RAIRO-Inf. Theor. Appl. **38** (2004) 277-319
DOI: 10.1051/ita:2004015

TERMINATION CHECKING WITH TYPES *

Andreas Abel[1]

Abstract. The paradigm of type-based termination is explored for functional programming with recursive data types. The article introduces $\mathbf{\Lambda}^{+}_{\boldsymbol{\mu}}$, a lambda-calculus with recursion, inductive types, subtyping and bounded quantification. Decorated type variables representing approximations of inductive types are used to track the size of function arguments and return values. The system is shown to be type safe and strongly normalizing. The main novelty is a bidirectional type checking algorithm whose soundness is established formally.

Mathematics Subject Classification. 68N15, 68N18, 68Q42.

Introduction

In interactive theorem provers like Alfa [26], Coq [29], or LEGO [33, 43], which are based on the Curry-Howard isomorphism, inductive proofs can be supplied as recursive functions. However, only functions which terminate on all inputs constitute valid proofs. In functional programming, functions are commonly defined *via* general recursion and pattern matching. This imposes some challenge on proof validation and, in general, it is undecidable whether a recursive program terminates.

Many termination checkers which analyze untyped program code follow methods from term rewriting and rely on term orderings. In previous work [1], we used the subterm ordering extended to higher-order functions to capture the class of structurally recursive functions over strictly positive inductive datatypes. For

Keywords and phrases. Type-based termination, sized types, inductive types, course-of-value recursion, bidirectional type checking, strong normalization.

* *Research supported by the Graduiertenkolleg Logik in der Informatik (Ph.D. Program Logic in Computer Science) of the Deutsche Forschungsgemeinschaft, the thematic networks TYPES (IST-1999-29001) and Applied Semantics II (IST-2001-38957) of the European Union and the project CoVer of the Swedish Foundation of Strategic Research.*

[1] Department of Computer Science, Chalmers University of Technology, Rännvägen 6, 41296 Göteborg, Sweden; e-mail: `abel@cs.chalmers.se`

polynomial inductive types, a greater class of functions is accepted in Telford and Turner's ESFP [46]. Their termination checker also incorporates a limited form of size-change and dataflow analysis which recognizes certain functions as *reducers* or *preservers*. Functions in these classes – first described by Walther [48] – have the property that the size of their output is bounded by the size of some input argument (strictly smaller in the case of reducers). Finally, Pientka [40] has implemented termination and reduction checking for higher-order logic programs based on the subterm ordering.

Such termination checkers, using syntactical conditions, have some drawbacks. First, for non-strictly positive inductive types it is not clear how a syntactic criterion should look like (see Sect. 7). Secondly, the acceptance of a program is often sensitive to small changes in the code (introduction of redexes, substitution of expressions by others of equal value).

Termination checking should abstract from the precise syntactical formulation of an algorithm; therefore Giménez [23] advocates a *type-based* termination method. The idea is to equip the types of recursive data structures with size information. A recursive function is only accepted if the sizes of arguments to recursive calls are bounded by the size of its inputs. Being an abstract property like *type*, the size information is insensitive to small reformulations of expressions. A nice feature of sized types is that function types can express whether functions are size-preserving. Hence, termination checking scales to higher-order languages which are problematic for purely syntactic methods.

Giménez was not the first to describe type-based termination; Hughes, Pareto and Sabry [28] described sized types for reactive functional programming, also in combination with region types [27, 37]. Amadio and Coupet-Grimal [7] used guarded types to ensure productivity of infinite structures like streams. Although these ideas have been around for some time, people are reluctant to pick up on them, probably because they shun changes in their type theories.

The aim of this article is to provide a gentle access to *sized types for termination* in order to increase their popularity. Therefore we restrict Giménez [23] proposal – sized types for the calculus of constructions – to a simply-typed setting, ignore coinductive types for now and confine subtyping to the necessary minimum. Once type-based termination is understood at this simple level, one can go ahead and upgrade to rich type theories with full subtyping, polymorphism or even dependent types.

For the system presented, we show type safety syntactically and termination semantically, by modeling types as sets of strongly normalizing terms. Independently, Barthe *et al.* [9] have defined a similar restriction of Giménez original proposal and proven subject reduction and strong normalization. Relative to their work, the novel contributions of this article are:

(1) An informal introduction into type based termination, starting with its simplest form, Mendler's rule for recursion [35], and ending with our restriction of Giménez' system.
(2) A bidirectional type checking algorithm for our calculus. This is the main technical novelty.

(3) A motivation why the result type of a recursive function can only *monotonically* depend on the type of its input. This side condition was missing in Giménez original formulation [23]. We give an example of a non-terminating function which is accepted by the type system if the side condition is dropped.

The article is structured as follows. In Section 1, we introduce a core language $\mathbf{\Lambda_\mu}$ with recursive datatypes and non-terminating functions and show that it has the strong type soundness property. By restricting the type of the fixed-point combinator, one can force recursive functions to be total. In Section 2, we start with Mendler's recursion rule [35], "the mother of type-based termination", and refine it step by step, enlarging the class of accepted functions, until we reach Giménez' rule. The formal presentation of the type system $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ is given in Section 3, together with a type-checking algorithm in Section 4. A proof of strong normalization follows in Section 5. In Section 6, we first discuss the above mentioned side condition. Secondly, we show how to represent productive streams in our type system as functions over natural numbers. Thus, we motivate, from the perspective of inductive types, the rules for sized coinductive types given by Giménez [23] and Barthe *et al.* [9]. Several applications for type-based termination are sketched in Section 7, before we conclude with an summary of related work in Section 8.

1. A core language with recursion

We consider termination in the setting of simply-typed functional programming with recursive datatypes. Table 1 shows the core language $\mathbf{\Lambda_\mu}$, a lambda-calculus with finite sums and products and positive recursive types. By the latter we mean recursive types $\mu X.\sigma$ such that on each walk from the binding site to an occurrence of the variable X we choose the left component of an arrow type an even number of times (see itypes in Sect. 3). The particular selection of type and term formers is inspired by category theory [25].

Throughout the paper we consider α-equivalent terms or types as identical. In all contexts we consider the variable names to be unique. Both properties can be ensured by considering variable names as just a sugar for de Bruijn indices, which we shall do for the whole of this article. We write $[N/x]M$ resp. $[\rho/Y]\tau$ (or $\tau(\rho)$) for capture-avoiding substitution in terms resp. types. We take the liberty to drop the dot in a binder if the expression after a binding consists of a single symbol (*e.g.*, $\lambda x M$, $\mu X \sigma$)[1].

Table 1 lists the rules of the static semantics, the typing judgement $\Gamma \vdash M : \tau$, and the axioms of the dynamic semantics, the computation relation $M \longrightarrow_\beta M'$. In our terminology, a β-redex, *i.e.*, the left hand side of a reduction axiom, shall denote any elimination of a matching introduction. This notion is a generalization of β-redexes in simply-typed λ-calculus to richer type systems. The full reduction relation $\longrightarrow^*$ is obtained by closing the relation $\longrightarrow_\beta$ under all term constructors,

[1] This goes well with the definition of "." which denotes an "opening parenthesis which closes as far to the right as syntactically possible".

TABLE 1. The core functional language $\mathbf{\Lambda_\mu}$.

Types:

$$\rho, \sigma, \tau ::= X \mid 1 \mid \sigma + \tau \mid \sigma \times \tau \mid \sigma \to \tau \mid \mu X.\rho \quad \text{(where } X \text{ appears only positively in } \rho)$$

Terms:

$$\begin{aligned}\mathbf{\Lambda_\mu} \ni M, N ::= {} & x \mid \lambda x.M \mid M_1\, M_2 \mid \mathsf{inl}\, M \mid \mathsf{inr}\, M \mid (\mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, x_1 \Rightarrow M_1 \mid \mathsf{inr}\, x_2 \Rightarrow M_2) \\ & \mid () \mid (M_1, M_2) \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid \mathsf{fold}\, M \mid \mathsf{unfold}\, M \mid \mathsf{fix}\, g(x).M\end{aligned}$$

Contexts (all variables distinct):

$$\Gamma ::= \cdot \mid \Gamma, x:\tau$$

Typing (τ and all types in Γ closed):

$$\Gamma \vdash M : \tau$$

Lambda-calculus:

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x:\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M_1 : \sigma \to \tau \qquad \Gamma \vdash M_2 : \sigma}{\Gamma \vdash M_1\, M_2 : \tau}$$

Sum types:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{inl}\, M : \sigma + \tau} \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathsf{inr}\, M : \sigma + \tau} \qquad \frac{\Gamma \vdash M : \sigma + \tau \qquad \Gamma, x_1:\sigma \vdash M_1 : \rho \qquad \Gamma, x_2:\tau \vdash M_2 : \rho}{\Gamma \vdash \mathsf{case}\, M\, \mathsf{of}\, \mathsf{inl}\, x_1 \Rightarrow M_1 \mid \mathsf{inr}\, x_2 \Rightarrow M_2 : \rho}$$

Product types:

$$\frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash M_1 : \sigma \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash (M_1, M_2) : \sigma \times \tau} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \mathsf{fst}\, M : \sigma} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \mathsf{snd}\, M : \tau}$$

Recursive types and terms:

$$\frac{\Gamma \vdash M : \sigma(\mu X.\sigma)}{\Gamma \vdash \mathsf{fold}\, M : \mu X.\sigma} \qquad \frac{\Gamma \vdash M : \mu X.\sigma}{\Gamma \vdash \mathsf{unfold}\, M : \sigma(\mu X.\sigma)} \qquad \frac{\Gamma,\ g:\sigma \to \tau,\ x:\sigma \vdash M : \tau}{\Gamma \vdash \mathsf{fix}\, g(x).M : \sigma \to \tau}$$

(Neutral) values:

$$\begin{aligned}\mathsf{Ne} \ni U ::= {} & x \mid U\, V \mid \mathsf{fst}\, U \mid \mathsf{snd}\, U \mid (\mathsf{case}\, U\, \mathsf{of}\, \mathsf{inl}\, x_1 \Rightarrow V_1 \mid \mathsf{inr}\, x_2 \Rightarrow V_2) \mid \mathsf{unfold}\, U \\ \mathsf{Val} \ni V ::= {} & U \mid \lambda x.V \mid \mathsf{inl}\, V \mid \mathsf{inr}\, V \mid (V_1, V_2) \mid \mathsf{fold}\, V \mid \mathsf{fix}\, g(x).V\end{aligned}$$

Reduction axioms:

$$\begin{aligned}(\lambda x.M)\, N &\longrightarrow_\beta [N/x]M \\ \mathsf{case}(\mathsf{inl}\, N)\, \mathsf{of}\, \mathsf{inl}\, x_1 \Rightarrow M_1 \mid \mathsf{inr}\, x_2 \Rightarrow M_2 &\longrightarrow_\beta [N/x_1]M_1 \\ \mathsf{case}(\mathsf{inr}\, N)\, \mathsf{of}\, \mathsf{inl}\, x_1 \Rightarrow M_1 \mid \mathsf{inr}\, x_2 \Rightarrow M_2 &\longrightarrow_\beta [N/x_2]M_2 \\ \mathsf{fst}(M_1, M_2) &\longrightarrow_\beta M_1 \\ \mathsf{snd}(M_1, M_2) &\longrightarrow_\beta M_2 \\ \mathsf{unfold}(\mathsf{fold}\, M) &\longrightarrow_\beta M \\ (\mathsf{fix}\, g(x).M)\, N &\longrightarrow_\beta [\mathsf{fix}\, g(x).M/g][N/x]M\end{aligned}$$

Reduction relations:

$\longrightarrow_\beta$	β-reduction (only axioms)
$\longrightarrow$	one-step reduction: closure of $\longrightarrow_\beta$ under all term constructors
$\longrightarrow^+$	transitive closure of $\longrightarrow$
$\longrightarrow^*$	reflexive-transitive closure of $\longrightarrow$
$\longrightarrow^\infty$	$\longrightarrow^*$ or divergence

reflexivity and transitivity. The proposition $M \longrightarrow^{\infty} M'$ shall denote that either $M \longrightarrow^{*} M'$, or M diverges, combining finite and infinite reduction sequences.

Recursive functions can be defined *via* the general fixed-point combinator fix. Note that $\mathsf{fix}\, g(x).M$ binds the two variables g (which stands for the recursively defined function) and x (which denotes the argument to that function) in M. The fixed-point combinator is the source of non-termination in our language, and in this work we will show a method to restrict its use by typing such that termination is regained.

Some significant subset of a functional language with recursive datatypes can be translated into $\mathbf{\Lambda_{\mu}}$. For example, consider the following Haskell program which sums up all elements in a list of natural numbers.

$$
\begin{array}{lll}
\mathsf{data} & \mathsf{Nat} & = \mathsf{Zero} \mid \mathsf{Succ}\ \mathsf{Nat} \\
\mathsf{data} & \mathsf{ListN} & = \mathsf{Nil} \mid \mathsf{Cons}\ \mathsf{Nat}\ \mathsf{ListN} \\
 & & \\
\mathsf{sum} & (\mathsf{Nil}) & = \mathsf{Zero} \\
\mathsf{sum} & (\mathsf{Cons}\ n\ l) & = \mathsf{sum}'\ n \\
\mathsf{where} & \mathsf{sum}'\ (\mathsf{Zero}) & = \mathsf{sum}\ l \\
 & \mathsf{sum}'\ (\mathsf{Succ}\ n') & = \mathsf{Succ}\ (\mathsf{sum}'\ n')
\end{array}
$$

In $\mathbf{\Lambda_{\mu}}$, the defined datatypes are represented by the type expressions $\mathsf{Nat} := \mu X.\ 1 + X$ and $\mathsf{ListN} := \mu Y.\ 1 + \mathsf{Nat} \times Y$. We simulate the Haskell constructors by the following term abbreviations.

$$
\begin{array}{lcll}
\mathsf{Zero} & := & \mathsf{fold}(\mathsf{inl}()) & : \ \mathsf{Nat} \\
\mathsf{Succ} & := & \lambda x.\, \mathsf{fold}(\mathsf{inr}\, x) & : \ \mathsf{Nat} \to \mathsf{Nat} \\
 & & & \\
\mathsf{Nil} & := & \mathsf{fold}(\mathsf{inl}()) & : \ \mathsf{ListN} \\
\mathsf{Cons} & := & \lambda x.\lambda xs.\, \mathsf{fold}(\mathsf{inr}(x, xs)) & : \ \mathsf{Nat} \to \mathsf{ListN} \to \mathsf{ListN}
\end{array}
$$

What our representation of datatypes loses in succinctness, it gains in conceptual clarity: it separates the notion of sum $(+)$ and product $(\times)$ from recursion (μ), which simplifies both the presentation of typing and the interpretation of types (see Sect. 5).

The Haskell program translates into the following $\mathbf{\Lambda_{\mu}}$ code which contains two interleaving uses of fix.

$$
\begin{array}{l}
\mathsf{sum}\ :\ \ \mathsf{ListN} \to \mathsf{Nat} \\
\mathsf{sum}\ :=\ \mathsf{fix}\, f(l).\, \mathsf{case}(\mathsf{unfold}\, l)\ \mathsf{of} \\
\qquad\quad \mathsf{inl}\, u \Rightarrow \mathsf{Zero} \\
\qquad \mid \mathsf{inr}\, p \Rightarrow \big(\mathsf{fix}\, g(n).\, \mathsf{case}(\mathsf{unfold}\, n)\ \mathsf{of} \\
\qquad\qquad\qquad\quad \mathsf{inl}\, u \Rightarrow f(\mathsf{snd}\, p) \\
\qquad\qquad\qquad \mid \mathsf{inr}\, n' \Rightarrow \mathsf{Succ}(g\, n')\big) \\
\qquad\qquad\quad (\mathsf{fst}\, p)
\end{array}
$$

We sketch a syntactical proof of type soundness, also called type safety, following the method of Wright and Felleisen [49] which is explained in detail in Pierce [41].

The type system of $\mathbf{\Lambda_\mu}$ guarantees that the language is free of *junk* terms (or *faulty* terms [49]) such as $\mathsf{fst}\,(\lambda x.x)$ which would cause evaluation to get stuck. By showing that welltypedness is preserved under reduction, we ensure that junk terms never arise during evaluation, and each term M reaches a value V or diverges.

Theorem 1.1 (type preservation and progress)**.** *Let $\Gamma \vdash M : \tau$ be a well-typed term.*

(1) *Types are preserved under reduction: If $M \longrightarrow M'$ then $\Gamma \vdash M' : \tau$.*
(2) *Evaluation can progress: Either $M \in \mathsf{Val}$ or $M \longrightarrow M'$ for some term M'.*

Proof. Both properties can be shown by induction on $\Gamma \vdash M : \tau$. Preservation and progress are text-book results, see Pierce's Exercise 20.2.2 [41]. □

Corollary 1.2 (strong type soundness for $\mathbf{\Lambda_\mu}$)**.** *If $\Gamma \vdash M : \tau$ then $M \longrightarrow^\infty V$ for some value V with $\Gamma \vdash V : \tau$.*

Proof. By coinduction. At this point, we should put forth a more precise definition of possibly infinite reductions $M \longrightarrow^\infty V$. The relation $\longrightarrow^\infty$ is obtained as the *greatest* fixed point $\mathcal{R}$ of the following rules:

$$\frac{}{M\ \mathcal{R}\ M}\ \mathsf{refl} \qquad \frac{M_1 \longrightarrow M_2 \qquad M_2\ \mathcal{R}\ M_3}{M_1\ \mathcal{R}\ M_3}\ \mathsf{step}$$

Note that if we take the least fixed point, we get $\longrightarrow^*$. Taking the greatest fixed point instead of the least adds infinite reduction sequences.

To prove the corollary, we first apply Theorem 1.1(2) to the assumption $\Gamma \vdash M : \tau$. If M is a value, we are done since $V \longrightarrow^\infty V$ by rule refl. Otherwise, $M \longrightarrow M'$ where $\Gamma \vdash M' : \tau$ by Theorem 1.1(1). By coinduction hypothesis, $M' \longrightarrow^\infty V$ for some value V with $\Gamma \vdash V : \tau$. Thus, $M \longrightarrow^\infty V$ by rule step. The use of the coinduction hypothesis is justified by the fact that we used at least one generating rule of $\longrightarrow^\infty$ to produce the goal $M \longrightarrow^\infty V$ *after* appealing to the coinduction hypothesis. This principle is called the *guardedness condition* by Coquand [17]. □

Thus, *well-typed $\mathbf{\Lambda_\mu}$ programs cannot go wrong* (Milner [36]). But they can diverge, like the totally undefined function $\mathsf{fix}\,g(x).g\,x$. In the remainder of the paper, we develop a more restrictive set of typing rules in which only terminating programs are typable.

2. From Mendler iteration to size-preserving recursion

Starting at Mendler's formulation of iteration [35], we stepwise motivate our principle of recursion through size-preserving functions inspired by Giménez [23]. We both give a semantical motivation as well as an interesting application: termination verification of quicksort by our type system.

