# A Polymorphic Lambda-Calculus with Sized Higher-Order Types

Andreas Abel

June 19, 2006

# Contents

# Chapter 1

# Introduction

This thesis is on termination of computer programs. By *termination* we mean that the task given to a computer is processed in a certain amount of time, as opposed to taking infinitely long. Termination is the expected behavior of a computer; it does not make sense to wait infinitely long for its response. Some tasks take longer, e. g., booting, or calculating an airline schedule. Some take shorter, like, after a key press in a word processing program, displaying the new letter on the screen. Some take a variable time, like executing a command at the command prompt. Some tasks might indeed never end, like producing an even number greater or equal to four which is *not* the sum of two prime numbers. This task, call it `goldbach`, will run forever, if, as Goldbach conjectured,[1] there exists no such number. Since the Goldbach conjecture is neither proven correct nor refuted,[2] with our current knowledge we cannot judge whether `goldbach` will terminate or not.

Of course, we would like to know whether a computer program terminates *before we start it*. Otherwise we might sit there and wait for termination like one waits for a late bus, which will, by Murphy's law, arrive exactly the minute after one has taken a taxi instead. The only way to tell whether a program will terminate without running it is to look at its internal structure, e. g., the sequence of instructions which make up the program. Compare this to a biologist who cuts up a frog in order to understand its behavior. Although a frog is magnitudes more complex in functionality[3] than the biggest information system ever constructed, this analogy gives some intuition on what it means to grasp the meaning of a large computer program. In this light, the following negative solution to the *Halteproblem*, which is as old as the first computers, does not surprise:

> There is no computer program which can take any (other) computer program and decide whether it terminates on some given input.

[1] The "Goldbach conjecture" in this form is actually due to Euler, see MathWorld [Wei05].

[2] As of today, the Goldbach conjecture has been confirmed for number up to $2 \times 10^{17}$ [Wei05].

[3] The reader may excuse my comparison of a higher living being to something entirely mechanical like a computer program.

This theorem has a simple proof by contradiction: Suppose there is such a program $D$. Then we can write a program $E$ which takes the code of some program $P$ as input, asks $D$ whether $P$ terminates on input $P$, and enters an infinite loop if $D$ says "yes". Now run $E$ on its own code. If $E$ terminates on $E$, then $D$ says "yes", hence $E$ does not terminate on $E$. Otherwise, if $E$ does not terminate on $E$, $D$ will say "no", hence $E$ will terminate on $E$. This is a contradiction, [4] and thus, program $D$ cannot exist.

The negative solution to the *Halteproblem* destroys the hope of finding a automated termination checker which works on any kind of program. There are two escapes from this dilemma:

1. Use *human intuition* and reasoning to check termination.
2. Write a termination checker which works on a certain, *restricted* class of programs.

In practice, both methods are indispensable. In this thesis, we describe some class of terminating programs. But however big one may make the class, there will always be an interesting program whose termination cannot be verified completely automatically but requires at least some hint by a human who understands the program.

## 1.1 Why Termination?

But is *termination* really interesting? Would we not rather want to know *how long* the execution of a program takes? This is certainly true in real-time and embedded systems, like the hard- and software assisting the navigation of automobiles and planes. There, the system must react *instantaneously*, e. g., within milliseconds; termination is surely not sufficient. But there are other applications for termination:

1. Verified termination increases the chance that the program is correct. This argument has been put forward by Xi [Xi01] and has an analogy in *static typing*. Empirically, programs in strongly typed languages like Haskell and ML often work correctly already after they passed the compiler. The type checker spots many bugs, which would—in the absence of type checking— have surfaced one after another at runtime. In the same way, a termination checker could spot additional bugs which cannot be detected by the type checker. [5]

[4] The propositions $A \implies \neg A$ and $\neg A \implies A$ are contradictory even in intuitionistic logics.

[5] For example, I had written the following ML function.

```
fun fromto (from, to) = if from > to then [] else from :: fromto (from*1, to)
```

This program should output a list of integers starting with `from` and ending in `to`. The type inference of ML assigns it the expected type `int * int -> int list`, but upon execution, e. g., `fromto (2, 641)`, it loops. It is supposed to return the empty list if `from` is greater or equal to `to`, and, otherwise, the element `from` plus the sequence `from+1..to` which is computed recursively. Here

2. It is agreeable that compilers and program generators should terminate. However, to optimize the target program, they might have to partially evaluate the source code. Partial evaluation has to be restricted to these parts of the source which terminate; hence, termination has to be checked before evaluation. This is the motivation for Neil Jones' research on termination. [JG02]

3. Program verification has been traditionally separated into two parts: proving partial correctness, meaning "if the program terminates then it satisfies its specification", and proving termination. If termination can be established in a canonical way, a proof of partial correctness implies total correctness.

4. Interactive theorem provers with inductive types like Agda [CC99], Coq [BC04], Lego [Pol94] and Twelf [PS99] make use of the Curry-Howard isomorphism and allow proofs by induction to be written as recursive functions. But only terminating functions correspond to valid proofs, hence, termination is vital to maintain soundness of the proof system. This is the original motivation for my continuing research on termination and strong normalization.

In the following we consider *systems*—these could be programming languages, abstract calculi or logics—which have an *expression* language or a *term* language and a notion of *computation*.

## 1.2 Approaches to Termination

There are two fundamental kinds of systems: *systems with partiality* and *total systems*. In the first class, each expression has a meaning, even those expressions whose computation does not terminate. This class covers most programming languages, but also, e. g., the Logic of Computable Functions (LCF) and logics with undefinedness (Beeson [Bee04], Farmer [Far04]). The fact that an expression terminates (i. e., is defined) has to be *proven* in the logics.

In total systems, each valid expression is terminating. We can distinguish systems with or without recursion. For example, the simply-typed $\lambda$-calculus is a system without recursion. So is Girard's [Gir72] System $\mathsf{F}^\omega$, but a limited form of recursion, *iteration*, is definable in it [GLT89, chap. 11] [BB85, RP93, PDM89, Geu92, Mat98, AMU05]. In HOL, even well-founded recursion is definable [Sli96]. Other systems have explicit recursion, which has to be tamed by some formalism to preserve termination of all expressions. Most notably, there are *term rewriting systems* (TRS). For TRS, many criteria have been developed

we spot the bug: I had typed an asterix $*$ instead of a plus + who are only a shift apart on a German keyboard. To locate this error in a bigger program took me more than an hour, since I had not expected the slip in such a simple function. A termination checker could have warned me that there might be a problem.

for termination: polynomial interpretations, the recursive-path ordering, simplification orderings [Ste95], and dependency pairs [AG00]. These criteria are *syntactical*, as opposed to *type-based*; they work on untyped rewrite rules. Syntactical methods have also been applied to termination of logic programs and higher-order logic programs [Pie01]. The `Fixpoint` declaration of Coq, which introduces dependently-typed recursive functions, uses as of today a syntactical termination check, and Agda uses an implementation of the termination checker `foetus` [AA02].

This thesis investigates *type-based termination*; by this we mean that termination checking is integrated with type checking: a program which passes the type-checker is guaranteed to terminate on all inputs.

## 1.3 Why Type-Based Termination Matters

I am convinced that the types paradigm is fruitful also for termination. The integration of termination checking into type checking has the following advantages.

**Communication.** The idea that each wellformed program fragment has a certain *type* seems still to be spreading in computing. Most firmly rooted is this view in strongly typed functional programming: in ML, each program must have a type which is inferable by the compiler. In practical software development, it has been fruitful to write down types as a documentation for the programmer. The type of a program gives an abstract idea which operation it implements, and this type, since checked by the compiler, is documentation which is always up-to-date. JAVA is a success story for a strongly typed language. Its design is quite contrary to ML, however: The type system is rather simple. It lacks higher-order functions, and subtyping is based on names, not on structure. The type of each variable has to be written down by the programmer and each exception a method can throw has to be declared, which makes JAVA a bit bureaucratic. A good compromise has been found in the design of Haskell: not each type must be inferable, but also not each type must be declared. Haskell's type system not only features higher-order functions, higher-order and nested data types, and type classes, but also polymorphic recursion and higher-rank polymorphism with explicit quantification. Through programming practice, a Haskell programmer learns to think in a quite complex type language.

Once a programmer has learned to think in types, type-based termination of a program can be easily communicated to him. Once a program can be given a certain type, its termination ensues. The main thing a programmer needs to understand in order to master type-based termination as presented in this thesis, is that each data structure carries an upper bound for its size in its type, and that in a recursive call, this size must decrease.

**Certification.** Besides communicating termination to a human there is the problem of communicating termination to a machine, or more precisely, providing a machine-checkable certificate that a program terminates. For type-based termination, this question is already answered: a correct *typing derivation* for a program certifies its termination. The proof of this fact for a polymorphic functional language with higher-order data types, which we call $\mathsf{F}_{\widehat{\omega}}$, is the main technical contribution of this thesis.

**A simple theoretical justification.** Type-based termination can be reduced to a simple concept: induction on the size parameter. For polynomial datatypes like lists and binary search trees, natural numbers suffice as size indices, for infinitely branching trees, streams, and processes we need ordinal indices and transfinite induction. However, even in this case the programmer does not need ordinal notations and can continue to think of sizes as natural numbers.

**Orthogonality.** Type-based termination rests on the principle of sized data types and requires subtyping and restricted use of recursion. But these are the only parts of a purely functional language which are affected.[6] Other language constructs, like higher-order functions, tuples and records, variants and disjoint sums, native types and operations like integers and reals can be added without having to change the termination machinery. All one has to do is to extend type checking and subtyping for the language—and this has to be done anyway. This is not true if termination checking is a separate module, e. g., some syntactic method. Then each extension of the language requires an extension of the termination checker, or even new research how termination checking can be extended to the new language constructs.

**Robustness.** A consequence of orthogonality is that the soundness of type-based termination is robust with regard to language extensions. As we show in this thesis, it is compatible with *impredicative polymorphism*. This does not hold for certain syntactic methods based on structural term orderings, as already pointed out by Coquand [Coq92] [AA02, page 3].

**Higher-order functions.** As another consequence of orthogonality, type-based termination plays its strength when applied to higher-order functions or higher-rank polymorphism. Since sizes are integrated into types, one can specify that a higher-order argument of a function should be a non size-increasing operation, which can be safely applied to the recursion argument without destroying size information important for termination. In Section 6.5 we will see an example where typing with rank-2 polymorphism will be crucial for the termination of a generic merge function.

[6] It is well-known that higher-order references can simulate recursion, see, e. g., Xi [Xi02, section 4.6]. I/O and process communication can also be a source of non-termination, for instance, in form of deadlocks. Such impure language features are not treated in this thesis.

## 1.4 Informal Account of Type-Based Termination

What do we mean by *type-based termination*? In its broadest sense, the phrase refers to any type system which ensures that well-typed programs terminate. Very basic representatives of such systems are the simply or the polymorphically typed lambda-calculus. We want be more specific and refer only to languages with *recursion* (or, on the imperative side, loops), whose use is restricted by a type system such that it cannot introduce non-termination. Again, there are different ways to implement this idea. This thesis is about about type-based termination as coined by Barthe et al. [BFG$^+$04], which builds on work by Mendler [Men87, Men91], Giménez [Gim98], and Amadio and Coupet-Grimal [ACG98], and has independently been invented by Hughes, Pareto, and Sabry [HPS96]. To simplify the following explanation of type-based termination, we will first refer to types in an informal, semantical fashion as sets of terms, but later we will be more precise and use types as *syntactical* objects, which are *interpreted* as sets of terms.

### 1.4.1 A Semantical Account of Type-Based Termination

*General recursion* can be implemented by adding a fixed-point combinator $\mathsf{fix}$ to a functional language based on a typed $\lambda$-calculus. This combinator is axiomatized as follows:

$$\frac{f \in A \to A}{\mathsf{fix}\, f \in A} \qquad \mathsf{fix}\, f = f\,(\mathsf{fix}\, f)$$

That means: If $f$ is a endo-function on type $A$, then $\mathsf{fix}\, f$ inhabits $A$; and $\mathsf{fix}\, f$ behaves as $f\,(\mathsf{fix}\, f)$. General recursion makes a language inconsistent as a logic, since every type is inhabited, and introduces non-termination, if the equation is read as a computation rule. (For both claims, take $f$ to be the identity function.) To maintain termination, the use of $\mathsf{fix}$ has to be restricted in some way.

Keeping only the axiom $\mathsf{fix}\, f = f\,(\mathsf{fix}\, f)$, we make the following observation. Let $i$ and $n$ range over natural numbers, and $A^i$ be some type dependent on $i$. Further let $\top$ denote the all-type, i. e., each program is of this type without further requirements. Then the following typing rule for $\mathsf{fix}$ is *admissible*, i. e., provable using just the typing rules of $\lambda$-calculus:

$$\frac{A^0 = \top \qquad f \in A^i \to A^{i+1} \text{ for all } i}{\mathsf{fix}\, f \in A^n}$$

The proof proceeds by induction on $n$: If $n = 0$, then $\mathsf{fix}\, f \in \top = A^0$. Now assume $\mathsf{fix}\, f \in A^n$. Since $f \in A^n \to A^{n+1}$ by assumption, $f\,(\mathsf{fix}\, f) = \mathsf{fix}\, f \in A^{n+1}$. Now the rough idea behind the work of Hughes, Pareto, and Sabry [HPS96] is: if $\bigcap_{n\in\mathbb{N}} A^n$ is "interesting" [Par00, p. 129], then $\mathsf{fix}\, f$ has interesting properties, like termination or productivity. Interesting types are, e.g., $A^n = \{m \mid m < n\} \to \mathbb{N}$; then $\bigcap_{n\in\mathbb{N}} A^n = \mathbb{N} \to \mathbb{N}$, and we can use the typing rule for $\mathsf{fix}$ to introduce total functions over natural numbers.

A typical application would be the following. Let $L^n$ denote the type of lists of natural numbers of length $< n$ and $L^\infty = \bigcup_{n\in\mathbb{N}} L^n$ the type of all natural number lists. Using the type $A^n := L^n \to L^\infty$, we can define a recursive function which eliminates all zeros from a list. Since $L^0$ is empty, $A^0$ is isomorphic to the all-type (under the standard semantics of the function type, $A \to B = \{f \mid f\,a \in B \text{ for all } a \in A\}$). Using pattern matching notation we can define the following functional $f$.

$$\begin{array}{lcl} f & \in & \bigcap_i((L^i \to L^\infty) \to L^{i+1} \to L^\infty) \\ f & := & \lambda \mathit{filter}0\lambda l.\,\mathsf{match}\ l\ \mathsf{with} \\ & & \quad \mathsf{nil} \mapsto \mathsf{nil} \\ & & \quad \mathsf{cons}\ 0\,t \mapsto \mathit{filter}0\,t \\ & & \quad \mathsf{cons}\ h\,t \mapsto \mathsf{cons}\,h\,(\mathit{filter}0\,t) \end{array}$$

Then $\mathsf{filter}0 := \mathsf{fix}\ f \in \bigcap_n(L^n \to L^\infty)$ is the desired filtering function which can be applied to lists of any length. Of course, we need to make sure $f$ is well-typed. For example, if the input list $l = \mathsf{cons}\,h\,t \in L^{i+1}$ has length $< i+1$, then the length of the tail $t$ is certainly below $i$, which implies that the recursive call $\mathit{filter}0\,t$ is well-typed and, thus, justified.

Lists are a special case of an *inductive type*; list of bounded length are a special case of a *sized* inductive type. In general, if $F$ is an isotone operator on term sets, then $\mu^\infty F$ denotes the inductive type which is the least fixed point of $F$. In case of lists of natural numbers, we have $F(X) = \{\mathsf{nil}\} \cup \{\mathsf{cons}\,h\,t \mid h \in \mathbb{N} \text{ and } t \in X\}$. A sized inductive type is obtained by ordinal iteration of $F$:

$$\begin{array}{lcll} \mu^0 F & = & \bot & \text{(the empty type)} \\ \mu^{\alpha+1} F & = & F(\mu^\alpha F) & \\ \mu^\lambda F & = & \bigcup_{\alpha<\lambda} \mu^\alpha F & (\lambda \text{ limit ordinal}) \end{array}$$

It is easy to see that $\mu^n F$, for the list-specific $F$ given above, contains exactly the lists of length $< n$, hence $L^n = \mu^n F$. Since all lists have finite length, the least fixed point $L^\infty = \bigcup_{n\in\mathbb{N}} L^n = \bigcup_{n<\omega} \mu^n F = \mu^\omega F$ is reached at the smallest infinite ordinal $\omega$. However, there are inductive types with a higher *closure ordinal*, for example, the second number class with the generating operator $F(X) = \{\mathsf{ozero}\} \cup \{\mathsf{osucc}\ o \mid o \in X\} \cup \{\mathsf{olim}\,f \mid f \in \mathbb{N} \to X\}$. If $\mathsf{Ord}^\alpha$ denotes the $\alpha$-iterate of this operator, then the constructors have the following types:

$$\begin{array}{lcl} \mathsf{ozero} & \in & \bigcap_\alpha \mathsf{Ord}^{\alpha+1} \\ \mathsf{osucc} & \in & \bigcap_\alpha(\mathsf{Ord}^\alpha \to \mathsf{Ord}^{\alpha+1}) \\ \mathsf{olim} & \in & \bigcap_\alpha((\mathbb{N} \to \mathsf{Ord}^\alpha) \to \mathsf{Ord}^{\alpha+1}) \end{array}$$

If $h \in \mathbb{N} \to \mathsf{Ord}^\omega$ is given by $h(0) = \mathsf{ozero}$ and $h(n+1) = \mathsf{osucc}(h(n))$, then $\mathsf{olim}\,h \in \mathsf{Ord}^{\omega+1}$ appears for the first time at the iterate $\omega + 1$, hence, there are terms of transfinite height in the second number class.

Consequently, a recursive function over elements of type $\mathsf{Ord}$ must treat also transfinite iterates $\mathsf{Ord}^\alpha$ for $\alpha \geq \omega$. To account for the limit iterates, we

extend our introduction rule for recursive functions as follows:

$$\frac{A^0 = \top \qquad f \in A^\alpha \to A^{\alpha+1} \text{ for all } \alpha \qquad \bigcap_{\alpha<\lambda} A^\alpha \subseteq A^\lambda \text{ for all limits } \lambda > 0}{\mathsf{fix}\, f \in A^\beta}$$

Again, this rule is provable, this time by transfinite induction on $\beta$: Base and step case work as before, for the limit case we need to show $\mathsf{fix}\, f \in A^\lambda$ under the induction hypothesis $\mathsf{fix}\, f \in A^\alpha$ for all $\alpha < \lambda$. But exactly this is given by the new assumption. Now, if we summarize the two conditions on the type $A$,

1. $A^0 = \top$, and
2. $\bigcap_{\alpha<\lambda} A^\alpha \subseteq A^\lambda$ for all limits $\lambda$,

into the predicate $A\ \mathsf{adm}$, we get a rule which looks quite similar to the rule for general recursion:

$$\frac{f \in A^\alpha \to A^{\alpha+1} \text{ for all } \alpha}{\mathsf{fix}\, f \in A^\beta}\ A\ \mathsf{adm}$$

### 1.4.2 From Semantics to Syntax

The goal of this thesis is to turn the semantical recursion rule into syntax, i. e., into a typing rule, and show that all well-typed programs are strongly normalizing. If we take a close look at the syntactical components of the semantical types we used in the constructors and the recursion rule in the last section, we find the following elements:

- types indexed by ordinal variables $\alpha$, their successor $\alpha + 1$ or the closure ordinal $\infty$,
- function types $\to$,
- infimum $\bigcap_\alpha$ of a family of types indexed by $\alpha$, and
- a condition $A\ \mathsf{adm}$ on types for recursion.

Surprisingly, there are only these few elements and they can be turned into syntax directly. To the type language of a simple or polymorphic lambda-calculus we add a language of size expressions $a : \mathsf{ord}$ composed from size variables $\iota$, successor $+1$, and $\infty$, which denotes a large ordinal at which the iteration processes for all inductive types of the system close. The infimum $\bigcap_\alpha$ is expressed by size polymorphism $\forall\iota$. Finally, we formulate a syntactic criterion on types $A\ \mathsf{adm}$ which entails the semantic one. This leads to the syntactic recursion rule

$$\frac{f : \forall\iota.\, A^\iota \to A^{\iota+1} \qquad a : \mathsf{ord}}{\mathsf{fix}\ f : A^a}\ A\ \mathsf{adm}.$$

There is natural order on size expressions $a$ induced by $a \leq a + 1$ and $a \leq \infty$. Since the approximation stages of an inductive type are ordered ascendingly by inclusion, we get a natural subtype relation $\mu^a F \leq \mu^b F$ for $a \leq b$.

The first part of this thesis deals with the subtyping induced by sizes in the presence of co- and contravariant type constructors.

