure features, it has become a device to program imperatively in Haskell.

gets a type system that accepts structurally recursive functions, hence, guarantees termination of well-typed programs. This idea has been taken up by Hughes, Pareto, and Sabry [17], Giménez [16], Amadio and Coupet-Grimal [9], Barthe et al. [10], Blanqui [14] and myself [1, 2]. My thesis [5] describes $\mathsf{F}^{\widehat{}}_{\omega}$, an extension of the higher-order polymorphic lambda-calculus F_ω by sized inductive types and structural recursion. A *sized inductive type* $\mu^a F$ is the syntactic equivalent of an iteration stage $\mu^\alpha F$, only that a is now a *syntactic* ordinal expression and F is a *syntactic* type constructor whose monotonicity is established *syntactically*.

In this article, we give a non-trivial example of a structurally recursive function whose termination is automatically established using sized types: a β-normalizer for simply-typed λ-terms. The heart of the normalizer are *hereditary substitutions* [25]: substitution of a normal form into another one, potentially triggering new substitutions, until a normal form is returned. Surprisingly, this process can be formulated by a lexicographic recursion on the type of the substituted value and the normal form substituted into. Our contribution is a generalization to *hereditary simultaneous substitutions*, necessary to handle de Bruijn-style λ-terms with static guarantee of well-scopedness. Such de Bruijn terms can be represented using a data structure of heterogeneous type [8, 13]. In $\mathsf{F}^{\widehat{}}_{\omega}$, heterogeneous types can be expressed by *higher-kinded* inductive types, i. e., type $\mu^a F$ where F is an operator on type constructors instead of just types. As a result, we obtain an implementation of a normalizer in $\mathsf{F}^{\widehat{}}_{\omega}$ whose termination and well-scopedness is ensured by type-checking in $\mathsf{F}^{\widehat{}}_{\omega}$.

The remainder of this article is organized as follows. In Section 2, we briefly present system $\mathsf{F}^{\widehat{}}_{\omega}$. Then, we specify and verify hereditary substitutions and normalization for simply-typed λ-terms in Section 3. In Section 4 we modify the specification to account for simultaneous hereditary substitutions. An implementation in $\mathsf{F}^{\widehat{}}_{\omega}$ is provided in Section 5. We conclude by discussing related and further work.

2. $\mathsf{F}^{\widehat{}}_{\omega}$: A POLYMORPHIC λ-CALCULUS WITH SIZED TYPES

In this section, we briefly introduce the most important concepts of $\mathsf{F}^{\widehat{}}_{\omega}$. We assume familiarity with system F_ω and inductive types.

Kinds. Kinds classify type constructors. In F_ω, one has the kind $*$ of types and function kinds $\kappa \to \kappa'$ for type constructors. In $\mathsf{F}^{\widehat{}}_{\omega}$ we additionally have a kind ord for syntactic ordinals. And we distinguish type constructors by their variance: they can be covariant (monotonic), contravariant (antitonic), or mixed-variant (no monotonicity information).

κ	$::=$	$*$	kind of types
	$\mid$	ord	kind of ordinals
	$\mid$	$\kappa \xrightarrow{+} \kappa'$	kind of covariant type constructors
	$\mid$	$\kappa \xrightarrow{-} \kappa'$	kind of contravariant type constructors
	$\mid$	$\kappa \xrightarrow{\circ} \kappa'$	kind of mixed-variant type constructors

Type constructors. Type constructors are expressions of a type-level λ-calculus, given by the following grammar.

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

The constants C are drawn from a signature Σ (see Figure 1). We use $+$, $\times$, and $\to$ infix and write $\forall X : \kappa.A$ for $\forall_\kappa(\lambda X.A)$. If clear from the context or inessential, we omit the kind annotation κ in $\forall X : \kappa.A$ and μ_κ. We also write the first argument to μ_κ—the size index—superscript. For instance, $\mu^a(\lambda X.\, 1 + A \times X)$ denotes the lists of length $< a$ containing elements of type A.

Constructors $a : \mathsf{ord}$ that denote ordinal expressions are *essentially* following the grammar

$$a, b ::= \imath \mid \infty \mid \mathsf{s}\,a$$

where we use $\imath$ and $\jmath$ for variables of kind ord. (Of course, there are non-normal expressions like $(\lambda X.\,\infty)\,G : \mathsf{ord}$, or neutral expression such as $X\,\vec{G} : \mathsf{ord}$, but they do not matter for our purposes.)

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
∞	:	ord	infinity ordinal

FIGURE 1: Signature Σ.

The ordinal expression ∞ denotes an ordinal large enough such that all inductive types have reached their fixed-point, thus, we have

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

The ordinal expression $\mathsf{s}\,a$ denotes the ordinal successor of a. We also write $a + n$ as shorthand for $\mathsf{s}\,(\ldots(\mathsf{s}\,a))$ (n successors). In general, we have the type equation

$$\mu^{a+1} F = F\,(\mu^a F).$$

for sized inductive types.

Subtyping. The sized inductive type $\mu^a F : *$ contains data trees of height $< a$ (this can be seen from its semantics). Since the size index denotes an upper bound, we have a natural subtyping relation

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

This relation is extended appropriately to all type constructors and to higher-order, taking the variance information into account. (For more details, see the article about higher-order subtyping in $\mathsf{F}^{\widehat{}}_\omega$ [4].)