2.1. Iteration à la Mendler

Let L denote the set of lists of natural numbers and $L \to L$ denote the set of all total functions from lists to lists. Let f be some (possibly partial) function over lists and $n \in \mathbb{N}$ arbitrary. Assume we can prove that *if f is total for lists of length less than n, then f is total for lists of length less than $n+1$*. By induction on n, and since there are no lists of length less than 0, if follows that f is total for all lists. More formally, let $L_n := \{l \in L : |l| < n\}$ be the nth approximation to the set of all lists L. The proof principle can be expressed *via* the following natural deduction rule:

$$\frac{\begin{array}{c}[n \in \mathbb{N}][f \in L_n \to L] \\ \vdots \\ f \in L_{n+1} \to L\end{array}}{f \in L \to L}$$

Mendler [35] first turned this principle into a typing rule for recursive functions which by their typing are bound to terminate. For the case of functions from lists to lists, his rule reads as follows:

$$\frac{\Gamma,\ Y\!:\!\mathsf{type},\ g\!:\!Y \to \mathsf{ListN},\ x\!:\!1 + \mathsf{Nat} \times Y \vdash M : \mathsf{ListN}}{\Gamma \vdash \mathsf{fix}\, g(x).M : \mathsf{ListN} \to \mathsf{ListN}}$$

In the assumptions of the premise, the (fresh) type variable Y stands for some approximation L_n. The to-be-defined recursive function g is assumed to be element of $L_n \to L$. The premise now shows that the function is in $L_{n+1} \to L$. To this end, an argument $x : 1 + \mathsf{Nat} \times Y$ is assumed; its type denotes L_{n+1} in unfolded form. Under these assumptions, the body M of the recursive function is shown to be well-typed. This implies that within M, recursive calls of g can only happen with arguments of type Y, which can only arise as subterms of x, since Y is a type variable. During evaluation of a function call *via* the reduction rule

$$(\mathsf{fix}\, g(x).M)\ (\mathsf{fold}\, N) \longrightarrow [\mathsf{fix}\, g(x).M/g][N/x]M,$$

the type variable Y gets instantiated by ListN, g by the recursive function, and x by the unfolded argument.

A closer look on the general form of Mendler's rule, given in the following, reveals that it types exactly the *iterative* programs (*cf.* Matthes [32], Spławski and Urzyczyn [44]):

$$\frac{\Gamma,\ Y\!:\!\mathsf{type},\ g\!:\!Y \to \tau,\ x\!:\!\sigma(Y)\ \vdash M : \tau}{\Gamma \vdash \mathsf{fix}\, g(x).M : \mu X \sigma \to \tau}$$

Above, we used an instance of this rule with $\mu X\sigma = \tau = \mathsf{ListN}$. Some number-theoretic functions which have a very natural iterative implementation are addition, multiplication and exponentiation. For instance:

$$\begin{array}{lcl} \mathsf{add} & : & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{add} & := & \lambda y.\ \mathsf{fix}\, add(x).\ \mathsf{case}\ x\ \mathsf{of} \\ & & \qquad\qquad \mathsf{inl}\, u \ \Rightarrow\ y \\ & & \qquad\quad \mid\ \mathsf{inr}\, x' \ \Rightarrow\ \mathsf{Succ}(add\, x') \end{array}$$

The definition is well-typed with the following type assignments to the bound variables: $y : \mathsf{Nat}$, $add : Y \to \mathsf{Nat}$, $x : 1 + Y$, $u : 1$ and $x' : Y$.

From now, since we interpret $\mu X\sigma$ as the least fixed-point of the type operator $\sigma(X)$, and since we allow $\mu X\sigma$ only in the domain of *total* functions, we speak of *inductive* types instead of recursive types[2].

2.2. Primitive recursion à la Mendler

Mendler also gave a more expressive rule which types all *primitive recursive* programs [34]. The premise of the recursion rule has an additional hypothesis $i : Y \to \mu X\sigma$ which can be used to convert a predecessor $r : Y$ of the input x into the inductive type $\mu X\sigma$. After conversion, it can either become part of the output or be used as argument to previously defined functions. During reduction, i gets instantiated with the identity $\lambda x x$. We achieve the same gain of expressivity by making Y a *subtype* of $\mu X\sigma$ in the iteration rule.

$$\frac{\Gamma,\ Y \leq \mu X\sigma,\ g : Y \to \tau,\ x : \sigma(Y)\ \vdash M : \tau}{\Gamma \vdash \mathsf{fix}\, g(x).M : \mu X\sigma \to \tau}$$

The reduction rule remains unchanged – but we need to add subsumption to our typing judgement. The following axiom and rule have to be added.

$$\frac{(Y \leq \mu X\sigma) \in \Gamma}{\Gamma \vdash Y \leq \mu X\sigma} \qquad \frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash \rho \leq \sigma}{\Gamma \vdash M : \sigma}$$

To equip subtyping with the standard properties of transitivity and reflexivity (in our case only for types $\rho \leq \mu X\sigma$), we add the following axioms:

$$\frac{(Y \leq \mu X\sigma) \in \Gamma}{\Gamma \vdash Y \leq Y} \qquad \frac{}{\Gamma \vdash \mu X\sigma \leq \mu X\sigma}$$

[2] There is a Curry-Howard-correspondence of – on the programming side – elimination schemes for $\mu X\sigma$ and – on the logical side – induction axioms for inhabitants of $\mu X\sigma$, hence the name "inductive types". For more details on this correspondence, the reader is referred to Spławski and Urzyczyn [44].

Examples of number-theoretic functions which have a natural primitive recursive implementation are predecessor, subtraction and factorial. For instance,

$$\begin{array}{lcl} \mathsf{pred} & : & \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{pred} & := & \mathsf{fix}\, pred(x).\ \mathsf{case}\ x\ \mathsf{of} \\ & & \qquad \mathsf{inl}\, u \ \Rightarrow \mathsf{Zero} \\ & & \qquad \mid \mathsf{inr}\, x' \ \Rightarrow x' \end{array}$$

with types $pred : Y \to \mathsf{Nat}$, $x : 1 + Y$, $u : 1$ and $x' : Y$. Observe that this definition would not have type-checked using the Mendler *iteration* rule, since we need to make use of subtyping to convert $x' : Y$ to type Nat.

2.3. Course-of-value recursion

In our introductory example, the type variable Y denotes some approximation L_n of the set of number lists. The next stage of approximation L_{n+1} is only available in its unfolded version $\sigma(Y)$. To make approximation stages first-class citizens, we follow Giménez [23] and extend our type language by a "next stage" operator $(\cdot)^+$. The inclusions $L_n \subseteq L_{n+1} \subseteq L_{n+2} \subseteq \cdots \subseteq L$ are reflected by a chain of subtyping relations $Y \leq Y^+ \leq Y^{++} \leq \cdots \leq \mu X\sigma$ which can be derived using the additional rules:

$$\frac{\Gamma \vdash \rho \leq \mu X\sigma}{\Gamma \vdash \rho^+ \leq \mu X\sigma} \qquad \frac{\Gamma \vdash \rho \leq \mu X\sigma}{\Gamma \vdash \rho \leq \rho^+} \qquad \frac{\Gamma \vdash \rho \leq \mu X\sigma}{\Gamma \vdash \rho \leq \rho} \qquad \frac{\Gamma \vdash \rho \leq \sigma \qquad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash \rho \leq \tau}$$

Folding and unfolding now must be applicable to inhabitants of approximations of inductive types as well. We replace the original rules from Table 1 by

$$\frac{\Gamma \vdash \rho \leq \mu X\sigma \qquad \Gamma \vdash M : \sigma(\rho)}{\Gamma \vdash \mathsf{fold}\, M : \rho^+} \qquad \frac{\Gamma \vdash \rho \leq \mu X\sigma \qquad \Gamma \vdash M : \rho^+}{\Gamma \vdash \mathsf{unfold}\, M : \sigma(\rho)}$$

The original rules can be derived using the new ones plus the fact that subtyping admits $(\mu X\sigma)^+ \leq \mu X\sigma$. With the new notation for approximation stages the typing rule for recursion reads as follows.

$$\frac{\Gamma,\ Y \leq \mu X\sigma,\ g : Y \to \tau,\ x : Y^+ \ \vdash M : \tau}{\Gamma \vdash \mathsf{fix}\, g(x).M : \mu X\sigma \to \tau}$$

Since the argument x of the recursive function is no longer of the unfolded type $\sigma(Y)$, but inhabits Y^+, we need to substitute the full $\mathsf{fold}\, N$ for x during reduction:

$$(\mathsf{fix}\, g(x).M)\ (\mathsf{fold}\, N) \longrightarrow [\mathsf{fix}\, g(x).M/g][\mathsf{fold}\, N/x]M.$$

Silently, we have turned primitive recursion into course-of-value recursion. This can be seen from the following argument: If we have computed the predecessor $r : Y$ of input $x : Y^+$ within the body M of our recursive function, we can apply the subtyping law $Y \leq Y^+$ on the type of r, unfold r, and compute its predecessor

in turn. Going on like this, we can analyze x arbitrarily deep to obtain subcomponents which all qualify as arguments to recursive calls of g since they are assigned type Y. Course-of-value recursion is useful, for instance, to program *division by two*, or to implement the specification of the Fibonacci numbers directly:

$$
\begin{array}{lcl}
\mathsf{fib} & : & \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{fib} & := & \mathsf{fix}\, \mathit{fib}(x).\ \mathsf{case}\,(\mathsf{unfold}\ x)\ \mathsf{of} \\
& & \qquad\quad \mathsf{inl}\, u \ \Rightarrow\ \mathsf{Zero} \\
& & \qquad \mid \mathsf{inr}\, y \ \Rightarrow\ \mathsf{case}\,(\mathsf{unfold}\ y)\ \mathsf{of} \\
& & \qquad\qquad \mathsf{inl}\, u \ \Rightarrow\ \mathsf{Succ}(\mathsf{Zero}) \\
& & \qquad\qquad \mid \mathsf{inr}\, z \ \Rightarrow\ \mathsf{add}\,(\mathit{fib}\, y)\,(\mathit{fib}\, z)
\end{array}
$$

In this definition, we have the following types for the bound variables: $\mathit{fib} : Y \to \mathsf{Nat}$, $x : Y^+$, $u : 1$, $y : Y \leq Y^+$ and $z : Y$. Since we can cast the type of y to Y^+, we can unfold it again, hence, look deeper into the recursion argument. This is the essence of course-of-value recursion. Note also that in contrast to the fixed-point rules before, the function argument x is not automatically in an unfolded state any more; we need to unfold it manually in order to analyze it.

To be precise, fib is already an instance of the weaker scheme of course-of-value *iteration*, a notion coined by Uustalu and Vene [47]. We obtain a rule for course-of-value iteration by replacing the hypothesis $Y \leq \mu X\sigma$ by $Y_{X.\sigma} \leq Y^+_{X.\sigma}$. The subscript $X.\sigma$ to the variable Y expresses that Y is an approximation type of $\mu X.\sigma$, an information that was implicit in the old hypothesis.

2.4. Recursion *via* size-preserving functions

Some functions like Euclidean division or functional quicksort have a succinct recursive implementation where the argument r to the recursive call is derived from the input argument x through another function. For example, assume a function $\mathsf{pivot}\ a\ xs$ which splits the input list xs into two output lists $(l,\ r)$ where l contains all elements $< a$ and r the remainder. Using pivot, we define $\mathsf{qsapp}\ xs\ ys$ which quick-sorts list xs and prepends it to ys.

$$
\begin{array}{lcl}
\mathsf{qsapp}\ [\,]\ \ \ \ \ \ \ ys & = & ys \\
\mathsf{qsapp}\ (x :: xs)\ ys & = & \mathsf{let}\ (l,\ r) = \mathsf{pivot}\ x\ xs \\
& & \mathsf{in}\ \ \mathsf{qsapp}\ l\ (x :: \mathsf{qsapp}\ r\ ys)
\end{array}
$$

(We use a sugared syntax for Nil and Cons in this example.) This function is defined by recursion on its first argument. However, the arguments l and r of the recursive calls are not derived from the input *directly*, *i.e.*, using only pattern matching. They are connected to xs, a direct subterm of the input, *via* the function pivot. If we know that both output lists $(l,\ r)$ of the application $\mathsf{pivot}\, x\, xs$ are at most as long as the input xs, we can justify that qsapp is defined by course-of-value recursion on the length of its first argument and constitutes a total function. We say (a bit sloppily) that $\mathsf{pivot}\, x$ needs to be a *size-preserving* function[3].

[3] A similar terminology is used by Walther [48] and Pientka [40].

In terms of approximations, $\mathsf{pivot}\,x$ is size preserving if for every input $xs \in L_n$ both output lists l and r are in the same approximation stage L_n as the input xs. More formally, we require $\mathsf{pivot}\,x \in \bigcap_n (L_n \to L_n \times L_n)$. Giménez [23] observed that one can use bounded quantification

$$\mathsf{pivot}\,x \;:\; \forall Y \leq \mathsf{ListN}.\, Y \to Y \times Y$$

to express this semantical property on the syntax level. How can we show a function to be size preserving? Semantically, we can use the following proof by induction ($f = \mathsf{pivot}\,x$):

$$\frac{\begin{array}{c}[f \in L_n \to L_n \times L_n] \\ \vdots \\ f \in L_{n+1} \to L_{n+1} \times L_{n+1}\end{array}}{f \in \bigcap_n (L_n \to L_n \times L_n)}$$

The difference to the proof scheme given in Section 2.1 is that the result type now also mentions approximation stages. We can carry this modification over to the syntactical level by allowing occurrences of Y in the result type τ of recursion. In this article, we restrict to positive occurrences of Y in τ (so do Barthe *et al.* [9]). As we will see in Section 6, negative occurrences can lead to non-termination. The recursion rule now takes the following shape:

$$\frac{\Gamma,\ Y \leq \mu X\sigma,\ g : Y \to \tau(Y),\ x : Y^+ \ \vdash M : \tau(Y^+) \qquad Y \text{ pos. in } \tau(Y)}{\Gamma \vdash \mathsf{fix}\, g(x).M : \forall Y \leq \mu X\sigma.\ Y \to \tau(Y)}$$

It is crucial to keep the quantifier in the conclusion. If its type were just $\mu X\sigma \to \tau(\mu X\sigma)$, we would have lost the information about size relations between the input and the outputs of the recursive function. To use functions defined with this rule, we need a means to instantiate bounded quantification:

$$\frac{\Gamma \vdash \rho \leq \mu X\sigma \qquad \Gamma \vdash M : \forall Y \leq \mu X\sigma.\ Y \to \tau(Y)}{\Gamma \vdash M : \rho \to \tau(\rho)}$$

In Giménez formulation [23], bounded quantification is not part of the type syntax and consequently, instantiation is merged with the fixed-point rule. Size relations can still be used this way, but local recursive functions can then not always be lifted to the top-level to be made available for other functions. For instance, pivot would have to be defined locally within qsapp. Since we have bounded quantification as a first-class citizen, we can even abstract the pivot function out of the qsapp function. We define a function qsapp' such that $\mathsf{qsapp} = \mathsf{qsapp}'\,\mathsf{pivot}$ and assign it the type

$$\mathsf{qsapp}' \;:\; (\forall Y \leq \mathsf{ListN}.\, Y \to Y \times Y) \to \forall Y \leq \mathsf{ListN}.\, Y \to \mathsf{ListN} \to \mathsf{ListN}$$

In the following, we complete the quicksort example and demonstrate that our type system can verify its totality. We give the program in a Haskell-like language

with type annotations, which is more readable than a direct representation in the core language. However, there is a simple one-to-one correspondence between type annotations in the clauses below and the typing rule for fix given above: the principal function argument is assumed to have type Y^+, recursive calls can only occur with arguments of type Y, and the result must have type $\tau(Y^+)$.

$$\mathsf{pivot} \;:\; \mathsf{Nat} \to \forall Y{\leq}\mathsf{ListN}.\; Y \to Y \times Y$$

$$\begin{array}{lll}
\mathsf{pivot}\;\; a\;\; []^{Y^+} & = & ([]^{Y^+},\; []^{Y^+}) \\
\mathsf{pivot}\;\; a\;\; (x :: xs^{Y})^{Y^+} & = & \mathsf{let}\;\; (l^{Y},\; r^{Y}) = \mathsf{pivot}\;\; a\;\; xs\;\; \mathsf{in} \\
 & & \quad \mathsf{if}\;\; x < a\;\; \mathsf{then}\;\; ((x :: l)^{Y^+},\; r^{Y \leq Y^+}) \\
 & & \quad \mathsf{else}\;\; (l^{Y \leq Y^+},\; (x :: r)^{Y^+})
\end{array}$$

$$\mathsf{qsapp} : \forall Y{\leq}\mathsf{ListN}.\, Y \to \mathsf{ListN} \to \mathsf{ListN}$$

$$\begin{array}{lll}
\mathsf{qsapp}\;[]^{Y^+}\;\; ys & = & ys \\
\mathsf{qsapp}\;(x :: xs^{Y})^{Y^+}\;\; ys & = & \mathsf{let}\;\; (l^{Y},\; r^{Y}) = \mathsf{pivot}\;\; x\;\; xs\;\; \mathsf{in} \\
 & & \quad \mathsf{qsapp}\;\; l^{Y}\;\; (x :: \mathsf{qsapp}\;\; r^{Y}\;\; ys)
\end{array}$$

$$\mathsf{quicksort} : \mathsf{ListN} \to \mathsf{ListN}$$

$$\mathsf{quicksort}\;\; l = \mathsf{qsapp}\;\; l\;\; []$$

All annotations in the function bodies can be inferred by the type checking algorithm presented in Section 4, once the types of the functions are given.

3. Type system

In the following we formally enrich the type system of $\mathbf{\Lambda_\mu}$ by approximation types, subtyping, and bounded quantification, as we have explained informally in the last section. This constitutes the language $\mathbf{\Lambda^+_\mu}$. Due to the restricted fix-rule, it types only strongly normalizing terms of $\mathbf{\Lambda_\mu}$.

3.1. Contexts, types and subtyping

Table 2 summarizes types and type-related judgements of $\mathbf{\Lambda^+_\mu}$. The set of raw types is an extension of the types of $\mathbf{\Lambda_\mu}$ by decorations τ^+ and bounded quantifications $\forall Y{\leq}\mu X\sigma.\, Y \to \tau(Y)$. Additionally to term variables x, wellformed contexts Γ can bind constrained type variables $Y{\leq}\mu X\sigma$. As before, all variables bound by Γ are assumed to be distinct. By two judgements, we distinguish two kinds of types:

itype: Datatypes. These are essentially the positive inductive types of $\mathbf{\Lambda_\mu}$. Additionally, they may contain type variables Y which are approximations $Y \leq \mu X\sigma$ of an inductive datatype $\mu X\sigma$. For example, in Section 7.1 we will see the type $Y\,\mathsf{list} = \mu X.\, 1 + Y \times X$ referring to an approximation type Y. Note that the itype-formation rule for $\sigma \to \tau$ swaps positivity and negativity of variables in σ.