The distinguished feature of our approach is that we speak about ordinal indices in our type system without requiring an ordinal notation system, like Cantor normal forms. The user can think of sizes as natural numbers instead of ordinals without making mistakes.

## 1.5 Contribution

This thesis is not the first one on typed-based termination. Pareto [Par00] has explored this paradigm in the area of functional programming. He developed *Synchronous Haskell*, the core of a functional language with type-based termination and productivity checking, which is summarized in Hughes et al. [HPS96]. He proves the soundness of his system by non-standard denotational semantics, interpreting types as upward-closed sets of values. Pareto considers only ordinals up to $\omega$, and, thus only inductive data types *without* embedded function spaces, excluding the type Ord.

Frade's thesis [Fra03] introduces $\lambda\hat{}$, an extension of $\lambda$-calculus with ML-polymorphism by sized inductive types, which is summarized in Barthe et al. [BFG$^+$04]. She does treat inductive types with embedded function spaces and she proves subject reduction and strong normalization for $\lambda\hat{}$.

Our contribution, in relation to the above theses and to other systems of type-based termination, is progress in the following areas:

**Higher-order inductive types.** Our inductive types can contain embedded function spaces and, at the same time, be of higher kind. Therefore, neither iteration to the first infinite ordinal, $\omega$, as in Pareto's case, nor to the first uncountable ordinal, $\Omega$, as in Frade's case seems sufficient. The upper bound for our closure ordinal is the $\omega$th uncountable.

**A theory of semi-continuous types.** Frade restricts result types of recursive functions to be monotone in their size argument. Pareto is more liberal, allowing types which are, in his terminology, $\omega$-undershooting. He gives some criteria how to classify such admissible types, but in a rather ad-hoc manner. We show that similar types are admissible in our semantics where types are sets of strongly normalizing terms, which is quite distinct from his domain-theoretic semantics. However, since we allow infinite branching, inductive types will fall in a different class in our case. We put his ad-hoc analysis of admissible types on a more solid basis by classifying types as upper or lower semi-continuous, which are standard concepts in analysis. Furthermore, we cast or results in kinding rules for semi-continuous types, thus, mechanizing the quest for admissible types.

**Heterogeneous data types.** We are the first to consider typed-based termination for heterogeneous or nested data types. Heterogeneous data types can be viewed as fixed points of type constructors of a higher kind, such that the fixed point is not a type but a type constructor. Such heterogeneous types have been analyzed in the mathematics of program construction and in Haskell by Hinze [Hin98, Hin99, Hin00a, Hin00b, Hin01] Bird and Meertens [BM98], Bird and Paterson [BP99a], Martin and Gibbons [MG01] and Bayley [MGB04] and in the context of System $\mathsf{F}^\omega$ by Pierce, Dietzen, and Michaylov [PDM89], Matthes [Mat01], Matthes and myself [AM03, AM04], and with Uustalu [AMU03, AMU05]. Note that heterogeneous data types are not a special case of inductive families [Dyb94].

**Equi-(co)inductive types.** In contrast to many predecessors [Geu92, Alt93, Mat98, Alt99, AA00, BJO01, BFG$^+$04] of our work, we do not use the common iso-recursive approach to (co)inductive types, but the equi-recursive one, hence, we consider *equi-(co)inductive* types. For example, take the Haskell data type of binary trees:

```
data BT a = Leaf
          | Node (BT a) a (BT a)
```

The finite binary trees can be modeled by an inductive type. Using *equi-inductive types*, one can decompose a data type into recursion and a labeled sum. For this example, the decomposition would read

$$\begin{array}{lcl} \mathsf{BTF} & : & * \to * \to * \\ \mathsf{BTF} & := & \lambda A \lambda X.\, \mathsf{Leaf}\,1 + \mathsf{Node}\,(X \times A \times X) \end{array}$$

$$\begin{array}{lcl} \mathsf{BT} & : & * \to * \\ \mathsf{BT} & := & \lambda A.\, \mu(\mathsf{BTF}\,A), \end{array}$$

where 1 is the unit type, $c_1\,A_1 + \cdots + c_n\,A_n$ is the labeled sum of the types $A_{1..n}$, and $\mu$ is the least fixed-point combinator, fulfilling the type equation $\mu\,F = F\,(\mu\,F)$. The constructors of the labeled sum can be reused directly as constructors for binary trees.

Iso-inductive types, however, fold and unfold noisily: Special terms $\mathsf{in} : F\,(\mu\,F) \to \mu\,F$ and $\mathsf{out} : \mu\,F \to F\,(\mu\,F)$ construct and destruct iso-inductive types, and $\mu\,F$ is only *isomorphic* to $F\,(\mu\,F)$ (hence, the name). The constructors for binary trees now read $\mathsf{in} \circ \mathsf{Leaf}$ and $\mathsf{in} \circ \mathsf{Node}$. This seems less direct and less-intuitive than the equi-inductive approach. On the other hand, the metatheory of iso-inductive types is simpler, because we know that each canonical inhabitant of an inductive type is of the form $\mathsf{in}\,t$.

In this thesis, we analyze the more challenging metatheory of equi-inductive and -coinductive types. Especially the combination of equi-induction and equi-coinduction will have some unpleasant effects when it comes to a strongly normalizing reduction system. Anyhow, we fight our way through, but if you want to avoid hardship, stick to iso-coinductive types!

**A unified language for types and size expressions.** In contrast to previous work on sized types, we unify the languages of sizes and types by introducing a new base kind ord of sizes into $\mathsf{F}^{\omega}$. This enables us to give precise types to polymorphic higher-order functions such as foldr. In our calculus, it can be given the type

$$\begin{aligned}\mathsf{foldr} \quad : \quad & \forall A\!:\!*.\ \forall B\!:\!\mathsf{ord} \xrightarrow{+} *. \\ & (\forall \imath\!:\!\mathsf{ord}.\ A \to B\,\imath \to B\,(\imath+1)) \to \\ & (\forall \imath\!:\!\mathsf{ord}.\ B\,(\imath+1)) \to \\ & (\forall \imath\!:\!\mathsf{ord}.\ \mathsf{List}^{\imath}\,A \to B\,\imath).\end{aligned}$$

By giving size information to polymorphic combinators such as foldr, even functions built-up from these combinators can have sized types. This is a novelty with regard to existing systems of type-based termination, like Barthe et al. [BFG+04], Hughes et al. [HPS96], and Xi [Xi02].

# Chapter 2

# Sized Higher-Order Subtyping

In this chapter, we define the kind and constructor level of $\mathsf{F}_{\widehat{\omega}}$, the higher-order polymorphic $\lambda$-calculus with sized types. We present rules for equality and subtyping and show their soundness through a set-theoretic model.

## 2.1 Constructors and Polarized Kinds

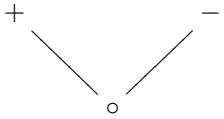
The higher-order polymorphic lambda-calculus introduces the notion of *type constructor* or just *constructor*. In the first intuition, it is a function which takes one or several types and produces a type, a so called *type transformer*. For example, the Cartesian product $\times$ is a constructor: It takes two types $A$ and $B$ and produces the product type $A \times B$ whose canonical elements are pairs $(r, s)$ of terms $r : A$ and $s : B$. The notion of constructor is then taken to higher orders: A type is a constructor, a function on constructors is a constructor itself.

### 2.1.1 Polarities

We aim to distinguish constructors with regard to their *monotonicity* or *variance*. For instance, the product constructor $\times$ is *isotone* or *covariant* in both of its arguments. If one enlarges the type $A$ or $B$, more terms inhabit $A \times B$. The opposite behavior is called *antitone* or *contravariant*. Finally, a function $F$ might not exhibit a uniform behavior, it might grow or shrink with its argument, or we just do not know how $F$ behaves. This is the general case, we call it *non-variant*. Each of the behaviors is called a *polarity* and abbreviated by one of the following symbols:

$$\begin{array}{rcll} \mathsf{Pol} \ni p, q & ::= & \circ & \text{non-variant} \\ & \mid & + & \text{covariant} \\ & \mid & - & \text{contravariant} \end{array}$$

The polarities are related: Since "non-variant" just means we do not have any information about the function, and we can forget our knowledge, each function is non-variant. The inclusion order between the three sets of co-, contra-, and non-variant functions induces a partial *information order* $\leq$ on $\mathsf{Pol}$. The smaller a set is, the more information it carries. Hence $\circ \leq p$, and $p \leq p$ for all $p$. We visualize the partial order on $\mathsf{Pol}$ as follows:




**Remark 2.1** In notation and order of polarities we follow Duggan and Compagnoni [DC99]. Detailed studies of polarized subtyping have been performed by Steffen [Ste98]. He uses a different notation and considers the dual ordering.

**Polarity of composed functions.** Let $F, G$ be two functions such that the composition $F \circ G$ is well-defined. If $F$ has polarity $p$ and $G$ has polarity $q$, we denote the polarity of the composed function $F \circ G$ by $pq$. It is clear that polarity composition is isotone: if one gets more information about $F$ or $G$, certainly one cannot have *less* information about $F \circ G$. Then, if one of the functions is non-variant, the same holds for the composition. In the remaining cases, the composition is covariant if $F$ and $G$ have the same variance, otherwise it is contravariant. We obtain the following multiplication table:

| | $\circ$ | $+$ | $-$ |
|---|---|---|---|
| $\circ$ | $\circ$ | $\circ$ | $\circ$ |
| $+$ | $\circ$ | $+$ | $-$ |
| $-$ | $\circ$ | $-$ | $+$ |

Polarity composition, as function composition, is associative. It is even commutative, but not *a priori*, since function composition is not commutative.

**Remark 2.2** Instead of polarities we could take signs $s \in \{0, +1, -1\}$ with ordinary multiplication and the non-standard order $s \leq s' \iff s = s' \vee |s| < |s'|$.

**Remark 2.3 ($\top$ polarity)** Steffen [Ste98] and Duggan and Compagnoni [DC99] include a fourth polarity, called $\top$ in second *loc. cit.*, which indicates that a function is invariant or constant. The symbol "$\top$" is justified since a constant function is both isotone and antitone. The ordering on polarities is extended by $p \leq \top$ for all $p$. For our purposes, $\top$ is not interesting.

### 2.1.2 Kinds

Constructors are classified by their *kind*, i. e., as types, functions on types, functions on such functions etc. Extending the kinds of $\mathsf{F}^\omega$, we introduce a second

base kind ord. Constructors of this kind are syntactic representations of ordinals.

$$\begin{array}{lcll} \mathsf{Kind} \ni \kappa & ::= & * & \text{types} \\ & \mid & \mathsf{ord} & \text{ordinals} \\ & \mid & p\kappa_1 \to \kappa_2 & p\text{-variant constructor transformers} \end{array}$$

A constructor of kind $p\kappa_1 \to \kappa_2$ is a $p$-variant function which maps constructors of kind $\kappa_1$ to constructors of kind $\kappa_2$. We introduce the following abbreviations:

$$\begin{array}{lll} \vec{p}\vec{\kappa} \to \kappa' & \text{for} & p_1\kappa_1 \to \cdots \to p_n\kappa_n \to \kappa', \\ \kappa \xrightarrow{p} \kappa' & \text{for} & p\kappa \to \kappa', \text{ and} \\ \vec{\kappa} \xrightarrow{\vec{p}} \kappa' & \text{for} & \vec{p}\vec{\kappa} \to \kappa'. \end{array}$$

Using the vector notation (first line) for kinds includes the assumption $|\vec{p}| = |\vec{\kappa}| = n$. Our *base kinds*, sometimes denoted by $\kappa 0$, are $*$ and ord. It is clear that every kind can be written in vector notation $\vec{p}\vec{\kappa} \to \kappa'$ with potentially empty vectors $\vec{p}$ and $\vec{\kappa}$. Especially, every kind can be written in this vector notation such that $\kappa'$ is a base kind.

*Pure kinds* are kinds which do not mention base kind "ord". These are the kinds of Steffen's polarized $\mathsf{F}^{\omega}_{\leq}$ and given by the following grammar.

$$\kappa_* ::= * \mid p\kappa_* \to \kappa'_*$$

The *rank* $\mathsf{rk}(\kappa) \in \mathbb{N}$ of a kind $\kappa$ is defined recursively as follows:

$$\begin{array}{lcll} \mathsf{rk}(\kappa 0) & = & 0 & \kappa 0 \in \{*, \mathsf{ord}\} \\ \mathsf{rk}(p\kappa \to \kappa') & = & \max(\mathsf{rk}(\kappa) + 1, \mathsf{rk}(\kappa')) & \end{array}$$

With $n := |\vec{\kappa}|$ we have $\mathsf{rk}(\vec{p}\vec{\kappa} \to \kappa 0) = \max\{1 + \mathsf{rk}(\kappa_i) \mid 1 \leq i \leq n\}$. If $n \neq 0$, then even $\mathsf{rk}(\vec{p}\vec{\kappa} \to \kappa 0) = 1 + \max\{\mathsf{rk}(\kappa_i) \mid 1 \leq i \leq n\}$.

### 2.1.3 Constructors

*Constructors* are given by the following Curry-style type-level lambda-calculus with some constants. The meta-variable $X$ ranges over a countably infinite set TyVar of constructor variables.

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

In most cases, we will use $F$, $G$, and $H$ for constructors of arbitrary kind, $A$ and $B$ for types (kind $*$) and $a$ and $b$ for ordinal expressions, i. e., constructors of kind ord. Ordinal variables will be denoted by $\imath$ and $\jmath$ whereas constructor variables of arbitrary kind will be denoted by $X$, $Y$ and $Z$. As usual, $\lambda X F$ binds variable $X$ in $F$. We identify constructors under $\alpha$-equivalence, i. e., under renaming of bound variables. $\mathsf{FV}(F)$ shall denote the set of free variables of constructor $F$.

**Signature.** The constructor constants $C$ are taken from a fixed *signature* $\Sigma$ which contains at least the following constants together with their kinding.

| | | |
|---|---|---|
| $\to$ | $: * \xrightarrow{-} * \xrightarrow{+} *$ | function space |
| $\forall_\kappa$ | $: (\kappa \xrightarrow{\circ} *) \xrightarrow{+} *$ | quantification |
| $\mu_{\kappa_*}$ | $: \mathsf{ord} \xrightarrow{+} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_*$ | inductive constructors |
| $\nu_{\kappa_*}$ | $: \mathsf{ord} \xrightarrow{-} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_*$ | coinductive constructors |
| $\mathsf{s}$ | $: \mathsf{ord} \xrightarrow{+} \mathsf{ord}$ | successor of ordinal |
| $\infty$ | $: \mathsf{ord}$ | infinity ordinal |

We often denote one of $\mu_{\kappa_*}$, $\nu_{\kappa_*}$ as $\nabla_{\kappa_*}$. Note that the indices $\kappa_*$ of these constants are pure kinds. Otherwise, we could define ordinals as least or greatest fixed points of some ordinal function. When clear from the context of discourse, we omit the indices of inductive and coinductive constructors and quantification. We introduce the following notations:

| | | |
|---|---|---|
| $\forall X{:}\kappa.\, A$ | for | $\forall_\kappa\, \lambda X A$, |
| $\forall X\, A$ | for | $\forall\, \lambda X A$, |
| $\nabla^a X{:}\kappa.\, F$ | for | $\nabla_\kappa\, a\, \lambda X F$, |
| $\nabla^a X\, F$ | for | $\nabla\, a\, \lambda X F$, and |
| $\nabla^a_\kappa$ | for | $\nabla_\kappa\, a$. |

The last notation is generalized to any constructor $F : \mathsf{ord} \xrightarrow{p} \kappa$ on ordinals: We allow the notation $F^a$ for application $F\, a$.

**Remark 2.4** In the above signature, the only *closed* constructors of kind ord are of the form $\mathsf{s}\,(\mathsf{s}\ldots(s\,\infty))$ for a finite number of successors $\mathsf{s}$. Semantically, all these ordinals are equal to $\infty$. Hughes, Pareto, and Sabry [HPS96] add a constant $0 : \mathsf{ord}$ denoting the ordinal zero, in order to get a more precise typing for constant objects of inductive type. Our syntax of ordinals is equivalent to the *stage expressions* of Barthe et al. [BFG+04].

**Example 2.5 (Impredicative encodings)** Constructors for Cartesian product $\times$, disjoint sum $+$, and existential type $\exists$ can be defined by the following impredicative encodings.

$$
\begin{array}{lcl}
\times & : & * \xrightarrow{+} * \xrightarrow{+} * \\
\times & := & \lambda X \lambda Y \forall Z{:}*.\, (X \to Y \to Z) \to Z \\
 & & \\
+ & : & * \xrightarrow{+} * \xrightarrow{+} * \\
+ & := & \lambda X \lambda Y \forall Z{:}*.\, (X \to Z) \to (Y \to Z) \to Z \\
 & & \\
\exists_\kappa & : & (\kappa \xrightarrow{\circ} *) \xrightarrow{+} * \\
\exists_\kappa & := & \lambda F \forall Z{:}*.\, (\forall X{:}\kappa.\, F\, X \to Z) \to Z
\end{array}
$$

We will use $\times$ and $+$ in infix notation and right associative. Later we will present a method to check that these definitions are well-kinded and do possess their claimed variance.

**Example 2.6 (Types with finitely many inhabitants)** We define $\mathbf{0} := \forall X\!:\!*.\, X$, $\mathbf{1} := \forall X\!:\!*.\, X \to X$, and $\mathsf{Bool} := \mathbf{2} := \mathbf{1} + \mathbf{1}$. It is clear how we could go on and define **3**, **4**, etc.

**Example 2.7 (Regular data types)** In the following examples, we define some inductive types. They are called *regular data types* because all of them are fixed points of a *type* transformer ($\mu_*$). The first type represents *natural numbers* in unary encoding, the second type polymorphic lists, the third one trees branching over type $B$ with node-labels of type $A$, and the fourth one Brouwer ordinals.

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

The first argument of the constructors in this example is an ordinal number describing the bound for the *height* of elements, when viewed as trees, in this data type. The height of a tree is defined as one plus the least upper bound of the heights of its immediate subtrees. There are no trees of height zero. Natural numbers and lists are linear trees, hence, their height is equal to one plus the length of the list resp. the unary encoding of the natural number. For finitely branching trees, the height is a finite ordinal, but for infinitely branching trees, it might be infinite. For instance, consider a $\mathsf{Nat}^\infty$ branching tree whose $n$th immediate subtree has height $n$. Then the height of the whole tree is $\omega + 1$.

Types like $\mathsf{Nat}^a$ and $\mathsf{List}^a\ A$ are called *sized types* since they carry a bound for the size of their inhabitants. Closely related sized type systems are $\lambda\hat{}$ (Barthe et al. [BFG$^+$04]) and the one of Hughes, Pareto, and Sabry [HPS96]. The latter one rejects infinitely branching trees and, hence, considers only ordinals up to $\omega$.

**Example 2.8 (Non-regular data types)** The following *heterogeneous* or *nested data types* are well-known from the literature on functional programming. Elements of $\mathsf{PList}^a\ A$ are *power lists* [BGJ00], i. e., list of length $2^n$ for some $n < a$, or, alternatively, perfect trees [Hin99], i. e., perfectly balanced leaf-labeled binary trees. The second type $\mathsf{Bush}^a\ A$, bushy lists, models finite maps from unlabeled binary trees of height $< a$ into $A$ [Alt01, Hin00b]. An the third type, $\mathsf{Lam}^a\ A$, is inhabited by de Bruijn representations of untyped lambda terms of height $< a$

with free variables in $A$ [BP99b, AR99].

$$\begin{array}{lcl} \mathsf{PList} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\ \mathsf{PList} & := & \lambda\iota.\, \mu^{\iota} X \lambda A.\, A + X\,(A \times A) \end{array}$$

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

$$\begin{array}{lcl} \mathsf{Lam} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\ \mathsf{Lam} & := & \lambda\iota.\, \mu^{\iota} X \lambda A.\, A + X\,A \times X\,A + X\,(\mathbf{1} + A) \end{array}$$

**Example 2.9 (Coinductive types)** The prime example of a coinductive type, $\mathsf{Stream}^a\ A$, contains infinite lists of elements in $A$ that are at least defined up to depth $a$. In other words, retrieving the $n$th element of a stream is guaranteed to succeed if $n < a$.

$$\begin{array}{lcl} \mathsf{Stream} & : & \mathsf{ord} \xrightarrow{-} * \xrightarrow{+} * \\ \mathsf{Stream} & := & \lambda\iota\lambda A.\, \nu^{\iota} \lambda X.\, A \times X \end{array}$$

### 2.1.4 Kinding

In this section, we present rules of *kinding*, i. e., assigning kinds to constructors. The rules extend the kinding rules of $\mathsf{F}^{\omega}$ by the treatment of polarities.

