Implementing a Normalizer Using Sized Heterogeneous Types

ANDREAS ABEL[*]
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, GERMANY
(*e-mail:* `andreas.abel@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 implement hereditary substitutions in a functional programming language with sized heterogeneous inductive types, $\widehat{\mathsf{F}_\omega}$, arriving at an interpreter whose termination can be tracked by the type system of its host programming language.

1 Introduction

An interpreter for a total programming language, i. e., a language in which only terminating programs can be written, will naturally terminate on all executions. However, this fact is sometimes hard to prove, as the abundant literature on normalization results shows. Often such a termination proof requires considerable insight into the semantics of the considered language. In this article, we consider a particularly easy example of a total language: the simply-typed lambda-calculus (STL). Its termination has been proven long ago and there are many different proofs. However, we will implement an interpreter for the STL whose termination can be checked *automatically*! As host programming language for the implementation we use $\widehat{\mathsf{F}_\omega}$, a polymorphic, purely functional language with sized types. Each well-typed $\widehat{\mathsf{F}_\omega}$-program is terminating (Abel, 2006b), hence, it is sufficient to show that the interpreter is a well-typed program. Provided the functions of the implementation are supplied with Haskell-style type signatures, well-typedness can be checked mechanically. Before we detail the structure of the interpreter, let us have a look at the idea behind $\widehat{\mathsf{F}_\omega}$.

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

[*] 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).

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

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

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 defined by *course-of-value recursion*, 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 (1987) first observed that by turning these semantical concepts into *syntax*,[1] one 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 (1996), Giménez (1998), Amadio and Coupet-Grimal (1998), Barthe et al. (2004), Blanqui (2004) and myself (Abel, 2004a). In this article we use $\mathsf{F}_\omega^{\widehat{\ }}$, an extension of the higher-order polymorphic lambda-calculus F_ω by sized inductive types and recursion on sizes. 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. At the heart of the normalizer are *hereditary substitutions* (Watkins *et al.*, 2003): substitution of a normal form into another one, triggering new substitutions to remove freshly created redexes, 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. We generalize them

[1] 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.

to *hereditary simultaneous substitutions* in order 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 (Bellegarde & Hook, 1994; Altenkirch & Reus, 1999; Bird & Paterson, 1999). In $\mathsf{F}_\omega^{\frown}$, 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 $\mathsf{F}_\omega^{\frown}$ whose termination and well-scopedness is ensured by type-checking in $\mathsf{F}_\omega^{\frown}$.

The remainder of this article is organized as follows. In Section 2, we briefly present system $\mathsf{F}_\omega^{\frown}$. 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 $\mathsf{F}_\omega^{\frown}$ is provided in Section 5. We conclude by discussing related and further work.

A preliminary version of this article has been presented at MSFP'06 (Abel, 2006a).

2 $\mathsf{F}_\omega^{\frown}$: A Polymorphic λ-calculus with Sized Types

In this section, we briefly introduce the most important concepts of $\mathsf{F}_\omega^{\frown}$. We assume some 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 $\mathsf{F}_\omega^{\frown}$ we additionally have a kind ord for syntactic ordinals. Moreover, we distinguish type constructors by their variance: they can be covariant (monotonic), contravariant (antitonic), constant (both mono- and antitonic) or mixed-variant (no monotonicity information). This is achieved by annotating function kinds with a polarity p.

p, q	$::=$	$\circ$	mixed-variant (no monotonicity information)
	$\mid$	$+$	covariant (monotone)
	$\mid$	$-$	contravariant (antitone)
	$\mid$	$\top$	constant (both mono- and antitone)
κ	$::=$	$*$	kind of types
	$\mid$	ord	kind of ordinals
	$\mid$	$\kappa \xrightarrow{p} \kappa'$	kind of p-variant type constructors

The order on polarities is the reflexive-transitive closure of the axioms $\circ \leq p$ and $p \leq \top$. This means that the bigger a polarity, the more information it provides about the functions: just a function ($\circ$), a mono-/antitone function ($+/-$), a constant function ($\top$).

If one composes a function in $\kappa_1 \xrightarrow{p} \kappa_2$ with a function in $\kappa_2 \xrightarrow{q} \kappa_3$ one obtains a function in $\kappa_1 \xrightarrow{pq} \kappa_3$. For the associative and commutative polarity composition pq we have the laws $\top p = \top$, $\circ p = \circ$ (for $p \neq \top$), $+p = p$, and $-- = +$. Inverse application $p^{-1}q$ of a polarity p to a polarity q is defined as the solution of

$$\forall q, q'.\ \ p^{-1}q \leq q' \iff q \leq pq'.$$

In other words, the operations $f(q) = p^{-1}q$ and $g(q') = pq'$ form a Galois connection. It is not hard to see that the unique solution is given by the equations: $+^{-1}q = q$, $-^{-1}q = -q$, $\top^{-1}q = \circ$, $\circ^{-1}\circ = \circ$ and $\circ^{-1}q = \top$ (for $q \neq \circ$).

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

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

$Z\,\lambda Y \lambda X X\, Y$ is to be read as $Z\,((\lambda Y(\lambda X X))Y)$; we use the dot "." as an opening parenthesis which closes as far to the right as syntactically possible, e.g. $\lambda X \lambda Y.\, Y\, X$ means $\lambda X \lambda Y\,(Y\, X)$. The constants C are drawn from a signature Σ. It contains at least the following symbols together with their kinding:

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

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, $\mu^\alpha(\lambda X.\, 1 + A \times X)$ denotes the lists of length $< \alpha$ containing elements of type A.

Kinding. A kinding context Δ is a finite map from type constructor variables X to pairs $p\kappa$ of a polarity p and a kind κ. Inverse application $p^{-1}\Delta$ of a polarity p to a context Δ is defined by $(p^{-1}\Delta)(X) = p^{-1}(\Delta(X))$ (inverse-apply p to the polarity component of $\Delta(X)$). The judgement $\Delta \vdash F : \kappa$ assigns kind κ to constructor F in context Δ. It is given inductively by the following rules:

$$\frac{(C : \kappa) \in \Sigma}{\Delta \vdash C : \kappa} \qquad \frac{\Delta(X) = p\kappa \qquad p \leq +}{\Delta \vdash X : \kappa}$$

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

The judgement $X_1 : p_1\kappa_1, \dots, X_n : p_n\kappa_n \vdash F : \kappa$ means that $\lambda X_1 \dots \lambda X_n.F$ is a function which is p_i-variant in its ith argument. Hence, one can only extract variables of polarity $+$ or $\circ$ from the context; $X\!:\!-\kappa \vdash X : \kappa$ would state that the identity function is antitone, which is certainly wrong. The application rule forms a function $H' = \lambda\vec{X}.\, F'\,\vec{X}\,(G\,\vec{X})$ from functions $F' = \lambda\vec{X}.\, F$ and $G' = \lambda\vec{X}.\, G$. The variance p_i' of H' in its ith argument depends on the variance p_i of F' and q_i of G' in this argument and the variance p of F' in its last argument: p_i' is the infimum of

p_i and pq_i, thus, $p'_i = p_i$ if $p_i \leq pq_i$, which is equivalent to $p^{-1}p_i \leq q_i$. So the most liberal setting for q_i is $p^{-1}p_i$, which is what happens in the application rule.