TABLE 2. Contexts, types and subtyping.

Raw types:

$$\rho, \sigma, \tau ::= X \mid 1 \mid \sigma + \tau \mid \sigma \times \tau \mid \sigma \to \tau \mid \mu X\sigma \mid \tau^+ \mid \forall Y{\leq}\mu X\sigma.\, Y \to \tau(Y)$$

Judgments:

Γ cxt	Γ is a wellformed context.
$\Gamma; \vec{X}; \vec{X}' \vdash \sigma : \mathsf{itype}$	σ is a datatype with free positive variables $\vec{X}$ and free negative variables $\vec{X}'$.
$\Gamma \vdash \tau : \mathsf{type}$	τ is a wellformed decorated type.
$\Gamma \vdash \rho \leq \sigma$	ρ is a subtype of σ.

Wellformed contexts Γ cxt.

$$\frac{}{\cdot\ \mathsf{cxt}} \qquad \frac{\Gamma \vdash \tau : \mathsf{type}}{\Gamma, x{:}\tau\ \mathsf{cxt}} \qquad \frac{\Gamma; \cdot; \cdot \vdash \mu X\sigma : \mathsf{itype}}{\Gamma, Y{\leq}\mu X\sigma\ \ \mathsf{cxt}}$$

Datatypes $\Gamma; \vec{X}; \vec{Y} \vdash \sigma : \mathsf{itype}$.

$$\frac{\Gamma\ \mathsf{cxt} \qquad (Y{\leq}\mu X\sigma) \in \Gamma}{\Gamma; \vec{X}; \vec{X}' \vdash Y : \mathsf{itype}} \qquad \frac{\Gamma\ \mathsf{cxt}}{\Gamma; \vec{X}; \vec{X}' \vdash 1 : \mathsf{itype}} \qquad \frac{\Gamma; \vec{X}; \vec{X}' \vdash \sigma : \mathsf{itype} \quad \Gamma; \vec{X}; \vec{X}' \vdash \tau : \mathsf{itype}}{\Gamma; \vec{X}; \vec{X}' \vdash \sigma \star \tau : \mathsf{itype}}\ \star \in \{+, \times\}$$

$$\frac{\Gamma\ \mathsf{cxt}}{\Gamma; \vec{X}; \vec{X}' \vdash X_i : \mathsf{itype}} \qquad \frac{\Gamma; \vec{X}'; \vec{X} \vdash \sigma : \mathsf{itype} \qquad \Gamma; \vec{X}; \vec{X}' \vdash \tau : \mathsf{itype}}{\Gamma; \vec{X}; \vec{X}' \vdash \sigma \to \tau : \mathsf{itype}} \qquad \frac{\Gamma; \vec{X}, X; \vec{X}' \vdash \sigma : \mathsf{itype}}{\Gamma; \vec{X}; \vec{X}' \vdash \mu X\sigma : \mathsf{itype}}$$

Subtyping I: $\Gamma \vdash \rho \leq \mu X\sigma$.

$$\frac{\Gamma; \cdot; \cdot \vdash \mu X\sigma : \mathsf{itype}}{\Gamma \vdash \mu X\sigma \leq \mu X\sigma} \qquad \frac{\Gamma\ \mathsf{cxt} \qquad (Y{\leq}\mu X\sigma) \in \Gamma}{\Gamma \vdash Y \leq \mu X\sigma} \qquad \frac{\Gamma \vdash \rho \leq \mu X\sigma}{\Gamma \vdash \rho^+ \leq \mu X\sigma}$$

Wellformed types $\Gamma \vdash \tau : \mathsf{type}$.

$$\frac{\Gamma\ \mathsf{cxt}}{\Gamma \vdash 1 : \mathsf{type}} \qquad \frac{\Gamma \vdash \sigma : \mathsf{type} \qquad \Gamma \vdash \tau : \mathsf{type}}{\Gamma \vdash \sigma \star \tau : \mathsf{type}}\ \star \in \{+, \times, \to\}$$

$$\frac{\Gamma \vdash \rho \leq \mu X\sigma}{\Gamma \vdash \rho : \mathsf{type}} \qquad \frac{\Gamma, Y{\leq}\mu X\sigma \vdash \tau : \mathsf{type}}{\Gamma \vdash \forall Y{\leq}\mu X\sigma.\, \tau : \mathsf{type}}$$

Subtyping II: $\Gamma \vdash \rho \leq \sigma$.

$$\frac{(Y{\leq}\mu X\sigma) \in \Gamma}{\Gamma \vdash Y \leq Y} \qquad \frac{\Gamma \vdash \mu X\sigma \leq \rho}{\Gamma \vdash \mu X\sigma \leq \rho^+} \qquad \frac{\Gamma \vdash Y \leq \rho}{\Gamma \vdash Y \leq \rho^+} \qquad \frac{\Gamma \vdash \tau \leq \rho}{\Gamma \vdash \tau^+ \leq \rho^+}$$

type: (Decorated) types. They can contain the standard type formers 1, $+$, $\times$ and $\to$, inductive types $\mu X\sigma$ and their approximations ρ – this is handled *via* the judgement $\rho \leq \mu X\sigma$ – and bounded quantification. All free variables Y for a wellformed type $\Gamma \vdash \tau : \mathsf{type}$ are approximations of datatypes and bound in Γ.

Separation of the types in two classes simplifies the theory: first, we do not need to define positivity for bounded quantification. Secondly, it brings some stratification that simplifies the semantics given in Section 5. For instance, monotonicity is only required of itypes.

Subtyping of the form $\rho \leq \mu X\sigma$ (I) determines ρ to be an approximation of the inductive type $\mu X\sigma$. Setting $\rho^0 := \rho$ and $\rho^{i+1} = (\rho^i)^+$, the approximations can only take the shape Y^i or $(\mu X\sigma)^i$ for some natural number i. We observe that

$$\exists\sigma.\ \Gamma \vdash \rho \leq \mu X\sigma$$

is decidable for all Γ and ρ. This entails that the other four judgements are also decidable (only the rule for wellformed approximations ρ is critical). Furthermore, if $\rho \leq \mu X\sigma$ and $\rho \leq \mu X\sigma'$ then $\sigma = \sigma'$.

There is another set of subtyping rules $\rho \leq \sigma$ (II) for the cases where the right hand side σ is an approximation of the shape Y or ρ^+. These rules are never invoked by the type-judgement, but by the subsumption rule (see below). The presentation of subtyping differs from the one in the previous section, but the same types are related. The formulation now is syntax-directed; transitivity and reflexivity for approximations $\rho \leq \mu X\sigma$ are admissible.

The judgements have a number of technical properties. Wellformedness of the context is a byproduct of the wellformedness of a type or a subtyping relation between two types in the given context:

Lemma 3.1 (Wellformedness of contexts)**.** *If either* $\Gamma; \vec{X}; \vec{X}' \vdash \sigma : \mathsf{itype}$ *or* $\Gamma \vdash \tau : \mathsf{type}$ *or* $\Gamma \vdash \rho \leq \sigma$ *then* $\Gamma\ \mathsf{cxt}$.

Proof. Simultaneously by induction on the derivation. □

All judgements allow weakening with wellformed contexts and substitution of approximation variables Y by the full inductive type $\mu X\sigma$:

Lemma 3.2 (Type substitution)**.** *Let* $\Gamma \vdash \rho \leq \mu X\sigma$. *Then*

(1) *If* $\Gamma, Y \leq \mu X\sigma, \Gamma'\ \mathsf{cxt}$ *then* $\Gamma, [\rho/Y]\Gamma'\ \mathsf{cxt}$.
(2) *If* $\Gamma, Y \leq \mu X\sigma, \Gamma'; \vec{X}; \vec{X}' \vdash \tau : \mathsf{itype}$ *then* $\Gamma, [\rho/Y]\Gamma'; \vec{X}; \vec{X}' \vdash [\rho/Y]\tau : \mathsf{itype}$.
(3) *If* $\Gamma, Y \leq \mu X\sigma, \Gamma' \vdash \tau : \mathsf{type}$ *then* $\Gamma, [\rho/Y]\Gamma' \vdash [\rho/Y]\tau : \mathsf{type}$.
(4) *If* $\Gamma, Y \leq \mu X\sigma, \Gamma' \vdash \tau_1 \leq \tau_2$ *then* $\Gamma, [\rho/Y]\Gamma' \vdash [\rho/Y]\tau_1 \leq [\rho/Y]\tau_2$.

Proof. By simultaneous induction over the derivation. □

3.2. Terms and type assignment

$\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ features the same terms as $\mathbf{\Lambda}_{\boldsymbol{\mu}}$, but comes with a more precise typing of inductive data (fold, unfold) and recursive functions (fix). Furthermore there are typing rules which arise from subtyping and quantification. Table 3 summarizes the changes to $\mathbf{\Lambda}_{\boldsymbol{\mu}}$ which have been introduced in Section 2 already. Due to the richer type grammar, not every context is automatically wellformed and we have to add the judgement $\Gamma\ \mathsf{cxt}$ to the two *axioms* of typing, var and unit. Similarly, not every type is wellformed, so we also have to refine the injection rules for sums.

The new typing rules for inductive data now also treat inhabitants of approximations of inductive types. Let us explain the fold-rule for the example of lists: let $Y \leq \mathsf{ListN} = \mu X.\ 1 + \mathsf{Nat} \times X$ be an approximation of the type of lists over natural

TABLE 3. Typing and reduction rules for $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$.

Typing axioms:

$$\frac{\Gamma\ \mathsf{cxt} \qquad x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{var} \qquad \frac{\Gamma\ \mathsf{cxt}}{\Gamma \vdash () : 1}\ \mathsf{unit}$$

Sum types:

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \tau : \mathsf{type}}{\Gamma \vdash \mathsf{inl}\, M : \sigma + \tau}\ \mathsf{inl} \qquad \frac{\Gamma \vdash \sigma : \mathsf{type} \qquad \Gamma \vdash M : \tau}{\Gamma \vdash \mathsf{inr}\, M : \sigma + \tau}\ \mathsf{inr}$$

Inductive types:

$$\frac{\Gamma \vdash \rho \leq \mu X.\sigma(X) \qquad \Gamma \vdash M : \sigma(\rho)}{\Gamma \vdash \mathsf{fold}\, M : \rho^+}\ \mathsf{fold} \qquad \frac{\Gamma \vdash \rho \leq \mu X.\sigma(X) \qquad \Gamma \vdash M : \rho^+}{\Gamma \vdash \mathsf{unfold}\, M : \sigma(\rho)}\ \mathsf{unfold}$$

Recursive functions:

$$\frac{\Gamma,\ Y{\leq}\mu X\sigma,\ g{:}Y \to \tau(Y),\ x{:}Y^+ \vdash M : \tau(Y^+) \qquad Y \text{ pos. in } \tau}{\Gamma \vdash \mathsf{fix}\, g(x).M : \forall Y{\leq}\mu X\sigma.\, Y \to \tau(Y)}\ \mathsf{fix}$$

Subtyping:

$$\frac{\Gamma \vdash \rho \leq \sigma \qquad \Gamma \vdash M : \rho}{\Gamma \vdash M : \sigma}\ \mathsf{sub} \qquad \frac{\Gamma \vdash \rho \leq \mu X\sigma \qquad \Gamma \vdash M : \forall Y{\leq}\mu X\sigma.\, \tau(Y)}{\Gamma \vdash M : \tau(\rho)}\ \mathsf{inst}$$

Neutral values:

$$\mathsf{Ne} \ni U ::= \ldots \mid (\mathsf{fix}\, g(x).V)\, U$$

Reduction axiom:

$$(\mathsf{fix}\, g(x).M)\, (\mathsf{fold}\, N) \longrightarrow_\beta [\mathsf{fix}\, g(x).M/g][\mathsf{fold}\, N/x]M$$

numbers and $x : \mathsf{Nat}$ a natural number. Let $y : Y$ a list in this approximation, *i.e.*, a list with size $< n$ for an unknown n. Then $\mathsf{inr}(x, y) : 1 + \mathsf{Nat} \times Y$, hence, by the folding rule, the list $\mathsf{Cons}(x, y) = \mathsf{fold}(\mathsf{inr}(x, y)) : Y^+$ is of size $< n + 1$. The unfold-rule works analogously. As mentioned in Section 2, the old folding rules (of $\mathbf{\Lambda}_{\boldsymbol{\mu}}$) are a special case of the new ones: set $\rho = \mu X\sigma$.

The reduction axioms of $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ are the same as for $\mathbf{\Lambda}_{\boldsymbol{\mu}}$ with the exception of fixed-point unrolling. To ensure normalization, recursive functions $\mathsf{fix}\, g(x).M$ are now only reduced if applied to a value $\mathsf{fold}\, N$ of an inductive type. Consequently, we get a new neutral value $(\mathsf{fix}\, g(x).V)\, U$ for a value U which is not of the form $\mathsf{fold}\, V$. The one-step reduction relation $\longrightarrow$ and multi-step reductions $\longrightarrow^+$, $\longrightarrow^*$ and $\longrightarrow^\infty$ are to be understood with regard to the modified axiom.

Example 3.3 (empty type)**.** The empty type is definable as the least fixed-point of the identity function on the type level.

$$\begin{array}{lcl} 0 & := & \mu X.X \\ \mathsf{abort} & := & \mathsf{fix}\, g(x).g(\mathsf{unfold}\, x)\ :\ 0 \to \tau \end{array}$$

Its elimination function abort is accepted by the type system in the above form and is indeed no threat to strong normalization. Why this holds will be informally explained by the following failed counterexample. It is formally established in Section 5. Assume a free variable $y : \mu X.X$. Then we have $\mathsf{fold}\, y : \mu X.X$ and the

following reduction sequence.

$$\begin{array}{rcl} \mathsf{abort}\,(\mathsf{fold}\,y) = & (\mathsf{fix}\,g(x).g(\mathsf{unfold}\,x))\ (\mathsf{fold}\,y) \\ \longrightarrow & (\mathsf{fix}\,g(x).g(\mathsf{unfold}\,x))\ (\mathsf{unfold}(\mathsf{fold}\,y)) \\ \longrightarrow & (\mathsf{fix}\,g(x).g(\mathsf{unfold}\,x))\ y \\ = & \mathsf{abort}\ y \end{array}$$

We observe that the present definition of abort in $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ does not produce loops, in contrast to the function $\mathsf{fix}\,g(x).g\,x$ which would be the simplest definition of abort in $\mathbf{\Lambda}_{\boldsymbol{\mu}}$.

The typing calculus enjoys the usual properties of wellformedness, restricted exchange, weakening, strengthening and substitution for term variables x. Substitution also holds for type variables Y:

Lemma 3.4 (type substitution)**.** *If* $\Gamma, Y \leq \mu X\sigma, \Gamma' \vdash M : \tau$ *and* $\Gamma \vdash \rho \leq \mu X\sigma$ *then* $\Gamma, [\rho/Y]\Gamma' \vdash M : [\rho/Y]\tau$.

Proof. By induction of the typing derivation and case distinction on the subtyping derivation. □

Important for the soundness of typing is the fact that all contexts and types appearing in the typing judgement are wellformed:

Lemma 3.5. *If* $\Gamma \vdash M : \tau$ *then* $\Gamma\ \mathsf{cxt}$ *and* $\Gamma \vdash \tau : \mathsf{type}$.

Proof. By induction on the typing derivation. □

Example 3.6 (tree ordinals)**.** The second number class can be represented as the strictly positive inductive type $\mathsf{Ord} = \mu X.\ 1 + (X + (\mathsf{Nat} \to X))$ with three constructors.

OZero	:=	$\mathsf{fold}(\mathsf{inl}())$	:	Ord	zero ordinal
OSucc	:=	$\lambda a.\ \mathsf{fold}(\mathsf{inr}(\mathsf{inl}\,a))$	:	$\mathsf{Ord} \to \mathsf{Ord}$	successor ordinal
OLim	:=	$\lambda f.\ \mathsf{fold}(\mathsf{inr}(\mathsf{inr}\,f))$	:	$(\mathsf{Nat} \to \mathsf{Ord}) \to \mathsf{Ord}$	limit ordinal

Example 3.7 (representation of ω)**.** Assuming $\mathsf{toOrd} : \mathsf{Nat} \to \mathsf{Ord}$ is defined as the function which converts a natural number $\mathsf{Succ}^n(\mathsf{Zero})$ into its ordinal equivalent $\mathsf{OSucc}^n(\mathsf{OZero})$ we obtain a representation of the least infinite ordinal ω as $\mathsf{omega} = \mathsf{OLim}\,\mathsf{toOrd}$.

Example 3.8 (addition of tree ordinals)**.** In $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$, ordinal addition can be defined as follows:

$$\begin{array}{lcl} \mathsf{oadd} & : & \mathsf{Ord} \to \forall Y \leq \mathsf{Ord}.\ Y \to \mathsf{Ord} \\ \mathsf{oadd} & := & \lambda x.\ \mathsf{fix}\,oadd(y).\ \mathsf{case}(\mathsf{unfold}\,y)\ \mathsf{of} \\ & & \qquad \mathsf{inl}\,u \ \Rightarrow\ x \\ & & \quad \mid\ \mathsf{inr}\,y' \ \Rightarrow\ \mathsf{case}\,y'\ \mathsf{of} \\ & & \qquad\quad \mathsf{inl}\,a \ \Rightarrow\ \mathsf{OSucc}(oadd\,a) \\ & & \qquad\quad \mid\ \mathsf{inr}\,f \ \Rightarrow\ \mathsf{OLim}(\lambda n.\ oadd\,(f\,n)) \end{array}$$

During type checking, the bound variables receive the following types: $x : \mathsf{Ord}$, $\mathsf{oadd} : Y \to \mathsf{Ord}$, $y : Y^+$, $u : 1$, $y' : Y + (\mathsf{Nat} \to Y)$, $a : Y$, $f : \mathsf{Nat} \to Y$ and $n : \mathsf{Nat}$. Also, the positivity condition on the result type Ord of recursion is trivially satisfied, since Y is not mentioned in Ord.

Datatypes with embedded function spaces, like in Example 3.6, are explicitely excluded in the work on size types by Hughes, Pareto and Sabry [28]. The reason is that in their their approach, size annotations are interpreted as ordinals $\leq \omega$. But Example 3.7 needs larger ordinals, as explained in Section 5.