**Positive and negative occurrence.** In the simply typed lambda-calculus and in System F there are simple syntactic definitions of positive and negative occurrence of type variables in types. Let $A$ be a type expression, viewed as a tree. A type variable $X$ is said to occur *positively* in $A$ if the path from the root to $X$ takes the left branch of an $\to$-node an even number of times; otherwise, $X$ occurs *negatively*. This simple syntactic criterion does not scale to a higher-order type system like $\mathsf{F}^{\widehat{\omega}}$, the system under consideration, hence we will define positivity and negativity via the kinding judgement, following previous work [AM04]. Nevertheless, the simple criterion might serve the reader as a first intuition.

**Polarized contexts.** A polarized context $\Delta$ fixes a polarity $p$ and a kind $\kappa$ for each free variable $X$ of a constructor $F$. If $p = +$, then $X$ may only appear positively in $F$; this ensures that $\lambda X F$ is an isotone function. Similarly, if $p = -$, then $X$ may only occur negatively, and if $p = \circ$, then $X$ may appear in both positive and negative positions.

$$\begin{array}{lcll} \mathsf{PCxt} \ni \Delta & ::= & \diamond & \text{empty context} \\ & \mid & \Delta,\, X : p\kappa & \text{extended context } (X \notin \mathsf{dom}(\Delta)) \end{array}$$

The domain $\mathsf{dom}(\Delta)$ is the set of constructor variables $\Delta$ mentions. Naturally, each variable can appear in the context only once.

**Ordering on contexts.** We say context $\Delta'$ is more *liberal* than context $\Delta$, written $\Delta' \leq \Delta$, iff

$$(X : p\kappa) \in \Delta \text{ implies } (X : p'\kappa) \in \Delta' \text{ for some } p' \leq p$$

In particular, $\Delta'$ may declare more variables than $\Delta$ and assign weaker polarities to them. The intuition is that all constructors which are well-kinded in $\Delta$ are also well-kinded in a more permissive context $\Delta'$.

The empty context is the most strict context since $\Delta \leq \diamond$ for any $\Delta$. In contrast, there is no most liberal context.

**Application of polarities to contexts.** We define application $p\Delta$ of a polarity $p$ to a polarized context $\Delta$. It composes $p$ with every polarity assigned to a variable in $\Delta$. It is easy to see that $+\Delta = \Delta$ and $p(q\Delta) = (pq)\Delta$ since $+$ is the neutral element of $\mathsf{Pol}$ and polarity composition is associative, hence, $--\Delta = \Delta$ and $\circ\circ\Delta = \circ\Delta$. The last operation removes all polarity information from a context, making it isomorphic to a classical kinding context for System $\mathsf{F}^\omega$, except for the polarities within types. Application inherits monotonicity from polarity composition: $p\Delta \leq p'\Delta'$ if $p \leq p'$ and $\Delta \leq \Delta'$. An instance of monotonicity is the law $-\Delta \leq \Delta' \iff \Delta \leq -\Delta'$.

**Kinding.** We will introduce a judgement $\Delta \vdash F : \kappa$ which combines the usual notions of well-kindedness and positive and negative occurrences of type variables. A candidate for the application rule is

$$\frac{\Delta \vdash F : p\kappa \to \kappa' \qquad \Delta' \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}\ \Delta \leq p\Delta'.$$

The side condition is motivated by polarity composition. Consider the case that $X \notin \mathsf{FV}(F)$. If $G$ is viewed as a function of $X$, then $F\,G$ is the composition of $F$ and $G$. Now if $G$ is $q$-variant in $X$, then $F\,G$ is $pq$-variant in $X$. This means that all $q$-variant variables of $\Delta'$ must appear in $\Delta$ with a polarity of at most $pq$. Now if $X \in \mathsf{FV}(F)$, it could be that it is actually declared in $\Delta$ with a polarity smaller than $pq$. Also, variables which are not free in $G$ are not affected by the application $F\,G$, hence they can carry the same polarity in $F\,G$ as in $F$. Together this motivates the condition $\Delta \leq p\Delta'$.

To eliminate the side condition from the application rule, we need a function which computes the most liberal context $\Delta'$ from $\Delta$ which satisfies both conditions.

**Inverse application of polarities.** We are looking for an inverse to application, $p^{-1}\Delta$, which should be the least $\Delta'$ such that $\Delta \leq p\Delta'$. If such an inverse application exists, it is monotone and forms a *Galois connection* with the application operation, i. e.,

$$p^{-1}\Delta \leq \Delta' \iff \Delta \leq p\Delta'.$$

This inverse indeed exists and obeys the equations $+^{-1}\Delta = \Delta$, $-^{-1}\Delta = -\Delta$, and $\circ^{-1}\Delta = \{X : p\kappa \in \Delta \mid p = \circ\}$, respectively.[1] In all three cases, $p^{-1}(p\Delta) \leq \Delta$, and the side condition $\Delta \leq p(p^{-1}\Delta)$ holds.

**Kinding (continued).** Now we are ready to introduce a judgement $\Delta \vdash F : \kappa$ inductively by the following rules:

$$\text{KIND-C}\ \frac{C : \kappa \in \Sigma}{\Delta \vdash C : \kappa} \qquad \text{KIND-VAR}\ \frac{X : p\kappa \in \Delta \qquad p \leq +}{\Delta \vdash X : \kappa}$$

$$\text{KIND-ABS}\ \frac{\Delta, X : p\kappa \vdash F : \kappa'}{\Delta \vdash \lambda X F : p\kappa \to \kappa'}$$

$$\text{KIND-APP}\ \frac{\Delta \vdash F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

To understand these rules, it is useful to think of $F$ in the context $\Delta = \vec{X} : \vec{p\kappa}$ as a function $F(\vec{X})$. The judgement $\Delta \vdash F : \kappa$ should be valid if both $F$ has kind $\kappa$ and $F$ grows (weakly) whenever the positive arguments to $F$ grow, the negative shrink and the non-variant stay fixed.

Hence, rule KIND-VAR can only allow non-negative variables to be fetched from the context. In contrast, constants may appear with any polarity (KIND-C). The rule KIND-ABS for abstraction is suggestive.

**Explanation of the application rule.** Because of its central role, rule KIND-APP is given again a very detailed explanation. First, observe that whenever $F$ grows (with fixed $G$), then also the application $F\,G$ grows. Hence, the polarity of the variables in $F$ is the same as those in $F\,G$. To see how the application behaves if we modify $G$, let us consider the cases $p = +, -, \circ$ separately:

$$\text{KIND-APP}+\ \frac{\Delta \vdash F : +\kappa \to \kappa' \qquad \Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

In the first case, $F$ is a monotone function, hence whenever its argument $G$ grows, the application $F\,G$ grows as well. Hence, the polarity of the variables in $G$ matches the polarity of the variables in $F\,G$.

$$\text{KIND-APP}-\ \frac{\Delta \vdash F : -\kappa \to \kappa' \qquad -\Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

If $F$ is antitone, the application $F\,G$ will grow if $G$ shrinks. Hence, the polarity of the variables in $G$ must be opposite ($-\Delta$) to the one of the variables in the application ($\Delta$). Imagine a variable $X$ appearing negatively in $G$. If it grows, $G$

[1] Unlike for integers, reversing the sign does not reverse the inequality: We have $-\Delta \leq \Delta' \iff \Delta \leq -\Delta'$.

will shrink, hence $F\,G$ will grow. Therefore $X$ appears positively in the application.

$$\text{KIND-APP}\circ\ \frac{\Delta \vdash F : \circ\kappa \to \kappa' \qquad \circ^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

In the non-variant case, we do not know how $F$ behaves if we modify its argument. To satisfy our informal semantics, $F\,G$ needs to grow if we grow the variables declared in $\Delta$ to be positive, shrink the negative variables, and leave the non-variant ones fixed. If one of the positive or negative variables appeared in $G$, the argument to $F$ would change which would result in an unpredictable shift of the value of $F\,G$. If we want to verify that $F\,G$ grows we need to ensure that no positive or negative variables occur in $G$. This is done by erasing all non-variant variables from the context through the operation $\circ^{-1}\Delta$.

**Example 2.10 (Derived rules for function space and quantification)** The following rules are derivable:

$$\frac{-\Delta \vdash A : * \qquad \Delta \vdash B : *}{\Delta \vdash A \to B : *} \qquad \frac{\Delta, X : \circ\kappa \vdash A : *}{\Delta \vdash \forall X{:}\kappa.\, A : *}$$

### 2.1.5 Equality

Type constructors are considered equal modulo $\beta\eta$. Further, the successor of the closure ordinal is considered equal to the closure ordinal. The judgement $\Delta \vdash F = F' : \kappa$ is defined inductively by the following rules:

**Computation axioms.**

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

$$\text{EQ-}\eta\ \frac{\Delta \vdash F : p\kappa \to \kappa'}{\Delta \vdash (\lambda X.\, F\,X) = F : p\kappa \to \kappa'}$$

$$\text{EQ-}\infty\ \frac{}{\Delta \vdash \mathsf{s}\,\infty = \infty : \mathsf{ord}}$$

**Congruence rules.** In rule EQ-APP we have to modify the polarities of the variables in $G$ and $G'$ in the same way as in the application rule of the kinding judgement: We inverse-apply $p$ to the context $\Delta$.

$$\text{EQ-C}\ \frac{C : \kappa \in \Sigma}{\Delta \vdash C = C : \kappa} \qquad \text{EQ-VAR}\ \frac{X : p\kappa \in \Delta \qquad p \leq +}{\Delta \vdash X = X : \kappa}$$

$$\text{EQ-}\lambda\ \frac{\Delta, X : p\kappa \vdash F = F' : \kappa'}{\Delta \vdash \lambda X F = \lambda X F' : p\kappa \to \kappa'}$$

$$\text{EQ-APP}\ \frac{\Delta \vdash F = F' : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G = G' : \kappa}{\Delta \vdash F\,G = F'\,G' : \kappa'}$$

**Symmetry and transitivity.**

$$\text{EQ-SYM}\ \frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F' = F : \kappa}$$

$$\text{EQ-TRANS}\ \frac{\Delta \vdash F_1 = F_2 : \kappa \qquad \Delta \vdash F_2 = F_3 : \kappa}{\Delta \vdash F_1 = F_3 : \kappa}$$

The rule for reflexivity is admissible thanks to the congruence rules:

**Lemma 2.11 (Reflexivity)** *If $\mathcal{D} :: \Delta \vdash F : \kappa$ then $\Delta \vdash F = F : \kappa$.*

*Proof.* By induction on $\mathcal{D}$. □

**Lemma 2.12 (Validity)** *If $\mathcal{D} :: \Delta \vdash F = F' : \kappa$ then $\Delta \vdash F : \kappa$ and $\Delta \vdash F' : \kappa$.*

*Proof.* By induction on $\mathcal{D}$. □

Together, $\Delta \vdash F : \kappa$ holds if and only if $\Delta \vdash F = F : \kappa$.

**Remark 2.13** The rules for kinding and equality are in essence those used in previous work of the author with Matthes [AM04], extended by EQ-$\infty$. Note that in loc. cit., the inverse application $p^{-1}\Delta$ of a polarity to a context was written $p\Delta$.

## 2.2 Higher-Order Subtyping

In this section, we specify subtyping for constructors of polarized kinds. The rules are inspired by Steffen [Ste98].

**Reflexivity, transitivity and antisymmetry.** These three properties make subtyping a partial order on constructors of the same kind.

$$\text{LEQ-REFL}\ \frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F \leq F' : \kappa}$$

$$\text{LEQ-TRANS}\ \frac{\Delta \vdash F_1 \leq F_2 : \kappa \qquad \Delta \vdash F_2 \leq F_3 : \kappa}{\Delta \vdash F_1 \leq F_3 : \kappa}$$

$$\text{LEQ-ANTISYM}\ \frac{\Delta \vdash F \leq F' : \kappa \qquad \Delta \vdash F' \leq F : \kappa}{\Delta \vdash F = F' : \kappa}$$

The reflexivity rule includes the subtyping axioms for variables and constants as special cases. Reflexivity and transitivity together ensure that subtyping is compatible with equality. The antisymmetry rule potentially enlarges our notion of equality.

**Abstraction.**

$$\text{LEQ-}\lambda\ \frac{\Delta,\ X : p\kappa \vdash F \leq F' : \kappa'}{\Delta \vdash \lambda X F \leq \lambda X F' : p\kappa \to \kappa'}$$

**Application.** There are two kinds of congruence rules for application: one kind states that if functions $F$ and $F'$ are in the subtyping relation, so are their values $F\,G$ and $F'\,G$ at a certain argument $G$.

$$\text{LEQ-APP}\ \frac{\Delta \vdash F \leq F' : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G \leq F'\,G : \kappa'}$$

The other kind of rules concern the opposite case: If $F$ is a function and two arguments $G$ and $G'$ are in a subtyping relation, so are the values $F\,G$ and $F\,G'$ of the function at these arguments. However, such a relation can only exist if $F$ is either covariant or contravariant.

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

$$\text{LEQ-APP}-\ \frac{\Delta \vdash F : -\kappa \to \kappa' \qquad -\Delta \vdash G' \leq G : \kappa}{\Delta \vdash F\,G \leq F\,G' : \kappa'}$$

What about a comparable rule for non-variant constructors? It is derivable:

$$\dfrac{\Delta \vdash F : \circ\kappa \to \kappa' \qquad \dfrac{\circ^{-1}\Delta \vdash G \leq G' : \kappa \qquad \circ^{-1}\Delta \vdash G' \leq G : \kappa}{\circ^{-1}\Delta \vdash G = G' : \kappa}}{\dfrac{\Delta \vdash F\,G = F\,G' : \kappa'}{\Delta \vdash F\,G \leq F\,G' : \kappa'}}$$

**Successor and infinity.**

$$\text{LEQ-S-R}\ \frac{\Delta \vdash a : \mathsf{ord}}{\Delta \vdash a \leq \mathsf{s}\,a : \mathsf{ord}} \qquad \text{LEQ-}\infty\ \frac{\Delta \vdash a : \mathsf{ord}}{\Delta \vdash a \leq \infty : \mathsf{ord}}$$

**Lemma 2.14 (Validity II)** *If $\mathcal{D} :: \Delta \vdash F \leq F' : \kappa$ then $\Delta \vdash F : \kappa$ and $\Delta \vdash F' : \kappa$.*

*Proof.* By induction on $\mathcal{D}$, using validity of equality (Lemma 2.12) in case of LEQ-REFL. □

## 2.3 Semantics and Soundness

In this section we give a semantics to kinds and constructors and show soundness of kinding, constructor equality and subtyping.

### 2.3.1 Interpretation of Kinds

Constructors $F$ of kind $\kappa$ will be interpreted as operators $\mathcal{F}$ which live in the denotation $[\![\kappa]\!]$ of their kinds. Each kind will be interpreted as a poset (partially ordered set) $([\![\kappa]\!], \sqsubseteq^\kappa)$, which is even a complete lattice in each case.

**Interpretation of base kind $*$.** For the moment, we assume a complete lattice $[\![*]\!]$ of countable sets $\mathcal{A} \in [\![*]\!]$ ordered by inclusion, with a maximal set $\top^* \in [\![*]\!]$ such that $\mathcal{A} \subseteq \top^*$ for all $\mathcal{A} \in [\![*]\!]$. Later, we will let $[\![*]\!]$ be the collection of all saturated sets $\subseteq \mathsf{SN}$ where $\top^* = \mathsf{SN}$ is the set of strongly normalizing terms. So, let

$$\begin{array}{rcl} [\![*]\!] & : & \text{complete lattice of sets} \\ \mathcal{A} \sqsubseteq^* \mathcal{A}' & :\Longleftrightarrow & \mathcal{A} \subseteq \mathcal{A}' \\ \bigsqcap^* \mathfrak{A} & := & \bigcap \mathfrak{A} \ \text{(where } \mathfrak{A} \subseteq [\![*]\!]\text{)}. \end{array}$$

By assumption, the poset $([\![*]\!], \sqsubseteq^*)$ is closed under infima, i. e., for a non-empty subset $\mathfrak{A} \subseteq [\![*]\!]$ the infimum $\bigsqcap^* \mathfrak{A} \in [\![*]\!]$ exists and is equal to the intersection $\bigcap \mathfrak{A}$. Intersection can be extended to empty collections by letting $\bigsqcap^* \emptyset = \top^*$. Once empty intersections are defined, we can define arbitrary suprema by $\bigsqcup^* \mathfrak{A} := \bigsqcap^* \{\mathcal{B} \in [\![*]\!] \mid \mathcal{B} \sqsupseteq^* \mathcal{A} \text{ for all } \mathcal{A} \in \mathfrak{A}\}$. Note that we do not require that the supremum is the union of sets; it might actually be something bigger. On the set $[\![*]\!]$ we assume a binary operation "$\to$" (function space construction) such that $\mathcal{A} \to \mathcal{B} \sqsubseteq^* \mathcal{A}' \to \mathcal{B}'$ if $\mathcal{A}' \sqsubseteq^* \mathcal{A}$ and $\mathcal{B} \sqsubseteq^* \mathcal{B}'$.

**Interpretation of base kind** $\mathsf{ord}$**.** Constructors "$a$" of kind $\mathsf{ord}$ denote set-theoretic ordinals in our semantics. We choose an initial segment $[0; \top^{\mathsf{ord}}] =: [\![\mathsf{ord}]\!]$ of the ordinals for the interpretation of $\mathsf{ord}$. At the moment we leave it open which ordinal $\top^{\mathsf{ord}}$ denotes; we will fill it in later.

$$\begin{array}{rcl} [\![\mathsf{ord}]\!] & := & \top^{\mathsf{ord}} + 1 \\ \alpha \sqsubseteq^{\mathsf{ord}} \alpha' & :\Longleftrightarrow & \alpha \leq \alpha' \end{array}$$

**Notation.** We introduce a notation $\mathcal{F} \sqsubseteq^{p\kappa} \mathcal{F}'$ for polarized inclusion and the notion $\mathcal{F} \sqsubseteq^{p} \mathcal{F}' \in [\![\kappa]\!]$ which expresses polarized inclusion for two operators $\mathcal{F}, \mathcal{F}'$ plus the fact that both are in the set $[\![\kappa]\!]$.

$$\begin{array}{lcl} \mathcal{F} \sqsubseteq^{+\kappa} \mathcal{F}' & :\Longleftrightarrow & \mathcal{F} \sqsubseteq^{\kappa} \mathcal{F}' \\ \mathcal{F} \sqsubseteq^{-\kappa} \mathcal{F}' & :\Longleftrightarrow & \mathcal{F}' \sqsubseteq^{\kappa} \mathcal{F} \\ \mathcal{F} \sqsubseteq^{\circ\kappa} \mathcal{F}' & :\Longleftrightarrow & \mathcal{F} \sqsubseteq^{\kappa} \mathcal{F}' \text{ and } \mathcal{F}' \sqsubseteq^{\kappa} \mathcal{F} \\ \\ \mathcal{F} \sqsubseteq^{p} \mathcal{F}' \in [\![\kappa]\!] & :\Longleftrightarrow & \mathcal{F}, \mathcal{F}' \in [\![\kappa]\!] \text{ and } \mathcal{F} \sqsubseteq^{p\kappa} \mathcal{F}' \\ \mathcal{F} \sqsubseteq \mathcal{F}' \in [\![\kappa]\!] & :\Longleftrightarrow & \mathcal{F} \sqsubseteq^{+} \mathcal{F}' \in [\![\kappa]\!] \end{array}$$

**Interpretation of function kinds.** Semantically, a constructor $F$ of kind $p\kappa \to \kappa'$ is a covariant ($p = +$), contravariant ($p = -$) or non-variant ($p = \circ$) operator. We define the posets $(\llbracket\kappa\rrbracket, \sqsubseteq^\kappa)$ for higher kinds by induction on $\kappa$.

$$\begin{array}{rcl} \llbracket p\kappa \to \kappa' \rrbracket & := & \{\mathcal{F} \in \llbracket\kappa\rrbracket \to \llbracket\kappa'\rrbracket \mid \mathcal{F}(\mathcal{G}) \sqsubseteq \mathcal{F}(\mathcal{G}') \in \llbracket\kappa'\rrbracket \\ & & \qquad \text{for all } \mathcal{G} \sqsubseteq^p \mathcal{G}' \in \llbracket\kappa\rrbracket\} \\ \mathcal{F} \sqsubseteq^{p\kappa\to\kappa'} \mathcal{F}' & :\Longleftrightarrow & \mathcal{F}(\mathcal{G}) \sqsubseteq^{\kappa'} \mathcal{F}'(\mathcal{G}) \text{ for all } \mathcal{G} \in \llbracket\kappa\rrbracket \end{array}$$

**Lemma 2.15 (Partial order)** *For each kind $\kappa$, the relation $\sqsubseteq^\kappa$ denotes a partial order on $\llbracket\kappa\rrbracket$.*