Programs (terms). $\mathsf{F}^{\widehat{}}_\omega$ is a purely functional language with categorical datatypes[3] and recursion. Programs are given by the following grammar.

e, f	::=	$x \mid \lambda x.e \mid f\,e$	λ-calculus
	$\mid$	$\langle\rangle$	inhabitant of type 1
	$\mid$	$\langle e_1, e_2\rangle \mid \mathsf{fst}\,e \mid \mathsf{snd}\,e$	pairing and projections
	$\mid$	$\mathsf{inl}\,e \mid \mathsf{inr}\,e \mid \mathsf{case}\,e\,f_1\,f_2$	injections into disjoint sum, case distinction
	$\mid$	$\mathsf{fix}\,f$	recursion

The typing rules for the λ-calculus part are inherited from Curry-style system F_ω. Unit type, cartesian product and disjoint sum are introduced and eliminated using the respective terms given in the grammar. Inductive types can be introduced and eliminated using the above type equations. The outstanding feature of $\mathsf{F}^{\widehat{}}_\omega$ is the type-based recursion rule which has been semantically

[3] Categorical datatypes do not contain *names*, just *structure*. Constructors of data structures are derived from the program primitives. This is in opposition to nominal languages where data constructors are themselves primitives (e. g., in Haskell).

motivated in the introduction:

$$\frac{f : \forall \imath{:}\mathsf{ord}.\, (\forall \vec{X}.\, \mu^\imath F\, \vec{X} \to G\, \imath) \to \forall \vec{X}.\, \mu^{\imath+1} F\, \vec{X} \to G\, (\imath+1) \qquad G : \mathsf{ord} \xrightarrow{+} * \qquad a : \mathsf{ord}}{\mathsf{fix}\, f : \forall \vec{X}.\, \mu^a F\, \vec{X} \to G\, a}$$

In comparison with the semantic rule given in the introduction, we now allow polymorphic recursion, and the result type $G\,a$ may mention the ordinal index a, but only positively [10, 2, 14]. Note that the size index a in the conclusion can be arbitrary. (Alternatively, one could formulate the rule with conclusion $\mathsf{fix}\, f : \forall \imath \forall \vec{X}.\, \mu^\imath F\, \vec{X} \to G\, \imath$. This size-polymorphic function can then instantiated to any size a.)

This has been a short introduction to $\mathsf{F}^{\widehat{}}_{\omega}$, more details can be found in my thesis [5]. In the next section, we start developing an interpreter for some object language, in our case, the simply-typed λ-calculus, which we will later implement in our meta language, $\mathsf{F}^{\widehat{}}_{\omega}$.

3. A TERMINATING NORMALIZER FOR SIMPLY-TYPED LAMBDA-TERMS

In this section, we formally define hereditary substitution for the simply-typed λ-calculus. We show its termination, soundness, and completeness.

Types and terms. The following grammars introduce our object language, the simply-typed λ-calculus. To distinguish it from our meta-language, $\mathsf{F}^{\widehat{}}_{\omega}$, we use lower case letters for the types.

a, b, c	$::=$	$o \mid a \to b$	simple types
r, s, t	$::=$	$x \mid \lambda x{:}a.t \mid r\, s$	simply-typed terms
n	$::=$	$x \mid n\, s$	neutral terms (required in Section 3.3)
Γ	$::=$	$\diamond \mid \Gamma, x{:}a$	typing contexts

Ordinary (capture-avoiding) substitution $[s/x]t$ of s for x in t, the set $\mathsf{FV}(t)$ of free variables of term t, and β-equality $t =_\beta t'$ of terms t, t' are defined as usual, as well as the typing judgement $\Gamma \vdash t : a$. Let $|a| \in \mathbb{N}$ denote a measure on types with $|b| < |b \to c|$ and $|c| \leq |b \to c|$ (e. g., *size*, or *order*).

3.1. Hereditary substitution

We define a 4-ary function $[s^a/x]t$, called *hereditary substitution*, which returns a *result* $\hat{r}$. A result is either just a term r or a term annotated with a type, written r^a. The intention is that if s and t are β-normal and well-typed terms, then the result will also be β-normal (and well-typed). Our definition is a simplification of Watkins et al.'s hereditary substitutions for terms of the logical framework LF extended to linearity and concurrency [25]. Implicitly, hereditary substitutions are present already in Joachimski and Matthes' normalization proof for the simply-typed λ-calculus [18].

Let us first introduce some overloaded notation on results $\hat{r}$. The operation $\underline{\hat{r}}$ discards the type annotation on the result if present, i. e., $\underline{r^a} = r$ and $\underline{r} = r$. This operation is to be applied implicitly when the context demands it. For example, application of two results $\hat{r}$ and $\hat{s}$ implicitly discards the type annotations: $\hat{r}\, \hat{s} = \underline{\hat{r}}\, \underline{\hat{s}}$. Similarly for abstraction: $\lambda x : a.\hat{r} = \lambda x : a.\underline{\hat{r}}$. Finally, reannotation $\hat{r}^a = (\underline{\hat{r}})^a$ puts a fresh type annotation a onto a result.

Using these notations, we can compactly define hereditary substitution:

$$\begin{array}{lcll}
[s^a/x]x & = & s^a & \\
[s^a/x]y & = & y & \text{if } x \neq y \\
[s^a/x](\lambda y{:}b.r) & = & \lambda y{:}b.\, [s^a/x]r & \text{where } y \text{ fresh for } s, x \\
 & & & \\
[s^a/x](t\, u) & = & ([\hat{u}^b/y]r')^c & \text{if } \hat{t} = (\lambda y{:}b'.r')^{b \to c} \\
 & & \hat{t}\, \hat{u} & \text{otherwise} \\
 & & & \\
\text{where } \hat{t} & = & [s^a/x]t & \\
\hat{u} & = & [s^a/x]u &
\end{array}$$

Lemma 1 (Invariant) *If* $[s^a/x]t = r^c$ *then* $|c| \leq |a|$.