Example 1
Using the rules, we can derive

$$\lambda F \lambda G \lambda X.\, F\,X \to G\,X : (* \xrightarrow{\circ} *) \xrightarrow{-} (* \xrightarrow{\circ} *) \xrightarrow{+} (* \xrightarrow{\circ} *).$$

Let $\Delta = F : -(* \xrightarrow{\circ} *), G : +(* \xrightarrow{\circ} *), X : \circ *$. We show $\Delta \vdash F\,X \to G\,X : *$ which means that $F\,X \to G\,X$ is antitone in F and monotone in G. (It is mixed-variant in X.) The type-functions F and G are assumed to be mixed-variant. Observe that $+^{-1}\Delta = \Delta$ and $-^{-1}\Delta = -\Delta = F : +(* \xrightarrow{\circ} *), G : -(* \xrightarrow{\circ} *), X : \circ *$ and $\circ^{-1}\Delta = \circ^{-1}(-\Delta) = F : \top(* \xrightarrow{\circ} *), G : \top(* \xrightarrow{\circ} *), X : \circ *$.

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash \to : * \xrightarrow{\circ} * \xrightarrow{+} *} \qquad -^{-1}\Delta \vdash F\,X : *}{\Delta \vdash (\to)\,(F\,X) : * \xrightarrow{+} *} \qquad \Delta \vdash G\,X : *}{\Delta \vdash (\to)\,(F\,X)\,(G\,X) : *}$$

Herein, we use the subderivations

$$\dfrac{\dfrac{-^{-1}\Delta(F) = +(* \xrightarrow{\circ} *)}{-^{-1}\Delta \vdash F : * \xrightarrow{\circ} *} \qquad \dfrac{\circ^{-1}(-^{-1}\Delta)(X) = \circ *}{\circ^{-1}(-^{-1}\Delta) \vdash X : *}}{-^{-1}\Delta \vdash F\,X : *}$$

and

$$\dfrac{\dfrac{\Delta(G) = +(* \xrightarrow{\circ} *)}{\Delta \vdash G : * \xrightarrow{\circ} *} \qquad \dfrac{\circ^{-1}\Delta(X) = \circ *}{\circ^{-1}\Delta \vdash X : *}}{\Delta \vdash G\,X : *}$$

Observe the polarity change of F from $-$ to $+$ as we step into the domain part of the arrow.

Example 2
Doubly negative counts as positive: $\lambda X.(X \to 1) \to 1 : * \xrightarrow{+} *$.

Type constructor equality. The judgement $\Delta \vdash F = F' : \kappa$ states that the two constructors F, F' which have kind κ in context Δ, are considered equal in $\mathsf{F}_\omega^{\widehat{}}$. It is the least congruence over the following axioms:

$$\dfrac{\Delta, X : p\kappa \vdash F : \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash (\lambda X F)\,G = [G/X]F : \kappa'} \qquad \dfrac{\Delta \vdash F : \kappa \xrightarrow{p} \kappa'}{\Delta \vdash \lambda X.\, F\,X = F : \kappa \xrightarrow{p} \kappa'}\ X \notin \mathsf{FV}(F)$$

$$\dfrac{\Delta \vdash F : \kappa \xrightarrow{\top} \kappa' \qquad \Delta \vdash G : \kappa \qquad \Delta \vdash G' : \kappa}{\Delta \vdash F\,G = F\,G' : \kappa'}$$

$$\dfrac{}{\Delta \vdash \mathsf{s}\,\infty = \infty : \mathsf{ord}} \qquad \dfrac{\Delta \vdash \alpha : \mathsf{ord}}{\Delta \vdash \mu^{\mathsf{s}\alpha} = \lambda F.\, F\,(\mu^{\alpha}\,F) : (* \xrightarrow{+} *) \xrightarrow{+} *}$$

The first rule encodes β- and the second η-equality. The third rule states that a constant function has the same value for all arguments. Let us look at the remaining two rules.

The normal forms of constructors $\alpha : \mathsf{ord}$ that denote ordinal expressions are *essentially* following the grammar

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

where we use $\imath$ and $\jmath$ for variables of kind ord. Not captured by this grammar are neutral expressions like $X\,\vec{G}$; they stem from variables $X : \vec{\kappa} \xrightarrow{\vec{p}} \mathsf{ord}$ which are not forbidden, but of no use in the type system to follow. The ordinal expression ∞ denotes an ordinal large enough such that all inductive types have reached their fixed-point, thus, we have

$$F\,(\mu^{\infty} F) = \mu^{\infty} F. \tag{1}$$

The ordinal expression $\mathsf{s}\,\alpha$ denotes the ordinal successor of α. We also write $\alpha + n$ as shorthand for $\mathsf{s}\,(\ldots(\mathsf{s}\,\alpha))$ (n successors). The closure ordinal ∞ is the largest size we need to consider, so we set

$$\mathsf{s}\,\infty = \infty. \tag{2}$$

In general, we have the type equation

$$\mu^{\alpha+1} F = F\,(\mu^{\alpha} F) \tag{3}$$

for sized inductive types, which reflects the semantical construction of inductive types given in the introduction. Equations (2) and (3) are axioms of our judgmental constructor equality $\Delta \vdash F = F' : \kappa$, the equation (1) follows from the other two.

Subtyping $\Delta \vdash F \leq F' : \kappa$ is induced by axioms expressing relations between ordinals and equipped with congruence rules that respect polarities.

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

$$\frac{\Delta \vdash F \leq F' : \kappa \xrightarrow{p} \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G \leq F'\,G : \kappa'}$$

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

Additionally, we have a congruence rule for λ-abstraction and rules for reflexivity, transitivity and antisymmetry. The rules for ordinal expressions are self-explanatory; the third rule states that functions are compared point-wise; and the last two rules generate inequalities from mono-/antitone type functions.