*Proof.* By induction on $\kappa$. For base kinds $\kappa 0 \in \{*, \mathsf{ord}\}$ reflexivity, transitivity and antisymmetry hold by definition. To prove transitivity for higher kinds, assume $\kappa = p\kappa_1 \to \kappa_2$ and $\mathcal{F}_1 \sqsubseteq \mathcal{F}_2 \in \llbracket\kappa\rrbracket$, $\mathcal{F}_2 \sqsubseteq \mathcal{F}_3 \in \llbracket\kappa\rrbracket$, and an arbitrary $\mathcal{G} \in \llbracket\kappa_1\rrbracket$. Since by ind. hyp. $\mathcal{G} \sqsubseteq^{\kappa_1} \mathcal{G}$, we have $\mathcal{F}_1(\mathcal{G}) \sqsubseteq \mathcal{F}_2(\mathcal{G}) \in \llbracket\kappa_2\rrbracket$ and $\mathcal{F}_2(\mathcal{G}) \sqsubseteq \mathcal{F}_3(\mathcal{G}) \in \llbracket\kappa_2\rrbracket$ by definition. By induction hypothesis $\mathcal{F}_1(\mathcal{G}) \sqsubseteq^{\kappa_2} \mathcal{F}_3(\mathcal{G})$, and since $\mathcal{G}$ was arbitrary $\mathcal{F}_1 \sqsubseteq^{p\kappa_1\to\kappa_2} \mathcal{F}_3$. Reflexivity and antisymmetry are proven analogously. □

**Pointwise infima, upper bounds and suprema.** For higher kinds, we define inductively pointwise infimum and maximal element as follows.

$$\begin{array}{lcl} \bigsqcap^{p\kappa\to\kappa'} \mathfrak{F} & \in & \llbracket\kappa\rrbracket \to \llbracket\kappa'\rrbracket \quad \text{for } \mathfrak{F} \subseteq \llbracket p\kappa \to \kappa'\rrbracket \\ (\bigsqcap^{p\kappa\to\kappa'} \mathfrak{F})(\mathcal{G}) & := & \bigsqcap^{\kappa'} \{\mathcal{F}(\mathcal{G}) \mid \mathcal{F} \in \mathfrak{F}\} \\ \\ \top^{p\kappa\to\kappa'} & \in & \llbracket p\kappa \to \kappa'\rrbracket \\ \top^{p\kappa\to\kappa'}(\mathcal{G}) & := & \top^{\kappa'} \end{array}$$

A simple proof by induction on $\kappa$ shows that $\top^\kappa$ is really the maximal element of $\llbracket\kappa\rrbracket$ for any kind $\kappa$. Extending the observations for kind $*$, we can now define empty infima and arbitrary suprema for all kinds.

$$\begin{array}{lcl} \bigsqcap^\kappa \emptyset & := & \top^\kappa \\ \bigsqcup^\kappa \mathfrak{F} & := & \bigsqcap^\kappa \{\mathcal{H} \in \llbracket\kappa\rrbracket \mid \mathcal{H} \sqsupseteq^\kappa \mathcal{F} \text{ for all } \mathcal{F} \in \mathfrak{F}\} \end{array}$$

**Lemma 2.16 (Supremum is pointwise)** $(\bigsqcup^{p\kappa\to\kappa'} \mathfrak{F})(\mathcal{G}) = \bigsqcup^{\kappa'} \{\mathcal{F}(\mathcal{G}) \mid \mathcal{F} \in \mathfrak{F}\}$.

The posets $\llbracket\kappa\rrbracket$ now are equipped with everything required for complete lattices.

**Lemma 2.17 (Complete lattice)** *For all kinds $\kappa$, the triple $(\llbracket\kappa\rrbracket, \bigsqcap^\kappa, \bigsqcup^\kappa)$ forms a complete lattice.*

*Proof.* We only need to show that $\bigsqcap^\kappa \mathfrak{F} \in \llbracket\kappa\rrbracket$ is the well-defined greatest lower bound for $\mathfrak{F} \subseteq \llbracket\kappa\rrbracket$ by induction on $\kappa$. For base kinds, there is nothing to prove.

1. Well-definedness: Show $\bigsqcap^{p\kappa\to\kappa'}\mathfrak{F} \in [\![p\kappa \to \kappa']\!]$. Assume $\mathcal{G} \sqsubseteq^{p} \mathcal{G}' \in [\![\kappa]\!]$. Then $\mathcal{F}(\mathcal{G}) \sqsubseteq \mathcal{F}(\mathcal{G}') \in [\![\kappa']\!]$ for all $\mathcal{F} \in \mathfrak{F}$. Since the infimum is well-defined at kind $\kappa'$ by induction hypothesis, this entails

$$\begin{aligned}(\textstyle\bigsqcap^{p\kappa\to\kappa'}\mathfrak{F})(\mathcal{G}) &= \textstyle\bigsqcap^{\kappa'}\{\mathcal{F}(\mathcal{G}) \mid \mathcal{F} \in \mathfrak{F}\} \sqsubseteq^{\kappa'} \\ &\sqsubseteq^{\kappa'} \textstyle\bigsqcap^{\kappa'}\{\mathcal{F}(\mathcal{G}') \mid \mathcal{F} \in \mathfrak{F}\} = (\bigsqcap^{p\kappa\to\kappa'}\mathfrak{F})(\mathcal{G}').\end{aligned}$$

2. Lower bound: Show $\bigsqcap^{p\kappa\to\kappa'}\mathfrak{F} \sqsubseteq^{p\kappa\to\kappa'} \mathcal{F}$ for all $\mathcal{F} \in \mathfrak{F}$. Assume $\mathcal{G} \in [\![\kappa]\!]$ arbitrary. Since by induction hypothesis, $\bigsqcap^{\kappa'}$ is a lower bound,

$$(\textstyle\bigsqcap^{p\kappa\to\kappa'}\mathfrak{F})(\mathcal{G}) = \bigsqcap^{\kappa'}\{\mathcal{F}(\mathcal{G}) \mid \mathcal{F} \in \mathfrak{F}\} \sqsubseteq^{\kappa'} \mathcal{F}(\mathcal{G})$$

   for any $\mathcal{F} \in \mathfrak{F}$.

3. Greatest lower bound: Let $\mathcal{H} \sqsubseteq \mathcal{F} \in [\![p\kappa \to \kappa']\!]$ for all $\mathcal{F} \in \mathfrak{F}$. Show $\mathcal{H} \sqsubseteq^{p\kappa\to\kappa'} \bigsqcap^{p\kappa\to\kappa'}\mathfrak{F}$. For $\mathcal{G} \in [\![\kappa]\!]$ arbitrary, $\mathcal{H}(\mathcal{G}) \sqsubseteq^{\kappa'} \mathcal{F}(\mathcal{G})$ for any $\mathcal{F} \in \mathfrak{F}$ by assumption. Since by induction hypothesis $\bigsqcap^{\kappa'}$ is a greatest lower bound,

$$\mathcal{H}(\mathcal{G}) \sqsubseteq^{\kappa'} \textstyle\bigsqcap^{\kappa'}\{\mathcal{F}(\mathcal{G}) \mid \mathcal{F} \in \mathfrak{F}\} = (\bigsqcap^{p\kappa\to\kappa'}\mathfrak{F})(\mathcal{G}).$$

□

### 2.3.2 Semantics of Constructors

In the following we develop a semantics of constructors through their derivations of well-kindedness. This indirect path is necessary since the constructors are domain-free. E. g., it is not determined which function is denoted by the constructor $\lambda XX$; it could be the identity function on $[\![\kappa]\!]$ for any kind $\kappa$. In joint work with Ralph Matthes I have investigated polarized kinding and semantics of Church-style constructors [AM04]. There, $\lambda X\!:\!+\kappa.X$ denotes exactly one set-theoretic function: the identity on $[\![\kappa]\!]$. The following development resembles closely the cited work, however, we take the detour via derivations here.

**Sound valuations.** Let $\theta$ be a mapping from constructor variables to sets. We say $\theta \in [\![\Delta]\!]$ if $\theta(X) \in [\![\kappa]\!]$ for all $(X\!:\!p\kappa) \in \Delta$. A partial order on valuations is established as follows:

$$\theta \sqsubseteq \theta' \in [\![\Delta]\!] \;:\Longleftrightarrow\; \theta(X) \sqsubseteq^{p} \theta'(X) \in [\![\kappa]\!] \text{ for all } (X\!:\!p\kappa) \in \Delta$$

We use $\sqsubseteq^{-}$ for $\sqsupseteq$ and $\sqsubseteq^{\circ}$ for $=$, and $\sqsubseteq^{+}$ as synonym for $\sqsubseteq$. It is clear that $\theta \sqsubseteq^{q} \theta' \in [\![\Delta]\!]$ iff $\theta(X) \sqsubseteq^{pq} \theta'(X) \in [\![\kappa]\!]$ for all $(X\!:\!p\kappa) \in \Delta$.

**Lemma 2.18** *If $\theta \sqsubseteq \theta' \in [\![\Delta]\!]$, then $\theta \sqsubseteq^p \theta' \in [\![p^{-1}\Delta]\!]$.*

*Proof.* By cases on $p$. Interesting is only case $p = \circ$. Assume $X : q\kappa \in [\![\circ^{-1}\Delta]\!]$, which is only possible if $q = \circ$ and $X : \circ\kappa \in [\![\Delta]\!]$. We have to show $\theta(X) \sqsubseteq^{\circ\circ} \theta'(X) \in [\![\kappa]\!]$ which follows from the premise of the lemma. □

**Remark 2.19** The opposite implication does *not* hold in case $p = \circ$.

**Denotation of constructors.** If $\mathcal{D} :: \Delta \vdash F : \kappa$ and $\theta$ is a function from type variables to sets, we define the set $[\![\mathcal{D}]\!]_\theta$ by recursion on $\mathcal{D}$ as follows.

*Case*

$$\mathcal{D} = \frac{X : p\kappa \in \Delta \qquad p \leq +}{\Delta \vdash X : \kappa}$$

We define $[\![\mathcal{D}]\!]_\theta = \theta(X)$.

*Case*

$$\mathcal{D} = \frac{C : \kappa \in \Sigma}{\Delta \vdash C : \kappa}$$

In this case, we simply return the semantics of $C$, which is defined elsewhere: $[\![\mathcal{D}]\!]_\theta = \mathsf{Sem}(C)$.

*Case*

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}' \\ \Delta, X : p\kappa \vdash F : \kappa'\end{array}}{\Delta \vdash \lambda X F : p\kappa \to \kappa'}$$

The semantics of $\mathcal{D}$ is a function over $[\![\kappa]\!]$, defined by $[\![\mathcal{D}]\!]_\theta(\mathcal{G} \in [\![\kappa]\!]) := [\![\mathcal{D}']\!]_{\theta[X \mapsto \mathcal{G}]}$. Note that this is only possible if we know the domain of the function ($\kappa$, in this case). This is the reason why we define the semantics of *derivations* instead of constructors (where we would not have the domain available).

*Case*

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 \\ \Delta \vdash F : p\kappa \to \kappa'\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ p^{-1}\Delta \vdash G : \kappa\end{array}}{\Delta \vdash F\,G : \kappa'}$$

Let $\mathcal{F} := [\![\mathcal{D}_1]\!]_\theta$ and $\mathcal{G} := [\![\mathcal{D}_2]\!]_\theta$. We simply define $[\![\mathcal{D}]\!]_\theta := [\![\mathcal{D}_1]\!]_\theta([\![\mathcal{D}_2]\!]_\theta)$. If $\mathcal{F}$ is not a function or $\mathcal{G}$ not in the domain of $\mathcal{F}$, then the application is defined to be the empty set.

**Lemma 2.20 (Well-definedness and monotonicity)** *Let $\mathcal{D} :: \Delta \vdash F : \kappa$. If $\theta \sqsubseteq \theta' \in [\![\Delta]\!]$ then $[\![\mathcal{D}]\!]_\theta \sqsubseteq [\![\mathcal{D}]\!]_{\theta'} \in [\![\kappa]\!]$.*

The lemma generalizes to $p$-monotonicity:

**Corollary 2.21 ($p$-monotonicity)** *Let $\mathcal{D} :: p^{-1}\Delta \vdash F : \kappa$. If $\theta \sqsubseteq \theta' \in [\![\Delta]\!]$ then $[\![\mathcal{D}]\!]_\theta \sqsubseteq^p [\![\mathcal{D}]\!]_{\theta'} \in [\![\kappa]\!]$.*

*Proof.* By case distinction on $p$, using Lemma 2.18. □

*Proof of the lemma.* By induction on $\mathcal{D}$.

*Case*

$$\mathcal{D} = \frac{X : p\kappa \in \Delta \qquad p \leq +}{\Delta \vdash X : \kappa}$$

Recall that $[\![\mathcal{D}]\!]_\theta = \theta(X)$. Assuming $\theta \sqsubseteq \theta' \in [\![\Delta]\!]$, we implicitly require—since $p \leq +$—that $\theta(X) \sqsubseteq \theta'(X) \in [\![\kappa]\!]$. Hence $[\![\mathcal{D}]\!]_\theta \sqsubseteq [\![\mathcal{D}]\!]_{\theta'} \in [\![\kappa]\!]$.

*Case*

$$\mathcal{D} = \frac{C : \kappa \in \Sigma}{\Delta \vdash C : \kappa}$$

We require that $\mathsf{Sem}(C) \in [\![\kappa]\!]$ for all $(C : \kappa) \in \Sigma$.

*Case*

$$\mathcal{D} = \frac{\mathcal{D}'}{\Delta \vdash \lambda X F : p\kappa \to \kappa'} \quad \Delta, X : p\kappa \vdash F : \kappa'$$

Recall that $[\![\mathcal{D}]\!]_\theta(\mathcal{G} \in [\![\kappa]\!]) = [\![\mathcal{D}']\!]_{\theta[X \mapsto \mathcal{G}]}$. Assuming $\mathcal{G} \sqsubseteq^p \mathcal{G}' \in [\![\kappa]\!]$, we have $\theta[X \mapsto \mathcal{G}] \sqsubseteq \theta'[X \mapsto \mathcal{G}'] \in [\![\Delta, X : p\kappa]\!]$. By induction hypothesis, $[\![\mathcal{D}]\!]_\theta(\mathcal{G}) \sqsubseteq [\![\mathcal{D}]\!]_{\theta'}(\mathcal{G}') \in [\![\kappa']\!]$, which implies $[\![\mathcal{D}]\!]_\theta \sqsubseteq [\![\mathcal{D}]\!]_{\theta'} \in [\![p\kappa \to \kappa']\!]$.

*Case*

$$\mathcal{D} = \frac{\mathcal{D}_1 \qquad\qquad \mathcal{D}_2}{\Delta \vdash F\, G : \kappa'} \quad \Delta \vdash F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa$$

Assume $\theta \sqsubseteq \theta' \in [\![\Delta]\!]$. By induction hypothesis, $[\![\mathcal{D}_1]\!]_\theta \sqsubseteq [\![\mathcal{D}_1]\!]_{\theta'} \in [\![p\kappa \to \kappa']\!]$. By second induction hypothesis, using the corollary, we have $[\![\mathcal{D}_2]\!]_\theta \sqsubseteq^p [\![\mathcal{D}_2]\!]_{\theta'} \in [\![\kappa]\!]$. Hence, the application $[\![\mathcal{D}]\!]_\theta = [\![\mathcal{D}_1]\!]_\theta([\![\mathcal{D}_2]\!]_\theta) \in [\![\kappa']\!]$ is well-defined, and further, $[\![\mathcal{D}]\!]_\theta \sqsubseteq [\![\mathcal{D}]\!]_{\theta'} \in [\![\kappa']\!]$. □

The remainder of this section is devoted to the proof that two well-kindedness derivations for the same constructors do not yield different semantics. More precisely, assume two derivations $\mathcal{D} :: \Delta \vdash F : \kappa$ and $\mathcal{D}' :: \Delta' \vdash F : \kappa$ and two valuations $\theta \in [\![\Delta]\!]$ and $\theta' \in [\![\Delta']\!]$. If $\theta(X) = \theta'(X)$ for all $X \in \mathsf{FV}(F)$, then $[\![\mathcal{D}]\!]_\theta = [\![\mathcal{D}']\!]_{\theta'}$. This result is not completely trivial, i. e., cannot be proven directly by induction on the derivations, since in derivations for non-$\beta$-normal $F$, some kinds in the middle of derivations can be canceled out. For example, consider $F = (\lambda X Y)\, G$. Some kind $\kappa$ for $X$ (and $G$) is mentioned in the middle of a well-kindness derivation of $F$, but it can differ from derivation to derivation. Still, the semantics of $F$ in environment $\theta$ should be just $\theta(Y)$, independent of kind $\kappa$.

**$\beta$-normal forms** are given by the grammar

$$V ::= C\,\vec{V} \mid X\,\vec{V} \mid \lambda X V.$$

One step $\beta$-reduction on constructorsbeta-red@$\beta$-reduction of constructors $F \longrightarrow_\beta F'$ is defined as usual. Since the simply-typed $\lambda$-calculus is strongly normalizing, it is clear that well-kinded constructors $\Delta \vdash F : \kappa$ reach a normal form: $F \longrightarrow_\beta^* V$ for some $V$.

**Lemma 2.22 (Derivation-independence for normal forms)** *Assume two derivations $\mathcal{D}^1 :: \Delta^1 \vdash V : \kappa$ and $\mathcal{D}^2 :: \Delta^2 \vdash V : \kappa$ of well-kindedness for the same normal form $V$, and two valuations $\theta^1 \in [\![\Delta^1]\!]$ and $\theta^2 \in [\![\Delta^2]\!]$. If $\theta^1(X) = \theta^2(X)$ for all $X \in \mathsf{FV}(F)$, then $[\![\mathcal{D}^1]\!]_{\theta^1} = [\![\mathcal{D}^2]\!]_{\theta^2}$.*

*Proof.* By induction on $V$. This is easy, since both derivations are deterministic.

*Case* $V = X\,\vec{V}$. For $j = 1, 2$ the derivation $\mathcal{D}^j$ has the shape

$$\frac{(X : \vec{p}\vec{\kappa} \to \kappa) \in \Delta^j \qquad \begin{matrix}\mathcal{D}_i^j \\ p_i^{-1}\Delta^j \vdash V_i : \kappa_i \quad \text{for } i = 1, \ldots, |\vec{\kappa}|\end{matrix}}{\Delta^j \vdash X\,\vec{V} : \kappa}.$$

For all $i$ we have $\mathcal{G}_i := [\![\mathcal{D}_i^1]\!]_{\theta^1} = [\![\mathcal{D}_i^2]\!]_{\theta^2}$ by induction hypothesis. Since $\mathcal{F} := \theta^1(X) = \theta^2(X)$ by assumption, it follows that $[\![\mathcal{D}^1]\!]_{\theta^1} = \mathcal{F}\,\vec{\mathcal{G}} = [\![\mathcal{D}^2]\!]_{\theta^2}$.

*Case* $V = C\,\vec{V}$. Analogously.

*Case* $V = \lambda X V'$. For $j = 1, 2$ the derivation $\mathcal{D}^j$ has the shape

$$\frac{\begin{matrix}\mathcal{E}^j \\ \Delta^j, X : p\kappa_1 \vdash V' : \kappa_2\end{matrix}}{\Delta^j \vdash \lambda X V' : p\kappa_1 \to \kappa_2}.$$

Let $\mathcal{F}^j(\mathcal{G}) := [\![\mathcal{E}^j]\!]_{\theta^j[X \mapsto \mathcal{G}]}$. Since by induction hypothesis $\mathcal{F}^1(\mathcal{G}) = \mathcal{F}^2(\mathcal{G})$ for all $\mathcal{G} \in [\![\kappa_1]\!]$, we have $[\![\mathcal{D}^1]\!]_{\theta^1} = \mathcal{F}^1 = \mathcal{F}^2 = [\![\mathcal{D}^2]\!]_{\theta^2}$. □

**Lemma 2.23 (Substitution)** *Let $\mathcal{D}_1 :: \Delta, X : p\kappa \vdash F : \kappa'$ and $\mathcal{D}_2 :: p^{-1}\Delta \vdash G : \kappa$. Then there exists a derivation $\mathcal{E} :: \Delta \vdash [G/X]F : \kappa'$. Further, assume $\theta \in [\![\Delta]\!]$ and let $\mathcal{F}(\mathcal{G}) := [\![D_1]\!]_{\theta[X \mapsto \mathcal{G}]}$ and $\mathcal{G} := [\![\mathcal{D}_2]\!]_\theta$. Then $[\![\mathcal{E}]\!]_\theta = \mathcal{F}(\mathcal{G})$.*

*Proof.* By induction on $\mathcal{D}_1$. □

**Lemma 2.24 (Subject reduction)** *Let $\theta \in [\![\Delta]\!]$. If $\mathcal{D} :: \Delta \vdash F : \kappa$ and $F \longrightarrow_\beta F'$ then exists a derivation $\mathcal{E} :: \Delta \vdash F' : \kappa$ with $[\![\mathcal{E}]\!]_\theta = [\![\mathcal{D}]\!]_\theta$.*

*Proof.* By induction on $F \longrightarrow_\beta F'$, using Lemma 2.23 in case of a $\beta$-contraction. □

**Corollary 2.25** *The lemma generalizes to multi-step reduction* $F \longrightarrow_\beta^* F'$.

**Theorem 2.26 (Derivation-independence of semantics)** *Assume two derivations* $\mathcal{D} :: \Delta \vdash F : \kappa$ *and* $\mathcal{D}' :: \Delta' \vdash F : \kappa$ *and two valuations* $\theta \in [\![\Delta]\!]$ *and* $\theta' \in [\![\Delta']\!]$. *If* $\theta(X) = \theta'(X)$ *for all* $X \in \mathsf{FV}(F)$, *then* $[\![\mathcal{D}]\!]_\theta = [\![\mathcal{D}']\!]_{\theta'}$.