3.3. Weak type soundness of $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$

In this section, we show that in the extended system $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$, programs still cannot go wrong. Unfortunately, the type preservation property fails for the new type system, since our notion of subtyping is too weak. Consider the following typing derivation:

$$\dfrac{\dfrac{M : \sigma(\rho_1)}{\mathsf{fold}\, M : \rho_1^+} \qquad \dfrac{\rho_1 \leq \rho_2}{\rho_1^+ \leq \rho_2^+}}{\dfrac{\mathsf{fold}\, M : \rho_2^+}{\mathsf{unfold}(\mathsf{fold}\, M) : \sigma(\rho_2)}}$$

Since $\mathsf{unfold}(\mathsf{fold}\, M) \longrightarrow_\beta M$, type preservation requires that $M : \sigma(\rho_2)$. However, our syntactic subtyping lacks the necessary property $\sigma(\rho_1) \leq \sigma(\rho_2)$ for arbitrary $\rho_1 \leq \rho_2$ and $\sigma(X)$ in which X appears only positively[4]. One way to overcome the problem would be to add the necessary closure properties to the subtyping relation, as we have done in other work [3]. Instead, we make use of the fact that we have the simpler type system $\mathbf{\Lambda}_{\boldsymbol{\mu}}$ which already ensures that well-typed programs cannot go wrong. We show that each typing derivation of $\mathbf{\Lambda}_{\boldsymbol{\mu}}$ can be turned into a typing derivation of $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ by replacing approximation types by the full inductive types. Since the progress property – Theorem 1.1(2) – also holds for the slightly restricted reduction relation and the slightly enlarged value set of $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$, type soundness of $\mathbf{\Lambda}_{\boldsymbol{\mu}}$ ensures that $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ programs cannot get stuck as well.

Table 4 specifies how to erase approximations in both types and contexts. The judgement $\Delta\ \mathsf{tcxt}$ singles out such contexts Δ – called translation contexts – which map approximation type variables Y into closed approximation-free inductive types $\mu X\sigma$. Given a translation context Δ and a $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$-type τ with $\Delta \vdash \tau : \mathsf{type}$, judgement $\Delta \vdash \tau \,||\rangle\, \tau'$ outputs a wellformed $\mathbf{\Lambda}_{\boldsymbol{\mu}}$-type τ' in which all approximation types $\rho \leq \mu X\sigma$ have been replaced by the full inductive type $\mu X\sigma$. Similarly, $\Gamma \,||\rangle\, \Gamma'; \Delta$ separates a wellformed $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$-context $\Gamma\ \mathsf{cxt}$ into a approximation-free $\mathbf{\Lambda}_{\boldsymbol{\mu}}$-context Γ' and a translation context $\Delta\ \mathsf{tcxt}$.

Theorem 3.9 (soundness of $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$-typing with regard to $\mathbf{\Lambda}_{\boldsymbol{\mu}}$-typing)**.** *If* $\Gamma \vdash M : \tau$ *in* $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$, $\Gamma \,||\rangle\, \Gamma'; \Delta$ *and* $\Delta \vdash \tau \,||\rangle\, \tau'$, *then* $\Gamma' \vdash M : \tau'$ *in* $\mathbf{\Lambda}_{\boldsymbol{\mu}}$.

[4] Semantically, this property does hold and is the content of Lemma 5.5.

TABLE 4. Erasure of approximation types.

Translation contexts Δ tcxt.

$$\frac{}{\cdot \text{ tcxt}} \qquad \frac{\Delta \text{ tcxt} \qquad \cdot;\cdot;\cdot \vdash \mu X\sigma : \text{itype}}{\Delta,\, Y{\leq}\mu X\sigma \text{ tcxt}}\; Y \notin \Delta$$

Approximation erasure in types $\Delta \vdash \tau \;||\rangle\; \tau'$.

$$\frac{}{\Delta \vdash 1 \;||\rangle\; 1} \quad \frac{X \notin \Delta}{\Delta \vdash X \;||\rangle\; X} \quad \frac{\Delta \vdash \sigma \;||\rangle\; \sigma'}{\Delta \vdash \mu X\sigma \;||\rangle\; \mu X\sigma} \quad \frac{\Delta \vdash \rho \;||\rangle\; \rho' \qquad \Delta \vdash \sigma \;||\rangle\; \sigma'}{\Delta \vdash \rho \star \sigma \;||\rangle\; \rho' \star \sigma'}\; \star \in \{+, \times, \to\}$$

$$\frac{\Delta \vdash \rho \leq \mu X\sigma}{\Delta \vdash \rho \;||\rangle\; \mu X\sigma} \qquad \frac{\Delta \vdash \sigma \;||\rangle\; \sigma' \qquad \Delta, Y{\leq}\mu X\sigma' \vdash \tau \;||\rangle\; \tau'}{\Delta \vdash \forall Y{\leq}\mu X\sigma.\tau \;||\rangle\; \tau'}$$

Approximation erasure in contexts $\Gamma \;||\rangle\; \Gamma';\Delta$.

$$\frac{}{\cdot \;||\rangle\; \cdot;\cdot} \qquad \frac{\Gamma \;||\rangle\; \Gamma';\Delta \qquad \Delta \vdash \tau \;||\rangle\; \tau'}{\Gamma, x{:}\tau \;||\rangle\; \Gamma', x{:}\tau';\Delta} \qquad \frac{\Gamma \;||\rangle\; \Gamma';\Delta \qquad \Delta \vdash \sigma \;||\rangle\; \sigma'}{\Gamma, Y{\leq}\mu X\sigma \;||\rangle\; \Gamma';\Delta, Y{\leq}\mu X\sigma'}$$

Proof. By induction on $\Gamma \vdash M : \tau$. The proof amounts to removing all applications of subsumption and instantiation from the $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ typing derivation. □

Corollary 3.10 (weak type soundness for $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$)**.** *If* $\Gamma \vdash M : \tau$ *in* $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$*, then* $M \longrightarrow^\infty V$*.*

Proof. By combining the last theorem with Corollary 1.2. □

Since we are lacking type preservation in $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$, we cannot show $\Gamma \vdash V : \tau$ and therefore only obtain the weak type soundness property. This is enough for our purposes; in Section 5 we show that infinite reduction sequences are excluded by the advanced type system – with the result that each well-typed term reduces to a value in $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$.

4. TYPE CHECKING

In order to use our type system as a termination checker, we need a suitable type checking algorithm. One possibility would be to extend the Hindley-Milner algorithm to bounded quantification and approximation types. This algorithm would produce a set of existential type variables together with a set of equations these variables have to satisfy. The problem with such an algorithm (as implemented in SML 1997, for instance) is that in case of unsolvable equations, the location "responsible" (from a human perspective) for the type error is hard to locate[5].

[5] Recently, type inference algorithms which give better error messages have been put forward by Haack and Wells [24] and Yang, Michaelson and Trinder [51].

Pierce and Turner [42] advocate *local type inference*, which restricts propagation of type information to adjacent nodes of the syntax tree of the term which is to be type-checked – in contrast to Hindley-Milner which collects type equations globally. Local type inference is based on *bidirectional type checking*, a folklore method which is for instance used by Coquand [18] for dependent types and Davies and Pfenning [20] for intersection types. Recently, Dunfield and Pfenning [21] have extended this method to tridirectional checking.

Bidirectional type checking is incomplete, it can decide typing only for some terms – in the case of the simply-typed lambda-calculus, it is exactly the *normal* terms. Non-normal terms need type annotations to type-check. Hence, we extend our calculus $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ by some means to aid the type-checker and obtain the language $\mathbf{L}^+_{\boldsymbol{\mu}}$. The new constructs allow annotation of bound variables $\lambda x : \sigma.\, M$ and of whole terms $(M : \tau)$. Additionally, we introduce let-binding $\mathsf{let}\, x = N \,\mathsf{in}\, M$ which has the same operational meaning as $(\lambda x.M)\, N$, but better behavior w.r.t. type checking.

We adapt bidirectional type checking to our calculus $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$, the rules are listed in Table 5. The checking judgement $\Gamma \vdash M \Uparrow \tau$ requires all of Γ, M and τ as input and its generating rules are to be read upwards as the arrow suggests. It is defined simultaneously with type inference $\Gamma \vdash M \downarrow \tau$, which computes the type τ from given Γ and M – the rules for inference should be read downwards.

The general principle of bidirectional checking is that the type of variables and eliminations can be inferred: rules $\downarrow\mathsf{var}$, $\downarrow\mathsf{fst}$, $\downarrow\mathsf{snd}$, $\downarrow\mathsf{unfold}$, $\downarrow\mathsf{app}$ and $\downarrow\mathsf{app}\mu 1$. In contrast, the type of introductions must be checked against: rules $\Uparrow\mathsf{lam}$, $\Uparrow\mathsf{pair}$, $\Uparrow\mathsf{fold}$, $\Uparrow\mathsf{inl}$, $\Uparrow\mathsf{inr}$, $\Uparrow\mathsf{fix}^-$ and $\Uparrow\mathsf{fix}$ (annotated lambda-abstractions $\lambda x{:}\sigma.M$ are an exception and admit inference by rule $\downarrow\mathsf{lam}$). The rule $\Uparrow\!\downarrow$ allows one to switch from checking mode into inference mode and should be applied if no other checking rule matches. In this case, the infered type τ must be more general than the type τ' supplied to the checking judgement. This is expressed be the extended subtyping relation $\tau \lhd \tau'$, which encompasses subtyping and instantiation. Hence, the non-deterministic character of subsumption in the typing judgement is tamed in the type checking algorithm by shifting applications of the subsumption rule to the specific point where inference and checking meet.

A positive abnormality to the introduction/elimination dichotomy are tuples: they permit inference although they are constructors ($\downarrow\mathsf{unit}$, $\downarrow\mathsf{pair}$). One of the negative abnormalities is sum-elimination: the type of the side clauses in the case-construct is arbitrary, and cannot be inferred in general ($\Uparrow\mathsf{case}$). Let-binding ($\Uparrow\mathsf{let}$) can be seen as elimination of a unary sum and is therefore treated like case. Another problem occurs with bounded quantification: in the rule $\downarrow\mathsf{app}\mu 2$, the size ρ of the argument needs to be inferred to give a size-bound $\tau(\rho)$ on the result of the application. If the size of the result is known, it can be checked (rule $\Uparrow\mathsf{app}\mu$). This rule involves simple matching on the level of types: $\tau' = \tau(\rho)$ for some unknown ρ. However, since types are first-order expressions without reduction, the matching problem is decidable.

TABLE 5. Type checking.

Terms.

$$\mathsf{L}^{+}_{\mu} \ni M, N ::= \cdots \mid \lambda x{:}\sigma.M \mid (M{:}\tau) \mid \mathsf{let}\, x = N\, \mathsf{in}\, M$$

Judgments.

$\Gamma \vdash \tau \lhd \tau'$	Extended subtyping.
$\Gamma \vdash M \downarrow \tau$	The type of term M is inferred as τ.
$\Gamma \vdash M \Uparrow \tau$	Term M is checked against type τ.

Type inference.

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x \downarrow \tau}\ \downarrow\mathsf{var} \qquad \frac{\Gamma \vdash \sigma : \mathsf{type} \qquad \Gamma, x{:}\sigma \vdash M \downarrow \tau}{\Gamma \vdash \lambda x{:}\sigma.M \downarrow \sigma \to \tau}\ \downarrow\mathsf{lam} \qquad \frac{\Gamma \vdash M \downarrow \sigma \to \tau \qquad \Gamma \vdash N \Uparrow \sigma}{\Gamma \vdash M\, N \downarrow \tau}\ \downarrow\mathsf{app}$$

$$\frac{}{\Gamma \vdash () \downarrow 1}\ \downarrow\mathsf{unit} \qquad \frac{\Gamma \vdash M \downarrow \sigma \qquad \Gamma \vdash N \downarrow \tau}{\Gamma \vdash (M, N) \downarrow \sigma \times \tau}\ \downarrow\mathsf{pair} \qquad \frac{\Gamma \vdash M \downarrow \sigma \times \tau}{\Gamma \vdash \mathsf{fst}\, M \downarrow \sigma}\ \downarrow\mathsf{fst} \qquad \frac{\Gamma \vdash M \downarrow \sigma \times \tau}{\Gamma \vdash \mathsf{snd}\, M \downarrow \tau}\ \downarrow\mathsf{snd}$$

$$\frac{\Gamma \vdash M \downarrow \rho^{+} \qquad \Gamma \vdash \rho \le \mu X.\sigma(X)}{\Gamma \vdash \mathsf{unfold}\, M \downarrow \sigma(\rho)}\ \downarrow\mathsf{unfold}^{+} \qquad \frac{\Gamma \vdash M \downarrow \rho \qquad \Gamma \vdash \rho \le \mu X.\sigma(X)}{\Gamma \vdash \mathsf{unfold}\, M \downarrow \sigma(\rho)}\ \downarrow\mathsf{unfold}$$

$$\frac{\Gamma \vdash \tau : \mathsf{type} \qquad \Gamma \vdash M \Uparrow \tau}{\Gamma \vdash (M{:}\tau) \downarrow \tau}\ \downarrow\mathsf{ann} \qquad \frac{\Gamma \vdash M \downarrow \forall Y \le \mu X\sigma.\, Y \to \tau \qquad Y \notin \mathsf{FV}(\tau) \qquad \Gamma \vdash N \Uparrow \mu X\sigma}{\Gamma \vdash M\, N \downarrow \tau}\ \downarrow\mathsf{app}\mu 1$$

$$\frac{\Gamma \vdash M \downarrow \forall Y \le \mu X\sigma.\, Y \to \tau(Y) \qquad \Gamma \vdash N \downarrow \rho \qquad \Gamma \vdash \rho \le \mu X\sigma}{\Gamma \vdash M\, N \downarrow \tau(\rho)}\ \downarrow\mathsf{app}\mu 2$$

Type checking.

$$\frac{\Gamma, x{:}\sigma \vdash M \Uparrow \tau}{\Gamma \vdash \lambda x.M \Uparrow \sigma \to \tau}\ \Uparrow\mathsf{lam} \qquad \frac{\Gamma \vdash M \Uparrow \sigma}{\Gamma \vdash \mathsf{inl}\, M \Uparrow \sigma + \tau}\ \Uparrow\mathsf{inl} \qquad \frac{\Gamma \vdash M \Uparrow \tau}{\Gamma \vdash \mathsf{inr}\, M \Uparrow \sigma + \tau}\ \Uparrow\mathsf{inr}$$

$$\frac{\Gamma \vdash N \downarrow \sigma_1 + \sigma_2 \qquad \Gamma, x_i{:}\sigma_i \vdash M_i \Uparrow \rho}{\Gamma \vdash \mathsf{case}\, N\, \mathsf{of}\, \mathsf{inl}\, x_1 \Rightarrow M_1 \mid \mathsf{inr}\, x_2 \Rightarrow M_2 \Uparrow \rho}\ \Uparrow\mathsf{case} \qquad \frac{\Gamma \vdash N \downarrow \sigma \qquad \Gamma, x{:}\sigma \vdash M \Uparrow \tau}{\Gamma \vdash \mathsf{let}\, x = N\, \mathsf{in}\, M \Uparrow \tau}\ \Uparrow\mathsf{let}$$

$$\frac{\Gamma \vdash \rho \le \mu X.\sigma(X) \quad \Gamma \vdash M \Uparrow \sigma(\rho)}{\Gamma \vdash \mathsf{fold}\, M \Uparrow \rho^{+}}\ \Uparrow\mathsf{fold} \qquad \frac{\Gamma \vdash M \Uparrow \sigma(\mu X.\sigma(X))}{\Gamma \vdash \mathsf{fold}\, M \Uparrow \mu X.\sigma(X)}\ \Uparrow\mathsf{fold}\mu$$

$$\frac{\Gamma \vdash M \Uparrow \sigma \quad \Gamma \vdash N \Uparrow \tau}{\Gamma \vdash (M, N) \Uparrow \sigma \times \tau}\ \Uparrow\mathsf{pair} \qquad \frac{\Gamma, Y \le \mu X\sigma, g{:}Y \to \tau, x{:}Y^{+} \vdash M \Uparrow \tau}{\Gamma \vdash \mathsf{fix}\, g(x).M \Uparrow \mu X\sigma \to \tau}\ \Uparrow\mathsf{fix}^{-}$$

$$\frac{\Gamma, Y \le \mu X\sigma, g{:}Y \to \tau(Y), x{:}Y^{+} \vdash M \Uparrow \tau(Y^{+}) \qquad Y \text{ pos. in } \tau(Y)}{\Gamma \vdash \mathsf{fix}\, g(x).M \Uparrow \forall Y \le \mu X\sigma.\, Y \to \tau(Y)}\ \Uparrow\mathsf{fix}$$

$$\frac{\Gamma \vdash M \downarrow \forall Y \le \mu X\sigma.\, Y \to \tau(Y) \quad \text{match } \tau' \text{ with } \tau(Y) \Longrightarrow Y = \rho \quad \Gamma \vdash \rho \le \mu X\sigma \quad \Gamma \vdash N \Uparrow \rho}{\Gamma \vdash M\, N \Uparrow \tau'}\ \Uparrow\mathsf{app}\mu$$

$$\frac{\Gamma \vdash M \downarrow \sigma \to \tau \qquad \Gamma \vdash \tau \lhd \tau' \qquad \Gamma \vdash N \Uparrow \sigma}{\Gamma \vdash M\, N \Uparrow \tau'}\ \Uparrow\mathsf{app}$$

Direction reversal.

$$\frac{\Gamma \vdash M \downarrow \tau \qquad \Gamma \vdash \tau \lhd \tau'}{\Gamma \vdash M \Uparrow \tau'}\ \Uparrow\downarrow$$

Extended subtyping.

$$\frac{}{\Gamma \vdash \tau \lhd \tau}\ \lhd\mathsf{refl} \qquad \frac{\Gamma \vdash \rho \le \sigma}{\Gamma \vdash \rho \lhd \sigma}\ \lhd\le \qquad \frac{\Gamma \vdash \rho \le \mu X\sigma}{\Gamma \ \vdash\ \forall Y \le \mu X\sigma.\, Y \to \tau(Y) \ \lhd\ \rho \to \tau(\rho)}\ \lhd\mathsf{inst}$$

Disregarding the abnormalities, the algorithm can check the types of most *normal* expressions, which is the kind of expressions that dominates in practical programming. Non-normal expressions need to be annotated with their type at some sensible position (rules $\downarrow\mathsf{ann}$ and $\downarrow\mathsf{lam}$). All top-level expressions and all recursive functions, need to be defined with their type. This is not too heavy a load for the programmer; in Haskell it is quite common.