Proof. By induction on t. There are only two cases which return an annotated term:

- $[s^a/x]x = s^a$. Trivially $|a| \leq |a|$.
- $[s^a/x](t\,u) = ([\hat{u}^b/y]r')^c$ where $[s^a/x]t = (\lambda y : b'.r')^{b \to c}$. By induction hypothesis, $|b \to c| \leq |a|$. This proves the invariant, since $|c| \leq |b \to c|$ by definition of the measure $|\cdot|$.

□

Lemma 2 (Termination and soundness) $[s^a/x]t =_\beta [s/x]t$ *for all* a, s, x, t.

Proof. By lexicographic induction on $(|a|, t)$. We consider the case of application, $[s^a/x](t\,u)$: By induction hypothesis, $\hat{t} = [s^a/x]t$ and $\hat{u} = [s^a/x]u$ are both defined. We consider the subcase $\hat{t} = (\lambda y : b'.r')^{b \to c}$. Using the invariant, we infer $|b| < |b \to c| \leq |a|$. Hence, we can again apply the induction hypothesis to infer that $[\hat{u}^b/y]r'$ terminates, thus, by definition, also $[s^a/x](t\,u)$. Soundness holds by the induction hypotheses, since $(\lambda y : b'.r')\,\hat{u} =_\beta [\hat{u}/y]r'$. □

3.2. Full Normalization

We define a function $[\![t]\!]$ which β-normalizes term t, provided it is well-typed. Even if it is not well-typed, the normalizer terminates and returns a term which is β-equal to the input.

$$
\begin{array}{lcll}
[\![x]\!] & = & x & \\
[\![\lambda x{:}a.r]\!] & = & \lambda x{:}a.\,[\![r]\!] & \\
 & & & \\
[\![r\,s]\!] & = & [[\![s]\!]^a/x]t & \text{if } [\![r]\!] = \lambda x{:}a.t \\
 & & [\![r]\!]\,[\![s]\!] & \text{otherwise}
\end{array}
$$

Lemma 3 (Termination and soundness) $[\![t]\!] =_\beta t$ *for all* t.

Proof. By induction on t, using termination and soundness of hereditary substitution. □

Example 4 (Behavior of normalizer)

$$[\![(\lambda x{:}o \to o.\lambda y{:}b.\,x\,y)\,(\lambda z{:}a.z)]\!] = \lambda y{:}b.y$$

Hereditarily substituting $(\lambda z{:}a.z)$ *at function type* $o \to o$ *for* x *into* $x\,y$ *triggers another substitution of* y *for* z *in* z*. However, if variable* x *is annotated with base type* o *only, this second substitution is not invoked and we get a non-normal result:*

$$[\![(\lambda x{:}o.\lambda y{:}b.\,x\,y)\,(\lambda z{:}a.z)]\!] = \lambda y{:}b.\,(\lambda z{:}a.z)\,y$$

Evaluation of terms that are diverging under β*-reduction also stops, e.g.,*

$$[\![(\lambda x{:}o.\,x\,x)\,(\lambda x{:}o.\,x\,x)]\!] = (\lambda x{:}o.\,x\,x)\,(\lambda x{:}o.\,x\,x).$$

3.3. Completeness of the Normalizer

In the following we show that the normalizer actually computes normal forms for well-typed terms.

Typed normal forms. We introduce a judgement $\Gamma \vdash t \Downarrow a$ which expresses that t is a β-normal form of type a in context Γ.

$$\frac{(x{:}a) \in \Gamma}{\Gamma \vdash x \Downarrow a} \qquad \frac{\Gamma \vdash n \Downarrow a \to b \qquad \Gamma \vdash s \Downarrow a}{\Gamma \vdash n\,s \Downarrow b} \qquad \frac{\Gamma, x{:}a \vdash r \Downarrow b}{\Gamma \vdash \lambda x{:}a.r \Downarrow a \to b}$$

Note that in the second rule, the head n of the application $n\,s$ is a neutral term, in particular, not an abstraction.

Lemma 5 (Completeness of hereditary substitution) *Let $\Gamma \vdash s \Downarrow a$ and $\Gamma, x:a \vdash t \Downarrow c$. Then exists an r with the following properties: If t is neutral then either $[s^a/x]t = r^c$, or r is also neutral and $[s^a/x]t = r$. Otherwise, if t is not neutral, $[s^a/x]t = r$. In all cases, $\Gamma \vdash r \Downarrow c$.*

Proof. By lexicographic induction on $(|a|, t)$. We consider the interesting case $t = n\,u$:

$$\frac{\Gamma, x:a \vdash n \Downarrow b \to c \qquad \Gamma, x:a \vdash u \Downarrow b}{\Gamma, x:a \vdash n\,u \Downarrow c}$$

Let $\hat{u} = [s^a/x]u$. If $[s^a/x]n = r$ and r is neutral, then $\Gamma \vdash r\,\hat{u} : c$ follows easily by induction hypothesis. Otherwise, $[s^a/x]n = r^{b\to c}$. If r is neutral then we conclude as before, otherwise $r = \lambda y{:}b.r'$. By the invariant, $|b| < |b \to c| \leq a$, and we can apply the induction hypothesis to infer $\Gamma \vdash [\hat{u}^b/y]r' \Downarrow c$, which is by definition equivalent to $\Gamma \vdash [s^a/x](n\,u) \Downarrow c$. □

Theorem 6 (Completeness of the normalizer) *If $\Gamma \vdash t : a$ then $\Gamma \vdash [\![t]\!] \Downarrow a$.*

Proof. By induction on t, using the previous lemma in case of a β-redex. □

4. ADAPTATION TO SIMULTANEOUS SUBSTITUTIONS

Our aim is to implement the normalizer of the last section for a representation of terms using de Bruijn indices. Following Altenkirch and Reus [8] and Bird and Paterson [13], untyped λ-terms over the set of free variables A can be implemented by a nested datatype $\mathsf{Tm}\,A$ with the three constructors:

$$\begin{array}{lcl} \mathsf{var} & : & \forall A.\, A \to \mathsf{Tm}\,A \\ \mathsf{abs} & : & \forall A.\, \mathsf{Tm}\,(1+A) \to \mathsf{Tm}\,A \\ \mathsf{app} & : & \forall A.\, \mathsf{Tm}\,A \to \mathsf{Tm}\,A \to \mathsf{Tm}\,A \end{array}$$

The second constructor, abs, expects a term with one more free variable $(1 + A)$ and binds this variable such that the result will only have free variables in A.