*Proof.* By normalization of $\beta$-reduction $F \longrightarrow_\beta{}^* V$. The theorem follows from Lemma 2.22 and the last corollary. □

This result justifies the notation $[\![F]\!]_\theta^\kappa$ as shorthand for $[\![\mathcal{D}]\!]_\theta$ where $\mathcal{D} :: \Delta \vdash F : \kappa$ for some $\Delta$ such that $\theta \in [\![\Delta]\!]$. If $\kappa$ is clear form the context of discourse, we will omit it.

**Theorem 2.27 (Soundness of constructor equality and subtyping)**

1. *If* $\mathcal{D} :: \Delta \vdash F = F' : \kappa$ *and* $\theta \sqsubseteq^p \theta' \in [\![\Delta]\!]$, *then* $[\![F]\!]_\theta \sqsubseteq^p [\![F]\!]_{\theta'} \in [\![\kappa]\!]$.
2. *If* $\mathcal{D} :: \Delta \vdash F \leq F' : \kappa$ *and* $\theta \sqsubseteq \theta' \in [\![\Delta]\!]$, *then* $[\![F]\!]_\theta \sqsubseteq [\![F]\!]_{\theta'} \in [\![\kappa]\!]$.

*Proof.* Simultaneously by induction on $\mathcal{D}$. □

# Chapter 3

# Type-Based Termination

In this chapter, we present the term or program level of $\mathsf{F}_{\widehat{\omega}}$. We introduce rules for typing and reduction and show their soundness through a term model. As a consequence, each program of $\mathsf{F}_{\widehat{\omega}}$ is terminating, and even strongly normalizing.

## 3.1 Specification

In this section, we introduce the terms of $\mathsf{F}_{\widehat{\omega}}$ with their typing and reduction rules.

### 3.1.1 The Language

**Terms.** The term language we use in this chapter is quite simple, it consists of the Curry-style $\lambda$-calculus with two special constants for recursion and corecursion.

| | | | | |
|---|---|---|---|---|
| $\mathsf{Tm}$ | $\ni r, s, t$ | $::=$ | $x \mid \lambda x t \mid r\, s$ | $\lambda$-calculus |
| | | $\mid$ | $c$ | constant |
| $\mathsf{Const}$ | $\ni c$ | $::=$ | $\mathsf{fix}_n^{\mu}$ | funct. def. by recursion on $n+1$st argument |
| | | $\mid$ | $\mathsf{fix}_n^{\nu}$ | corecursive function with $n$ arguments |

We use $\mathsf{fix}_n^{\nabla}$ to denote either $\mathsf{fix}_n^{\mu}$ or $\mathsf{fix}_n^{\nu}$. The default for subscript $n$ is 0, hence, $\mathsf{fix}^{\mu}$ denotes recursion on the first argument, and $\mathsf{fix}^{\nu}$ means the construction of an infinite object.

In spite of its economic spirit, the untyped $\lambda$-calculus can express all computable functions: it is Turing-complete. We heavily restrict its power through typing, but it is well-known that already in the simply-typed $\lambda$-calculus, many data structures can be simulated through continuation passing.

**Example 3.1 (Pairs and Variants)** Pairs with projections and variants (elements of a disjoint sum) with case distinction can be implemented as follows:

| | | | |
|---|---|---|---|
| $\mathsf{K}$ | $:=$ | $\lambda x \lambda y x$ | constant function constructor |
| $\mathsf{K}'$ | $:=$ | $\lambda x \lambda y y$ | identity function constructor |
| $\mathsf{pair}$ | $:=$ | $\lambda x \lambda y.\, \lambda k.\, k\, x\, y$ | constructor |
| $\mathsf{fst}$ | $:=$ | $\lambda p.\, p\, \mathsf{K}$ | first |
| $\mathsf{snd}$ | $:=$ | $\lambda p.\, p\, \mathsf{K}'$ | second projection |
| $\mathsf{inl}$ | $:=$ | $\lambda x.\, \lambda k \lambda l.\, k\, x$ | left injection |
| $\mathsf{inr}$ | $:=$ | $\lambda y.\, \lambda k \lambda l.\, l\, y$ | right injection |
| $\mathsf{case}$ | $:=$ | $\lambda i \lambda k \lambda l.\, i\, k\, l$ | |

### 3.1.2 Typing

**Typing contexts.** We extend our notion of context: Now a context may not only assign kinds and polarities to type variables $X$, but also types to term variables $x$.

| | | | |
|---|---|---|---|
| $\Gamma$ | $::=$ | $\diamond$ | empty context |
| | $\mid$ | $\Gamma, X : p\kappa$ | extension by constructor variable |
| | $\mid$ | $\Gamma, x : A$ | extension by term variable |

The default polarity is *non-variant*; extension of a context by a non-variant type variable may be written $\Gamma, X : \kappa$ instead of $\Gamma, X : \circ\kappa$.

Typing contexts $\Gamma$ will also be used in kinding judgements like $\Gamma \vdash F : \kappa$, as defined in Section 2.1.4. In this case, the term variable declarations are irrelevant and should be considered absent.

**Well-formed typing contexts** $\Gamma$ cxt. Contexts for typing must contain only non-variant type variable declarations; and each term variable must be declared with a well-formed type.

$$\textsc{cxt-empty}\ \frac{}{\diamond\ \mathsf{cxt}} \qquad \textsc{cxt-tyvar}\ \frac{\Gamma\ \mathsf{cxt}}{\Gamma, X : \circ\kappa\ \mathsf{cxt}}$$

$$\textsc{cxt-var}\ \frac{\Gamma\ \mathsf{cxt} \qquad \Gamma \vdash A : *}{\Gamma, x : A\ \mathsf{cxt}}$$

The restriction to non-variant type variables in rule CXT-TYVAR will be understood when the typing rules are given below. The only typing rule which introduces type variables is generalization, TY-GEN, and it only introduces non-variant variables. However, allowing all polarities in the *grammar* for typing contexts $\Gamma$ pays off when we pass from typing to kinding judgements $\Gamma \vdash F : \kappa$, because those derivations introduce also co- and contravariant type variables (and they do not impose a well-formedness criterion).

**Lemma 3.2** *If* $\mathcal{D} :: \Gamma\ \mathsf{cxt}$ *and* $(x : A) \in \Gamma$ *then* $\Gamma \vdash A : *$.

*Proof.* By induction on $\mathcal{D}$. □

**Notation.** Addition $a + n$ of a size expression $a : \mathsf{ord}$ and a natural number $n \in \mathbb{N}$.

$$\begin{array}{lcl} a + 0 & = & a \\ a + (n+1) & = & \mathsf{s}\, a + n \end{array}$$

**Typing** $\Gamma \vdash t : A$.

**Lambda-calculus.** These rules are standard:

$$\textsc{ty-var}\ \frac{(x{:}A) \in \Gamma \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash x : A} \qquad \textsc{ty-abs}\ \frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x t : A \to B}$$

$$\textsc{ty-app}\ \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B}$$

**Quantification.** Since $\forall_\kappa : (\kappa \xrightarrow{\circ} *) \xrightarrow{+} *$ (see Section 2.1.3), the constructor $F$ is of kind $\kappa \xrightarrow{\circ} *$ in the following rules:

$$\textsc{ty-gen}\ \frac{\Gamma, X{:}\kappa \vdash t : F\, X}{\Gamma \vdash t : \forall_\kappa F}\ X \notin \mathsf{FV}(F) \qquad \textsc{ty-inst}\ \frac{\Gamma \vdash t\ :\ \forall_\kappa F \qquad \Gamma \vdash G : \kappa}{\Gamma \vdash t : F\, G}$$

For rule TY-INST note that since $\Gamma \vdash F : \kappa \xrightarrow{\circ} *$, the application $F\, G$ is only well-kinded in $\Gamma$ if the context for $G : \kappa$ mentions only non-variant variables (see kinding rule KIND-APP$\circ$ in Section 2.1.4). But this is the case, since $\Gamma$ is well-formed (see Lemma 3.4).

**Subsumption.** (Subtyping has been defined in Section 2.2.)

$$\textsc{ty-sub}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leq B : *}{\Gamma \vdash t : B}$$

**Folding and unfolding** for (co)inductive types ($\nabla \in \{\mu, \nu\}$).

$$\textsc{ty-fold}\ \frac{\Gamma \vdash t : F\,(\nabla_\kappa\, a\, F)\,\vec{G}}{\Gamma \vdash t : \nabla_\kappa\,(a+1)\,F\,\vec{G}} \qquad \textsc{ty-unfold}\ \frac{\Gamma \vdash r : \nabla_\kappa\,(a+1)\,F\,\vec{G}}{\Gamma \vdash r : F\,(\nabla_\kappa\, a\, F)\,\vec{G}}$$

**Recursion** ($\nabla = \mu$) and corecursion ($\nabla = \nu$).

$$\textsc{ty-rec}\ \frac{\Gamma \vdash A\ \mathsf{fix}_n^\nabla\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord} \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash \mathsf{fix}_n^\nabla : (\forall \iota{:}\mathsf{ord}.\, A\,\iota \to A\,(\iota+1)) \to A\, a}$$

The condition $A\ \mathsf{fix}_n^\nabla\text{-}\mathsf{adm}$ ensures that we only use TY-REC to define recursive ($\nabla = \mu$) or corecursive ($\nabla = \nu$) functions. Dropping the condition

we could immediately introduce non-terminating programs, e. g., for $A(\iota) = \mathsf{Nat}^\infty \to \mathsf{Nat}^\infty$ which does not depend on the size index, the diverging program $\mathsf{fix}_0^\mu\, \lambda f \lambda x.\, \mathsf{succ}\,(f\, x)$ would be accepted.

For $A$ $\mathsf{fix}_n^\mu$-$\mathsf{adm}$ we say that *$A$ is admissible for recursion on the $n+1$st argument*. Similarly, $A$ $\mathsf{fix}_n^\nu$-$\mathsf{adm}$ is pronounced *$A$ is admissible for corecursion with $n$ arguments*. A definition of these predicates is given below.

**Example 3.3 (Typing for Pairs and Variants)** The following type assignments show that the constructors and destructors given in Example 3.1 introduce and eliminate Cartesian product and disjoint sum type as presented in Example 2.5 (impredicative encodings).

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

**Lemma 3.4** *If $\mathcal{D} :: \Gamma \vdash t : A$ then $\Gamma$ cxt.*

*Proof.* By induction on $\mathcal{D}$. At the leaves (TY-VAR and TY-REC) of the typing derivation, $\Gamma$ cxt is checked, and no rule, read downwards, extends the context. □

**Lemma 3.5** *If $\mathcal{D} :: \Gamma \vdash t : A$ then $\Gamma \vdash A : *$.*

*Proof.* By induction on $\mathcal{D}$. In most rules the type of the conclusion is simply assembled from components of the type(s) of the premise(s). In rule TY-SUB, the premise $\Gamma \vdash A \leq B : *$ entails $\Gamma \vdash B : *$ (by Lemma 2.14). □

**Notation for size index.** We sometimes write the size index superscript, e. g., $\mu^\iota$ instead of $\mu\,\iota$, or $\nu^\iota$ instead of $\nu\,\iota$.

**Natural transformations.** Let $n \geq 0$. For constructors $F_1, \ldots F_n$ and $G$ with $\vec{F}, G : \vec{p}\vec{\kappa} \to *$, let

$$\vec{F} \Rightarrow G \;:\Longleftrightarrow\; \forall \vec{X} : \vec{\kappa}.\, F_1\, \vec{X} \to \ldots F_n\, \vec{X} \to G\, \vec{X}.$$

Also, we abbreviate $\lambda \vec{X}.\, F(H_1\, \vec{X}) \ldots (H_n\, \vec{X})$ by $F \circ \vec{H}$.

**Admissible types for recursion and corecursion** must meet certain requirements. For completeness of the typing system, we give the judgements $\Gamma \vdash A\ \mathsf{fix}_n^{\nabla}\text{-}\mathsf{adm}$ for $\nabla \in \{\mu, \nu\}$ here. We will motivate and repeat them in sections 3.5.1 and 3.5.2.

$$\begin{aligned}\Gamma \vdash A\ \mathsf{fix}_n^{\mu}\text{-}\mathsf{adm} \quad :\Longleftrightarrow \quad & \Gamma, \iota{:}\mathsf{ord} \vdash A\,\iota = (\vec{G}, (\mu^{\iota} F) \circ \vec{H} \Rightarrow G) : * \quad (\iota \notin \mathsf{FV}(A)) \\ & \text{for some } F, G, \vec{G}, \vec{H} \text{ with } |\vec{G}| = n \text{ and} \\ & \Gamma \vdash F : +\kappa \to \kappa \text{ for some pure } \kappa = \vec{p}\vec{\kappa} \to *, \\ & \Gamma, \iota{:}{+}\mathsf{ord} \vdash G : \kappa' \text{ for some } \kappa' = \circ\vec{\kappa}' \to *, \\ & \Gamma, \iota{:}{-}\mathsf{ord} \vdash G_i : \kappa' \text{ for } 1 \le i \le n, \text{ and} \\ & \Gamma \vdash H_i : \circ\vec{\kappa}' \to \kappa_i \text{ for } 1 \le i \le |\vec{\kappa}|.\end{aligned}$$

$$\begin{aligned}\Gamma \vdash A\ \mathsf{fix}_n^{\nu}\text{-}\mathsf{adm} \quad :\Longleftrightarrow \quad & \Gamma, \iota{:}\mathsf{ord} \vdash A\,\iota = (\vec{G} \Rightarrow (\nu^{\iota} F) \circ \vec{H}) : * \quad (\iota \notin \mathsf{FV}(A)) \\ & \text{for some } F, \vec{G}, \vec{H} \text{ with } |\vec{G}| = n \text{ and} \\ & \Gamma \vdash F : +\kappa \to \kappa \text{ for some pure } \kappa = \vec{p}\vec{\kappa} \to *, \\ & \Gamma, \iota{:}{-}\mathsf{ord} \vdash G_i : \kappa' \text{ (all } i\text{) for some } \kappa' = \circ\vec{\kappa}' \to *, \text{ and} \\ & \Gamma \vdash H_i : \circ\vec{\kappa}' \to \kappa_i \text{ for } 1 \le i \le |\vec{\kappa}|.\end{aligned}$$

**Example 3.6 (Admissible types for recursive functions)** These $A(\iota)$ are $\mathsf{fix}_n^{\mu}\text{-}\mathsf{adm}$ for some $n$.

| | |
|---|---|
| $\mathsf{Nat}^{\iota} \to \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\infty}$ | addition and multiplication |
| $\mathsf{Nat}^{\iota} \to \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota}$ | subtraction and division |
| $\mathsf{List}^{\iota} A \to \mathsf{List}^{\iota} B$ | list map |
| $\mathsf{List}^{\iota} A \to \mathsf{List}^{\iota} A \times \mathsf{List}^{\iota} A$ | list splitting |

The following types types are not admissible according to the current criterion, but will be admissible after a relaxation in Chapter 5.

| | |
|---|---|
| $\mathsf{Nat}^{\iota} \to \mathsf{Nat}^{\iota} \to \mathsf{Nat}^{\iota}$ | minimum and maximum |
| $\mathsf{List}^{\iota} A \to \mathsf{List}^{\iota} B \to \mathsf{List}^{\iota} C$ | list zip-with |

**Example 3.7 (Repeat function)** The term $\mathsf{repeat}\ a$ constructs an infinite stream of $a$s.

$$\begin{aligned}\mathsf{repeat}\ a &:= \mathsf{fix}_0^{\nu}\ \lambda \mathit{repeat}.\ \mathsf{pair}\ a\ \mathit{repeat} \\ \lambda a.\ \mathsf{repeat}\ a &: \forall A.\ A \to \mathsf{Stream}^{\infty} A\end{aligned}$$

It is well-typed, since $\lambda \mathit{repeat}.\ \mathsf{pair}\ a\ \mathit{repeat}$ can be assigned type $\forall \iota.\ \mathsf{Stream}^{\iota} A \to \mathsf{Stream}^{\iota+1} A$, and $\lambda \iota.\ \mathsf{Stream}^{\iota} A$ is trivially $\mathsf{fix}_0^{\nu}\text{-}\mathsf{adm}$ (empty $\vec{G}$, $\vec{H}$).

### 3.1.3 Operational Semantics

In the following, we give the operational semantics of $\mathsf{F}_{\widehat{\omega}}$. It is clear that, in order to obtain a strongly normalizing calculus, the unrolling of recursion has to be restricted. It is safe to unroll a recursive function if all of its arguments

are values, however, it is sufficient if the *recursive argument* is a value (i. e., the argument which decreases in the recursive call). In $\mathsf{F}_\omega^{\widehat{}}$, the number of the recursive argument is associated with the index $n$ of the recursion operator. A candidate for the unrolling rule is

$$\mathsf{fix}_n^\mu\, s\, t_{1..n}\, (\lambda x r) \longrightarrow s\, (\mathsf{fix}_n^\mu\, s)\, t_{1..n}\, (\lambda x r),$$

since a $\lambda$-abstraction is the canonical value in $\mathsf{F}_\omega^{\widehat{}}$. Considering this for another moment, we see that recursive and corecursive functions must also count as values, since they are not eagerly unrolled, and under-applied constants, such as a bare $\mathsf{fix}_m^\mu$, should also be included. Anyhow, when we extend $\mathsf{F}_\omega^{\widehat{}}$ with more primitive type constructors, more values will arise. So we trade the rule for the following:

$$\mathsf{fix}_n^\mu\, s\, t_{1..n}\, v \longrightarrow s\, (\mathsf{fix}_n^\mu\, s)\, t_{1..n}\, v.$$

What about corecursion? A corecursive function should also only be unrolled on demand. Could we use the same rule for $\mathsf{fix}_n^\nu$—which would make the distinction between $\mathsf{fix}^\mu$ and $\mathsf{fix}^\nu$ superfluous? Actually, the above rule is too strict for corecursion, it prevents sensible reductions. Consider, the type of Burroni conatural numbers[1] and their mapping function:

$$\begin{array}{lcl}
\mathsf{Bur} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Bur} & := & \lambda\iota\lambda A.\, \nu^\iota \lambda X.\, A + X \\
 & = & \lambda\iota\lambda A.\, \nu^\iota \lambda X \forall C.\, (A \to C) \to (X \to C) \to C \\
 & & \\
\mathsf{bmap} & : & \forall A \forall B.\, (A \to B) \to \forall \iota.\, \mathsf{Bur}^\iota\, A \to \mathsf{Bur}^\iota\, B \\
\mathsf{bmap} & := & \lambda f.\ \mathsf{fix}_1^\nu\, \lambda bmap \lambda n.\, \mathsf{case}\, n\, (\lambda a.\, \mathsf{inl}\, (f\, a))\, (\lambda m.\, \mathsf{inr}\, (bmap\ m)) \\
 & = & \lambda f.\ \mathsf{fix}_1^\nu\, \lambda bmap \lambda n.\, n\, (\lambda a.\, \lambda g \lambda h.\, g\, (f\, a))\, (\lambda m.\, \lambda g \lambda h.\, h\, (bmap\ m))
\end{array}$$

Now we expect $\mathsf{case}\, (\mathsf{bmap}\, f\, (\mathsf{inl}\, a))\, g\, h$ to reduce to $g\, (f\, a)$ even for variables $f$, $g$, and $h$. Expanding case and inl this simplifies to

$$(\mathsf{fix}_1^\nu\, (\lambda bmap \ldots)\, (\lambda g' \lambda h'.\, g'\, a))\, g\, h.$$

Since variable $g$ is not a value, this term would be stuck with the evaluation rule for $\mathsf{fix}^\mu$. Unrolling *on demand* means for corecursive elements: unrolling in any evaluation context. This can be application to a variable like $g$, or being applied to a recursive function.