Example 4.1 (type-checkable version of sum example)**.** To make the function sum from Section 1 acceptable by the type checker we need to introduce one annotation. Using the let construct, we can write it as follows in $\mathbf{L}_{\boldsymbol{\mu}}^{+}$:

$$
\begin{array}{lcl}
\mathsf{sum} & : & \mathsf{ListN} \to \mathsf{Nat} \\
\mathsf{sum} & := & \mathsf{fix}\, f(l).\ \mathsf{case}\,(\mathsf{unfold}\, l)\ \mathsf{of} \\
 & & \quad \mathsf{inl}\, u \ \Rightarrow\ \mathsf{Zero} \\
 & & \quad \mid\ \mathsf{inr}\, p \ \Rightarrow\ \mathsf{let}\ \mathsf{sum}' = (\mathsf{fix}\, g(n).\ \ \mathsf{case}\,(\mathsf{unfold}\, n)\ \mathsf{of} \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{inl}\, u \ \Rightarrow\ f\,(\mathsf{snd}\ p) \\
 & & \qquad\qquad\qquad\qquad\qquad \mid\ \mathsf{inr}\, n' \ \Rightarrow\ \mathsf{Succ}(g\, n')) \\
 & & \qquad\qquad\qquad\qquad : \mathsf{Nat} \to \mathsf{Nat} \\
 & & \qquad\qquad \mathsf{in}\ \ \mathsf{sum}'(\mathsf{fst}\, p)
\end{array}
$$

The rules have been implemented in the higher-order logical framework Twelf [39] to yield a prototypical type checker which is available on the homepage of the author [4]. It is easy to turn the rules into a deterministic algorithm, namely into two mutually recursive functions inf and chk with the following specification.

$$
\mathsf{inf}\,(\Gamma, M) \quad = \quad \begin{cases} \tau & \text{if } \ \Gamma \vdash M \downarrow \tau \\ \mathsf{error} & \text{otherwise} \end{cases}
$$

$$
\mathsf{chk}\,(\Gamma, M, \tau) \quad = \quad \begin{cases} \mathsf{ok} & \text{if } \ \Gamma \vdash M \Uparrow \tau \\ \mathsf{error} & \text{otherwise.} \end{cases}
$$

Function inf is defined by case distinction on M and function chk by cases on both M and τ. The tests and recursive calls necessary in each case can be read off the rules in Table 5. In some cases several rules seem applicable. This source of non-determinism can be resolved by fusing the respective rules into a single one:

(1) Case $\mathsf{inf}\,(\Gamma, \mathsf{unfold}\, M)$, rules $\downarrow\mathsf{unfold}^{+}$ and $\downarrow\mathsf{unfold}$: After inferring the type ρ of M, distinguish whether ρ is of shape ρ^{+} or not.

(2) Case $\mathsf{inf}\,(\Gamma,\, M\,N)$, rules $\downarrow\mathsf{app}$, $\downarrow\mathsf{app}\mu 1$ and $\downarrow\mathsf{app}\mu 2$: Distinguish on the infered type of M.
(3) Case $\mathsf{chk}\,(\Gamma,\, M\,N,\, \tau')$, rules $\Uparrow\mathsf{app}\mu$ and $\Uparrow\mathsf{app}$: similarly.
(4) Case $\mathsf{chk}\,(\Gamma,\, M,\, \tau')$, rule $\Uparrow\downarrow$: Switch to inference only if no checking rule is applicable.

The resulting function $\mathsf{chk}(\Gamma, M, \tau)$ checks types in time linear in the size of the input expression M plus type τ (modulo context lookup, type matching and subtype checking).

Table 6 shows a trace of the type checker as it verifies the welltypedness of the ordinal addition function oadd. The first column displays which rules have fired, the second column lists the hypotheses which the rules have added to the context and the third column show the new state of the checker. To simplify the presentation we have ignored the inl branches of the case expressions in this trace.

The algorithm is not complete, but it can check many interesting programs, for instance, all examples of this article. In the following we show soundness of the algorithm.

Definition 4.2 (erasure)**.** Let $|M| \in \mathbf{\Lambda}^{+}_{\boldsymbol{\mu}}$ denote the result of erasing all type annotations and let-bindings from $M \in \mathbf{L}^{+}_{\boldsymbol{\mu}}$. We define $|\cdot|$ by

$$\begin{array}{lcl} |\lambda x{:}\sigma.M| & = & \lambda x.|M|, \\ |(M:\tau)| & = & |M|, \text{ and} \\ |\mathsf{let}\, x = N\, \mathsf{in}\, M| & = & (\lambda x.|M|)\,|N|, \end{array}$$

adding congruences for all other term constructors.

Proposition 4.3 (soundness of type checking)**.** *Let Γ be a wellformed context, M a term of $\mathbf{L}^{+}_{\boldsymbol{\mu}}$ and τ a type.*

(1) *If $\Gamma \vdash M \downarrow \tau$, then $\Gamma \vdash \tau : \mathsf{type}$ and $\Gamma \vdash |M| : \tau$.*
(2) *If $\Gamma \vdash M \Uparrow \tau$ and $\Gamma \vdash \tau : \mathsf{type}$, then $\Gamma \vdash |M| : \tau$.*

Proof. Simultaneously by induction on $\Gamma \vdash M \downarrow\Uparrow \tau$. For example:

Case

$$\frac{\Gamma \vdash M \downarrow \rho \qquad \Gamma \vdash \rho \leq \mu X.\sigma(X)}{\Gamma \vdash \mathsf{unfold}\, M \downarrow \sigma(\rho)}\ \downarrow\mathsf{unfold}$$

$$\begin{array}{ll} \Gamma \vdash M : \rho & \text{by induction hypothesis} \\ \Gamma \vdash \rho \leq \rho^{+} & \text{from assumption by subtyping} \\ \Gamma \vdash M : \rho^{+} & \text{by subsumption rule} \\ \Gamma \vdash \mathsf{unfold}\, M : \sigma(\rho) & \text{by unfolding rule} \end{array}$$

Case

$$\frac{\Gamma \vdash M \downarrow \forall Y \leq \mu X\sigma.\, Y \to \tau \qquad Y \notin \mathsf{FV}(\tau) \qquad \Gamma \vdash N \Uparrow \mu X\sigma}{\Gamma \vdash M\,N \downarrow \tau}\ \downarrow\mathsf{app}\mu 1$$

TABLE 6. Example run of type checker.

$$
\begin{array}{lcl}
\mathsf{oadd} & : & \mathsf{Ord} \to \forall Y{\leq}\mathsf{Ord}.\ Y \to \mathsf{Ord} \\
\mathsf{oadd} & := & \lambda x.\ \mathsf{fix}\, oadd(y).\ \mathsf{case}(\mathsf{unfold}\, y)\ \mathsf{of} \\
 & & \qquad \mathsf{inl}\, u \ \Rightarrow\ x \\
 & & \quad \mid\ \mathsf{inr}\, y' \ \Rightarrow\ \mathsf{case}\, y'\ \mathsf{of} \\
 & & \qquad\quad \mathsf{inl}\, a \ \Rightarrow\ \mathsf{OSucc}(oadd\, a) \\
 & & \qquad\quad \mid\ \mathsf{inr}\, f \ \Rightarrow\ \mathsf{OLim}(\lambda n.\ oadd\, (f\, n))
\end{array}
$$

$$\Gamma_0 \ :=\ \mathsf{OZero}{:}\mathsf{Ord},\ \mathsf{OSucc}{:}\mathsf{Ord} \to \mathsf{Ord}, \mathsf{OLim}{:}(\mathsf{Nat} \to \mathsf{Ord}) \to \mathsf{Ord}$$

Rule	Context ext.	Judgement
start	Γ_0	$\lambda x.\ \mathsf{fix}\ldots \Uparrow \mathsf{Ord} \to \forall Y{\leq}\mathsf{Ord}.\ Y \to \mathsf{Ord}$
$\Uparrow$lam	$x : \mathsf{Ord}$	$\mathsf{fix}\, oadd(y)\ldots \Uparrow \forall Y{\leq}\mathsf{Ord}.\ Y \to \mathsf{Ord}$
$\Uparrow$fix	$Y \leq \mathsf{Ord}$	
	$oadd : Y \to \mathsf{Ord}$	
	$y : Y^+$	$\mathsf{case}(\mathsf{unfold}\, y)\ldots \Uparrow \mathsf{Ord}$
$\downarrow$var		$y \downarrow Y^+$
$\downarrow$unfold		$\mathsf{unfold}\, y \downarrow 1 + (Y + (\mathsf{Nat} \to Y))$
$\Uparrow$case	$y' : Y + (\mathsf{Nat} \to Y)$	$\mathsf{case}\, y'\ldots \Uparrow \mathsf{Ord}$
$\downarrow$var		$y' \downarrow Y + (\mathsf{Nat} \to Y)$
$\Uparrow$case	$f : \mathsf{Nat} \to Y$	$\mathsf{OLim}(\lambda n\ldots) \Uparrow \mathsf{Ord}$
$\downarrow$var		$\mathsf{OLim} \downarrow (\mathsf{Nat} \to \mathsf{Ord}) \to \mathsf{Ord}$
$\Uparrow$app		$\lambda n.\, oadd\ldots \Uparrow \mathsf{Nat} \to \mathsf{Ord}$
$\Uparrow$lam	$n : \mathsf{Nat}$	$oadd\, (f\, n) \Uparrow \mathsf{Ord}$
$\downarrow$var		$oadd \downarrow Y \to \mathsf{Ord}$
$\Uparrow$app		$f\, n \Uparrow Y$
$\downarrow$var		$f \downarrow \mathsf{Nat} \to Y$
$\Uparrow$app		$n \Uparrow \mathsf{Nat}$
$\Uparrow\downarrow, \downarrow$var		$n \downarrow \mathsf{Nat}$

$$
\begin{array}{ll}
\Gamma \vdash M : \forall Y{\leq}\mu X\sigma.\, Y \to \tau & \text{by induction hypothesis} \\
\Gamma \vdash M : \mu X\sigma \to \tau & \text{by instantiation} \\
\Gamma \vdash \mu X\sigma \to \tau : \mathsf{type} & \text{by Lemma 3.5} \\
\Gamma \vdash N : \mu X\sigma & \text{by induction hypothesis} \\
\Gamma \vdash M\, N : \tau & \text{by application rule}
\end{array}
$$

The other cases are similarly straightforward. □

Most of the soundness proof has been formally verified in Twelf and can be obtained from the author [4]. For the remainder of this article, we forget about the additional constructs in $\mathbf{L}^+_{\boldsymbol{\mu}}$ and return to our core calculus $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$.

5. Strong normalization

In this section, we present a semantical soundness proof for $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$. More precisely, we assign to each wellformed type τ a set $[\![\tau]\!]$ of strongly normalizing terms such that each term M which can be assigned type τ inhabits $[\![\tau]\!]$. The consequence is that such M are strongly normalizing, which in combination with the weak type soundness result of Section 3.3 guarantees that they reduce to values V in a finite number of steps.

Proofs of strong normalization have a long tradition and several methods have been developed. We use the *saturated sets* method based on the notion of *weak head reduction* as found *e.g.* in Luo [31] and Altenkirch [6].

5.1. Preliminaries: saturated sets

The set of *strongly normalizing terms* $\mathsf{SN} \subseteq \mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ is defined inductively by the following rule:

$$\frac{\forall M' \in \mathbf{\Lambda}^+_{\boldsymbol{\mu}}.\ M \longrightarrow M' \Rightarrow M' \in \mathsf{SN}}{M \in \mathsf{SN}}$$

In other words, SN is the wellfounded part of the set of terms w. r. t. the reduction relation $\longrightarrow$. All variables x are strongly normalizing as well as all subterms of strongly normalizing terms. The set SN is closed under reduction.

The notion of weak head reduction can be elegantly defined with *evaluation contexts* $E[\bullet]$, which are generated by the following grammar:

$$\begin{aligned} E[\bullet] ::= {} & \bullet \mid E[\bullet]\, M \mid \mathsf{fst}\, E[\bullet] \mid \mathsf{snd}\, E[\bullet] \\ & \mid (\mathsf{case}\, E[\bullet]\, \mathsf{of}\, \mathsf{inl}\, x_1 \Rightarrow M_1 \mid \mathsf{inr}\, x_2 \Rightarrow M_2) \\ & \mid \mathsf{unfold}\, E[\bullet] \mid (\mathsf{fix}\, g(x).M)\, E[\bullet] \end{aligned}$$

Weak head reduction is reduction of the form $E[M_0] \longrightarrow E[M_1]$ where $M_0 \longrightarrow_\beta M_1$. In this situation, we say that $E[M_0]$ has a head redex M_0. Terms of the form $E[x]$ are called *neutral*, they possess no head redex and produce no head redex if substituted for some variable into some term. In the process of normalization, weak head redexes $E[M_0]$ *must* be resolved, there is no way of skipping them by performing other reductions first. This is expressed by the following standardization lemma (Altenkirch [6]).

Lemma 5.1 (weak standardization)**.** *If $M_0 \longrightarrow_\beta M_1$ and $E[M_0] \longrightarrow N$, then either $N = E[M_1]$ or $N = E'[M_0']$ for some $E'[\bullet]$, M_0' and there exists an M_1' such that $M_0' \longrightarrow_\beta M_1'$ and $E[M_1] \longrightarrow^* E'[M_1']$.*

Proof. By induction on the generation on $E[\bullet]$, and in the base case $E[\bullet] = \bullet$ by case distinction on $M_0 \longrightarrow N$. □

A consequence of weak standardization is that the set SN is closed under weak head expansion.

Lemma 5.2 (weak head expansion)**.** *If $E[M_1] \in \mathsf{SN}$ then for all $M_0 \in \mathsf{SN}$ with $M_0 \longrightarrow_\beta M_1$ it holds that $E[M_0] \in \mathsf{SN}$.*

Proof. By simultaneous induction on $E[\bullet], M_0 \in \mathsf{SN}$ we show $N \in \mathsf{SN}$ for all N with $E[M_0] \longrightarrow N$, using Lemma 5.1. □

A set of terms $P \subseteq \mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ is called *saturated*, written $P \in \mathcal{SAT}$, if it contains only strongly normalizing terms, all strongly normalizing neutral terms, and if it is closed under weak head expansion. The *saturation* of a set P is defined as the closure under the following rules:

$$\frac{M \in P}{M \in P^*} \qquad \frac{E[x] \in \mathsf{SN}}{E[x] \in P^*} \qquad \frac{M \in \mathsf{SN} \qquad M \longrightarrow_\beta M' \qquad E[M'] \in P^*}{E[M] \in P^*}$$

For sets of terms P, Q we define the *function space*

$$P \to Q := \{M \in \mathsf{SN} : \forall N \in P.\ M\,N \in Q\}.$$

In the following we reuse the well-known and easily verified fact that the function space constructor $\to$ operates on saturated sets, *i.e.*, for $P, Q \in \mathcal{SAT}$ also $P \to Q \in \mathcal{SAT}$.

5.2. Type interpretation

In this section, we give an interpretation $[\![\tau]\!]$ for every wellformed type τ. For datatypes σ, the semantics $[\![\sigma]\!]$ is a monotone operator on saturated sets. In particular, $[\![\mu X.\sigma]\!]$ is the fixpoint of an operator derived from $[\![\sigma]\!]$. In previous work [5], we obtained this fixpoint by the theorem of Knaster and Tarski, resp. by accessibility inductive definitions. This time we follow an idea of Mendler [35] (reused by Amadio and Coupet-Grimal [7]) and reach the fixpoint by transfinite iteration. This technique enables us to give an interpretation to approximation types $\rho \leq \mu X.\sigma$.

Let O be some set which we call *origin set* and Φ a monotonic operator on sets. For an ordinal number α we define the *α-iterate* Φ^α by transfinite recursion on α:

$$\begin{array}{rcll} \Phi^0 & = & O & \\ \Phi^{\alpha+1} & = & \Phi(\Phi^\alpha) & \\ \Phi^\lambda & = & \bigcup_{\alpha<\lambda} \Phi^\alpha & \text{for } \lambda \text{ limit ordinal.} \end{array}$$

If $O \subseteq \Phi(O)$, then the hierarchy (Φ^α) is *cumulative*, that is, $\Phi^\alpha \subseteq \Phi^\beta$ for all $\alpha < \beta$. A cumulative hierarchy, which is also called a *chain*, will become stationary at some point in the iteration process, *i.e.*, it reaches a fixed-point. Since we consider

sets over a countable domain, the fixed-point will be reached latest at the least uncountable ordinal Ω and we have

$$\Phi(\Phi^\Omega) = \Phi^\Omega = \bigcup_{\alpha<\Omega} \Phi^\alpha.$$

If O is least in the considered collection of sets, then Φ^Ω is guaranteed to be the least fixed point. In the following, we consider operators on saturated sets and fix $O = \emptyset^*$ to be the least saturated set, namely the saturation of the empty set.

In order to define the interpretation $[\![\tau]\!]$ of type τ we need a valuation θ for the free type variables occurring in τ. Since we later also interpret terms which have free term variables, we introduce valuations $\theta : \Gamma$ which provide a substitute for all variables bound in context Γ by the following rules:

$$\frac{}{\cdot : \cdot} \qquad \frac{\theta : \Gamma \qquad M \in \mathbf{\Lambda}^+_{\boldsymbol{\mu}}}{(\theta, x \mapsto M) : (\Gamma, x{:}\tau)} \qquad \frac{\theta : \Gamma \qquad P \subseteq \mathbf{\Lambda}^+_{\boldsymbol{\mu}}}{(\theta, Y \mapsto P) : (\Gamma, Y{\leq}\mu X\sigma)}$$

For context triples, $\theta : (\Gamma; \vec{X}; \vec{X}')$ holds if θ additionally provides a set of terms Q for each variable in $\vec{X}, \vec{X}'$.

Note that $\theta : \Gamma$ only expresses that $\mathsf{dom}(\theta) = \mathsf{dom}(\Gamma)$ and that each term variable is mapped to a term and each type variable to a set of terms. We do not require that the assigned terms are type-correct nor are there any conditions on the assigned term sets. We do not even require Γ to be a well-formed context. Still, this notion of valuation is sufficient to define a type interpretation and prove monotonicity of this interpretation for all itypes, which are positive by definition. In the next section, we will sharpen our requirements on valuations by an additional judgement $\theta \in [\![\Gamma]\!]$.

Now to the semantics of types: Let τ be a type with free type variables in $\Delta = (\Gamma; \vec{X}; \vec{X}')$ and $\theta : \Delta$. We define the *interpretation* $[\![\tau]\!]\theta$ by recursion on τ:

$$\begin{aligned}
[\![Z]\!]\theta &= \theta(Z) \\
[\![1]\!]\theta &= \{()\}^* \\
[\![\sigma + \tau]\!]\theta &= \{\mathsf{inl}\, M : M \in [\![\sigma]\!]\theta\}^* \cup \{\mathsf{inr}\, M : M \in [\![\tau]\!]\theta\}^* \\
[\![\sigma \times \tau]\!]\theta &= \{(M_1, M_2) : M_1 \in [\![\sigma]\!]\theta, M_2 \in [\![\tau]\!]\theta\}^* \\
[\![\sigma \to \tau]\!]\theta &= [\![\sigma]\!]\theta \to [\![\tau]\!]\theta \\
[\![\mu X\sigma]\!]\theta &= \Phi^\Omega && \text{where} \quad \Phi = \Phi_{X.\sigma,\theta} \\
[\![\rho^+]\!]\theta &= \Phi^{\alpha+1} && \text{if} \quad [\![\rho]\!]\theta = \Phi^\alpha \\
[\![\forall Y{\leq}\mu X\sigma.\, \tau]\!]\theta &= \bigcap_{\alpha<\Omega} [\![\tau]\!](\theta, Y \mapsto \Phi^\alpha) && \text{where} \quad \Phi = \Phi_{X.\sigma,\theta} \\
\text{and} \quad \Phi_{X.\sigma,\theta}(Q) &= \{\mathsf{fold}\, M : M \in [\![\sigma]\!](\theta, X \mapsto Q)\}^*.
\end{aligned}$$

It can be shown that for $\to$-free inductive types the least-fixed point is reached at ordinal ω. Examples include $\mathsf{Nat} = \mu X.\, 1 + X$ and $\mathsf{ListN} = \mu X.\, 1 + \mathsf{Nat} \times X$, but also the empty type.