In this representation of de Bruijn terms, which statically ensures well-scoping, substitution $[s/x]t$ for a single variable x is not directly implementable. Instead, one has to define the simultaneous substitution $t\rho$ for all free variables in t: If t has type $\mathsf{Tm}\,A$ and ρ has type $A \to \mathsf{Tm}\,B$, the result $t\rho$ of the substitution has type $\mathsf{Tm}\,B$.[4]

Hereditary simultaneous substitution. A *valuation* ρ is a function from variables to results $\hat{r}$. Let the update $\rho[x \mapsto \hat{r}]$ of valuation ρ in x by result $\hat{r}$ be defined as usual:

$$\begin{array}{lcll} \rho[x \mapsto \hat{r}](x) & = & \hat{r} & \\ \rho[x \mapsto \hat{r}](y) & = & \rho(y) & \text{if } x \neq y \end{array}$$

The singleton valuation which maps x to $\hat{r}$ and all other variables to themselves shall be denoted by $(x \mapsto \hat{r})$. A single substitution $[s/x]t$ can be implemented using the simultaneous substitution $t(x \mapsto s)$ with a singleton valuation.

The *hereditary simultaneous substitution* $t!\rho$ returns a result $\hat{r}$ and is defined by the following equations:

$$\begin{array}{rcll} x!\rho & = & \rho(x) & \\ (\lambda y{:}b.r)!\rho & = & \lambda y{:}b.\,(r!\rho[y \mapsto y]) & \text{where } y \text{ fresh for any } \rho(x) \text{ with } x \in \mathsf{FV}(r) \setminus \{y\} \\ (t\,u)!\rho & = & (r'!(y \mapsto \hat{u}^b))^c & \text{if } \hat{t} = (\lambda y{:}b'.r')^{b\to c} \\ & & \hat{t}\,\hat{u} & \text{otherwise} \\ \text{where } \hat{t} & = & t!\rho & \\ \hat{u} & = & u!\rho & \end{array}$$

[4] The type constructor Tm forms a Kleisli triple with unit var and simultaneous substitution as the *bind*-operation [8].

A closer look reveals that we use only three operations on valuations: *lookup*, $\rho(x)$, *lifting*, $\rho[y \mapsto y]$ for y fresh, and creation of a *singleton* valuation, $(y \mapsto \hat{u}^b)$. In particular, if $t!\rho$ is invoked with a singleton valuation ρ, all recursive calls will also just involve a singleton valuation. We could therefore restrict ourselves to singleton valuations. However, for termination, a weaker requirement is sufficient:

In the following we consider only valuations ρ where $\rho(x) = r^a$ for only *finitely many* variables x. For such valuations,

$$|\rho| := \max\{|a| \mid \rho(x) = r^a\}$$

is a well-defined measure, $|\rho| \in \mathbb{N}$. Trivially, $|(y \mapsto s^a)| = |a|$ for a singleton valuation.

Hereditary simultaneous substitutions always terminate, and the respective proof for hereditary singleton substitutions can be adopted:

Lemma 7 (Invariant) *If $t!\rho = r^c$, then $|c| \leq |\rho|$.*

Proof. By induction on t. □

Lemma 8 (Termination and soundness of hereditary simultaneous substitutions)
For all terms r and valuations ρ such that $|\rho|$ exists, we have $r!\rho =_\beta r\rho$.

Proof. By lexicographic induction on $(|\rho|, r)$. In case $r = t\,u$ and $\hat{t} = t!\rho = (\lambda y{:}b'.r')^{b\to c}$, use the invariant to establish $|(y \mapsto \hat{u}^b)| = |b| < |b \to c| \leq |\rho|$ and apply the induction hypothesis. □

By setting $[s^a/x]t := t!(x \mapsto s^a)$ we can reuse the code for the normalization function $[\![r]\!]$ from the last section.

5. IMPLEMENTATION USING SIZED HETEROGENEOUS TYPES

In this section, we implement hereditary substitutions in $\mathsf{F}^{\widehat{}}_{\omega}$. As a result, we will get a normalizer whose termination is certified by the type system of $\mathsf{F}^{\widehat{}}_{\omega}$.

Simple types over a single base type o can be defined as follows in $\mathsf{F}^{\widehat{}}_{\omega}$:

$$\begin{array}{lcl} \mathsf{Ty} & : & \mathsf{ord} \xrightarrow{+} * \\ \mathsf{Ty} & := & \lambda\imath.\,\mu^{\imath}_{*}\lambda X.\,1 + X \times X \end{array} \qquad \begin{array}{lcl} \mathsf{o} & : & \forall\imath.\,\mathsf{Ty}^{\imath+1} \\ \mathsf{o} & := & \mathsf{inl}\,\langle\rangle \\ \mathsf{arr} & : & \forall\imath.\,\mathsf{Ty}^{\imath} \to \mathsf{Ty}^{\imath} \to \mathsf{Ty}^{\imath+1} \\ \mathsf{arr} & := & \lambda a\lambda b.\,\mathsf{inr}\,\langle a, b\rangle \end{array}$$

The size index $\imath$ in $\mathsf{Ty}^{\imath}$ implements a measure on simple types. The requirements $|b| < |\mathsf{arr}\,b\,c|$ and $|c| \leq |\mathsf{arr}\,b\,x|$ hold since $b, c : \mathsf{Ty}^{\imath}$ implies $\mathsf{arr}\,b\,c : \mathsf{Ty}^{\imath+1}$. But note that the only kind of measure that can be captured by the type system of $\mathsf{F}^{\widehat{}}_{\omega}$ is the *structural* one.