It is also clear that a corecursive function cannot be unrolled before all of its arguments have been supplied and the result is demanded. For instance, we cannot allow a reduction rule like

$$e\, (\mathsf{fix}_n^\nu\, s\, t_{1..m}) \longrightarrow e\, (s\, (\mathsf{fix}_n^\nu\, s)\, t_{1..m}) \qquad \text{where } m \leq n.$$

[1] Capretta [Cap05] interprets $\mathsf{Bur}^\infty A$ as computations of type $A$: Either it is a value of type $A$ (inl) or it is a "step" containing a computation of type $A$ (inr). "Step" could be read as "hello, I know you are waiting for the result, I just want to tell you that I am still alive and busy with the computation". A diverging computation is an infinite sequence of steps ($\mathsf{fix}_0^\nu$ inr).

This rule leads to divergence of well-typed terms if $m < n$. For example, consider the coinductive type of Bool-hungry functions $\nu^\iota \lambda X.\, \mathsf{Bool} \to X$ and the function (communicated to me by Tarmo Uustalu during the APPSEM meeting in Nottingham 2003)

$$\begin{array}{lll} \mathsf{veryHungry} & : & \forall \iota.\, \mathsf{Bool} \to \nu^\iota \lambda X.\, \mathsf{Bool} \to X \\ \mathsf{veryHungry} & := & \mathsf{fix}_1^\gamma\, \lambda \mathit{veryHungry} \lambda b \lambda b'.\, \mathit{veryHungry}\ \mathsf{true}. \end{array}$$

Then $\mathsf{veryHungry}\,\mathsf{true}$ would reduce in several steps to $\lambda b'.\, \mathsf{veryHungry}\,\mathsf{true}$ and, hence, diverge. But the above reduction rule is sound for $m = n$. This we will show in the remainder of the chapter.

**Evaluation frames and contexts.**

$$\begin{array}{llllll} \mathsf{Eframe} & \ni & e & ::= & \_\, s & \text{application} \\ & & & \mid & \mathsf{fix}_n^\mu\, s\, t_{1..n}\, \_ & \text{recursive function call} \\ \\ \mathsf{Ecxt} & \ni & E & ::= & \mathsf{Id} & \text{empty stack} \\ & & & \mid & E \circ e & \text{push frame} \end{array}$$

Frames and contexts can be interpreted as endo-functions on terms in the obvious way, i.e.,

$$\begin{array}{rcl} (\_\, s)(r) & := & r\, s \\ (\mathsf{fix}_n^\mu\, s\, \vec{t}\, \_)(r) & := & \mathsf{fix}_n^\mu\, s\, \vec{t}\, r \\ \mathsf{Id}(r) & := & r \\ (E \circ e)(r) & := & E(e(r)) \end{array}$$

Hence, by abuse of notation, $\mathsf{Eframe}, \mathsf{Ecxt} \subseteq \mathsf{Tm} \to \mathsf{Tm}$.

The *head* of a term $t$ is the shortest term $r$ such that $E(r) = t$ for some evaluation context $E$. For example if $t = \mathsf{fix}_n^\mu\, s\, t_{1..n}\, ((\lambda x r')\, s')$, then $\lambda x r'$ is the head of $t$.

**(Lazy) Values.** While a *normal form* is a term which cannot be reduced further, we will refer to a *canonical form* as a *value*.

$$\begin{array}{llll} \mathsf{Val} \ni v & ::= & \lambda x t & \\ & \mid & \mathsf{fix}_n^\nabla & \\ & \mid & \mathsf{fix}_n^\nabla\, s\, \vec{t} & \text{where } 0 \le |\vec{t}| \le n \end{array}$$

**Reduction.** The *contraction* relation $t \rightarrowtail t'$ is given by the following axiom schemata:

$$\begin{array}{llcll} \text{RED-}\beta & (\lambda x t)\, s & \rightarrowtail & [s/x]t & \\ \text{RED-REC} & \mathsf{fix}_n^\mu\, s\, t_{1..n}\, v & \rightarrowtail & s\, (\mathsf{fix}_n^\mu\, s)\, t_{1..n}\, v & \text{if } v \neq \mathsf{fix}_{n'}^\nu\, s'\, t_{1..n'} \\ \text{RED-COREC} & e(\mathsf{fix}_n^\gamma\, s\, t_{1..n}) & \rightarrowtail & e(s\, (\mathsf{fix}_n^\gamma\, s)\, t_{1..n}) & \text{if } e \neq \mathsf{fix}_{n'}^\mu\, s'\, t_{1..n'}\, \_ \end{array}$$

Furthermore, we define the *one-step reduction* relation $\longrightarrow$ as closure of $\rightarrowtail$ under all term constructors, $\longrightarrow^+$ as the transitive closure of $\longrightarrow$ and $\longrightarrow^*$ as the reflexive-transitive closure.

**Remark 3.8 (On the choice of RED-REC)** Why have we restricted the unfolding of recursive functions to call-by-value? First, note that the completely unrestricted unrolling of fixed points, $\mathsf{fix}^\mu_n\ s \longrightarrow s\,(\mathsf{fix}^\mu_n\ s)$, gives immediately rise to infinite reduction sequences. Secondly, we need recursive functions to be evaluation contexts, i. e., a recursive function applied to a variable may not unfold the function definition. Otherwise, each function definition which contains a recursive call will immediately loop under full reduction.

**Remark 3.9 (Confluence)** If we omit the side conditions in the fixed-point unrolling rules RED-REC and RED-COREC, then reduction would not be confluent, not even locally confluent for closed terms. Critical pairs would arise when in RED-REC, the value $v$ was a corecursive function, or in RED-COREC, the evaluation frame $e$ was a recursive function; in this case, both rules could fire. For instance, let $s = \lambda\_\lambda xx$, $f = \mathsf{fix}^\mu_0\ s$, and $v = \mathsf{fix}^\nu_0\ s$. On one hand, $f\,v \longrightarrow s\,f\,v$, whose only reducts are $(\lambda xx)\,v$ and $v$. On the other hand, $f\,v \longrightarrow f\,(s\,v)$ which reduces further only to $f\,\lambda xx$, $s\,f\,\lambda xx$, $(\lambda xx)\,\lambda xx$ and $\lambda xx$. The deeper reason for non-confluence is that we cannot unfold all fixed-points, since we want to ensure normalization.

In order to overcome this defect, we have inserted restrictions that remove the critical pairs, make reduction confluent and weak head reduction deterministic.

**Remark 3.10 (Why prevent both reductions?)** For a term with clashing fixed points, e. g., $t := \mathsf{fix}^\mu_0\ s\,(\mathsf{fix}^\nu_0\ s')$, we currently allow *no* contraction. As we have seen in the previous remark, allowing both fixed-point reductions would lead to non-confluence, and it would even break a crucial property of strongly normalizing terms: they are closed under weak head expansion[2] (see Remark 3.27). But could we not give priority to one kind of fixed-points and keep *one* reduction of the two? This would break symmetry, but does it do any harm? Yes, as we will see in the proofs of lemmata 3.32 and 3.37. The term $t$ would not be considered neutral; removing the reduction of $\mathsf{fix}^\mu$ would break the first lemma (3.32), and removing the reduction of $\mathsf{fix}^\nu$ would break the second one (3.37).

For system $\mathsf{F}^{\widehat{\omega}}$, we are only interested in strong normalization. We will present a better behaved, iso-(co)inductive, system in Section 4.1 and show that its normalization can be inherited from $\mathsf{F}^{\widehat{\omega}}$.

**Example 3.11 (Reduction for repeat)** Let $A$ be a type and $a : A$. Recall that

[2]Here, we mean a restricted form of weak head expansion, which usually preserves strong normalization. E.g., $[s/x]t$ only expands to $(\lambda xt)\,s$ if $s$ is strongly normalizing.

$\mathsf{repeat}\, a = \mathsf{fix}_0^\nu(\lambda \mathit{repeat}.\, \mathsf{pair}\, a\, \mathit{repeat})$. We have the following reduction sequences

$$\begin{array}{lll}
\mathsf{fst}\,(\mathsf{repeat}\, a) & \longrightarrow & (\mathsf{repeat}\, a)\,\mathsf{K} \\
& \longrightarrow & (\lambda \mathit{repeat}.\, \mathsf{pair}\, a\, \mathit{repeat})\,(\mathsf{repeat}\, a)\,\mathsf{K} \\
& \longrightarrow & \mathsf{pair}\, a\,(\mathsf{repeat}\, a)\,\mathsf{K} \\
& \longrightarrow^3 & \mathsf{K}\, a\,(\mathsf{repeat}\, a) \\
& \longrightarrow^2 & a \\
\\
\mathsf{snd}\,(\mathsf{repeat}\, a) & \longrightarrow^+ & \mathsf{K}'\, a\,(\mathsf{repeat}\, a) \\
& \longrightarrow & \mathsf{repeat}\, a
\end{array}$$

Note that $\mathsf{repeat}\, a$ does neither reduce by itself, nor as argument to a non-recursive function like pair, K, etc. (lines 2–4). It only reduces if it is applied to something (line 1). This fixed-point unfolding *on demand* is a generalization of the strategy of Amadio and Coupet-Grimal [ACG98], Giménez [Gim98], and Barthe et al. [BFG$^+$04], who unfold corecursive definition only under case distinction.

Although there are variations on the order of reductions, one will find that there is no possibility that the fixed-point $\mathsf{repeat}\, a$ is unfolded more than once. Each unfolding "eats" one destructor (like fst or snd), which guarantees termination.

**Lemma 3.12 (Substitution)** *Reduction is closed under substitution.*

1. *If* $t \longrightarrow t'$ *then* $[s/x]t \longrightarrow [s/x]t'$.
2. *If* $s \longrightarrow s'$ *then* $[s/x]t \longrightarrow^* [s'/x]t$.

*Proof.* The first by induction on $t \longrightarrow t'$, the second by induction on $t$. □

**Example 3.13 (Types not admissible for recursion)** There are types which are OK in the domain-theoretic semantics of Hughes, Pareto, and Sabry [HPS96], but not in our current reduction semantics:

| | |
|---|---|
| $\mathsf{Nat}^\iota \times \mathsf{Nat}^\infty \to \mathsf{Nat}^\infty$ | addition and multiplication (uncurried) |
| $\mathsf{List}^\iota A \times \mathsf{List}^\iota B \to \mathsf{List}^\iota C$ | list zip-with (uncurried) |

We discuss modification of our semantics in Section 7.2.

But these types are not admissible and lead to undefined recursive functions.

| | |
|---|---|
| $\mathsf{List}^\infty(\mathsf{Nat}^\iota) \to C$ | admits function $\mathsf{fix}_0^\mu\, \lambda f \lambda\_.\, f\, \mathsf{nil}$ |
| $\mathsf{Nat}^\iota \to (\mathsf{Nat} \to \mathsf{Nat}^\iota) \to C$ | see Section 5.1 |
| $(\mathsf{Nat} \to \mathsf{Nat}^\iota) \to C$ | ditto |

## 3.2 Examples

To flesh out the definition of $\mathsf{F}_{\widehat{\omega}}$, we present some simple examples in this section. More examples will be given in Chapter 6. If clear from the context, we will write Nat for $\mathsf{Nat}^\infty$ to denote the type of all natural numbers.

### 3.2.1 Fibonacci Numbers

A basic exercise in lazy functional programming is to code up the stream of Fibonacci numbers in one line, assuming we have the standard library function

$$\begin{array}{lcl} \mathsf{zipWith} & : & \forall A \forall B \forall C.\, (A \to B \to C) \to \\ & & \forall \iota.\, \mathsf{Stream}^{\iota} A \to \mathsf{Stream}^{\iota} B \to \mathsf{Stream}^{\iota} C \\ \mathsf{zipWith} & := & \lambda f.\ \mathsf{fix}_2^{\gamma}\, \lambda zip \lambda s \lambda t.\ \mathsf{pair}\ (f\ (\mathsf{fst}\, s)\ (\mathsf{fst}\, t))\ (zip\ (\mathsf{snd}\, s)\ (\mathsf{snd}\, t)). \end{array}$$

Note that the type of zipWith is admissible for corecursion, since $\iota$ appears negatively in $\mathsf{Stream}^{\iota} A$ and $\mathsf{Stream}^{\iota} B$. In $\mathsf{F}_{\widehat{\omega}}$, the Fibonacci stream can be defined as follows (assuming $0, 1 : \mathsf{Nat}$ and $+ : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$).

$$\begin{array}{lcl} \mathsf{fib} & : & \mathsf{Stream}^{\infty} \mathsf{Nat} \\ \mathsf{fib} & := & \mathsf{pair}\ 0\ (\mathsf{fix}_0^{\gamma}\, \lambda zip.\ \mathsf{pair}\ 1\ (\mathsf{zipWith}\ (+)\ zip\ (\mathsf{pair}\ 0\ zip))). \end{array}$$

The body of the corecursive definition is well-typed. We assign the following types:

$$\begin{array}{lcl} zip & : & \mathsf{Stream}^{\iota}\, \mathsf{Nat} \\ \mathsf{pair}\ 0\ zip & : & \mathsf{Stream}^{\iota+1}\, \mathsf{Nat} \leq \mathsf{Stream}^{\iota}\, \mathsf{Nat} \\ (\mathsf{zipWith}\ \ldots) & : & \mathsf{Stream}^{\iota}\, \mathsf{Nat} \\ \mathsf{pair}\ 1\ (\mathsf{zipWith}\ \ldots) & : & \mathsf{Stream}^{\iota+1}\, \mathsf{Nat} \end{array}$$

It is instructive to understand why the alternative definition of the Fibonacci stream as

$$\mathsf{fix}_0^{\gamma}\, \lambda fib.\ \mathsf{pair}\ 0\ (\mathsf{pair}\ 1\ (\mathsf{zipWith}\ (+)\ (\mathsf{snd}\, fib)\ fib)$$

is not well-typed in $\mathsf{F}_{\widehat{\omega}}$.

### 3.2.2 Co-Natural Numbers

Non-standard natural numbers are given by the type $\mathsf{CoNat} = \nu^{\infty} \lambda X.\, \mathbf{1} + X$. We can define all natural numbers plus infinity:

$$\begin{array}{lcl} \overline{0} & := & \mathsf{inl}() \\ \overline{1} & := & \mathsf{inr}\, \overline{0} \\ \overline{2} & := & \mathsf{inr}\, \overline{1} \\ \vdots & & \\ \overline{\omega} & := & \mathsf{fix}_0^{\gamma}\ \mathsf{inr} \end{array}$$

All these values inhabit CoNat.

### 3.2.3 Some Pathological Cases

Datatypes like $\mathsf{Empty}^{a} := \mu_{*}^{a} \lambda XX$ are called *unguarded* since their unfolding does not produce a "real" data type constructor like $+$, $\times$, or $\to$, which would enable production of inhabitants. Since $[\![\mathsf{Empty}]\!](\alpha + 1) = [\![\mathsf{Empty}]\!](\alpha)$, we have

$[\![\mathsf{Empty}]\!] = \bot^*$. It is easy to check that $\mathsf{fix}_0^\mu\, \lambda x x : \forall C \forall \iota.\, \mathsf{Empty}^\iota \to C$, so the Empty type can be eliminated through a recursive function. A bit counterintuitively, this function, which is the primary example of a looping definition in functional programming, is termination on all inputs—simply because we can never construct a value of $\mathsf{Empty}^a$.

Dually, the unguarded coinductive type $\mathsf{Unit}^a := \nu_*^a \lambda X X$ is inhabited by $\mathsf{fix}_0^\nu\, \lambda x x$. This function is normalizing as well because there is no well-typed evaluation context for elements of type $\mathsf{Unit}^a$.

### 3.2.4 Huffman Trees

Huffman codes are space-optimal and prefix-free. To decode a Huffman-coded bit stream, one uses a Huffman tree, i. e., a binary tree whose leaves are labeled by the characters whose code is the path to their leaf. In Haskell, we can implement the decoding function as follows.[3]

```
data Huffman b c = Leaf c
                 | Node (b -> Huffman b c)

decode :: Huffman b c -> [b] -> [c]
decode t0 = dec0
    where dec0 = dec1 t0
              where dec1 (Leaf c) bs     = c : dec0 bs
                    dec1 (Node f) (b:bs) = dec1 (f b) bs
```

There are two subroutines: the corecursive `dec0` of type `[b] -> [c]` decodes a stream starting with the full Huffman-tree `t0`, whereas `dec1`, of the same type as `decode`, is defined by recursion on subtrees of `t0`. Observe that the recursive call to `dec0` is *guarded* by the stream constructor `(:)`, although not according to Coquand's conditions [Coq93], because it is inside another fixed point.

In $\mathsf{F}_{\widehat{\omega}}$, we can define the type $\mathsf{Huffman}^a\, B\, C$ of $B$-branching $C$-leaf labeled trees of height $< a$ as follows:

$$
\begin{array}{lcl}
\mathsf{Huffman} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{-} * \xrightarrow{+} * \\
\mathsf{Huffman} & := & \lambda \iota \lambda B \lambda C.\, \mu_*^\iota \lambda X.\, C + (B \to X) \\
\\
\mathsf{leaf} & : & \forall B \forall C \forall \iota.\, C \to \mathsf{Huffman}^{\iota+1}\, B\, C \\
\mathsf{leaf} & := & \mathsf{inl} \\
\mathsf{node} & : & \forall B \forall C \forall \iota.\, (B \to \mathsf{Huffman}^\iota\, B\, C) \to \mathsf{Huffman}^{\iota+1}\, B\, C \\
\mathsf{node} & := & \mathsf{inr}
\end{array}
$$

[3] Yong Luo observed that `decode (Leaf c) []` produces an infinite stream of `c`s. This shows that `decode` is not necessarily terminating on finite input, although it is, as we will prove in the following, productive. One could ask whether `decode`'s behavior on the singleton Huffman tree `Leaf c` makes sense. I think "yes", because an infinite text over an alphabet with a single character contains no information. Regardless what the input stream of Huffman codes is, the output will just consist of `c`s. In this case, we do not even have to look at the input.

The decoding function takes a Huffman-tree $t$ and a $B$-stream $s$. If the $t$ is a node $\mathsf{inr}\, f$, we take the first element $b$ from the stream and continue with the $b$-th subtree of $t$, i. e., $f\, b$, and the remaining stream. If $t$ is a leaf $\mathsf{inl}\, c$, we can output $c$ and continue with the original Huffman tree $t_0$.

$$\begin{array}{lcl}
\mathsf{decode} & : & \forall B \forall C.\, \mathsf{Huffman}^\infty\, B\, C \to \mathsf{Stream}^\infty\, B \to \mathsf{Stream}^\infty\, C \\
\mathsf{decode} & := & \lambda t_0.\ \mathsf{fix}_1^\gamma\, \lambda dec_0. \\
 & & \quad \big(\mathsf{fix}_0^\mu\, \lambda dec_1.\lambda t \lambda s.\, \mathsf{case}\, t \\
 & & \quad\ (\lambda c.\, \mathsf{pair}\, c\, (dec_0\, s)) \\
 & & \quad\ (\lambda f.\, dec_1\, (f\, (\mathsf{fst}\, s))\, (\mathsf{snd}\, s))\big)\, t_0
\end{array}$$

This code is a close translation of the Haskell program, one clearly sees the outer corecursion ($\mathsf{fix}_1^\gamma\, \lambda dec_0$) and the inner recursion ($\mathsf{fix}_0^\mu\, \lambda dec_1$). We can assign the following types to variables and subterms:

$$\begin{array}{lcl}
t_0 & : & \mathsf{Huffman}^\infty\, B\, C \\
dec_0 & : & \mathsf{Stream}^\infty\, B \to \mathsf{Stream}^\iota\, C \\
dec_1 & : & \mathsf{Huffman}^\jmath\, B\, C \to \mathsf{Stream}^\infty\, B \to \mathsf{Stream}^{\iota+1}\, C \\
t & : & \mathsf{Huffman}^{\jmath+1}\, B\, C \\
s & : & \mathsf{Stream}^\infty\, B \\
c & : & C \\
\mathsf{pair}\, c\, (dec_0\, s) & : & \mathsf{Stream}^{\iota+1}\, C \\
f & : & B \to \mathsf{Huffman}^\jmath\, B\, C \\
f\, (\mathsf{fst}\, s) & : & \mathsf{Huffman}^\jmath\, B\, C \\
dec_1\, (f\, (\mathsf{fst}\, s))\, (\mathsf{snd}\, s) & : & \mathsf{Stream}^{\iota+1}\, C \\
\big(\mathsf{fix}_0^\mu \ldots\big) & : & \mathsf{Huffman}^\infty\, B\, C \to \mathsf{Stream}^\infty\, B \to \mathsf{Stream}^{\iota+1}\, C
\end{array}$$

The type of $dec_0$ is admissible for corecursion with one argument, $\mathsf{fix}_1^\gamma$-adm, since it is a function type into a coinductive type $\nu^\iota \ldots$, and the domain is monotone in the size variable $\iota$, since it does not mention it. Similarly, the type of $dec_1$ is admissible for recursion on the first argument, $\mathsf{fix}_0^\mu$-adm, since its domain is an inductive type and its codomain is trivially monotone in the size variable $\jmath$.