Using the application rules and the polarized kind κ of constants $C : \kappa$ from the signature, we can establish interesting subtyping relations: Since μ is covariant in its first argument, the rules allow us to derive

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

This reflects the fact that by definition of the semantics, ordinals are upper bounds on size. For instance $\mathsf{List}^\alpha\, A = \mu^\alpha(\lambda X.\, 1 + A \times X)$ contains lists of length $< \alpha$, $\mathsf{BTree}^\alpha\, A = \mu^\alpha(\lambda X.\, 1 + A \times X \times X)$ binary trees of height $< \alpha$, and $\mathsf{Nat}^\alpha = \mu^\alpha(\lambda X.\, 1 + X)$ natural numbers $< \alpha$. Transfinite sizes are needed for infinitely branching trees, e. g., $\mu^\alpha(\lambda X.\, 1 + \mathsf{Nat}^\infty \times X + (\mathsf{Nat}^\infty \to X))$.

Programs (terms). $\mathsf{F}_\omega^{\widehat{}}$ is a purely functional language with categorical datatypes[2] 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.

Typing. Wellformed typing contexts are generated by the rules

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

Typing contexts can always be viewed as kinding contexts, forgetting the bindings for the term variables. The typing rules for the λ-calculus part are inherited from Curry-style system F_ω.

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

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

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

Unit type, cartesian product and disjoint sum are introduced and eliminated using the respective terms given in the grammar (we omit the obvious typing rules). Inductive types can be introduced and eliminated using the type equations. For instance, the empty list is typed as

$$\mathsf{inl}\,\langle\rangle : 1 + A \times \mathsf{List}^\imath A = \mathsf{List}^{\imath+1} A.$$

The central feature of $\mathsf{F}_\omega^{\widehat{}}$ is the type-based recursion rule which has been semantically motivated in the introduction:

$$\frac{\begin{array}{c}\Gamma \vdash \alpha : \mathsf{ord} \\ \Gamma \vdash G : \mathsf{ord} \xrightarrow{+} * \\ \Gamma \vdash f : \forall \imath\!:\!\mathsf{ord}.\, (\forall \vec{X}.\, \mu^\imath F\, \vec{X} \to G\, \imath) \to \forall \vec{X}.\, \mu^{\imath+1} F\, \vec{X} \to G\,(\imath + 1)\end{array}}{\Gamma \vdash \mathsf{fix}\, f : \forall \vec{X}.\, \mu^\alpha F\, \vec{X} \to G\, \alpha}$$

[2] 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).

In comparison with the semantic rule given in the introduction, we now allow polymorphic recursion, and the result type $G\,\alpha$ may mention the ordinal index α, but only positively (Barthe *et al.*, 2004; Abel, 2004a; Blanqui, 2004). Note that the size index α 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 be instantiated to any size α.)

Typing is not decidable, since $\mathsf{F}^{\widehat{}}_\omega$ features impredicative polymorphism and polymorphic recursion, which are both undecidable by themselves already. However undecidability may not be a problem in practice, as the experience with *Haskell* shows. There, it is usually sufficient that the programmer provides the types of all recursive functions. Type information is then propagated using heuristics like *bidirectional type-checking.* In previous work (Abel, 2004a) I have described a bidirectional type-checking algorithm for a simply-typed language with sized types, and I expect that sized types can be integrated into Haskell type-checking without causing new undecidability issues.

We illustrate the potential of type-based termination à la $\mathsf{F}^{\widehat{}}_\omega$ with the following example.

Example 3 (*Quicksort*)
$\mathsf{F}^{\widehat{}}_\omega$ accepts the usual functional quicksort as terminating. The type system can track the size of the output of $\mathsf{filter}\, f\, l$ (the list of elements of l for which f holds) which is at most the size of l.

$$
\begin{array}{lll}
\mathsf{filter} : \forall A.\,(A \to \mathsf{Bool}) \to \forall\imath.\,\mathsf{List}^\imath A \to \mathsf{List}^\imath\, A \\
\mathsf{filter}\, f & = & \mathsf{fix}\,\lambda \mathit{filt}\lambda l.\ \mathsf{case}\ l \\
 & & \quad (\lambda\_.\, \mathsf{inl}\langle\rangle) \\
 & & \quad (\lambda p.\ \mathsf{if}\ f\,(\mathsf{fst}\,p)\ \mathsf{then}\ \mathsf{inr}\,\langle \mathsf{fst}\,p,\ \mathit{filt}\,(\mathsf{snd}\,p)\rangle\ \mathsf{else}\ \mathit{filt}\,(\mathsf{snd}\,p))
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{quicksort} : \forall\imath.\,\mathsf{List}^\imath\,\mathsf{Int} \to \mathsf{List}^\infty\,\mathsf{Int} \\
\mathsf{quicksort} & = & \mathsf{fix}\,\lambda \mathit{quicksort}\lambda l.\ \mathsf{case}\ l \\
 & & \quad (\lambda\_.\, \mathsf{inl}\langle\rangle) \\
 & & \quad (\lambda p.\ \mathsf{append}\ \ (\mathit{quicksort}\,(\mathsf{filter}\,(\le\,(\mathsf{fst}\,p))\,(\mathsf{snd}\,p))) \\
 & & \qquad\qquad\qquad (\mathsf{inr}\,\langle \mathsf{fst}\,p,\ \mathit{quicksort}\,(\mathsf{filter}\,(>\,(\mathsf{fst}\,p))\,(\mathsf{snd}\,p))\rangle))
\end{array}
$$

Using the size bound on filtered lists, quicksort is well-typed, hence, terminating.

In the next section, we start developing an interpreter for an 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' shall be defined as usual, as well as the typing judgement $\Gamma \vdash t : a$. We need these notions to prove correctness of the normalization algorithm which we are going to implement in $\mathsf{F}_\omega^{\widehat{}}$.

Let $|a| \in \mathbb{N}$ denote a measure on types with $|b| < |b \to c|$ and $|c| \le |b \to c|$. There are three natural candidates for this measure:

$\lvert o\rvert$	$\lvert a \to b\rvert$	measure
1	$\lvert a\rvert + \lvert b\rvert + 1$	tree *size*
1	$\max(\lvert a\rvert, \lvert b\rvert) + 1$	tree *height*, or *structural measure*
0	$\max(\lvert a\rvert + 1, \lvert b\rvert)$	*order* of the type

The last two can be expressed in $\mathsf{F}_\omega^{\widehat{}}$, see Section 5.

3.1 Hereditary substitution

We define a 4-ary function $[s/x]^a t$, 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^c. The intention is that if s and t are β-normal and well-typed terms and a is the type of s and x, then the result will also be β-normal (and well-typed). Our definition is a simplification of Watkins et al.'s (2003) hereditary substitutions for terms of the concurrent logical framework CLF. Implicitly, hereditary substitutions are present already in Joachimski and Matthes' (2003) normalization proof for the simply-typed λ-calculus. This proof is similar to normalization proofs before Tait (1967), e. g., Gentzen (1935)[3], Turing, Prawitz (1965)[4], and others.