We can express the sized type constructor Tm for de Bruijn terms by a least fixed point of kind $* \xrightarrow{+} *$.

$$\begin{array}{lcl} \mathsf{Tm} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\ \mathsf{Tm} & := & \lambda\imath.\,\mu^{\imath}_{*\xrightarrow{+}*}\lambda X\lambda A.\,A + (X\,A \times X\,A + \mathsf{Ty}^{\infty} \times X\,(1 + A)) \end{array}$$

$$\begin{array}{lcl} \mathsf{var} & : & \forall\imath\forall A.\,A \to \mathsf{Tm}^{\imath+1}\,A \\ \mathsf{var} & := & \lambda x.\,\mathsf{inl}\,x \end{array}$$

$$\begin{array}{lcl} \mathsf{app} & : & \forall\imath\forall A.\,\mathsf{Tm}^{\imath}\,A \to \mathsf{Tm}^{\imath}\,A \to \mathsf{Tm}^{\imath+1}\,A \\ \mathsf{app} & := & \lambda r\lambda s.\,\mathsf{inr}\,(\mathsf{inl}\,\langle r, s\rangle) \end{array}$$

$$\begin{array}{lcl} \mathsf{abs} & : & \forall\imath\forall A.\,\mathsf{Ty}^{\infty} \to \mathsf{Tm}^{\imath}\,(1 + A) \to \mathsf{Tm}^{\imath+1}\,A \\ \mathsf{abs} & := & \lambda a\lambda r.\,\mathsf{inr}\,(\mathsf{inr}\,\langle a, r\rangle) \end{array}$$

Note that the first argument to abs is the object-level type of the abstraction: $\mathsf{abs}\,a\,r$ represents $\lambda x\!:\!a.\,r$.

A whole article [6] has been devoted to functions defined by iteration over heterogeneous data types such as Tm. $\mathsf{F}^{\widehat{}}_{\omega}$ can simulate all the systems discussed in that article, hence, all examples in that article can be replayed in $\mathsf{F}^{\widehat{}}_{\omega}$. Even more, in $\mathsf{F}^{\widehat{}}_{\omega}$, some functions can be given more precise typings, e. g., the functoriality/monotonicity witness of Tm:

$$
\begin{array}{lcl}
\mathsf{map} & : & \forall\imath\forall A\forall B.\,(A \to B) \to \mathsf{Tm}^{\imath} A \to \mathsf{Tm}^{\imath} B \\
\mathsf{map} & := & \lambda f \lambda r.\ \mathsf{map}'\ r\ f \\
 & & \\
\mathsf{map}' & : & \forall\imath\forall A.\,\mathsf{Tm}^{\imath} A \to \forall B.\,(A \to B) \to \mathsf{Tm}^{\imath} B \\
\mathsf{map}' & := & \mathsf{fix}\,\lambda map'\lambda t\lambda f.\ \mathsf{match}\ t\ \mathsf{with} \\
 & & \quad \mathsf{var}\ x \ \mapsto\ \mathsf{var}\ (f\,x) \\
 & & \quad \mathsf{app}\ r\,s \ \mapsto\ \mathsf{app}\ (map'\ r\ f)\ (map'\ s\ f) \\
 & & \quad \mathsf{abs}\ a\,r \ \mapsto\ \mathsf{abs}\ a\ (map'\ r\ (\mathsf{lift}\ f)) \\
 & & \\
\mathsf{lift} & : & \forall A\forall B.\,(A \to B) \to (1 + A \to 1 + B) \\
\mathsf{lift} & := & \lambda f \lambda t.\,\mathsf{match}\ t\ \mathsf{with} \\
 & & \quad \mathsf{inl}\ \langle\rangle \ \mapsto\ \mathsf{inl}\ \langle\rangle \\
 & & \quad \mathsf{inr}\ x \ \mapsto\ \mathsf{inr}\ (f\,x)
\end{array}
$$

Note our use of pattern matching over t distinguishing the *defined* constructors var, app, and abs. Assuming the patterns resulting from expanding the constructors are non-overlapping, mutually exclusive, and complete, this can be mechanically translated into case expressions. For the sake of readability, we prefer pattern matching as syntactic sugar.

The call $\mathsf{map}\ f\,t$ renames all free variables in t according to f; the structure of t remains unchanged, which is partially reflected in the type of map: it expresses that the output term is not higher than the input term.

The type of recursion in map' is polymorphic:

$$C(\imath) := \forall A.\,\mathsf{Tm}^{\imath} A \to \forall B.\,(A \to B) \to \mathsf{Tm}^{\imath} B.$$

This is typical for functions over heterogeneous data types; in our example, since in the recursive call $\mathsf{map}\ r\,(\mathsf{lift}\ f)$ the argument r has type $\mathsf{Tm}^{\imath}\,(1 + A)$, the variable A of the recursion type $C(\imath)$ has to be instantiated to $1 + A$.