This example could be handled in the calculus of Barthe et al. [BFG+04] and in my TLCA system [Abe03], but not in the system of Hughes, Pareto, and Sabry [HPS96, Par00], since they forbid function spaces in data types.

In the following, we allow ourselves some syntactic sugar and write the case-expression as pattern matching:

$$\begin{array}{l}
\mathsf{match}\ t\ \mathsf{with} \\
\quad \mathsf{leaf}\, c \quad \mapsto \quad \mathsf{pair} \ldots \\
\quad \mathsf{node}\, f \quad \mapsto \quad dec_1 \ldots
\end{array}$$

**Creating a Huffman tree.** Huffman [Huf52] presented an algorithm how to create a optimal prefix-free binary code for a set of characters from their relative frequencies. This algorithm can be coded directly in $\mathsf{F}_{\widehat{\omega}}$. For simplicity,

assume that the characters and their frequencies are given as a list of pairs $(n_1, \mathsf{leaf}\, c_1), ..., (n_k\, \mathsf{leaf}\, c_k)$ of natural numbers and singleton Huffman trees. Assume further, this list is sorted ascendingly by frequencies (if not, sort it first—sorting can also be coded in $\mathsf{F}_{\widehat{\omega}}$, as we will see later).

We abbreviate the type of frequency-tree pairs by WT. Two pairs can be joined by adding the frequencies and constructing a node with the two given trees as subtrees:

$$\begin{array}{lcl} \mathsf{WT} & : & * \\ \mathsf{WT} & := & \mathsf{Nat} \times \mathsf{Huffman}^{\infty}\ \mathsf{Bool}\ \mathsf{Char} \end{array}$$

$$\begin{array}{lcl} \mathsf{join} & : & \mathsf{WT} \to \mathsf{WT} \to \mathsf{WT} \\ \mathsf{join} & := & \lambda x \lambda y.\ \mathsf{pair}\ (\mathsf{fst}\, x + \mathsf{fst}\, y)\ (\lambda b.\ \mathsf{snd}\ (\mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ y)) \end{array}$$

Huffman's algorithm works as follows: Pick the pairs with the least frequencies, join them and solve the remaining problem. If we start with a sorted list, we have to insert the joined pair into the remaining list in order, before we recursively process the remainder. It is crucial that insert is defined in such a way that our type system can "see" that the extended list is only one element larger. But this is the case for the obvious recursive definition of insert, as for example present in the Haskell prelude. Here, we give a version slightly optimized for our purposes: it returns head and tail of the result list separately, in order to statically exclude the impossible case that an empty list is returned. (In the following, we consider frequency comparison $x \leq y$ for $x, y \in \mathsf{WT}$ to be already defined.)

$$\begin{array}{lcl} \mathsf{insert} & : & \forall \iota.\ \mathsf{WT} \to \mathsf{List}^{\iota}\ \mathsf{WT} \to \mathsf{WT} \times \mathsf{List}^{\iota}\ \mathsf{WT} \\ \mathsf{insert} & := & \mathsf{fix}_1^{\mu}\ \lambda \mathit{insert} \lambda x \lambda xs.\ \mathsf{match}\ xs\ \mathsf{with} \\ & & \quad \mathsf{inl}\ \_ \mapsto \mathsf{pair}\, x\, xs \\ & & \quad \mathsf{inr}\ (\mathsf{pair}\, y\, ys) \mapsto \mathsf{if}\ x \leq y\ \mathsf{then}\ \mathsf{pair}\, x\, xs \\ & & \qquad\qquad\qquad\ \ \mathsf{else}\ \mathsf{pair}\, y\ (\mathsf{inr}\ (\mathit{insert}\, x\, ys)) \end{array}$$

$$\begin{array}{lcl} \mathsf{huffman} & : & \forall \iota.\ \mathsf{WT} \to \mathsf{List}^{\iota}\ \mathsf{WT} \to \mathsf{Huffman}^{\infty}\ \mathsf{Bool}\ \mathsf{Char} \\ \mathsf{huffman} & := & \mathsf{fix}_1^{\mu}\ \lambda \mathit{huffman} \lambda x \lambda xs.\ \mathsf{match}\ xs\ \mathsf{with} \\ & & \quad \mathsf{nil} \mapsto \mathsf{snd}\, x \\ & & \quad \mathsf{cons}\, y\, ys \mapsto \mathsf{uncurry}\ \mathit{huffman}\ (\mathsf{insert}\ (\mathsf{join}\, x\, y)\ ys) \end{array}$$

Herein, uncurry is an auxiliary function defined as usual.

$$\begin{array}{lcl} \mathsf{uncurry} & : & \forall A \forall B \forall C.\ (A \to B \to C) \to A \times B \to C \\ \mathsf{uncurry} & := & \lambda f \lambda p.\ f\ (\mathsf{fst}\, p)\ (\mathsf{snd}\, p) \end{array}$$

Using the impredicative encoding $A \times B = \forall C.\ (A \to B \to C) \to C$ of products, we can state the last case without uncurry directly as

$$\mathsf{insert}\ (\mathsf{join}\, x\, y)\ ys\ \mathit{huffman}.$$

It is easy to see that the recursive call to *huffman* is justified. Some types are:

$$\begin{array}{lcl} xs & : & \mathsf{List}^{\iota}\,\mathsf{WT} \leq \mathsf{List}^{\iota+1}\,\mathsf{WT} \\ ys & : & \mathsf{List}^{\iota}\,\mathsf{WT} \\ \mathsf{insert}\ (\mathsf{join}\,x\,y)\ ys & : & \mathsf{WT} \times \mathsf{List}^{\iota}\,\mathsf{WT} \\ \mathsf{uncurry}\ \mathit{huffman} & : & \mathsf{WT} \times \mathsf{List}^{\iota}\,\mathsf{WT} \to \mathsf{Huffman}^{\infty}\,\mathsf{Bool}\,\mathsf{Char} \end{array}$$

Finally, Huffman *encoding* can also be coded in $\mathsf{F}_{\widehat{\omega}}$, although for this, we need another data structure, mapping characters to Huffman codes.

### 3.2.5 Prime Numbers

Productivity of certain streams depends on non-trivial mathematical results. For example, the stream of prime numbers is productive since there are infinitely many primes. If the converse was true, i. e., if there were only $n$ primes and $m$ the product of these primes, then $m+1$ would be relatively prime to all $n$ primes. This proof by Euclid guarantees a prime number between $n+1$ and $m+1$. We can exploit this fact to define a stream of prime numbers in $\mathsf{F}_{\widehat{\omega}}$. The definition is an adaption of Miculan and Gianantonio's function $p(m,n)$ [GM03, tenth page]:

$$p(m,n) = \begin{cases} n :: p(m*n, n+1) & \text{if } \gcd(m,n) = 1 \\ p(m, n+1) & \text{else} \end{cases}$$

If $\mathsf{F}_{\widehat{\omega}}$, we require a dummy argument $k$ to count down the maximum number of steps until the next prime is found.

$$\begin{array}{lcl} \mathsf{pr} & : & \forall\iota.\,\mathsf{Nat} \to \forall\jmath.\,\mathsf{Nat} \to \mathsf{Nat}^{\jmath} \to \mathsf{Stream}^{\iota}\,\mathsf{Nat} \\ \mathsf{pr} & := & \mathsf{fix}_3^{\nu}\,\lambda pr\lambda m. \\ & & \quad \mathsf{fix}_1^{\mu}\,\lambda pr'\lambda n\lambda k.\ \mathsf{match}\ k\ \mathsf{with} \\ & & \qquad \mathsf{zero} \quad \mapsto \quad \ldots \\ & & \qquad \mathsf{succ}\ k' \quad \mapsto \quad \mathsf{if}\ \gcd m\,n = 1\ \mathsf{then} \\ & & \qquad\qquad \mathsf{pair}\ n\ (pr\,(m*n)\,(n+1)\,((m-1)*n+1)) \\ & & \qquad\qquad \mathsf{else}\ pr'\,(n+1)\,k' \end{array}$$

$$\begin{array}{lcl} \mathsf{primes} & : & \mathsf{Stream}^{\infty}\mathsf{Nat} \\ \mathsf{primes} & := & \mathsf{pr}\ 1\ 2\ 1 \end{array}$$

Herein, $\mathsf{gcd} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ computes the greatest common divisor and $+$ and $*$ denote sum and product of natural numbers. Since in each executed function instance $\mathsf{pr}\,m\,n\,k$ the invariant $m+2 = n+k$ holds, the case $k = 0$ is impossible; otherwise the counter $n$ would have stepped over $m+1$ without finding a prime. Hence, the dots "$\ldots$" in the first match-branch will never be reached and can be replace with any term of the right type.

### 3.2.6 Sorting by Merging

The sorting algorithm whose termination is most easy to establish is insertion sort: insertion is primitive recursive, and sorting is performed by iteratively

inserting elements into a list which is empty in the beginning. Quick sort is more difficult, since the recursive calls happen on *indirect* sublists. It has been treated elsewhere [Abe04]. This time, we treat merge sort. Let $A$ be some type with a total order $\leq$: $A \to A \to \mathsf{Bool}$. Sorted merging can be defined by lexicographic recursion on the two input lists.

$$
\begin{array}{lcl}
\mathsf{merge} & : & \forall\iota.\, \mathsf{List}^\iota A \to \forall\jmath.\, \mathsf{List}^\jmath A \to \mathsf{List}^\infty A \\
\mathsf{merge} & : & \mathsf{fix}_0^\mu\, \lambda merge_0 \lambda l_1.\ \mathsf{match}\ l_1\ \mathsf{with} \\
 & & \quad \begin{array}{lcl}
\mathsf{nil} & \mapsto & \lambda l_2.\, l_2 \\
\mathsf{cons}\, x\, xs & \mapsto & \mathsf{fix}_0^\mu\, \lambda merge_1 \lambda l_2.\ \mathsf{match}\ l_2\ \mathsf{with} \\
 & & \quad \begin{array}{lcl}
\mathsf{nil} & \mapsto & l_1 \\
\mathsf{cons}\, y\, ys & \mapsto & \mathsf{if}\ x \leq y\ \mathsf{then}\, \mathsf{cons}\, x\, (merge_0\ xs\ l_2) \\
 & & \mathsf{else}\, \mathsf{cons}\, y\, (merge_1\ ys)
\end{array}
\end{array}
\end{array}
$$

A more precise type for merge would be $\forall\iota\forall\jmath.\ \mathsf{List}^\iota\ A \to \mathsf{List}^{\jmath+1} A \to \mathsf{List}^{\iota+\jmath} A$ (since the index counts the number of constructors as opposed to the length of the list, the $+1$ in the type of the second list is necessary—one nil constructor is discarded during the merge). But our type system, unlike Hughes, Pareto, and Sabry's [HPS96], does not handle addition of sizes. [4]

Merge sort recursively splits the input list, until only singleton lists remain, and then merges the lists together in a sorted fashion. For our type system to accept merge sort, it is crucial to know that both output lists of the split operation are not longer than the input list (in most, but not all cases, they will be strictly shorter, namely half the length). The natural type for split is $\forall A\forall\iota.\ \mathit{List}^\iota A \to \mathsf{List}^\iota A \times \mathsf{List}^\iota A$; with the impredicative encoding of products split can be written tail-recursive, in continuation-passing style (CPS).

$$
\begin{array}{lcl}
\mathsf{split} & : & \forall A\forall\iota.\, \mathsf{List}^\iota\ A \to \forall C.\, (\mathsf{List}^\iota\ A \to \mathsf{List}^\iota\ A \to C) \to C \\
\mathsf{split} & := & \mathsf{fix}_0^\mu\, \lambda split \lambda l \lambda k.\ \mathsf{match}\ l\ \mathsf{with} \\
 & & \quad \begin{array}{lcl}
\mathsf{nil} & \mapsto & k\, \mathsf{nil}\, \mathsf{nil} \\
\mathsf{cons}\, x\, xs & \mapsto & split\ xs\ (\lambda ys \lambda zs.\ k\ (\mathsf{cons}\, x\, zs)\ ys)
\end{array}
\end{array}
$$

We assign the following types.

$$
\begin{array}{lcl}
split & : & \mathsf{List}^\iota\ A \to \forall C.\, (\mathsf{List}^\iota\ A \to \mathsf{List}^\iota\ A \to C) \to C \\
l & : & \mathsf{List}^{\iota+1}\ A \\
k & : & \forall C.\, (\mathsf{List}^{\iota+1}\ A \to \mathsf{List}^{\iota+1}\ A \to C) \to C \\
x & : & A \\
xs & : & \mathsf{List}^\iota\ A \\
split\ xs & : & (\mathsf{List}^\iota\ A \to \mathsf{List}^\iota\ A \to C) \to C \\
ys, zs & : & \mathsf{List}^\iota\ A \\
\mathsf{cons}\, x\, zs,\ ys & : & \mathsf{List}^{\iota+1}\ A \\
k\ (\mathsf{cons}\, x\, zs)\ ys & : & C
\end{array}
$$

[4] Although there is an addition on ordinals, it is not commutative and hence would be a bit misleading in a type system (since most programmers would think of natural number addition). We could, however, introduce another kind nat of finite sizes which supports addition.

The type of *split* is $\mathsf{fix}_0^\mu$-$\mathsf{adm}$ since the result type $\forall C.\,(\mathsf{List}^\iota\,A \to \mathsf{List}^\iota\,A \to C) \to C$ is positive in size variable $\iota$.

Finally, we define merge sort msort via an auxiliary function msort′ such that msort′ *a as* sorts the list cons *a as* by merging.

$$
\begin{array}{lcl}
\mathsf{msort} & : & \mathsf{List}^\infty\,A \to \mathsf{List}^\infty\,A \\
\mathsf{msort} & := & \lambda l.\ \mathsf{match}\ l\ \mathsf{with} \\
 & & \quad \mathsf{nil} \mapsto \mathsf{nil} \\
 & & \quad \mathsf{cons}\ a\ as \mapsto \mathsf{msort}'\ a\ as
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{msort}' & : & A \to \mathsf{List}^\infty\,A \to \mathsf{List}^\infty\,A \\
\mathsf{msort}' & := & \mathsf{fix}_1^\mu\ \lambda msort \lambda a \lambda xs.\ \mathsf{match}\ xs\ \mathsf{with} \\
 & & \quad \mathsf{nil} \mapsto \mathsf{cons}\ a\ \mathsf{nil} \\
 & & \quad \mathsf{cons}\ b\ l \mapsto \mathsf{split}\ l\ (\lambda as \lambda bs.\ \mathsf{merge}\ (msort\ a\ as)\ (msort\ b\ bs))
\end{array}
$$

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

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

Altenkirch, McBride, and McKinna [AMM05] demonstrate that sorting by merging can be implemented in a structurally recursive fashion by explicating the computation pattern into an inductive data type. Applying short-cut fusion[5] to their program, one arrives at the usual functional formulation of merge sort. They point out that Turner [Tur95] has also defused quick sort, arriving at the structural tree sort [Bur69].

### 3.2.7 A Heterogeneous Data Type of Lambda Terms

In Haskell, we can define *heterogeneous*, also called *nested*, data types, for instance:

```
data TLam a = Var a
            | App (TLam a) (TLam a)
            | Abs Ty (TLam (Maybe a))
```

(Herein, `Ty` denotes some Haskell type of object-level type expressions.) The type `TLam a` is called heterogeneous, since the argument to the data type constructor `TLam` varies in a recursive occurrence on the right hand side. It is inhabited by de Bruijn representations of typed lambda terms over a set of free variables "`a`". A similar type has been studied by Altenkirch and Reus [AR99] and

[5] Intermediate tree structures are eliminated by the *acid rain theorem*.

Bird and Paterson [BP99b]; a precursor has been considered already by Pfenning and Lee [PL89] and Pierce, Dietzen, and Michaylov [PDM89]. The constructor for lambda-abstraction `Abs` expects the type of the abstracted variable and a term over the extended set of free variables `Maybe a`, which is the Haskell representation of the sum type $1 + \mathtt{a}$. The disjoint sum reflects the choice for a bound variable under the abstraction: either it is the variable freshly bound (left injection into the unit set "1") or it is one of the variables that have been available already (right injection into "a").

In $\mathsf{F}^{\widehat{\omega}}$, we can express the type constructor TLam by a least fixed point of kind $* \xrightarrow{+} *$.

$$
\begin{array}{lcl}
\mathsf{Ty} & : & * \\
\mathsf{TLam} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{TLam} & := & \lambda\iota.\, \mu^{\iota}_{*\xrightarrow{+}*} \lambda X \lambda A.\, A + (X\,A \times X\,A + \mathsf{Ty} \times X\,(\mathbf{1} + A)) \\
 & & \\
\mathsf{var} & : & \forall\iota \forall A.\, A \to \mathsf{TLam}^{\iota+1}\,A \\
\mathsf{var} & := & \lambda x.\, \mathsf{inl}\, x \\
 & & \\
\mathsf{app} & : & \forall\iota \forall A.\, \mathsf{TLam}^{\iota}\,A \to \mathsf{TLam}^{\iota}\,A \to \mathsf{TLam}^{\iota+1}\,A \\
\mathsf{app} & := & \lambda r \lambda s.\, \mathsf{inr}\,(\mathsf{inl}\,\langle r, s\rangle) \\
 & & \\
\mathsf{abs} & : & \forall\iota \forall A.\, \mathsf{Ty}^{\infty} \to \mathsf{TLam}^{\iota}\,(\mathbf{1} + A) \to \mathsf{TLam}^{\iota+1}\,A \\
\mathsf{abs} & := & \lambda a \lambda r.\, \mathsf{inr}\,(\mathsf{inr}\,\langle a, r\rangle)
\end{array}
$$

A whole article [AMU05] has been devoted to functions defined by iteration over heterogeneous data types. $\mathsf{F}^{\widehat{\omega}}$ can simulate all the systems discussed in that article, hence, all examples in that article can be replayed in $\mathsf{F}^{\widehat{\omega}}$. In $\mathsf{F}^{\widehat{\omega}}$, some functions can be given more precise typings, e. g., the functoriality/monotonicity witness of TLam:

$$
\begin{array}{lcl}
\mathsf{map}_{\mathsf{TLam}} & : & \forall\iota \forall A \forall B.\, (A \to B) \to \mathsf{TLam}^{\iota}A \to \mathsf{TLam}^{\iota}B \\
\mathsf{map}_{\mathsf{TLam}} & := & \mathsf{fix}^{\mu}_{1}\, \lambda \mathit{map}_{\mathsf{TLam}} \lambda f \lambda t.\, \mathsf{match}\ t\ \mathsf{with} \\
 & & \quad \mathsf{var}\ x \ \mapsto\ \mathsf{var}\,(f\,x) \\
 & & \quad \mathsf{app}\ r\,s \ \mapsto\ \mathsf{app}\,(\mathit{map}_{\mathsf{TLam}}\,f\,r)\,(\mathit{map}_{\mathsf{TLam}}\,f\,s) \\
 & & \quad \mathsf{abs}\ a\,r \ \mapsto\ \mathsf{abs}\,a\,(\mathit{map}_{\mathsf{TLam}}\,(\mathsf{lift}\,f)\,r) \\
 & & \\
\mathsf{lift} & : & \forall A \forall B.\, (A \to B) \to (\mathbf{1} + A \to \mathbf{1} + B) \\
\mathsf{lift} & := & \lambda f \lambda t.\, \mathsf{match}\ t\ \mathsf{with} \\
 & & \quad \mathsf{inl}\,\langle\rangle \ \mapsto\ \mathsf{inl}\,\langle\rangle \\
 & & \quad \mathsf{inr}\ x \ \mapsto\ \mathsf{inr}\,(f\,x)
\end{array}
$$

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

$$C(\iota) = \forall A \forall B.\, (A \to B) \to \mathsf{TLam}^{\iota}A \to \mathsf{TLam}^{\iota}B.$$

This is typical for functions over heterogeneous data types; in our example, since in the recursive call $\mathsf{map}_{\mathsf{TLam}}\,(\mathsf{lift}\,f)\,r$ the argument $r$ has type $\mathsf{TLam}^{\iota}\,(\mathbf{1} +$

$A$), the variable $A$ of the recursion type $C(\iota)$ has to be instantiated to $\mathbf{1} + A$. It is well-known that type reconstruction for polymorphic recursion is undecidable [Hen93, KTU93], hence, we cannot hope for an autonomous type inference algorithm for $\mathsf{F}_{\widehat{\omega}}$.

This example will be continued in Section 6.3.