Example 5.3 (interpretation of the empty type)**.** A built-in empty type 0 would be interpreted as O. In our case, 0 is defined and $[\![0]\!]\theta = [\![\mu X.X]\!]\theta = \Phi^{\omega}_{X.X,\theta} = (Q \mapsto \{\mathsf{fold}\, M : M \in Q\}^*)^{\omega}$. But we can show that $[\![0]\!]$ is extensionally equal to O, i.e., for all $M \in [\![0]\!]$ exists some $N \in O$ such that $\mathsf{abort}\, M \longrightarrow^* \mathsf{abort}\, N$ (recall that $\mathsf{abort} = \mathsf{fix}\, g(x).\, g\,(\mathsf{unfold}\, x)$). The proof proceeds by induction on the approximations of $[\![0]\!]$ with a side induction on the saturation and makes use of the fact that $\mathsf{abort}\,(\mathsf{fold}\, N) \longrightarrow^+ \mathsf{abort}\, N$.

Inductive types involving function spaces can have a closure ordinal greater than ω. One instance is the type of tree ordinals Ord introduced in Example 3.6.

Example 5.4 (Ord requires iteration beyond ω)**.** Let Φ denote the generating operator for type Ord. Then the conversion function from natural numbers to tree ordinals $\mathsf{toOrd} \in [\![\mathsf{Nat}]\!] \to \Phi^{\alpha}$ if and only if $\alpha \geq \omega$. Hence the least approximation stage that $\mathsf{omega} = \mathsf{OLim}\ \mathsf{toOrd}$ inhabits is $\omega + 1$.

We continue by showing that the interpretation of a positive type is a monotone operator. But first, observe that the interpretation is preserved under substitution, that is, for types $\sigma(X)$ and ρ it holds that

$$[\![\sigma]\!](\theta, (X \mapsto [\![\rho]\!]\theta), \theta') = [\![\sigma(\rho)]\!](\theta, \theta').$$

Lemma 5.5 (monotonicity of itypes)**.** *Let* $\Gamma; \vec{X}; \vec{X}' \vdash \sigma : \mathsf{itype}$ *and* θ, θ' *be valuations with* $\theta(Y) = \theta'(Y)$ *for all* $Y \in \Gamma$, $\theta(X_i) \subseteq \theta'(X_i)$ *for all* $1 \leq i \leq |\vec{X}|$ *and* $\theta(X'_j) \supseteq \theta'(X'_j)$ *for all* $1 \leq j \leq |\vec{X}'|$. *Then* $[\![\sigma]\!]\theta \subseteq [\![\sigma]\!]\theta'$ *and* $\Phi^{\alpha}_{X.\sigma,\theta} \subseteq \Phi^{\alpha}_{X.\sigma,\theta'}$ *for all ordinals* α.

Proof. By induction on $\Gamma; \vec{X}; \vec{X}' \vdash \sigma : \mathsf{itype}$. □

For datatypes $\sigma : \mathsf{itype}$, the interpretation $[\![\sigma]\!]\theta$ is monotone in θ (in the sense of the previous lemma), hence also the operator $\Phi_{X.\sigma,\theta}$, which additionally has the property $O \subseteq \Phi_{X.\sigma,\theta}(O)$. This entails that the fixpoints exist and the iterates $(\Phi^{\alpha}_{X.\sigma,\theta})$ form a cumulative hierarchy. The following fact will be important for the proof of strong normalization: If a data value inhabits an approximation type, it is already found in a successor iterate.

Lemma 5.6. *If* $\mathsf{fold}\, M \in \Phi^{\alpha}$ *then* $\mathsf{fold}\, M \in \Phi^{\beta+1}$ *for some* $\beta < \alpha$.

Proof. By transfinite induction on α. □

5.3. Proof of strong normalization

To complete the proof of strong normalization, we first ensure that the interpretation of every type is a saturated set. Then we show that each well-typed term inhabits the interpretation of its type and therefore is strongly normalizing.

Wellformed contexts Γ cxt are interpreted as the set of all *sound valuations*. We define $\theta \in [\![\Gamma]\!]$ inductively by the following rules.

$$\frac{}{\cdot \in [\![\cdot]\!]} \qquad \frac{\theta \in [\![\Gamma]\!] \qquad M \in [\![\tau]\!]\theta}{(\theta, x \mapsto M) \in [\![\Gamma, x{:}\tau]\!]} \qquad \frac{\theta \in [\![\Gamma]\!] \qquad \Phi = \Phi_{X.\sigma,\theta}}{(\theta, Y \mapsto \Phi^\alpha) \in [\![\Gamma, Y \leq \mu X\sigma]\!]}$$

We now can verify that our subtyping calculus is sound:

Lemma 5.7 (soundness of subtyping)**.** *If $\Gamma \vdash \rho \leq \sigma$ then for all $\theta \in [\![\Gamma]\!]$ it holds that*

$$[\![\rho]\!]\theta = \Phi^\alpha \subseteq \Phi^\beta = [\![\sigma]\!]\theta$$

for some operator Φ and some ordinals α and β.

Proof. By induction over the derivation of $\Gamma \vdash \rho \leq \sigma$. □

Note that in the previous lemma we do not always have $\alpha \leq \beta$, since subtyping accepts $(\mu X\sigma)^+ \leq \mu X\sigma$, for example.

Lemma 5.8 (saturatedness of type interpretations)**.** *If $\Gamma \vdash \tau : \mathsf{type}$ then for all $\theta \in [\![\Gamma]\!]$ we have $[\![\tau]\!]\theta \in \mathcal{SAT}$.*

Proof. State similar propositions for the other three type-related judgements on Table 2 and prove all propositions by simultaneous induction. □

Lemma 5.9 (wellformed contexts are satisfiable)**.** *If Γ cxt then there exists some $\theta \in [\![\Gamma]\!]$.*

Proof. By induction on Γ cxt. For the empty context, we take the empty valuation. In case $\Gamma = (\Gamma', x{:}\tau)$ with $\Gamma' \vdash \tau : \mathsf{type}$ we know by Lemma 3.1 that Γ' cxt and obtain a $\theta' \in [\![\Gamma']\!]$ by induction hypothesis. Since by the last lemma $[\![\tau]\!]\theta'$ is a saturated set, it contains x and hence $\theta = (\theta', x \mapsto x)$ is the desired valuation for Γ. In the remaining case $\Gamma = (\Gamma', Y \leq \mu X\sigma)$ we similarly obtain a $\theta' \in [\![\Gamma']\!]$ by induction hypothesis and extend it by the mapping $Y \mapsto [\![\mu X\sigma]\!]\theta'$. □

Now we have assembled all the pieces to prove strong normalization.

Theorem 5.10 (soundness of typing)**.** *If $\Gamma \vdash M : \tau$ then for all $\theta \in [\![\Gamma]\!]$ we have $M\theta \in [\![\tau]\!]\theta$.*

Proof. By induction on $\Gamma \vdash M : \tau$. We only show the interesting cases of fold, unfold and fix.

Case

$$\frac{\Gamma \vdash \rho \leq \mu X.\sigma(X) \qquad \Gamma \vdash M : \sigma(\rho)}{\Gamma \vdash \mathsf{fold}\, M : \rho^+}$$

From the assumption follows $[\![\rho]\!]\theta = \Phi^\alpha$ for $\Phi(Q) = \{\mathsf{fold}\, N : N \in [\![\sigma]\!](\theta, X \mapsto Q)\}^*$ and some α. By induction hypothesis, $M\theta \in [\![\sigma]\!](\theta, X \mapsto \Phi^\alpha)$. Thus, $\mathsf{fold}\, M\theta \in \Phi^{\alpha+1} = [\![\rho^+]\!]\theta$.

Case

$$\frac{\Gamma \vdash \rho \leq \mu X.\sigma(X) \qquad \Gamma \vdash M : \rho^+}{\Gamma \vdash \mathsf{unfold}\, M : \sigma(\rho)}$$

As in the last case, $[\![\rho]\!]\theta = \Phi^\alpha$. By induction hypothesis, $M\theta \in \Phi^{\alpha+1}$. We show $\mathsf{unfold}\, M\theta \in [\![\sigma(\rho)]\!]\theta$ by side induction on the saturation of Φ^α.

Subcase $M\theta = E[x]$. Then $\mathsf{unfold}\, M\theta = \mathsf{unfold}\, E[x]$ is neutral and strongly normalizing and therefore element of $[\![\sigma(\rho)]\!]\theta$.

Subcase $M\theta = E[M_0]$, $M_0 \in \mathsf{SN}$, $M_0 \longrightarrow_\beta M_1$ and $E[M_1] \in \Phi^\alpha$. By side induction hypothesis, $\mathsf{unfold}\, E[M_1] \in [\![\sigma(\rho)]\!]\theta$ and by saturation also $\mathsf{unfold}\, E[M_0] \in [\![\sigma(\rho)]\!]\theta$.

Subcase $M\theta = \mathsf{fold}\, N$ with $N \in [\![\sigma]\!](\theta, X \mapsto \Phi^\alpha) = [\![\sigma(\rho)]\!]\theta$. Since we can reduce $\mathsf{unfold}(\mathsf{fold}\, N) \longrightarrow_\beta N$, by saturatedness it follows that $\mathsf{unfold}\, M\theta \in [\![\sigma(\rho)]\!]\theta$.

Case

$$\frac{\Gamma,\ Y{\leq}\mu X\sigma,\ g{:}\forall Y \to \tau(Y),\ x{:}Y^+ \ \vdash M : \tau(Y^+)}{\Gamma \vdash \mathsf{fix}\, g(x).M : \forall Y{\leq}\mu X\sigma.\, Y \to \tau(Y)}$$

where Y appears only positively in τ. Let $\Phi = \Phi_{X.\sigma,\theta}$. Then $[\![\forall Y \leq \mu X\sigma.\, Y \to \tau(Y)]\!]\theta = \bigcap_{\alpha<\Omega}(\Phi^\alpha \to [\![\tau]\!](\theta, Y \mapsto \Phi^\alpha))$. We show for all ordinals α,

$$(\mathsf{fix}\, g(x).M)\theta \in \Phi^\alpha \to [\![\tau]\!](\theta, Y \mapsto \Phi^\alpha)$$

by a first side induction on α. Fix the ordinal α and some term $N \in \Phi^\alpha$. We show

$$(\mathsf{fix}\, g(x).M)\theta\, N \in [\![\tau]\!](\theta, Y \mapsto \Phi^\alpha)$$

by a second side induction on the saturation of Φ^α.

Subcase $N = E[x]$. Then $(\mathsf{fix}\, g(x).M\theta)\, E[x]$ is neutral and in $[\![\tau]\!](\theta, Y \mapsto \Phi^\alpha)$.

Subcase $N = E[N_1]$ with $N_1 \longrightarrow_\beta N_2$ and $E[N_2] \in \Phi^\alpha$. By the second side induction hypothesis, $(\mathsf{fix}\, g(x).M\theta)\, E[N_2] \in [\![\tau]\!](\theta, Y \mapsto \Phi^\alpha)$, and by saturatedness also $(\mathsf{fix}\, g(x).M\theta)\, E[N_1] \in [\![\tau]\!](\theta, Y \mapsto \Phi^\alpha)$.

Subcase $N = \mathsf{fold}\, N' \in \Phi^{\beta+1} \subseteq \Phi^\alpha$ (with Lem. 5.6). By the first side induction hypothesis $\mathsf{fix}\, g(x).M \in (\Phi^\beta \to [\![\tau]\!](\theta, Y \mapsto \Phi^\beta))$. Hence,

$$\begin{aligned}\theta' &= (\theta, Y \mapsto \Phi^\beta, g \mapsto \mathsf{fix}\, g(x).M, x \mapsto N)\\ &\in [\![\Gamma,\ Y{\leq}\mu X\sigma,\ g{:}Y \to \tau(Y),\ x{:}Y^+]\!].\end{aligned}$$

By main induction hypothesis

$$\begin{aligned}M\theta' \in [\![\tau(Y^+)]\!]\theta' &= [\![\tau]\!](\theta, Y \mapsto \Phi^{\beta+1})\\ &\subseteq [\![\tau]\!](\theta, Y \mapsto \Phi^\alpha),\end{aligned}$$

since Y appears only positively in τ. We can conclude by saturation, since

$$(\mathsf{fix}\, g(x).M)\theta\, (\mathsf{fold}\, N) \longrightarrow_\beta M\theta'. \qquad \square$$

Remark 5.11 (transfinite induction)**.** The soundness proof for the fixed point rule uses a course-of-value transfinite induction on α and the fact that new canonical inhabitants of an inductive datatype are only found at successor iterates (Lem. 5.6). It is quite instructive to consider the less economic proof by a simple transfinite induction. In this case, the base case $\alpha = 0$ holds since Φ^0 contains no canonical forms $\mathsf{fold}\, N$, the step case $\alpha \rightsquigarrow \alpha + 1$ follows from the main induction hypothesis and the limit case $\alpha = \lambda$ needs monotonicity of $[\![\tau(Y)]\!]$ which is ensured by the positivity condition for τ.

Corollary 5.12 (strong normalization)**.** *If* $\Gamma \vdash M : \tau$ *then* $M \in \mathsf{SN}$.

Proof. By Lemma 3.5 the context Γ is wellformed. This entails that there exists a valuation $\theta \in [\![\Gamma]\!]$, as constructed in the proof of Lemma 5.9, which maps all term variables to themselves. Hence $M\theta = M$, and by soundness of typing $M \in [\![\tau]\!]\theta \subseteq \mathsf{SN}$, since the interpretation of τ is saturated. □

Remark 5.13 (type-based termination in DML)**.** Xi's [50] more sophisticated type system admits inconsistent contexts Γ. Consequently, strong normalization does not hold although he can prove soundness of typing analogously to our Theorem 5.10. But since in his case, a $\theta \in [\![\Gamma]\!]$ as in the last proof does not always exist, he can show normalization only for closed terms (empty Γ).

Strong normalization excludes infinite reduction sequences. More directly, we can show that each possibly infinite reduction sequence is in fact finite. As a consequence, each well-typed $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$-program reduces to a value.

Lemma 5.14 (termination)**.** *If* $M \in \mathsf{SN}$ *and* $M \longrightarrow^\infty M'$ *then* $M \longrightarrow^* M'$.

Proof. By induction on $M \in \mathsf{SN}$ and cases on $M \longrightarrow^\infty M'$. □

Corollary 5.15 (reduction to value)**.** *If* $\Gamma \vdash M : \tau$ *then* $M \longrightarrow^* V$.

Proof. By Corollary 1.2, $M \longrightarrow^\infty V$. Since $M \in \mathsf{SN}$ by Corollary 5.12, Lemma 5.14 implies $M \longrightarrow^* V$. □

6. Extensions

Two issues are briefly discussed in this section: The positivity requirement for result types in the recursion rule and approximations of coinductive types. Both topics are treated rather informally and incompletely, more systematic approaches can be found in Pareto [37], Barthe *et al.* [9] and Abel [3].

6.1. On non-monotonic result types of recursion

In Section 2 we promised to motivate why we required the result type $\tau(Y)$ of recursion to be positive in Y. In his first presentation, Giménez [23] had no such requirement, it was added later [9]. Indeed, if we simply drop the side condition,

$$\frac{\Gamma,\ Y \leq \mu X\sigma,\ g : Y \to \tau(Y),\ x : Y^+ \ \vdash M : \tau(Y^+)}{\Gamma \vdash \mathsf{fix}\, g(x).M : \forall Y \leq \mu X\sigma.\, Y \to \tau(Y)}$$

we can type non-terminating functions, which we will demonstrate in the following. Recall the type $\mathsf{Nat} = \mu X.\, 1 + X$ with the defined constructors Zero and Succ from Section 1. We introduce an abbreviation

$$\begin{array}{lcl} \mathsf{maybe} & : & (1+\sigma) \to \tau \to (\sigma \to \tau) \to \tau \\ \mathsf{maybe} & := & \lambda t \lambda b \lambda f.\ \mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}\, \_ \Rightarrow b \mid \mathsf{inr}\, a \Rightarrow f\, a \end{array}$$

for any types σ and τ. For $\rho \leq \mathsf{Nat}$, we define:

$$\begin{array}{lcl} \mathsf{shift} & : & (\mathsf{Nat} \to (1+\rho^+)) \to \mathsf{Nat} \to (1+\rho) \\ \mathsf{shift} & := & \lambda f \lambda n.\ \mathsf{maybe}\ (f\,(\mathsf{Succ}\, n))\ (\mathsf{inl}())\ (\lambda z.\ \mathsf{unfold}\, z) \end{array}$$

$$\begin{array}{lcl} \mathsf{inc} & : & \mathsf{Nat} \to (1 + \mathsf{Nat}) \\ \mathsf{inc} & := & \lambda n.\ \mathsf{inr}\,(\mathsf{Succ}\, n) \end{array}$$

The function inc is a fixed-point of shift as we can show by the following reduction sequence:

$$\begin{array}{rcl} \mathsf{shift}\ \mathsf{inc} & \longrightarrow^+ & \lambda n.\ \mathsf{maybe}\ (\mathsf{inr}\,(\mathsf{Succ}\,(\mathsf{Succ}\, n)))\ (\mathsf{inl}\,())\ (\lambda z.\ \mathsf{unfold}\, z) \\ & \longrightarrow^+ & \lambda n.\ \mathsf{unfold}\,(\mathsf{Succ}\,(\mathsf{Succ}\, n)) \\ & \longrightarrow^+ & \lambda n.\ \mathsf{inr}\,(\mathsf{Succ}\, n) \\ & = & \mathsf{inc} \end{array}$$

Using shift, we can construct a recursive function g with result type $\tau(Y) = (\mathsf{Nat} \to (1+Y)) \to 1$ which loops on input inc.

$$\begin{array}{lcl} \mathsf{g} & : & \forall Y \leq \mathsf{Nat}.\ Y \to (\mathsf{Nat} \to (1+Y)) \to 1 \\ \mathsf{g} & := & \mathsf{fix}\, g(\_).\, \lambda f.\ \mathsf{maybe}\ (f\ \mathsf{Zero})\ () \\ & & \qquad (\lambda x.\ \mathsf{maybe}\ (\mathsf{unfold}\ x)\ () \\ & & \qquad\quad (\lambda y.\ g\, y\, (\mathsf{shift}\, f))) \end{array}$$

This function is well typed, we can infer the following types for some subexpressions:

$$\begin{array}{rcl} g & : & Y \to (\mathsf{Nat} \to (1+Y)) \to 1 \\ f & : & \mathsf{Nat} \to (1 + Y^+) \\ x & : & Y^+ \\ y & : & Y \\ \mathsf{shift}\, f & : & \mathsf{Nat} \to (1+Y) \end{array}$$

Hence, the application of the recursive function $g\, y\, (\mathsf{shift}\, f)$ is accepted by the type system. By a simple computation,

$$\mathsf{g}\ \mathsf{Zero}\ \mathsf{inc} \longrightarrow^+ \mathsf{g}\ \mathsf{Zero}\ (\mathsf{shift}\ \mathsf{inc}) \longrightarrow^+ \mathsf{g}\ \mathsf{Zero}\ \mathsf{inc}$$

and we have created a loop.