The result of a hereditary substitution of a normal term into a neutral term is either a neutral term r or a normal term r plus (its) type a, which we have written as r^a. We encode these alternatives in the type $\mathsf{Res}^{\imath}\,A$, where $\imath$ is an upper bound on the size of the type a and A is the set of free variables that might occur in the result term r. We define two constructors: $\mathsf{ne}_{\mathsf{Res}}\,r$ for the first alternative, and $\mathsf{nf}_{\mathsf{Res}}\,r\,a$ for the second alternative.

$$
\begin{array}{lcl}
\mathsf{Res} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Res} & := & \lambda\imath\lambda A.\,\mathsf{Tm}^{\infty}\,A \times (1 + \mathsf{Ty}^{\imath}) \\
 & & \\
\mathsf{ne}_{\mathsf{Res}} & : & \forall\imath.\,\mathsf{Tm}^{\infty}\,A \to \mathsf{Res}^{\imath}\,A \\
\mathsf{ne}_{\mathsf{Res}} & := & \lambda r.\,\langle r,\ \mathsf{inl}\,\langle\rangle\rangle \\
 & & \\
\mathsf{nf}_{\mathsf{Res}} & : & \forall\imath.\,\mathsf{Tm}^{\infty}\,A \to \mathsf{Ty}^{\imath} \to \mathsf{Res}^{\imath}\,A \\
\mathsf{nf}_{\mathsf{Res}} & := & \lambda r\lambda a.\,\langle r,\ \mathsf{inr}\,a\rangle
\end{array}
$$

The destructor tm just extracts the term component. The function $\mathsf{weak}_{\mathsf{Res}}$ lifts the free variables in a result term by one.

$$
\begin{array}{lcl}
\mathsf{tm} & : & \forall\imath\forall A.\,\mathsf{Res}^{\imath}\,A \to \mathsf{Tm}^{\infty}\,A \\
\mathsf{tm} & := & \lambda\langle r, a\rangle.\,r \\
 & & \\
\mathsf{weak}_{\mathsf{Res}} & : & \forall\imath\forall A.\,\mathsf{Res}^{\imath}\,A \to \mathsf{Res}^{\imath}\,(1 + A) \\
\mathsf{weak}_{\mathsf{Res}} & := & \lambda\langle r, a\rangle.\,\langle \mathsf{map}\ \ \mathsf{inr}\ r,\ a\rangle
\end{array}
$$

Finally, we can mimic all term constructors on Res. They should all discard the type component, if present. This is why the size index $\imath$ on the result of these operations is arbitrarily small:

$$\begin{array}{lcl}
\mathsf{var}_{\mathsf{Res}} & : & \forall\imath\forall A.\, A \to \mathsf{Res}^{\imath}\, A \\
\mathsf{var}_{\mathsf{Res}} & := & \lambda a.\, \mathsf{ne}_{\mathsf{Res}}\,(\mathsf{var}\, a) \\
\mathsf{abs}_{\mathsf{Res}} & : & \forall\imath\forall A.\, \mathsf{Ty}^{\infty} \to \mathsf{Res}^{\infty}(1+A) \to \mathsf{Res}^{\imath} A \\
\mathsf{abs}_{\mathsf{Res}} & := & \lambda a\lambda r.\, \mathsf{ne}_{\mathsf{Res}}\,(\mathsf{abs}\, a\, (\mathsf{tm}\, r)) \\
\mathsf{app}_{\mathsf{Res}} & : & \forall A.\, \mathsf{Res}^{\infty}\, A \to \mathsf{Res}^{\infty}\, A \to \mathsf{Res}^{\imath}\, A \\
\mathsf{app}_{\mathsf{Res}} & := & \lambda t\lambda u.\, \mathsf{ne}_{\mathsf{Res}}\,(\mathsf{app}\,(\mathsf{tm}\, t)\,(\mathsf{tm}\, u))
\end{array}$$

We represent valuations ρ which map all variables in A to a result with variables in B by the sized type $\mathsf{Val}^{\imath}\, A\, B$. The size index $\imath$ is an upper bound for $|\rho|$.

$$\begin{array}{lcl}
\mathsf{Val} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{-} * \xrightarrow{+} * \\
\mathsf{Val} & := & \lambda\imath\lambda A\lambda B.\, A \to \mathsf{Res}^{\imath} B \\
\mathsf{sg}_{\mathsf{Val}} & : & \forall\imath\forall A.\, \mathsf{Tm}^{\infty} A \to \mathsf{Ty}^{\imath} \to \mathsf{Val}^{\imath}\,(1+A)\, A \\
\mathsf{sg}_{\mathsf{Val}} & := & \lambda s\lambda a\lambda my.\ \mathsf{match}\ my\ \mathsf{with} \\
 & & \quad \mathsf{inl}\,\langle\rangle \ \mapsto\ \mathsf{nf}_{\mathsf{Res}}\, s\, a \\
 & & \quad \mathsf{inr}\, y \ \mapsto\ \mathsf{var}_{\mathsf{Res}}\, y \\
\mathsf{lift}_{\mathsf{Val}} & : & \forall\imath\forall A\forall B.\, \mathsf{Val}^{\imath}\, A\, B \to \mathsf{Val}^{\imath}\,(1+A)\,(1+B) \\
\mathsf{lift}_{\mathsf{Val}} & := & \lambda\rho\lambda mx.\ \mathsf{match}\ mx\ \mathsf{with} \\
 & & \quad \mathsf{inl}\,\langle\rangle \ \mapsto\ \mathsf{var}_{\mathsf{Res}}\,(\mathsf{inl}\,\langle\rangle) \\
 & & \quad \mathsf{inr}\, x \ \mapsto\ \mathsf{weak}_{\mathsf{Res}}\,(\rho\, x)
\end{array}$$

The expression $\mathsf{sg}_{\mathsf{Val}}\, s\, a \ :\ \mathsf{Val}^{\imath}\,(1+A)\, A$ corresponds to the singleton valuation $(x \mapsto s^a)$; it generates a valuation which maps the variable x in 1 to $\mathsf{nf}_{\mathsf{Res}}\, s\, a$ and the variables y in A to $\mathsf{ne}_{\mathsf{Res}}\,(\mathsf{var}\, y)$. The extension $\rho[y \mapsto y]$ of a valuation ρ is implemented by $\mathsf{lift}_{\mathsf{Val}}\,\rho$.