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 the process $[s/x]^a t = \hat{r}$ of hereditarily substituting s of type a for variable x in t. The type a should be viewed as *fuel* which is spent on triggering *new* hereditary substitutions. If the result $\hat{r}$ is r^c, then the process returns fuel c. If the result is simply a term r, no fuel is returned,

[3] pp. 197ff
[4] Thm. III.2

i. e., it has all been spent or wasted.

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

In lines 2, 3, and 5, all fuel is wasted. In line 1, all fuel is returned. And in line 4, the fuel $(b \to c)$ returned by the hereditary substitution into t is partially spent (b) on a new substitution, and partially returned (c).

Example 4
Let us demonstrate hereditary substitution for a few cases. We will write λxt instead of $\lambda x\!:\!a.\, t$ if the type a of the abstracted variable does not matter for our purposes.

1. Different head variable $(x \neq y)$:

$$[s/x]^a (y\, t_1 \dots t_n) = y\, ([s/x]^a t_1) \dots ([s/x]^a t_n)$$

Hereditarily substituting for x into a term $y\, t_1 \dots t_n$ with a different head variable behaves on the surface level like ordinary substitution. (In the subterms t_i something more interesting might happen, of course.)

2. Same head variable, *out of fuel*: creates redexes.

$$[\lambda y.\, y\, \lambda zz/x]^o (x\, \lambda ff) = (\lambda y.\, y\, \lambda zz)\, \lambda ff.$$

We have $[\lambda y.\, y\, \lambda zz/x]^o x = (\lambda y.\, y\, \lambda zz)^o$, and since o is a base type, the application of the result to λff does not trigger a new substitution.

3. Same head variable, *some fuel left*:

$$[\lambda y.\, y\, \lambda zz/x]^{o \to o} (x\, \lambda ff) = (\lambda ff)\, \lambda zz.$$

This time, executing the hereditary substitution for x in x returns $o \to o$ fuel, so a new hereditary substitution $[\lambda ff/y]^o (y\, \lambda zz)$ is triggered. However, now we run out of fuel, and one redex remains.

4. Same head variable, *enough fuel*:

$$[\lambda y.\, y\, \lambda zz/x]^{(o \to o) \to o} (x\, \lambda ff) = (\lambda zz)^o.$$

Finally, we reach a normal form, since the hereditary substitution for y returns fuel $o \to o$, so a third and last hereditary substitution, $[\lambda zz/f]^o f$ is triggered.

5. Substituting into redexes:

$$[s/x]^a ((\lambda yt)\, u) = (\lambda y.\, [s/x]^a t)([s/x]^a u)$$

Already present redexes are preserved by hereditary substitution, only new redexes which are created by the substitution process can be eliminated. Similarly, redexes in s are kept:

$$[(\lambda xr)\, s/y]^a (y\, z) = (\lambda xr)\, s\, z$$

In the following we will prove termination of hereditary substitution and soundness, i. e., the hereditary substitution $[s/x]^a t$ returns a term with the same meaning as ordinary substitution $[s/x]t$. A necessary condition for termination is that *driving does not create fuel*, which we can more formally express as:

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

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

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

This entails termination, because whenever we want to execute a new hereditary substitution, we need to have $b \to c$ fuel left, from which we take b for the new substitution and keep the c for further substitutions which may arise.

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

Remark 1
The statement "$[s/x]^a t =_\beta$ *some term*" entails the statement "$[s/x]^a t$ is defined" (termination).

Proof
By lexicographic induction on $(|a|, t)$. In the case of application, $[s/x]^a (t\,u)$, by induction hypothesis, $\hat{t} = [s/x]^a t$ and $\hat{u} = [s/x]^a u$ 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 $[\hat{u}/y]^b r'$ terminates, thus, by definition, also $[s/x]^a (t\,u)$. Soundness holds by the induction hypotheses, since $(\lambda y{:}b'.r')\,\underline{\hat{u}} =_\beta [\underline{\hat{u}}/y]r'$. $\square$

In Sec. 3.3 we will show that hereditary substitution is complete, i. e., returns a normal form, if run on normal forms with sufficient fuel. For well-typed terms, the type of the substituted variable provides sufficient fuel. For non-well-typed terms, no amount of fuel might be sufficient; consider $[\lambda x.\, x\,x/x]^a (x\,x)$, which has no normal form.

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.

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

The normalizer is structurally recursive in its argument, thus, once we have established termination of hereditary substitution, its termination is trivial.

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

Proof
By induction on t, using termination and soundness of hereditary substitution. $\square$

Example 5 (*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.

$$\frac{(x\!:\!a) \in \Gamma}{\Gamma \vdash x \Downarrow a} \qquad \frac{\Gamma \vdash n \Downarrow a \to b \qquad \Gamma \vdash s \Downarrow a}{\Gamma \vdash n\,s \Downarrow b}\ n \text{ not a } \lambda \qquad \frac{\Gamma, x\!:\!a \vdash r \Downarrow b}{\Gamma \vdash \lambda x\!:\!a.r \Downarrow a \to b}$$

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

Lemma 4 (*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 $[s/x]^a t = r^c$, or r is also neutral and $[s/x]^a t = r$. Otherwise, if t is not neutral, $[s/x]^a t = r$. 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 $\hat{u} = [s/x]^a u$. If $[s/x]^a n = \hat{r}$ and $\underline{\hat{r}}$ is neutral, then $\Gamma \vdash \hat{r}\,\hat{u} : c$ follows easily by induction hypothesis. Otherwise, $[s/x]^a n = (\lambda y : b.r')^{b \to c}$. By the invariant, $|b| < |b \to c| \leq a$, and we can apply the induction hypothesis to infer $\Gamma \vdash [\hat{u}/y]^b r' \Downarrow c$, which is by definition equivalent to $\Gamma \vdash [s/x]^a(n\,u) \Downarrow c$. $\square$

Theorem 1 (*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. $\square$

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 Bellegarde and Hook (Bellegarde & Hook, 1994), Altenkirch and Reus (1999) and Bird and Paterson (1999), untyped λ-terms over the set of free variables A can be implemented by a heterogeneous 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 uses *types* A as indices of the family $\mathsf{Tm}\,A$, substitution $[s/x]t$ for a single variable x is most elegantly obtained as an instance of 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 $\mathsf{Tm}\,B$.[5]

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:

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

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:

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

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

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

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 (see also Appendix A). 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 5 (*Invariant*)
If $t!\rho = r^c$, then $|c| \leq |\rho|$.

Proof
By induction on t. $\square$

Lemma 6 (*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. $\square$

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

5 Implementation in $\mathsf{F}_\omega^{\frown}$

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

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

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

The definition of Ty forces the types of the constructors o and arr. The size index $\imath$ in $\mathsf{Ty}^\imath$ implements the structural measure on simple types. The requirements $|b| < |\mathsf{arr}\,b\,c|$ and $|c| \leq |\mathsf{arr}\,b\,c|$ hold since $b, c : \mathsf{Ty}^\imath$ implies $\mathsf{arr}\,b\,c : \mathsf{Ty}^{\imath+1}$.