What exactly causes non-termination? Not the mere negative occurrence of Y because the type $\forall Y \leq \mathsf{Nat}.\, Y \to (1+Y) \to 1$ would be unproblematic for terminating recursion (*cf.* Abel [3]). Only through the presence of an argument f of type

$\mathsf{Nat} \to (1+Y)$, arbitrary natural numbers n can "sneak through the back door f" being declared of type Y on the way and present themselves as safe arguments for recursion.

To be on the safe side, all arguments $a : \tau'(Y)$ (for $Y \leq \mu X\sigma$) to a recursive function must be *cocontinuous* [3] resp. *overshooting* (in the terminology of Hughes, Pareto and Sabry [28]). This means that for each limit ordinal λ,

$$[\![\tau']\!](Y \mapsto \Phi^\lambda) \subseteq \bigcup_{\alpha<\lambda} [\![\tau']\!](Y \mapsto \Phi^\alpha),$$

where $\Phi = \Phi_{X.\sigma}$. In our counterexample (where $\tau'(Y) = \mathsf{Nat} \to (1+Y)$), the argument inc violates this requirement for $\lambda = \omega$. While inc clearly inhabits the set $[\![\mathsf{Nat} \to (1+\mathsf{Nat})]\!]$ on the left hand side, it is not member of the right hand side set $[\![\mathsf{Nat} \to (1+Y)]\!](Y \mapsto \Phi^\alpha)$ for any $\alpha < \omega$, since Φ^α contains only codes for the natural numbers $< \alpha$.

A systematic treatment of admissible result types of recursion can be found in Pareto's Ph.D. thesis [37], for evaluation of closed terms, and in Abel [3], for evaluation under binders.

6.2. On coinductive datatypes

Approximation types have been used to check productivity of infinite structures like streams (*cf.* Hughes, Pareto and Sabry [28], Amadio and Coupet-Grimal [7], Giménez [23] and Barthe *et al.* [9]). In the following, we demonstrate how to simulate productivity checking for streams in our type system *via* an encoding of streams as functions over natural numbers.

Streams, being infinite sequences of elements of a fixed type, can – in finitary languages – only be defined by *recursion*. The dual to termination in the case of functions is *productivity* for streams. It means that at any time, one can unfold a stream and extract its first element, obtaining a remainder which is again productive. What the size is for inductive data is *definedness* for coinductive structures. If we define U to be the set of all partial streams of natural numbers,

$$S_n := \{s \in U \mid s \text{ can be unfolded } n \text{ times}\}$$

characterizes the streams of definedness n. While the corresponding predicate L_n for lists is covariant in its argument n (see Sect. 2), *i.e.*, $L_n \subseteq L_m$ for $n \leq m$, definedness is contravariant: $S_n \subseteq S_m$ for $m \leq n$. The set of *productive* streams $S = S_\omega$ is the bottom element of the approximation chain (S_n). Productive streams can be obtained *via* the induction rule "*if* $s \in S_n$ *implies* $s \in S_{n+1}$ *for all* n*, then* $s \in S$". In the following, we present rules for productive streams which could form an extension of $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$. Later we will show that these rules are admissible in a small extension of our original calculus.

Formation.

$$\frac{\Gamma; \vec{X}; \vec{X}' \vdash \sigma : \mathsf{itype}}{\Gamma; \vec{X}; \vec{X}' \vdash \mathsf{Stream}(\sigma) : \mathsf{itype}} \qquad \frac{\Gamma, Y \geq \mathsf{Stream}(\sigma) \vdash \tau(Y) : \mathsf{type}}{\Gamma \vdash \forall Y \geq \mathsf{Stream}(\sigma).\, \tau(Y) : \mathsf{type}}$$

Subtyping.

$$\begin{array}{l} \mathsf{Stream}(\sigma) \leq \ldots \leq \mathsf{Stream}(\sigma)^{++} \leq \mathsf{Stream}(\sigma)^{+} \leq \mathsf{Stream}(\sigma) \\ \mathsf{Stream}(\sigma) \leq \ldots \leq Y^{++} \leq Y^{+} \leq Y \qquad (\text{for } Y \geq \mathsf{Stream}(\sigma)) \end{array}$$

Elimination.

$$\frac{}{\Gamma \vdash \mathsf{hd} : \forall Y \geq \mathsf{Stream}(\sigma).\, Y^{+} \to \sigma} \qquad \frac{}{\Gamma \vdash \mathsf{tl} : \forall Y \geq \mathsf{Stream}(\sigma).\, Y^{+} \to Y}$$

Introduction.

$$\frac{}{\Gamma \vdash \mathsf{scons} : \sigma \to \forall Y \geq \mathsf{Stream}(\sigma).\, Y \to Y^{+}}$$

Recursion.

$$\frac{\Gamma,\, Y \geq \mathsf{Stream}(\sigma),\, g{:}\tau(Y) \to Y \vdash M : \tau(Y^{+}) \to Y^{+} \qquad Y \text{ pos. in } \tau(Y)}{\Gamma \vdash \mathsf{fix}_2\, g.M : \forall Y \geq \mathsf{Stream}(\sigma).\, \tau(Y) \to Y}$$

Reduction.

$$\begin{array}{rcl} \mathsf{hd}\ (\mathsf{scons}\ M\ N) & \longrightarrow_\beta & M \\ \mathsf{tl}\ (\mathsf{scons}\ M\ N) & \longrightarrow_\beta & N \\ \\ \mathsf{hd}\ ((\mathsf{fix}_2\, g.M)\, N) & \longrightarrow_\beta & \mathsf{hd}\ ([\mathsf{fix}_2\, g.M/g]M\ N) \\ \mathsf{tl}\ ((\mathsf{fix}_2\, g.M)\, N) & \longrightarrow_\beta & \mathsf{tl}\ ([\mathsf{fix}_2\, g.M/g]M\ N) \end{array}$$

These rules enable us, for instance, to code the sequence of natural numbers starting at a given n, and a sorted merging of streams.

$$\begin{array}{lcl} \mathsf{nats} & : & \forall Y \geq \mathsf{Stream}(\mathsf{Nat}).\, \mathsf{Nat} \to Y \\ \mathsf{nats} & := & \mathsf{fix}_2\, \mathit{nats}.\, \lambda n.\ \mathsf{scons}\ n\ (\mathit{nats}\, (\mathsf{Succ}\, n)) \\ \\ \mathsf{merge} & : & \forall Y \geq \mathsf{Stream}(\mathsf{Nat}).\, Y \times Y \to Y \\ \mathsf{merge} & := & \mathsf{fix}_2\, \mathit{merge}.\, \lambda p. \\ & & \quad \mathsf{let}\, x = \mathsf{hd}\, (\mathsf{fst}\, p)\, \mathsf{in} \\ & & \quad \mathsf{let}\, y = \mathsf{hd}\, (\mathsf{snd}\, p)\, \mathsf{in} \\ & & \quad \mathsf{if}\, x < y\ \mathsf{then}\ \ \mathsf{scons}\, x\, (\mathit{merge}\, (\mathsf{tl}\, (\mathsf{fst}\, p)),\ \mathsf{snd}\, p) \\ & & \qquad\qquad \mathsf{else}\ \ \mathsf{scons}\, y\, (\mathit{merge}\, (\mathsf{fst}\, p),\ \mathsf{tl}\, (\mathsf{snd}\, p)) \end{array}$$

We claim that all we need to simulate streams *via* the definition $\mathsf{Stream}(\sigma) = \mathsf{Nat} \to \sigma$ in our type system, is a little more liberal handling of subtyping and bounded quantification and a new fixed-point combinator which implements

recursion on the second argument. Thus, we extend $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ by the following rules instead:

Bounded quantification and subtyping.

$$\frac{\Gamma, X{\leq}\mu Z\sigma \vdash \tau(X) : \mathsf{type}}{\Gamma \vdash \forall X{\leq}\mu Z\sigma.\, \tau(X) : \mathsf{type}} \qquad \frac{\Gamma \vdash \sigma' \leq \sigma \qquad \Gamma \vdash \tau \leq \tau'}{\Gamma \vdash \sigma \to \tau \leq \sigma' \to \tau'} \qquad \frac{}{\Gamma \vdash \sigma \leq \sigma}$$

Recursion on the second argument.

$$\frac{\begin{array}{l} X \text{ neg. in } \tau \\ X \text{ pos. in } \tau' \\ \Gamma,\ X{\leq}\mu Z\sigma,\ g : \tau(X) \to X \to \tau'(X) \ \vdash\ M : \tau(X^+) \to X^+ \to \tau'(X^+) \end{array}}{\Gamma \vdash \mathsf{fix}_2\, g.M : \forall X{\leq}\mu Z\sigma.\, \tau(X) \to X \to \tau'(X)}$$

Reduction.

$$(\mathsf{fix}_2\, g.M)\ N_1\ (\mathsf{fold}\, N_2) \quad \longrightarrow_\beta \quad [\mathsf{fix}_2\, g.M/g]M\ N_1\ (\mathsf{fold}\, N_2)$$

To encode streams in the extended $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$, we replace all quantifications $\forall Y \geq \mathsf{Stream}(\sigma)$ by $\forall X \leq \mathsf{Nat}$, all occurrences of Y^i by $(X^i \to \sigma)$ and all remaining occurrences of $\mathsf{Stream}(\sigma)^i$ by $(\mathsf{Nat}^i \to \sigma)$. The formation rules for streams are immediately valid, and subtyping can be justified using the rule for arrow, for example $Y^+ \leq Y$:

$$\frac{X \leq X^+ \qquad \sigma \leq \sigma}{(X^+ \to \sigma) \leq (X \to \sigma)}$$

Destructors and the constructor for streams can be defined with the correct typing:

$$\begin{array}{lcl} \mathsf{hd} & : & \forall X{\leq}\mathsf{Nat}.\ (X^+ \to \sigma) \to \sigma \\ \mathsf{hd} & := & \lambda s.\, s\, \mathsf{Zero} \\ \mathsf{tl} & : & \forall X{\leq}\mathsf{Nat}.\ (X^+ \to \sigma) \to X \to \sigma \\ \mathsf{tl} & := & \lambda s \lambda n.\, s\, (\mathsf{Succ}\, n) \\ \mathsf{scons} & : & \sigma \to \forall X{\leq}\mathsf{Nat}.\, (X \to \sigma) \to X^+ \to \sigma \\ \mathsf{scons} & := & \lambda a \lambda s \lambda n.\ \mathsf{maybe}\ (\mathsf{unfold}\ n)\ a\ s \end{array}$$

In the typing of scons we need the new introduction rule for bounded quantification. The fixed-point combinator for streams is a special case of recursion over the second argument, with $\tau'(X) = \sigma$. The polarity conditions are satisfied, since Y positive in $\tau(Y)$ implies X negative in $\tau(X \to \sigma)$. By calculation, we verify the

computational behavior for streams:

$$\begin{array}{rcl}
\mathsf{hd}\ (\mathsf{scons}\ M\ N) & \longrightarrow & (\lambda n.\ \mathsf{maybe}\ (\mathsf{unfold}\, n)\ M\ N)\ \mathsf{Zero} \\
& \longrightarrow^{+} & M \\
\mathsf{tl}\ (\mathsf{scons}\ M\ N) & \longrightarrow & \lambda n.\ (\lambda n'.\ \mathsf{maybe}\ (\mathsf{unfold}\, n')\ M\ N)\ (\mathsf{Succ}\, n) \\
& \longrightarrow^{+} & \lambda n.\ N\, n \\
\mathsf{hd}\ ((\mathsf{fix}_2\, g.M)\, N) & \longrightarrow & ((\mathsf{fix}_2\, g.M)\, N)\ \mathsf{Zero} \\
& \longrightarrow & ([\mathsf{fix}_2\, g.M/g]M\, N)\ \mathsf{Zero} \\
& = & \mathsf{hd}\ ([\mathsf{fix}_2\, g.M/g]M\, N) \\
\mathsf{tl}\ ((\mathsf{fix}_2\, g.M)\, N) & \longrightarrow & \lambda n.\ ((\mathsf{fix}_2\, g.M)\, N)\ (\mathsf{Succ}\, n) \\
& \longrightarrow & \lambda n.\ ([\mathsf{fix}_2\, g.M/g]M\, N)\ (\mathsf{Succ}\, n) \\
& \longleftarrow & \mathsf{tl}\ ([\mathsf{fix}_2\, g.M/g]M\, N)
\end{array}$$

Reduction is not completely simulated by the encoding since $\mathsf{tl}\, M$ is equal to a lambda-abstraction. But for each axiom $M \longrightarrow_\beta M'$ for streams, M and M' are $\beta\eta$-equal in the encoding. The reader is invited to check that the programs for nats and merge also inhabit the translated types.

$$\begin{array}{rcl}
\mathsf{nats} & : & \forall X{\leq}\mathsf{Nat}.\, \mathsf{Nat} \to X \to \mathsf{Nat} \\
\mathsf{merge} & : & \forall X{\leq}\mathsf{Nat}.\, (X \to \mathsf{Nat}) \times (X \to \mathsf{Nat}) \to X \to \mathsf{Nat}
\end{array}$$

Summing up, we have managed to justify the use of approximation stages also for coinductive types in the example of streams. We hope that we hereby have provided an easy access to the rules of productivity checking for infinite structures, which are – in the experience of the author – harder to communicate than the rules for termination checking.

7. Applications

After establishing soundness, we show the effectiveness of the presented approach in this section. We sketch how the explored principles can be applied in functional programming, theorem proving, and, as a specific example, for resource bound certification.

7.1. Functional programming

Throughout the article we have demonstrated type-based termination on small functional programs already. In the following we present another example which shows that our system can handle mutual recursion, interleaving inductive types, and even non-strictly positive types.

We introduce two abbreviations for certain datatypes: $X\,\mathsf{list}$ for lists over some type X and term for first-order terms with continuations.

$$\begin{array}{lcl} X\,\mathsf{list} & = & \mu Y.\, 1 + X \times Y \\ \mathsf{term} & = & \mu X.\, \mathsf{int} + \mathsf{int} \times (X\,\mathsf{list}) + ((X \to 0) \to 0) \end{array}$$

The first-order terms we consider are constructed from variables, n-ary function symbols and continuations[6]. We have three constructors:

$$\begin{array}{lcl} \mathsf{Var} & : & \mathsf{int} \to \mathsf{term} \\ \mathsf{Func} & : & \mathsf{int} \times \mathsf{term}\,\mathsf{list} \to \mathsf{term} \\ \mathsf{Mu} & : & ((\mathsf{term} \to 0) \to 0) \to \mathsf{term} \end{array}$$

$\mathsf{Var}(i)$ denotes the ith variable, $\mathsf{Func}(i, l)$ the ith function symbol applied to the terms of the list l and

$$\mathsf{Mu}(\lambda a^{\mathsf{term}\to 0}.\, b^{\mathsf{term}\to 0}\, t)$$

the term that stores the current continuation in a, retrieves a continuation b and throws the term t at b.

Note that term is the rare example of a *non-strictly positive* type, furthermore term and list are *interleaving*, also called *nested*. Nevertheless, our system accepts the following mutually recursive substitution function.

$$\begin{array}{l} \mathsf{subst}\ :\ (\mathsf{int} \to \mathsf{term}) \to \forall X \leq \mathsf{term}.\, X \to \mathsf{term} \\ \\ \mathsf{subst}\ \ f\ \ (\mathsf{Var}(i))^{X^+} \quad = f(i) \\ \mathsf{subst}\ \ f\ \ (\mathsf{Func}(i,l))^{X^+} = \mathsf{Func}(i, \mathsf{sl}(l^{X\,\mathsf{list}})) \\ \mathsf{where}\ \mathsf{sl}\ :\ \forall Y \leq X\,\mathsf{list}.\, Y \to \mathsf{term}\,\mathsf{list} \\ \qquad \mathsf{sl}\ \ ([])^{Y^+} \quad = [] \\ \qquad \mathsf{sl}\ \ (t :: ts)^{Y^+} \quad = (\mathsf{subst}\ \ f\, t^X) :: \mathsf{sl}\ \ ts^Y \\ \mathsf{subst}\ \ f\ \ (\mathsf{Mu}(t))^{Y^+} \quad = \mathsf{Mu}\,(\lambda a^{\mathsf{term}\to 0}.\, t^{(Y\to 0)\to 0}\,(\lambda r^Y. \\ \qquad\qquad\qquad\qquad a\,(\mathsf{subst}\ \ f\, r^Y))) \end{array}$$

Recursion over interleaving inductive datatypes is most naturally expressed by mutually recursive functions. In our case it is crucial that this mutual recursion is not simultaneous but interleaving, as the approximation of the type of terms $X \leq \mathsf{term}$ must be available with in the inner function sl for the approximation $Y \leq X\,\mathsf{list}$. Informally, X contains terms upto a certain height and $X\,\mathsf{list}$ lists of such terms. Hence, Y contains lists of restricted length which consist of terms of bounded height. The typing ensures that the call $\mathsf{subst}\ f\, t$ within sl terminates, no call graphs are necessary (*cf.* Abel and Altenkirch [5] or Lee, Jones and Ben-Amram [30]).

[6] The continuations we consider are modeled after the $\lambda\mu$-calculus (Parigot [38]), which is very well presented in Bierman [10]. The chosen representation as positive type is derived from Abel [2].

The last line encodes substitution for continuations, applying our technique for non-strictly positive types. Here, termination is not obvious at all, also not for measure-based approaches – it is not clear which measure one should use. For sure, termination cannot be checked with the usual untyped notion of the subterm ordering, since r is no subterm of t. Type-based termination succeeds since the types provide us with some data flow information which is not available in subterm calculi.

In every-day programming, some functions are not terminating on all inputs. Others may be terminating, but the termination proof may involve complicated arguments and cannot be checked automatically. For that reason, it is desirable to allow both *total and partial* functions in a program. This can be realized naturally in our system by simply adding the fix-rule of $\mathbf{\Lambda_\mu}$ which does not impose any restriction on recursion. In the extended system, functions are total if their typing derivation does not use the liberal fix-rule. To keep efficiency in an implementation of the type system, functions could be tagged as *total* resp. *possibly partial* by the compiler.