For implementing hereditary substitutions, we have to take into account the limitations of recursion in $\mathsf{F}^{\widehat{}}_{\omega}$. Lexicographic recursion on type and term, $\mathsf{Ty}^{\imath} \times \mathsf{Tm}^{\jmath}\, A$, needs to be split up into an outer recursion on $\mathsf{Ty}^{\imath}$ and an inner recursion on $\mathsf{Tm}^{\jmath}$.

Thus, we define hereditary substitution $[s^a/x]r$ by a function $\mathsf{subst}\, a\, s\, r$ recursive in a. This outer function calls an inner function $\mathsf{simsubst}\, r\,(\mathsf{sg}_{\mathsf{Val}}\, a\, s)$ recursive in r, which performs hereditary simultaneous substitutions in r, starting with the singleton valuation $(x \mapsto s^a)$. In one case, the inner function calls the outer function, albeit with a smaller type b. That b is in fact smaller than a is tracked by the type system.

$$\begin{array}{l}
\mathsf{subst} : \quad \forall\imath.\, \mathsf{Ty}^{\imath} \to \forall A.\, \mathsf{Tm}^{\infty}\, A \to \mathsf{Tm}^{\infty}(1+A) \to \mathsf{Tm}^{\infty}\, A \\
\mathsf{subst} := \mathsf{fix}\, \lambda subst\lambda a\lambda s\lambda t.\ \mathsf{tm}\,(\mathsf{simsubst}\, t\,(\mathsf{sg}_{\mathsf{Val}}\, s\, a))
\end{array}$$

$$\begin{array}{lcll}
\mathsf{where}\ \mathsf{simsubst} : & \forall\jmath.\, \forall A\forall B.\, \mathsf{Tm}^{\jmath} A \to \mathsf{Val}^{\imath+1}\, A\, B \to \mathsf{Res}^{\imath+1} B & & \\
\mathsf{simsubst} := & \mathsf{fix}\, \lambda simsubst\lambda t\lambda\rho.\ \mathsf{match}\ t\ \mathsf{with} & & \\
 & \quad \mathsf{var}\, x & \mapsto & \rho\, x \\
 & \quad \mathsf{abs}\, b\, r & \mapsto & \mathsf{abs}_{\mathsf{Res}}\, b\,(simsubst\, r\,(\mathsf{lift}_{\mathsf{Val}}\,\rho)) \\
 & \quad \mathsf{app}\, t\, u & \mapsto & \mathsf{let}\ \hat{t} = simsubst\ t\ \rho \\
 & & & \qquad \hat{u} = simsubst\ u\ \rho \\
 & & & \mathsf{in}\ \mathsf{match}\ \hat{t}\ \mathsf{with} \\
 & & & \qquad \mathsf{nf}_{\mathsf{Res}}\,(\mathsf{abs}\, b'\, r')\,(\mathsf{arr}\, b\, c) \ \mapsto\ \mathsf{nf}_{\mathsf{Res}}\,(subst\, b\,(\mathsf{tm}\,\hat{u})\, r')\, c \\
 & & & \qquad _ \ \mapsto\ \mathsf{app}_{\mathsf{Res}}\,\hat{t}\,\hat{u}
\end{array}$$

The outer function is defined by induction on size $\imath$; we have the following important types of bound variables:

$$\begin{array}{lcl}
subst & : & \mathsf{Ty}^{\imath} \to \forall A.\, \mathsf{Tm}^{\infty}\, A \to \mathsf{Tm}^{\infty}(1+A) \to \mathsf{Tm}^{\infty}\, A \\
a & : & \mathsf{Ty}^{\imath+1}
\end{array}$$

This explains why in the type of the inner function, we have used size index $\imath + 1$ to Val and Res. The inner function is defined by induction on j. It is important that $\mathsf{lift}_{\mathsf{Val}}$ does not touch the size argument to Val. Also, $\mathsf{abs}_{\mathsf{Res}}$ and $\mathsf{app}_{\mathsf{Res}}$ can return results Res with any size argument, thus, $\imath + 1$ is fine. In the application $\rho\, x$, the size index on Val is returned as the size of Res. Finally, when we match $\hat{t} : \mathsf{Res}^{\imath+1}\, B$ as the result of a recursive call to $simsubst$, the expression $\mathsf{arr}\, b\, c$ has sized type $\mathsf{Ty}^{\imath+1}$, hence, $b : \mathsf{Ty}^{\imath}$, and the recursive call to $subst$ is legal. Since $c : \mathsf{Ty}^{\imath} \leq \mathsf{Ty}^{\imath+1}$ by subtyping, the result $\mathsf{nf}_{\mathsf{Res}}\, (\dots)\, c$ is well-typed.

The evaluator $[\![-]\!]$ can now be implemented straightforwardly. Its termination is guaranteed by sized types!

6. RELATED WORK

Type-based termination and sized types. Mendler [23] first devised a typing rule for general recursive programs that would single out the ones that are *iterative*—like `fold` for lists. Mendler's system was extended by Giménez [16] and Barthe et al. [10, 11] to account for course-of-value recursion—covering functions like quicksort. I have explored how the type of a recursive function may depend on the size index [1]—now functions like breadth-first traversal are recognized as terminating by the type system. In my thesis [5], I cover also heterogeneous types like this article's Tm. Blanqui [14, 15] extended type-based termination to dependent types and rewriting.

Independently of these "type-theoretic" developments, Hughes, Pareto, and Sabry [17, 24] have explored sized types for preventing crashes of functional programs. They arrived at a similar rule for type-based terminating recursion.