Remark 2

If we choose to implement types as (naked) rose trees, $\mathsf{Ty}^\imath = \mu^\imath \lambda X.\, \mathsf{List}^\infty X$, the constructors o and arr, defined as above, now receive the typing:

$$\begin{array}{lcl} \mathsf{o} & : & \forall \imath.\, \mathsf{Ty}^{\imath+1} \\ \mathsf{arr} & : & \forall \imath.\, \mathsf{Ty}^{\imath} \to \mathsf{Ty}^{\imath+1} \to \mathsf{Ty}^{\imath+1}. \end{array}$$

To see this, observe that $\mathsf{Ty}^{\imath+1} = \mathsf{List}^\infty \mathsf{Ty}^\imath$ and $\mathsf{o} = \mathsf{nil}$ and $\mathsf{arr} = \mathsf{cons}$. The resulting measure is an upper bound on the *order* of a type and fulfills the requirements as well. However, I do not see how one could code $\mathsf{Ty}^\imath$ such that $\imath$ would be an upper bound on the number of arr-nodes.

5.1 De Bruijn Terms as a Sized Heterogeneous Data Type

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

$$\begin{array}{lcl} \mathsf{Tm} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\ \mathsf{Tm} & := & \lambda \imath.\, \mu^\imath_{* \xrightarrow{+} *} \lambda X \lambda A.\; A + (X\,A \times X\,A + \mathsf{Ty}^\infty \times X\,(1 + A)) \\[1ex] \mathsf{var} & : & \forall \imath \forall A.\, A \to \mathsf{Tm}^{\imath+1}\,A \\ \mathsf{var} & := & \lambda x.\, \mathsf{inl}\,x \\[1ex] \mathsf{app} & : & \forall \imath \forall A.\, \mathsf{Tm}^\imath\,A \to \mathsf{Tm}^\imath\,A \to \mathsf{Tm}^{\imath+1}\,A \\ \mathsf{app} & := & \lambda r \lambda s.\, \mathsf{inr}\,(\mathsf{inl}\,\langle r, s\rangle) \\[1ex] \mathsf{abs} & : & \forall \imath \forall A.\, \mathsf{Ty}^\infty \to \mathsf{Tm}^\imath\,(1 + A) \to \mathsf{Tm}^{\imath+1}\,A \\ \mathsf{abs} & := & \lambda a \lambda r.\, \mathsf{inr}\,(\mathsf{inr}\,\langle a, r\rangle) \end{array}$$

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

Lifting the free de Bruijn indices of a term t by one is implemented as $\mathsf{map}_{\mathsf{Tm}}\,\mathsf{inr}$ where $\mathsf{map}_{\mathsf{Tm}}$ is the functorial action of Tm.

$$\begin{array}{lcl} \mathsf{map}_{\mathsf{Tm}} & : & \forall \imath \forall A \forall B.\, (A \to B) \to \mathsf{Tm}^\imath A \to \mathsf{Tm}^\imath B \\ \mathsf{map}_{\mathsf{Tm}} & := & \lambda f \lambda r.\, \mathsf{map}'_{\mathsf{Tm}}\,r\,f \\[2ex] \mathsf{map}'_{\mathsf{Tm}} & : & \forall \imath \forall A.\, \mathsf{Tm}^\imath A \to \forall B.\, (A \to B) \to \mathsf{Tm}^\imath B \\ \mathsf{map}'_{\mathsf{Tm}} & := & \mathsf{fix}\,\lambda \mathit{map}' \lambda t \lambda f.\; \mathsf{match}\;t\;\mathsf{with} \\ & & \quad \begin{array}{lcl} \mathsf{var}\;x & \mapsto & \mathsf{var}\;(f\,x) \\ \mathsf{app}\;r\,s & \mapsto & \mathsf{app}\;(\mathit{map}'\,r\,f)\,(\mathit{map}'\,s\,f) \\ \mathsf{abs}\;a\,r & \mapsto & \mathsf{abs}\;a\,(\mathit{map}'\,r\,(\mathsf{map}_{\mathsf{Maybe}}\,f)) \end{array} \\[2ex] \mathsf{map}_{\mathsf{Maybe}} & : & \forall A \forall B.\; (A \to B) \to (1 + A \to 1 + B) \\ \mathsf{map}_{\mathsf{Maybe}} & := & \lambda f \lambda t.\, \mathsf{match}\;t\;\mathsf{with} \\ & & \quad \begin{array}{lcl} \mathsf{inl}\;\langle\rangle & \mapsto & \mathsf{inl}\;\langle\rangle \\ \mathsf{inr}\;x & \mapsto & \mathsf{inr}\;(f\,x) \end{array} \\[2ex] \mathsf{lift}_{\mathsf{Tm}} & : & \forall \imath \forall A.\, \mathsf{Tm}^\imath A \to \mathsf{Tm}^\imath(1 + A) \\ \mathsf{lift}_{\mathsf{Tm}} & := & \mathsf{map}_{\mathsf{Tm}}\,\mathsf{inr} \end{array}$$

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

5.2 Implementation of Hereditary Simultaneous Substitution

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.

$$\begin{array}{lcl}
\mathsf{Res} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Res} & := & \lambda\imath\lambda A.\, \mathsf{Tm}^\infty\, A \times (1 + \mathsf{Ty}^\imath) \\[4pt]
\mathsf{ne}_{\mathsf{Res}} & : & \forall\imath.\, \mathsf{Tm}^\infty\, A \to \mathsf{Res}^\imath\, A \\
\mathsf{ne}_{\mathsf{Res}} & := & \lambda r.\, \langle r,\ \mathsf{inl}\, \langle\rangle\rangle \\[4pt]
\mathsf{nf}_{\mathsf{Res}} & : & \forall\imath.\, \mathsf{Tm}^\infty\, A \to \mathsf{Ty}^\imath \to \mathsf{Res}^\imath\, A \\
\mathsf{nf}_{\mathsf{Res}} & := & \lambda r\lambda a.\, \langle r,\ \mathsf{inr}\, a\rangle
\end{array}$$

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

$$\begin{array}{lcl}
\mathsf{tm} & : & \forall\imath\forall A.\, \mathsf{Res}^\imath A \to \mathsf{Tm}^\infty\, A \\
\mathsf{tm} & := & \lambda\langle r, a\rangle.\, r \\[4pt]
\mathsf{lift}_{\mathsf{Res}} & : & \forall\imath\forall A.\, \mathsf{Res}^\imath\, A \to \mathsf{Res}^\imath\, (1 + A) \\
\mathsf{lift}_{\mathsf{Res}} & := & \lambda\langle r, a\rangle.\, \langle \mathsf{lift}_{\mathsf{Tm}}\, r,\ a\rangle
\end{array}$$