7.2. Theorem proving

As pointed out in the introduction, the typed-based approach to termination is suitable for theorem provers based on theories with inductive definitions, *e.g.*, Agda [16], Coq [29], and LEGO [43]. However, it needs to be adapted to dependent type theories. A first proposal has been put forth by Blanqui [12]. He combines parts of the approaches of Xi [50], Giménez *et al.* [9, 23] and Hughes *et al.* [28] to obtain a type-based termination criterion for the Calculus of Algebraic Constructions.

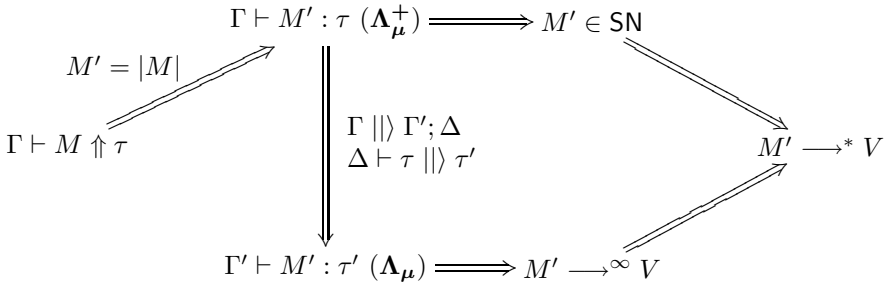
7.3. Resource bound certification

Crary and Weirich [19] describe a language LXres for resource bound certification. To estimate the cost of defined operations they use static *cost functions*. Technically these cost functions are so-called *constructors* in their type theory which is based on F^ω. For type-checking to be decidable, these constructors must be terminating. Currently they are restricted to Mendler-style primitive recursive functions (see Sect. 2). Our extension could be integrated into their type theory which would strictly increase the expressiveness of their language.

8. Conclusion

We have presented a method of ensuring termination by types, shown soundness, given an algorithm for type checking and discussed applications besides the intended use in theorem provers. Table 7 gives an overview over the technical results of this article. In Section 4 we have formalized a type checking algorithm which is sound for $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$. Typing of $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$ excludes infinite reduction sequences as proven in Section 5. Each $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$-program can also be assigned a $\mathbf{\Lambda_\mu}$-type which is

TABLE 7. Summary of technical results.



obtained by erasure (Sect. 3.3) and guarantees that evaluation does not get stuck (Sect. 1). Putting things together, each type-checkable $\mathbf{\Lambda}^+_{\boldsymbol{\mu}}$-program reduces to a value in finite time.

Type based termination combines three program analyses: ordinary type checking, termination checking and what is called *reduction checking* (Pientka [40]) resp. size checking (Chin and Khoo [15]). We expect that this combination will make implementations of normalizing calculi more succinct, efficient and transparent to the user. However, to give substance to our claim, implementations of type based termination have to be tested in practice.

We conclude with a discussion of related work on sized types and termination.

8.1. HUGHES, PARETO *ET AL.*

Hughes, Pareto [27] and Sabry [28] use sized types for termination and productivity checking of functional programs. A detailed treatment is given by Pareto in his thesis [37]. His type system refers to approximations of the type of natural numbers as $\mathsf{Nat}\, i$ where i is a size expression. The full type of natural numbers is represented as $\mathsf{Nat}\, \omega$. Instead of bounded quantification he uses quantification over size variables, and our next-stage operation $(\cdot)^+$ is simulated by a successor operation on sizes. For instance, the type $\mathsf{Nat} \to \forall Y \leq \mathsf{ListN}.\, Y \to Y \times Y$ of the pivot function would be written $\mathsf{Nat} \to \forall i.\, \mathsf{List}\, i\, \mathsf{Nat} \to \mathsf{List}\, i\, \mathsf{Nat} \times \mathsf{List}\, i\, \mathsf{Nat}$ in his system, where i is a size variable.

Pareto's type language is more expressive than ours in two aspects: First, since sizes are separated from approximation stages, he can assign, for instance, the precise type $\forall \alpha.\, \forall i.\, \mathsf{List}\, i\, \alpha \to \mathsf{Nat}\, i$ to the length function; furthermore, this enables size information for polymorphic functions like map: $\forall \alpha \forall \beta.\, (\alpha \to \beta) \to \forall i.\, \mathsf{List}\, i\, \alpha \to \mathsf{List}\, i\, \beta$. Secondly, size expressions can contain addition of sizes and multiplication with a constant, thus, a more precise type $\forall \alpha\, \forall i.\, \mathsf{List}\, i\, \alpha \to \forall j.\, \mathsf{List}\, (j+1)\, \alpha \to \mathsf{List}\, (i+j)\, \alpha$ of the append function is possible. On the other

hand, since sizes are bounded by ordinal ω, he can not treat inductive types with embedded function spaces like Ord (Ex. 3.6). Also, his semantics is denotational and based on domains, hence he only shows termination for closed programs, whereas we permit evaluation under binders.

Notwithstanding the different interpretations, the formation of recursive functions underlies the same constraints in Pareto's and our approach. Our side condition on the fixed point rule (see Sect. 6.1) is a restriction of his condition to instantiate a size quantifier $\forall i$ to the limit ordinal ω. He has another side condition on the fixed point rule, the so called *bottom check*. It expresses that the type $\forall i.\, \tau$ of a recursive function must, when i is instantiated to 0, be the universe of all terms in the semantics. In our setting, types of recursive functions are restricted to the shape $Y \to \tau'(Y)$, which trivially satisfy the bottom check [37] (p. 148) for empty types Y such as the 0th approximation of an inductive type. Our pendant to the bottom check is the base case $\alpha = 0$ of the transfinite induction over α in the soundness proof of the fixed point rule (see Rem. 5.11).

8.2. Giménez *et al.*

Giménez [23] provided the starting point for our work. He describes type-based termination using approximation types for an extension of the Calculus of Constructions, but provides no soundness proof. Showing strong normalization for his method was the original motivation for the present article.

In Giménez later work with Barthe *et al.* [9], the formulation of the type system follows Hughes *et al.* The set of size expressions is generated by size variables, the successor operation, and the symbol ∞, denoting the first uncountable ordinal Ω. The type system features function types and inductive datatypes, but excludes quantification over size variables, hence one cannot abstract over a size-preserving function (see qsapp' in Sect. 2.4). Except for bounded quantification, our system embeds into Barthe *et al.*, and they give both a proof of subject reduction and strong normalization. However, they do not describe a type checking algorithm. Thus, their type system is not ready to be used as a termination checker.

8.3. Xi's Dependent ML

Xi [50] considers termination of closed functional programs in DML, a functional language with lightweight dependent types. In contrast to our approach and the approaches described above, the size of a data structure is not determined to be its height, but it can be any integer expression defined by the user. For example, one could define the size of a tree as the total number of nodes, or even the sum of its labels. Recursive functions come with a termination measure which has the form of a lexicographic product of size expressions. His size annotations are exact in contrast to our notion of size which is just an upper bound. Therefore, he does not need subtyping; to compare integer expressions he relies on a constraint solver.

On one hand, due to exact size annotations, Xi can also perform size-checking for accumulation parameters in functions, which is impossible with the approaches

mentioned above. This qualifies Xi's approach for practical termination checking of realistic functional programs. On the other hand, since Xi has no ordinal sizes, he can not handle higher-order datatypes like infinite trees which are common in proof theory. This limits the applicability of his approach in theorem provers.

8.4. Other related work

On termination in general, a vast amount of literature has been published, so we can only give a non-representative overview of work related to ours.

Typed functional programming. In previous work [1] with Thorsten Altenkirch [5], we have described a syntactic check for structurally recursive functions in the context of simple types. We proved termination only for closed terms (no strong normalization). Telford and Turner [45, 46] check termination and productivity of recursive functions by an abstract interpretation. They track size-change on an abstract domain; the power of the approach is comparable to the one presented here. Giesl [22], also with Brauburger [14], generates termination predicates by static analysis of functional programs. Termination can be proven by some external automated theorem prover. Their approach is limited in power only by the theorem prover, hence very flexible, but also quite unpredictable.

Higher-order logic programming. Pientka [40] describes a calculus for termination and reduction checking of logic programs over a higher-order term language. Her calculus is based on the subterm ordering and used in the HOAS theorem prover Twelf [39].

Untyped functional programming. Lee, Jones and Ben-Amram [30] investigate termination of untyped first-order functional programs *via* size-change graphs. They show that the implicit complexity of termination-checking mutually recursive functions is PSPACE-hard.

Term rewriting. This is the classical area of investigating termination. We accentuate Giesl's work, for example, with Arts, on dependency pairs [8]. Blanqui, Jouannaud and Okada investigate term rewriting for systems with inductive types [13]. In his thesis [11], Blanqui proves strong normalization for the Calculus of Algebraic Constructions with rewrite rules.

Acknowledgements. The author thanks Thorsten Altenkirch for more than one year of joint work on termination, Thierry Coquand for pointing to Giménez' work, John Hughes and Lars Pareto for a conversation which eventually lead to the counterexample in Section 6, Martin Hofmann, Frank Pfenning and Brigitte Pientka for helpful discussions, and Frédéric Blanqui for encouraging feedback on this article. He is indebted to Ralph Matthes for careful reading of the draft and for conversation and advice which helped overcome unforeseen difficulties in the final preparation phase of this articles. Last but not least, thanks to the the anonymous referees for critical remarks which led to a clearer presentation.

References

[1] A. Abel, Specification and verification of a formal system for structurally recursive functions, in *Types for Proof and Programs, International Workshop, TYPES '99*, edited by T. Coquand, P. Dybjer, B. Nordström, J. Smith, Springer. *Lect. Notes Comput. Sci.* **1956** (2000) 1–20.

[2] A. Abel, A third-order representation of the $\lambda\mu$-calculus, edited by S. Ambler, R. Crole, A. Momigliano, Elsevier Science Publishers. *Electron. Notes Theor. Comput. Sci.* **58** (2001).

[3] A. Abel, Termination and guardedness checking with continuous types, in *Typed Lambda Calculi and Applications (TLCA 2003)*, edited by M. Hofmann, Valencia, Spain, Springer. *Lect. Notes Comput. Sci.* **2701** (2003) 1–15.

[4] A. Abel, *Soundness of a bidirectional typing algorithm*. Twelf code, available on the author's homepage, `http://www.tcs.informatik.uni-muenchen.de/~abel` (May 2004).

[5] A. Abel and T. Altenkirch, A predicative analysis of structural recursion. *J. Funct. Programming* **12** (2002) 1–41.

[6] T. Altenkirch, *Constructions, Inductive Types and Strong Normalization*. Ph.D. Thesis, University of Edinburgh (Nov. 1993).

[7] R.M. Amadio and S. Coupet-Grimal, Analysis of a guard condition in type theory, in *Foundations of Software Science and Computation Structures, First International Conference, FoSSaCS'98*, edited by M. Nivat, Springer. *Lect. Notes Comput. Sci.* **1378** (1998).

[8] T. Arts and J. Giesl, Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.* **236** (2000) 133–178.

[9] G. Barthe, M.J. Frade, E. Giménez, L. Pinto and T. Uustalu, Type-based termination of recursive definitions. *Math. Struct. Comput. Sci.* **14** (2004) 1–45.

[10] G.M. Bierman, A computational interpretation of the $\lambda\mu$-calculus, in *Proc. of Symposium on Mathematical Foundations of Computer Science*, edited by L. Brim, J. Gruska, J. Zlatuska, Brno, Czech Republic. *Lect. Notes Comput. Sci.* **1450** (1998) 336–345.

[11] F. Blanqui, *Type Theory and Rewriting*. Ph.D. Thesis, Université Paris XI (Sept. 2001).

[12] F. Blanqui, A type-based termination criterion for dependently-typed higher-order rewrite systems, in *15th International Conference on Rewriting Techniques and Applications (RTA 04)*, June 3–5, 2004, Aachen, Germany, Springer. *Lect. Notes Comput. Sci.* **3091** (2004) 24–39.

[13] F. Blanqui, J.-P. Jouannaud and M. Okada, Inductive data type systems. *Theor. Comput. Sci.* **277** (2001).

[14] J. Brauburger and J. Giesl, Termination analysis for partial functions, in *Proc. of the Third International Static Analysis Symposium (SAS'96)*, Aachen, Germany, Springer. *Lect. Notes Comput. Sci.* **1145** (1996).

[15] W.-N. Chin and S.-C. Khoo, Calculating sized types. *Higher-Order and Symbolic Computation* **14** (2001) 261–300.

[16] C. Coquand, Agda. WWW page (2000) `http://www.cs.chalmers.se/~catarina/agda/`

[17] T. Coquand, Infinite objects in type theory, in *Types for Proofs and Programs (TYPES '93)*, edited by H. Barendregt, T. Nipkow, Springer. *Lect. Notes Comput. Sci.* **806** (1993) 62–78.

[18] T. Coquand, An algorithm for type-checking dependent types, in *Mathematics of Program Construction. Selected Papers from the Third International Conference on the Mathematics of Program Construction*, July 17–21, 1995, Kloster Irsee, Germany, Elsevier Science. *Sci. Comput. Programming* **26** 167–177 (1996).

[19] K. Crary and S. Weirich, Resource bound certification, in *Proc. of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, Massachusetts, USA (Jan. 2000) 184–198.

[20] R. Davies and F. Pfenning, Intersection types and computational effects, in *Proc. of the International Conference on Functional Programming (ICFP 2000)*, Montreal, Canada (Sept. 2000) 198–208.

[21] J. Dunfield and F. Pfenning, Tridirectional typechecking, in *31st Annual Symposium on Principles of Programming Languages (POPL'04)*, edited by N.D. Jones and X. Leroy, Venice, Italy. ACM (Jan. 2004) 281–292.
[22] J. Giesl, Termination of nested and mutually recursive algorithms. *J. Automat. Reason.* **19** (1997) 1–29.
[23] E. Giménez, Structural recursive definitions in type theory, in *Automata, Languages and Programming, 25th International Colloquium, ICALP'98*, Aalborg, Denmark, July 13–17 1998, Proc., Springer. *Lect. Notes Comput. Sci.* **1443** (1998) 397–408.
[24] C. Haack and J.B. Wells, Type error slicing in implicitly typed, higher-order languages, in *Programming Languages and Systems, 12th European Symp. Programming*, Springer. *Lect. Notes Comput. Sci.* **2618** (2003) 284–301.
[25] T. Hagino, A typed lambda calculus with categorical type constructors, in *Category Theory and Computer Science*, edited by D.H. Pitt, A. Poigné, D.E. Rydeheard. *Lect. Notes Comput. Sci.* **283** (1987) 140–157.
[26] T. Hallgren, Alfa home page. `http://www.math.chalmers.se/~hallgren/Alfa/` (2003).
[27] J. Hughes and L. Pareto, Recursion and dynamic data-structures in bounded space: Towards embedded ML programming, in *International Conference on Functional Programming (ICFP'99)* (1999) 70–81.
[28] J. Hughes, L. Pareto and A. Sabry, Proving the correctness of reactive systems using sized types, in *Symposium on Principles of Programming Languages* (1996) 410–423.
[29] INRIA, *The Coq Proof Assistant Reference Manual*, version 8.0 edition (April 2004). `http://coq.inria.fr/doc/main.html`
[30] C.S. Lee, N.D. Jones and A.M. Ben-Amram, The size-change principle for program termination, in *ACM Symposium on Principles of Programming Languages (POPL'01)*, London, UK. ACM Press (Jan. 2001).
[31] Z. Luo, *ECC: An Extended Calculus of Constructions*. Ph.D. Thesis, University of Edinburgh (1990).
[32] R. Matthes, *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. Ph.D. Thesis, Ludwig-Maximilians-University (May 1998).
[33] C. McBride, *Dependently Typed Functional Programs and their Proofs*. Ph.D. Thesis, University of Edinburgh (1999).
[34] N.P. Mendler, Recursive types and type constraints in second-order lambda calculus, in *Proc. of the Second Annual IEEE Symposium on Logic in Computer Science*, Ithaca, New York. IEEE Computer Society Press (1987) 30–36.
[35] N.P. Mendler, Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Logic* **51** (1991) 159–172.
[36] R. Milner, A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* **17** (1978) 348–375.
[37] L. Pareto, *Types for Crash Prevention*. Ph.D. Thesis, Chalmers University of Technology (2000).
[38] M. Parigot, $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction, in *Logic Programming and Automated Reasoning: Proc. of the International Conference LPAR'92*, edited by A. Voronkov, Springer, Berlin, Heidelberg (1992) 190–201.
[39] F. Pfenning and C. Schürmann, System description: Twelf – a meta-logical framework for deductive systems, in *Proc. of the 16th International Conference on Automated Deduction (CADE-16)*, edited by H. Ganzinger, Springer, Trento, Italy. *Lect. Notes Artif. Intell.* **1632** (1999) 202–206.
[40] B. Pientka, Termination and reduction checking for higher-order logic programs, in *Automated Reasoning, First International Joint Conference, IJCAR 2001*, edited by R. Goré, A. Leitsch, and T. Nipkow, Springer. *Lect. Notes Artif. Intell.* **2083** (2001) 401–415.
[41] B.C. Pierce, *Types and Programming Languages*. MIT Press (2002).
[42] B.C. Pierce, D.N. Turner, Local type inference, in *POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California (1998).

[43] R. Pollack, *The Theory of LEGO*. Ph.D. Thesis, University of Edinburgh (1994).
[44] Z. Spławski and P. Urzyczyn, Type fixpoints: Iteration vs. recursion, in *Proc. of the 1999 International Conference on Functional Programming (ICFP)*, Paris, France. *SIGPLAN Notices* **34** (1999) 102–113.
[45] A.J. Telford and D.A. Turner, Ensuring streams flow, in *Algebraic Methodology and Software Technology (AMAST '97)*, Springer. *Lect. Notes Comput. Sci.* **1349** (1997) 509–523.
[46] A.J. Telford and D.A. Turner, Ensuring termination in ESFP, in *Proc. of BCTCS 15*, 1999. *J. Universal Comput. Sci.* **6** (2000) 474–488.
[47] T. Uustalu and V. Vene, Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica (Lithuanian Academy of Sciences)* **10** (1999) 5–26.
[48] C. Walther, Argument-Bounded Algorithms as a Basis for Automated Termination Proofs, in *9th International Conference on Automated Deduction*, edited by E.L. Lusk and R.A. Overbeek, Springer. *Lect. Notes Comput. Sci.* **310** (1988) 602–621.
[49] A.K. Wright and M. Felleisen, A syntactic approach to type soundness. *Inform. Comput.* **115** (1994) 38–94.
[50] H. Xi, Dependent types for program termination verification. *J. Higher-Order and Symbolic Computation* **15** (2002) 91–131.
[51] J. Yang, G. Michaelson and P. Trinder, Explaining polymorphic types. *Comput. J.* **45** (2002) 436–452.