De Bruijn representation by a heterogeneous type. Since the 1960s it is known that simultaneous substitution can be viewed as the bind operation of a suitable monad [19, VI.4] [20]. For terms with binders, the idea was taken up by Bird and Paterson [13] as a case study for heterogeneous (nested, resp.) types in Haskell. Altenkirch and Reus [8] implement simultaneous substitutions for de Bruijn terms in a more type-theoretic setting; they carry out the renaming—performed in substitution under a binder—also by an invocation of substitution. Thus, termination of substitution becomes more complicated; they justify it by a lexicographic argument. McBride [21] stratifies this situation by exhibiting a common scheme which substitution and renaming as special instances. This scheme is again structurally recursive. McBride considers the object language of simply-typed terms in a dependently typed meta language; this way, he even establishes that substitution and renaming are type-preserving.

Adams [7] has carried out metatheory of pure type systems using de Bruijn representations as a heterogeneous type. He found that he had to pass from *small-step* statements, which speak about a single variable, e.g., a single substitution, to *big-step* statements, that take all variables into account at the same time, like simultaneous substitution. We can only confirm his findings.

Arithmetic normalization proofs and functions. Joachimski and Matthes [18] present strong and weak normalization proofs of the simply-typed λ-calculus using a lexicographic induction on types and derivations, similar to the one we have given in Section 3. They hand-extract a general recursive normalization algorithm which may diverge on ill-typed input. Berghofer [12] implemented the weak normalization proof in Isabelle and automatically extracted a normalizer, being much more complicated and keeping some intermediate data structures. I have implemented a similar proof in Twelf [3]. The termination checker of Twelf is capable to certify termination of the normalization proof, if viewed as a recursive function.

Watkins et al. [25] define hereditary substitutions for terms of the logical framework LF, and its extensions to linearity and concurrency. These substitutions are terminating by a lexicographic induction on type and term—as in this article. Hereditary substitutions enable us to specify

a framework which only treats canonical forms (β-normal η-long objects), thus, simplifying its metatheory considerably.

7. CONCLUSION

We have presented $\mathsf{F}^{\widehat{\omega}}$, a pure polymorphic programming language with sized heterogeneous inductive types and type-based termination. As a non-trivial example, we have implemented a normalizer for simply-typed λ-terms in $\mathsf{F}^{\widehat{\omega}}$. The termination of the normalizer is statically ensured in $\mathsf{F}^{\widehat{\omega}}$. Additionally, the normalizer is well-scoped, by virtue of an implementation of λ-terms using a heterogeneous type.

One could go further and aim for a normalizer for statically well-typed λ-terms, "well-typed" referring here to typing in the *object* language of simple types. McBride has done a first step and implemented type-preserving renaming and substitution for the object language using a dependently-typed meta language [21]. Pursuing this approach, one could modify the normalizer such that it does not any more explicitly manipulate object-level types but considers only object-level terms. The object-level types would be visible only in the meta-level typing of the normalizer. The normalizer, viewed purely operationally, would have the possibility to diverge on ill-typed terms, but the static well-typedness condition would guarantee its termination for all actual inputs.

Acknowledgments. Thanks to the anonymous referees for their careful reading and their helpful and encouraging comments.

REFERENCES

[1] Andreas Abel. Termination and guardedness checking with continuous types. In M. Hofmann, editor, *Typed Lambda Calculi and Applications (TLCA 2003), Valencia, Spain*, volume 2701 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, June 2003.

[2] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS'03).

[3] Andreas Abel. Weak normalization for the simply-typed lambda-calculus in Twelf. In *Logical Frameworks and Metalanguages (LFM 04)*, IJCAR, Cork, Ireland, 2004.

[4] Andreas Abel. Polarized subtyping for sized types. In D. Grigoriev, J. Harrison, and E. A. Hirsch, editors, *Computer Science Symposium in Russia (CSR 2006), St. Petersburg, June 8-12, 2006*, volume 3967 of *Lecture Notes in Computer Science*, pages 381–392. Springer-Verlag, 2006.

[5] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006. Draft available under `http://www.tcs.ifi.lmu.de/~abel/diss.pdf`.

[6] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1–2):3–66, 2005.

[7] Robin Adams. Formalized metatheory with terms represented by an indexed family of types. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2006.

[8] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer-Verlag, 1999.

[9] Roberto M. Amadio and Solange Coupet-Grimal. Analysis of a guard condition in type theory. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structures, First International Conference, FoSSaCS'98*, volume 1378 of *Lecture Notes in*

Computer Science, pages 48–62. Springer-Verlag, 1998.

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

[11] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for type-based termination in a polymorphic setting. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*, pages 71–85. Springer-Verlag, 2005.

[12] Stefan Berghofer. Extracting a normalization algorithm in Isabelle/HOL. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 2006.

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

[14] Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3 – 5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 24–39. Springer-Verlag, 2004.

[15] Frédéric Blanqui. Decidability of type-checking in the Calculus of Algebraic Constructions with size annotations. In C.-H. Luke Ong, editor, *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3634 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 2005.

[16] Eduardo Giménez. Structural recursive definitions in type theory. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1998.

[17] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *23rd Symposium on Principles of Programming Languages, POPL'96*, pages 410–423, 1996.

[18] Felix Joachimski and Ralph Matthes. Short proofs of normalization. *Archive of Mathematical Logic*, 42(1):59–87, 2003.

[19] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[20] E. Manes. *Algebraic Theories*. Springer-Verlag, 1976.

[21] Conor McBride. Type-preserving renaming and substitution. *Journal of Functional Programming*, 2006. Functional Pearl. To appear.

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

[23] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[24] Lars Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, 2000.

[25] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgements and properties. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2003.