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:

$$\begin{array}{lcl}
\mathsf{var}_{\mathsf{Res}} & : & \forall\imath\forall A.\, A \to \mathsf{Res}^\imath\, A \\
\mathsf{var}_{\mathsf{Res}} & := & \lambda a.\, \mathsf{ne}_{\mathsf{Res}}\, (\mathsf{var}\, a) \\[4pt]
\mathsf{abs}_{\mathsf{Res}} & : & \forall\imath\forall A.\, \mathsf{Ty}^\infty \to \mathsf{Res}^\infty (1 + A) \to \mathsf{Res}^\imath A \\
\mathsf{abs}_{\mathsf{Res}} & := & \lambda a\lambda r.\, \mathsf{ne}_{\mathsf{Res}}\, (\mathsf{abs}\, a\, (\mathsf{tm}\, r)) \\[4pt]
\mathsf{app}_{\mathsf{Res}} & : & \forall A.\, \mathsf{Res}^\infty\, A \to \mathsf{Res}^\infty\, A \to \mathsf{Res}^\imath\, A \\
\mathsf{app}_{\mathsf{Res}} & := & \lambda t\lambda u.\, \mathsf{ne}_{\mathsf{Res}}\, (\mathsf{app}\, (\mathsf{tm}\, t)\, (\mathsf{tm}\, u))
\end{array}$$

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 $|\rho|$.

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

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 $\mathsf{lift}_{\mathsf{Val}}\,\rho$ and lookup of variable x in ρ by $\mathsf{lookup}_{\mathsf{Val}}$. Other implementations of valuations are possible, see Appendix A.

For implementing hereditary substitutions, we have to take into account the limitations of recursion in $\mathsf{F}_{\omega}^{\widehat{}}$. 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 $[s/x]^{a} r$ 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.

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

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

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

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

This explains why in the type of the inner function, we have used size index $\imath + 1$ for Val and Res. The inner function is defined by induction on $\jmath$. 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 $\mathsf{lookup}_{\mathsf{Val}}\, \rho\, x$, 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 $\mathsf{nf}_{\mathsf{Res}}\, (\ldots)\, c$ is well-typed.

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

```
norm :  $\forall \imath \forall A.\, \mathsf{Tm}^{\imath} A \to \mathsf{Tm}^{\infty} A$
norm := fix $\lambda norm \lambda t.$ match $t$ with
          var $x$    $\mapsto$ var $x$
          abs $a\, r$  $\mapsto$ abs $a\, (norm\, r)$
          app $r\, s$  $\mapsto$ let $r' = norm\, r$
                          $s' = norm\, s$
                    in match $r'$ with  abs $a\, t'$ $\mapsto$ subst $a\, s'\, t'$
                                        _       $\mapsto$ app $r'\, s'$
```

6 Discussion and Related Work

We have seen an interesting example for certifying termination with sized types. One may wonder, is this normalizer extensible beyond the simply typed lambda-calculus, and are sized types strictly necessary to implement terminating hereditary substitutions? The answer to the first question is, *yes, but not substantially.* One can add product and sum types and probably control operators such as the μ in the $\lambda\mu$-calculus (David & Nour, 2005). Yet System T is clearly out of reach, since the termination orderings are well beyond lexicographic. With respect to the second question: one can obtain a structurally recursive implementation of hereditary substitutions if one uses a spine representation of λ-terms: $r, s, t ::= x\, \vec{t} \mid \lambda x : a.\, t \mid r\, s$. This was pointed out to me by one of the referees. However, as we have seen, sized types offer more flexibility and means to abstract: we can always replace a piece of code by something different with the same type and maintain termination. For instance, we have factored out the implementation of valuations from the implementation of simultaneous substitutions, so now we can choose a completely different representation.

Terminating normalizers for object languages like System T or System F (Altenkirch, 1993) can be implemented in dependently typed meta-languages of appropriate strength, like Coq (2007) which is based on a type theory called the *Calculus of Inductive Constructions.* There have been proposals to integrate sized types into the type theory (Barthe *et al.*, 2006; Blanqui, 2005) to check termination of recursive definitions behind the scenes. In $\mathsf{F}_{\widehat{\omega}}$, sizes are *first-class*, i. e., one can speak about non-size-increasing functions like filter and pass them as parameters to other functions. Of course, dependent type theories already feature indexed types, thus, one can simulate sized types to a certain extent. E. g., sized types requiring only sizes $< \omega$ can be simulated by an inductive family indexed by a natural number, and size ω through an existential type. However, subtyping sized types is then not available, one has to insert explicit coercions instead, and the conveniences of a size ∞ are lost.[6] A lightweight integration of sized types into dependent types is still a topic of research.

We continue with a review of some related work.

Type-based termination and sized types. Mendler (1991) 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 (1998) and Barthe et al. (2004; 2005) 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 (Abel, 2006c)—now functions like breadth-first traversal are recognized as terminating by the type system. In my thesis (Abel, 2006b), I cover also heterogeneous types like this article's Tm. Blanqui (2004; 2005) extended type-based termination to dependent types and rewriting.

Independently of these "type-theoretic" developments, Hughes, Pareto, and Sabry (Hughes *et al.*, 1996; Pareto, 2000) 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 (Lane, 1971; Manes, 1976). For terms with binders, the idea was taken up by Bird and Paterson (1999) as a case study for heterogeneous types in Haskell. Altenkirch and Reus (1999) 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 (2006) stratifies this situation by exhibiting a common scheme with 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 (2006) 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,

[6] Implementing $\mathsf{arr}' : \mathsf{Ty}^\infty \to \mathsf{Ty}^\infty \to \mathsf{Ty}^\infty$ as $\mathsf{arr}' : (\exists\imath : \mathbb{N}.\mathsf{Ty}^\imath) \to (\exists\imath : \mathbb{N}.\mathsf{Ty}^\imath) \to (\exists\imath : \mathbb{N}.\mathsf{Ty}^\imath)$ involves computation of a maximum.

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 (2003) 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 (2006) implements the weak normalization proof in Isabelle and automatically extracts a normalizer, being much more complicated and keeping some intermediate data structures. I have implemented a similar proof in Twelf (Abel, 2004b). The termination checker of Twelf is capable to certify termination of the normalization proof, if viewed as a recursive function.

Watkins et al. (2003) define hereditary substitutions for terms of the logical framework LF, and its extensions to linearity and concurrency. These substitutions proved to be 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; see also the tutorial on LF by Harper and Licata (2007).

7 Conclusion

We have presented $\mathsf{F}_\omega^{\frown}$, 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 $\mathsf{F}_\omega^{\frown}$. The termination of the normalizer is statically ensured in $\mathsf{F}_\omega^{\frown}$. Additionally, the normalizer is well-scoped, by virtue of an implementation of λ-terms using a heterogeneous type.

One can 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 (2006) has done a first step and implemented type-preserving renaming and substitution for the object language using a dependently-typed meta language. Pursuing this approach, one can 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 are then visible only in the meta-level typing of the normalizer. The normalizer, viewed purely operationally, has the possibility to diverge on ill-typed terms, but the static well-typedness condition guarantees its termination for all actual inputs.[7]

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

[7] One referee has sent me Agda code which implements the type-preserving normalizer I describe here.

A Variation

In this section, we consider an alternative representation of a single substitution for de Bruijn terms. It will turn out that the alternative is not so different to what we had already; substitution $[s/i]t$ of a term s for a free de Bruijn index i in t is just an instance of simultaneous substitution $t\rho$, using a special representation of the valuation ρ.

Consider λ-terms as given by the grammar $\mathsf{Tm} \ni r, s, t ::= i \mid \lambda t \mid r\, s$, where $i \in \mathbb{N}$ is a de Bruijn index, and λt binds index 0 in t. The set of free indices and the lifting operation $\uparrow t$, which increases each free index by 1, shall be defined as usual.

Let $\rho \in \mathbb{N} \to \mathsf{Tm}$. Simultaneous substitution $t\rho$ is given by the three equations:

$$\begin{array}{lcl} i\rho & = & \rho(i) \\ (\lambda t)\rho & = & \lambda.\, t(\Uparrow \rho) \\ (r\, s)\rho & = & (r\rho)\,(s\rho) \end{array}$$

Herein, $(\Uparrow \rho)(0) = 0$ and $(\Uparrow \rho)(i+1) = \uparrow(\rho(i))$. Substitution $[s/i]t$ for a single index i is an instance of simultaneous substitution $t\rho$ with

$$\rho(j) = \begin{cases} s & \text{if } j = i \\ j & \text{if } j < i \\ j-1 & \text{if } j > i. \end{cases}$$

There is also a direct implementation of substitution $[s/i]t$ for a single index:

$$\begin{array}{lcl} [s/i]j & = & \begin{cases} s & \text{if } j = i \\ j & \text{if } j < i \\ j-1 & \text{if } j > i \end{cases} \\ [s/i](\lambda t) & = & \lambda.\, [\uparrow s/i+1]t \\ [s/i](t\, u) & = & ([s/i]t)\,([s/i]u) \end{array}$$

However, this algorithm differs not much from the instance of simultaneous substitution we had before. It just uses a representation of the valuation ρ as a pair (s, i), with lookup $\rho(j)$ and lifting $\Uparrow \rho$ defined accordingly.

We will now investigate how this implementation of a single substitution carries over to our representation of de Bruijn terms as a heterogeneous datatype. The operations $<$, $>$, and -1 cannot be implemented in $\mathsf{F}_\omega^{\widehat{}}$ on our type of variables, which has a shape of the form $1 + \cdots + 1 + A$; we would need advanced type-level programming features such type classes or inductive kinds. But we can massage our implementation of singleton valuations into a form which is implementable in $\mathsf{F}_\omega^{\widehat{}}$.

The new representation of a singleton valuation is a pair (s, ϕ) where $\phi \in \mathbb{N} \to (\mathbb{N} \cup \{*\})$, and we define lookup and lifting as follows:

$$(s, \phi)(j) \quad = \quad \begin{cases} s & \text{if } \phi(j) = * \\ \phi(j) & \text{otherwise} \end{cases}$$

$$\begin{array}{lclll} \Uparrow(s, \phi) & = & (\uparrow s, \phi') & \text{where } \phi'(0) & = \quad 0 \\ & & & \phi'(j+1) & = \quad \begin{cases} * & \text{if } \phi(j) = * \\ \phi(j) + 1 & \text{otherwise} \end{cases} \end{array}$$

Substitution $[s/0]t$ for the index 0 is then obtained by $t\rho$ with $\rho = (s, \phi)$, $\phi(0) = *$ and $\phi(j+1) = j$.

This representation of singleton valuations can also be used with the representation of de Bruijn terms as a heterogeneous type in Section 5. Then, $\phi(j) = *$ is represented as $\phi(y) = \mathsf{inl}\langle\rangle$. If y is not the one variable which is assigned to $\hat{s}$ in the singleton valuation, then $\phi(y) = \mathsf{inr}\, y'$ for some variable y' which is either identical to y or the variable just below y (thus, we have $y \in \{y',\ \mathsf{inr}\, y'\}$). A singleton valuation is defined for *at least* one variable, thus, the domain of ϕ is $1 + A$.

$$
\begin{array}{lll}
\mathsf{SgVal} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{\circ} * \\
\mathsf{SgVal} & := & \lambda\imath\lambda A.\, \mathsf{Tm}^{\infty} A \times \mathsf{Ty}^{\imath} \times (1 + A \to 1 + A) \\
\\
\mathsf{lookup}_{\mathsf{Val}} & : & \forall\imath\forall A.\, \mathsf{SgVal}^{\imath} A \to 1 + A \to \mathsf{Res}^{\imath} A \\
\mathsf{lookup}_{\mathsf{Val}} & := & \lambda\langle s, a, \phi\rangle\lambda y.\ \mathsf{match}\ \phi\, y\ \mathsf{with} \\
 & & \quad \mathsf{inl}\,\langle\rangle \quad \mapsto \quad \mathsf{nf}_{\mathsf{Res}}\, s\, a \\
 & & \quad \mathsf{inr}\, y' \quad \mapsto \quad \mathsf{ne}_{\mathsf{Res}}\, (\mathsf{var}\, y') \\
\\
\mathsf{sg}_{\mathsf{Val}} & : & \forall\imath\forall A.\, \mathsf{Tm}^{\infty} A \to \mathsf{Ty}^{\imath} \to \mathsf{SgVal}^{\imath}\, A \\
\mathsf{sg}_{\mathsf{Val}} & := & \lambda s\lambda a.\, \langle s,\ a,\ \lambda y.y\rangle \\
\\
\mathsf{lift}_{\mathsf{Val}} & : & \forall\imath\forall A.\, \mathsf{SgVal}^{\imath}\, A \to \mathsf{SgVal}^{\imath}\, (1 + A) \\
\mathsf{lift}_{\mathsf{Val}} & := & \lambda\langle s, a, \phi\rangle.\, \langle \mathsf{lift}_{\mathsf{Tm}}\, s,\ a,\ \phi'\rangle \\
 & & \mathsf{where}\ \ \phi' := \lambda y.\ \mathsf{match}\ y\ \mathsf{with} \\
 & & \qquad\qquad\qquad \mathsf{inl}\,\langle\rangle \quad \mapsto \quad \mathsf{inl}\,\langle\rangle \\
 & & \qquad\qquad\qquad \mathsf{inr}\, y' \quad \mapsto \quad \mathsf{map}_{\mathsf{Maybe}}\, \mathsf{inr}\, (\phi\, y')
\end{array}
$$

The code for subst can be reused, only simsubst now receives the less general type

$$\mathsf{simsubst} : \forall\jmath.\ \forall A.\, \mathsf{Tm}^{\jmath}(1 + A) \to \mathsf{SgVal}^{\imath+1}\, A \to \mathsf{Res}^{\imath+1} A.$$

By parametricity for the type of simsubst, SgVal needs to have a functional component with a domain that mentions A positively. Thus, no fundamentally different implementations of substitution are possible for the chosen, type-indexed representation of de Bruijn terms.

References

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

Abel, Andreas. (2004b). Weak normalization for the simply-typed lambda-calculus in Twelf. *Logical Frameworks and Metalanguages (LFM 2004), IJCAR Satellite Workshop, Cork, Ireland.*

Abel, Andreas. (2006a). Implementing a normalizer using sized heterogeneous types. McBride, Connor, & Uustalu, Tarmo (eds), *Workshop on Mathematically Structured Functional Programming, MSFP 2006, Kuressaare, Estonia, July 2, 2006.* electronic Workshop in Computing (eWiC). The British Computer Society (BCS).

Abel, Andreas. (2006b). *A polymorphic lambda-calculus with sized higher-order types.* Ph.D. thesis, Ludwig-Maximilians-Universität München.

Abel, Andreas. (2006c). Semi-continuous sized types and termination. *Pages 72–88 of:* Ésik, Zoltán (ed), *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 21-24, 2006, Proceedings*. Lecture Notes in Computer Science, vol. 4207. Springer-Verlag.

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

Altenkirch, Thorsten. (1993). A formalization of the strong normalization proof for System F in LEGO. *Pages 13–28 of:* Bezem, M., & Groote, J. F. (eds), *Typed Lambda Calculi and Applications, TLCA'93*. Lecture Notes in Computer Science, vol. 664. Springer-Verlag.

Altenkirch, Thorsten, & Reus, Bernhard. (1999). Monadic presentations of lambda terms using generalized inductive types. *Pages 453–468 of:* Flum, Jörg, & Rodríguez-Artalejo, Mario (eds), *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*. Lecture Notes in Computer Science, vol. 1683. Springer-Verlag.

Amadio, Roberto M., & Coupet-Grimal, Solange. (1998). Analysis of a guard condition in type theory (extended abstract). *Pages 48–62 of:* Nivat, Maurice (ed), *Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Lecture Notes in Computer Science, vol. 1378. Springer-Verlag.

Barthe, Gilles, Frade, Maria J., Giménez, Eduardo, Pinto, Luis, & Uustalu, Tarmo. (2004). Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, **14**(1), 97–141.

Barthe, Gilles, Grégoire, Benjamin, & Pastawski, Fernando. (2005). Practical inference for type-based termination in a polymorphic setting. *Pages 71–85 of:* Urzyczyn, Pawel (ed), *Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan*. Lecture Notes in Computer Science, vol. 3461. Springer-Verlag.

Barthe, Gilles, Grégoire, Benjamin, & Pastawski, Fernando. (2006). CICˆ: Type-based termination of recursive definitions in the Calculus of Inductive Constructions. *Pages 257–271 of:* Hermann, Miki, & Voronkov, Andrei (eds), *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*. Lecture Notes in Computer Science, vol. 4246. Springer-Verlag.

Bellegarde, Françoise, & Hook, James. (1994). Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, **23**(2-3), 287–311.

Berghofer, Stefan. (2006). Extracting a normalization algorithm in Isabelle/HOL. *Pages 50–65 of:* Filliâtre, Jean-Christophe, Paulin-Mohring, Christine, & Werner, Benjamin (eds), *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 3839. Springer-Verlag.

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

Blanqui, Frédéric. (2004). A type-based termination criterion for dependently-typed higher-order rewrite systems. *Pages 24–39 of:* van Oostrom, Vincent (ed), *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Ger-*

many, June 3 – 5, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3091. Springer-Verlag.

Blanqui, Frédéric. (2005). Decidability of type-checking in the Calculus of Algebraic Constructions with size annotations. *Pages 135–150 of:* Ong, C.-H. Luke (ed), *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings.* Lecture Notes in Computer Science, vol. 3634. Springer-Verlag.

David, René, & Nour, Karim. (2005). Arithmetical proofs of strong normalization results for the symmetric lambda-mu-calculus. *Pages 162–178 of:* Urzyczyn, Pawel (ed), *Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan.* Lecture Notes in Computer Science, vol. 3461. Springer-Verlag.

Gentzen, Gerhard. (1935). Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, **39**, 176–210, 405–431. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Giménez, Eduardo. (1998). Structural recursive definitions in type theory. *Pages 397–408 of:* Larsen, K. G., Skyum, S., & Winskel, G. (eds), *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings.* Lecture Notes in Computer Science, vol. 1443. Springer-Verlag.

Harper, Robert, & Licata, Daniel. (2007). Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, **17**(4–5), 613–673.

Hughes, John, Pareto, Lars, & Sabry, Amr. (1996). Proving the correctness of reactive systems using sized types. *Pages 410–423 of: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96.*

INRIA. (2007). *The coq proof assistant, version 8.1.* INRIA. http://coq.inria.fr/.

Joachimski, Felix, & Matthes, Ralph. (2003). Short proofs of normalization. *Archive of Mathematical Logic*, **42**(1), 59–87.

Lane, Saunders Mac. (1971). *Categories for the working mathematician.* Springer-Verlag.

Manes, Ernest G. (1976). *Algebraic theories.* Springer-Verlag.

McBride, Conor. (2006). *Type-preserving renaming and substitution.* Functional Pearl. Draft.

Mendler, Nax Paul. (1987). Recursive types and type constraints in second-order lambda calculus. *Pages 30–36 of: Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, N.Y.* IEEE Computer Society Press.

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

Pareto, Lars. (2000). *Types for crash prevention.* Ph.D. thesis, Chalmers University of Technology.

Prawitz, Dag. (1965). *Natural deduction.* Almqvist & Wiksell, Stockholm. Republication by Dover Publications Inc., Mineola, New York, 2006.

Tait, William W. (1967). Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, **32**(2), 198–212.

Watkins, Kevin, Cervesato, Iliano, Pfenning, Frank, & Walker, David. (2003). *A concurrent logical framework I: Judgements and properties.* Tech. rept. School of Computer Science, Carnegie Mellon University, Pittsburgh.