### 3.2.8 Substitution for Finite and Infinite Lambda-Terms

Altenkirch and Reus [AR99] describe a substitution function for de Bruijn-style lambda-terms as a heterogeneous data type. During substitution under a abstraction, a new free variable is temporarily introduced, hence, all free variables in the substitute terms have to be lifted (renamed). Altenkirch and Reus perform this lifting operation also using substitution, and give a lexicographic termination argument. McBride [McB06] has, besides generalizing it to typed lambda-terms, exhibited the primitive recursive structure behind this algorithm, by giving a single traversal function for lambda-terms that is then instantiated to yield renaming or substitution. In the following, we demonstrate that we can replay his development in $\mathsf{F}_{\widehat{\omega}}$, although only for untyped terms, and that the same algorithm is recognized to be productive for *infinite* lambda-terms.

The generating type constructor for untyped de Bruijn terms is $\mathsf{LamF}$, from which we get the types of finite ($\mathsf{Lam}$) and infinite ($\mathsf{CoLam}$) terms:

$$\begin{array}{lcl}
\mathsf{LamF} & : & (* \xrightarrow{+} *) \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{LamF} & := & \lambda X \lambda A.\ A + (X\,A \times X\,A + X\,(\mathbf{1} + A))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{Lam} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Lam} & := & \lambda \iota.\ \mu^{\iota}_{*\xrightarrow{+}*}\mathsf{LamF}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{CoLam} & : & \mathsf{ord} \xrightarrow{-} * \xrightarrow{+} * \\
\mathsf{CoLam} & := & \lambda \iota.\ \nu^{\iota}_{*\xrightarrow{+}*}\mathsf{LamF}
\end{array}$$

The data constructors are uniformly definable for finite ($\nabla = \mu$) and infinite ($\nabla = \nu$) terms:

$$\begin{array}{lcl}
\mathsf{var} & : & \forall \iota \forall A.\ A \to \nabla^{\iota+1}\mathsf{LamF}\ A \\
\mathsf{var} & := & \lambda x.\ \mathsf{inl}\ x
\end{array}$$

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

$$\begin{array}{lcl}
\mathsf{abs} & : & \forall \iota \forall A.\ \nabla^{\iota}\mathsf{LamF}\ (\mathbf{1} + A) \to \nabla^{\iota+1}\mathsf{LamF}\ A \\
\mathsf{abs} & := & \lambda r.\ \mathsf{inr}\ (\mathsf{inr}\ r)
\end{array}$$

In the absence of sized types, renaming and substitution can be subsumed under the common type

$$\forall A \forall B.\ (A \to F\ B) \to \nabla\mathsf{LamF}\ A \to \nabla\mathsf{LamF}\ B,$$

where $F$ is instantiated with the identity in case of renaming, and with Lam in case of substitution. We are dealing with sized types, however, and in the case of infinite terms we would like that both renaming and substitution are of type $\mathsf{CoLam}^\iota A \to \mathsf{CoLam}^\iota B$. This means that the result of renaming and substitution is defined at least up to the same depth than the input. Hence, we let $F$ depend on a size argument and obtain the type

$$\forall \iota \forall A \forall B.\, (A \to F^\iota\, B) \to \mathsf{CoLam}^\iota A \to \mathsf{CoLam}^\iota B.$$

McBride parameterizes the generic traversal function for de Bruijn terms by a *kit* that is passed as an extra argument to the traversal function. The kit contains three functions: a mapping from variables into $F$, a mapping from $F$ to terms, and a lifting function on $F$. For our purposes is more convenient to separate the second function from the rest. We say $F$ has *term structure* if it is pointed and supports lifting:

$$\begin{array}{lcl} \mathsf{TmStr} & : & (\mathsf{ord} \xrightarrow{-} * \xrightarrow{+} *) \xrightarrow{\circ} * \\ \mathsf{TmStr} & := & \lambda F \forall \iota \forall A. \\ & & (A \to F^{\iota+1} A) \times (F^\iota A \to F^\iota(\mathbf{1} + A)) \end{array}$$

We have chosen $F$ to be antitonic in the size index since we will later instantiate it with CoLam.

If $F$ has term structure, we can lift functions $f : A \to F^{\iota+1} B$ to make room for one extra variable:

$$\begin{array}{lcl} \mathsf{lift} & : & \forall F{:}\mathsf{ord} \xrightarrow{-} * \xrightarrow{+} *.\ \mathsf{TmStr}\, F \to \\ & & \forall \iota \forall A \forall B.\, (A \to F^{\iota+1} B) \to (\mathbf{1} + A \to F^{\iota+1}(\mathbf{1} + B)) \\ \mathsf{lift} & := & \lambda \langle vr, wk \rangle \lambda f \lambda ma.\ \mathsf{match}\ ma\ \mathsf{with} \\ & & \quad \mathsf{inl}\, \langle\rangle \ \mapsto\ vr\, (\mathsf{inl}\, \langle\rangle) \\ & & \quad \mathsf{inr}\, a \ \mapsto\ wk\, (f\, a) \end{array}$$

Generic traversal now works for all $F$ which can be converted to terms using the parameter $tm$. It is defined as follows:

$$\begin{array}{lcl} \mathsf{trav} & : & \forall F{:}\mathsf{ord} \xrightarrow{-} * \xrightarrow{+} *.\ \mathsf{TmStr}\, F \to (\forall \iota \forall B.\, F^{\iota+1} B \to \nabla^{\iota+1} \mathsf{LamF}\, B) \to \\ & & \forall \iota \forall A \forall B.\, (A \to F^\iota\, B) \to (\nabla^\iota \mathsf{LamF}\, A \to \nabla^\iota \mathsf{LamF}\, B) \\ \mathsf{trav} & : & \lambda k \lambda tm.\ \mathsf{fix}^\nabla\ \lambda trav \lambda f \lambda t.\ \mathsf{match}\ t\ \mathsf{with} \\ & & \quad \mathsf{var}\, a \ \mapsto\ tm\, (f\, a) \\ & & \quad \mathsf{abs}\, r \ \mapsto\ \mathsf{abs}\, (trav\, (\mathsf{lift}\, k\, f)\, r) \\ & & \quad \mathsf{app}\, r\, s \ \mapsto\ \mathsf{app}\, (trav\, f\, r)\, (trav\, f\, s) \end{array}$$

Note that $f : A \to F^{\iota+1} B$, but, in the recursive call, $trav\, f\, s$ expects $f$ to be of type $A \to F^\iota B$. This is fine because $F$ is antitone in its size argument, hence $A \to F^{\iota+1} B \leq A \to F^\iota B$. In case of $\nabla = \mu$, $\mathsf{fix}^\nabla$ has to be instantiated with $\mathsf{fix}^\mu_1$, and with $\mathsf{fix}^\nu_2$ in case of $\nabla = \nu$. In both cases, the type

$$C(\iota) = \forall A \forall B.\, (A \to F^\iota\, B) \to (\nabla^\iota \mathsf{LamF}\, A \to \nabla^\iota \mathsf{LamF}\, B)$$

is admissible for (co)recursion, again since $F$ is antitonic.

Instantiating $F$ with $\lambda\iota\lambda A.A$, we obtain the renaming function for free variables:

$$\begin{array}{lcl} \mathsf{rename} & : & \forall\iota\forall A\forall B.\,(A \to B) \to \nabla^\iota\mathsf{LamF}\,A \to \nabla^\iota\mathsf{LamF}\,B \\ \mathsf{rename} & := & \mathsf{trav}\,\langle \mathsf{id}, \mathsf{inr}\rangle\,\mathsf{var} \end{array}$$

Substitution for infinite terms is obtained by the instantiation $F = \mathsf{CoLam}$:

$$\begin{array}{lcl} \mathsf{subst} & : & \forall\iota\forall A\forall B.\,(A \to \mathsf{CoLam}^\iota B) \to \mathsf{CoLam}^\iota A \to \mathsf{CoLam}^\iota B \\ \mathsf{subst} & := & \mathsf{trav}\,\langle \mathsf{var}, \mathsf{rename}\,\mathsf{inr}\rangle\,\mathsf{id} \end{array}$$

The best type we can give to substitution of finite terms in $\mathsf{F}^{\widehat{}}_\omega$ is $(A \to \mathsf{Lam}^\infty B) \to \mathsf{Lam}^\infty A \to \mathsf{Lam}^\infty B$. We first have to type-check trav with a specialized type, and then instantiate it with $F = \lambda\iota.\,\mathsf{Lam}^\infty$:

$$\begin{array}{lcl} \mathsf{trav} & : & \forall F : * \xrightarrow{+} *.\,\mathsf{TmStr}\,(\lambda\iota F) \to (\forall B.\,F\,B \to \mathsf{Lam}^\infty B) \to \\ & & \forall\iota\forall A\forall B.\,(A \to F\,B) \to (\mathsf{Lam}^\iota A \to \mathsf{Lam}^\infty B) \\ \mathsf{subst} & : & \forall A\forall B.\,(A \to \mathsf{Lam}^\infty B) \to \mathsf{Lam}^\infty A \to \mathsf{Lam}^\infty B \\ \mathsf{subst} & := & \mathsf{trav}\,\langle \mathsf{var}, \mathsf{rename}\,\mathsf{inr}\rangle\,\mathsf{id}. \end{array}$$

## 3.3 Limits, Iteration, and Fixed-Points

In this section, we will define transfinite iteration to give a semantics to inductive and coinductive types. We will calculate the closure ordinal $\top^{\mathsf{ord}}$ of iteration, such that, $\mu\infty$ and $\nu\infty$ are indeed fixed-points. This will justify the rules TY-FOLD and TY-UNFOLD for the case $a = \infty$.

### 3.3.1 Limits

Let $(\mathsf{O}, \leq)$ be some complete linear ordering, e.g., the set of ordinals up to $\top^{\mathsf{ord}}$ and $(\mathfrak{L}, \sqsubseteq, \inf, \sup)$ some complete lattice. For $f \in \mathsf{O} \to \mathfrak{L}$ and $\lambda > 0$ a limit ordinal in $\mathsf{O}$ we define:

$$\begin{array}{lcll} \inf_\lambda f & := & \inf_{\alpha<\lambda} f(\alpha) & \text{infimum} \\ \sup_\lambda f & := & \sup_{\alpha<\lambda} f(\alpha) & \text{supremum} \\ \\ \liminf_\lambda f & := & \sup_{\alpha_0<\lambda} \inf_{\alpha_0\leq\alpha<\lambda} f(\alpha) & \text{limes inferior} \\ \limsup_\lambda f & := & \inf_{\alpha_0<\lambda} \sup_{\alpha_0\leq\alpha<\lambda} f(\alpha) & \text{limes superior} \end{array}$$

**Lemma 3.14 (Relationships)** *In the above context, if $\lambda \neq 0$,*

$$\inf_\lambda f \sqsubseteq \liminf_\lambda f \sqsubseteq \limsup_\lambda f \sqsubseteq \sup_\lambda f.$$

If $\liminf_\lambda f = \limsup_\lambda f = x$ for some $x \in \mathfrak{L}$, we say that the *limit* of $f$ at $\lambda$ exists and set $\lim_\lambda f = x$.

**Lemma 3.15 (Limes inferior of monotone function)** *If $f$ is monotone below $\lambda$, i. e., $f(\alpha) \sqsubseteq f(\beta)$ for all $\alpha < \beta < \lambda$, then $\liminf_\lambda f = \sup_\lambda f$.*

*Proof.* For monotone $f$ it holds that $\inf_{\alpha_0 \leq \alpha < \lambda} f(\alpha) = f(\alpha_0)$. □

**Corollary 3.16** *If $f$ is monotone below $\lambda$, then $\lim_\lambda f = \sup_\lambda f$.*

**Lemma 3.17 (Limes superior of antitone function)** *If $f$ is antitone below $\lambda$, then $\limsup_\lambda f = \inf_\lambda f$.*

*Proof.* For antitone $f$ it holds that $\sup_{\alpha_0 \leq \alpha < \lambda} f(\alpha) = f(\alpha_0)$. □

**Corollary 3.18** *If $f$ is antitone below $\lambda$, then $\lim_\lambda f = \inf_\lambda f$.*

A function $f$ is *continuous* in $\lambda$, if $f(\lambda) = \lim_\lambda f$. Here we mean continuous in the sense of classical analysis, not in the sense of domain theory, where continuity includes monotonicity.

### 3.3.2 Operator Iteration

By the theorem of Knaster and Tarski we know that in each complete lattice $\mathfrak{L}$ least and greatest fixed-points of monotone operators $\mathcal{F} \in \mathfrak{L} \to \mathfrak{L}$ exist. The least fixed-point of $\mathcal{F}$, for instance, can be defined either "from above" as $\inf\{\mathcal{G} \mid \mathcal{F}(\mathcal{G}) \sqsubseteq \mathcal{G}\}$ or from below as the $\alpha$th iterate $\mathcal{F}^\alpha(\bot)$ for some sufficiently large ordinal $\alpha$. We choose the second alternative since our system $\mathsf{F}_\omega^{\widehat{}}$ also speaks about approximations $\mathcal{F}^\beta(\bot)$ for $\beta < \alpha$. In the following we will make it clear what we mean by transfinite iteration $\mathcal{F}^\alpha(\mathcal{G})$ in arbitrary complete lattices $\mathfrak{L}$—we will of course only use the notion for our operator lattices $[\![\kappa]\!]$, but the specifics about the lattice do not matter for iteration and fixed-points.

**Transfinite iteration.** Let $\mathcal{F} \in \mathfrak{L} \to \mathfrak{L}$ be an endo-function, $\mathcal{G} \in \mathfrak{L}$ and $\alpha$ an ordinal number. We define the $\alpha$-iterate $\mathsf{I}_\alpha\, \mathcal{F}\, \mathcal{G}$ of $\mathcal{F}$ at $\mathcal{G}$ by transfinite recursion on $\alpha$.

$$\begin{array}{rcl} \mathsf{I}_0\, \mathcal{F}\, \mathcal{G} & := & \mathcal{G} \\ \mathsf{I}_{\alpha+1}\, \mathcal{F}\, \mathcal{G} & := & \mathcal{F}(\mathsf{I}_\alpha\, \mathcal{F}\, \mathcal{G}) \\ \mathsf{I}_\lambda\, \mathcal{F}\, \mathcal{G} & := & \liminf_{\alpha \to \lambda}(\mathsf{I}_\alpha\, \mathcal{F}\, \mathcal{G}) \end{array}$$

We use $\mathcal{F}^\alpha(\mathcal{G})$ as a shorthand for $\mathsf{I}_\alpha\, \mathcal{F}\, \mathcal{G}$. For the limit case, we chose to use the limes superior, as Danner [Dan99]. Note that iteration is well-defined for any $\mathcal{F}$, in particular, non-monotonic $\mathcal{F}$. However, many reasonable properties on iteration hold only for a monotonic $\mathcal{F}$.

**Lemma 3.19 (Iteration of monotonic function)** *If $\mathcal{F} \sqsubseteq \mathcal{F}' \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ then $\mathsf{I}_\alpha\, \mathcal{F} \sqsubseteq \mathsf{I}_\alpha\, \mathcal{F}' \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ for any $\alpha \in \mathsf{On}$.*

*Proof.* By induction on $\alpha$. □

For iteration to be monotonic in the ordinal index, we need to start iteration at an element $\mathcal{G} \in \mathfrak{L}$ for which $\mathcal{F}$ is *inflationary*, meaning $\mathcal{F}(\mathcal{G}) \sqsupseteq \mathcal{G}$. When we use iteration to define the semantics of $\mu$, iteration will start at $\mathcal{G} = \bot$, for which any $\mathcal{F}$ is trivially inflationary, hence the semantics of $\mu$ will be monotone in the ordinal argument for any monotone $\mathcal{F}$.

**Lemma 3.20 (Iteration of inflationary function[6])** *Let $\mathcal{F} \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ and $\mathcal{G} \in \mathfrak{L}$ with $\mathcal{G} \sqsubseteq \mathcal{F}(\mathcal{G})$. Then*

1. *$\mathcal{F}^\alpha(\mathcal{G}) \sqsubseteq \mathcal{F}^\beta(\mathcal{G})$ for $\alpha \leq \beta$, and*
2. *$\mathcal{F}^\beta(\mathcal{G}) \sqsubseteq \mathcal{F}^{\beta+1}(\mathcal{G})$.*

*Proof.* Simultaneously by induction on $\beta$. Since $\sqsubseteq$ is reflexive, it is sufficient to show $\mathcal{F}^\alpha(\mathcal{G}) \sqsubseteq \mathcal{F}^\beta(\mathcal{G})$ for $\alpha < \beta$ instead of proposition 1.

*Case* $\beta = 0$. For part 1, there is nothing to show, part 2 follows by assumption.

*Case* $\beta = \beta' + 1$. For part 1, assume $\alpha < \beta' + 1$. By induction hypothesis 1, $\mathcal{F}^\alpha(\mathcal{G}) \sqsubseteq \mathcal{F}^{\beta'}(\mathcal{G})$, and by induction hypothesis 2, $\mathcal{F}^{\beta'}(\mathcal{G}) \sqsubseteq \mathcal{F}^{\beta'+1}(\mathcal{G})$. Hence the claim follows by transitivity. For part 2, just use monotonicity of $\mathcal{F}$ on the induction hypothesis.

*Case* $\beta = \lambda$ limit ordinal. For part 1, assume $\alpha < \lambda$. Since by induction hypothesis 1 function $\alpha \mapsto \mathcal{F}^\alpha(\mathcal{G})$ is monotone below $\lambda$, by Lemma 3.15 it is sufficient to show that $\mathcal{F}^\alpha(\mathcal{G}) \sqsubseteq \sup_{\alpha<\lambda} \mathcal{F}^\alpha(\mathcal{G})$—which trivially holds. For part 2 we can apply Lemma 3.15 similarly which leaves us the goal $\sup_{\alpha<\lambda} \mathcal{F}^\alpha(\mathcal{G}) \sqsubseteq \mathcal{F}^{\lambda+1}(\mathcal{G})$. Assume an arbitrary $\alpha < \lambda$. By induction hypothesis 1, $\mathcal{F}^\alpha(\mathcal{G}) \sqsubseteq \mathcal{F}^\lambda(\mathcal{G})$, hence by monotonicity $\mathcal{F}^{\alpha+1}(\mathcal{G}) \sqsubseteq \mathcal{F}^{\lambda+1}(\mathcal{G})$. Since $\alpha$ was arbitrary and $\mathcal{F}^0(\mathcal{G}) \sqsubseteq \mathcal{F}^1(\mathcal{G})$ by assumption, it holds for all $\alpha < \lambda$ that $\mathcal{F}^\alpha(\mathcal{G}) \sqsubseteq \mathcal{F}^{\lambda+1}(\mathcal{G})$. Hence, we are done. □

[6] Our formulation of Lemma 3.20 is actually equivalent to Danner's Lemma 3.5 [Dan99]. He states instead (in our notation):

> If $\mathsf{I}_{\xi+1}\,\mathcal{F}\,x \sqsupseteq \mathsf{I}_\xi\,\mathcal{F}\,x$, then for all $\gamma > \alpha \geq \xi$, $\mathsf{I}_\gamma\,\mathcal{F}\,x \sqsupseteq \mathsf{I}_\alpha\,\mathcal{F}\,x$.

Letting $\mathcal{G} := \mathsf{I}_\xi\,\mathcal{F}\,x$, his premise states that $\mathcal{F}$ should be inflationary on $\mathcal{G}$. His conclusion can be reformulated to

> for all $\gamma' > \alpha'$, $\mathsf{I}_{\gamma'}\,\mathcal{F}\,\mathcal{G} \sqsupseteq \mathsf{I}_{\alpha'}\,\mathcal{F}\,\mathcal{G}$

where $\gamma = \xi + \gamma'$ and $\alpha = \xi + \alpha'$. The new formulation is identical to the old since ordinal addition distributes over iteration in the following way:

$$\mathsf{I}_{\xi+\alpha}\,\mathcal{F} = \mathsf{I}_\alpha\,\mathcal{F} \circ \mathsf{I}_\xi\,\mathcal{F}$$

This equation can be proven by transfinite induction on $\alpha$ (note that ordinal addition is defined by recursion on the second argument) and holds without any assumptions on $\mathcal{F}$. The equation also appears in Danner's Theorem 3.7, but only for monotone $\mathcal{F}$.

If $\mathcal{F}$ is monotone and deflationary, $\mathcal{F}(\mathcal{G}) \sqsubseteq \mathcal{G}$, for the start value $\mathcal{G}$, then iteration will be antitonic in the ordinal index. This is, for instance, the case if we choose $\mathcal{G} = \top$, as we will do for the interpretation of $\nu$.

**Lemma 3.21 (Iteration of deflationary function)** *Let $\mathcal{F} \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ and $\mathcal{G} \in \mathfrak{L}$ with $\mathcal{G} \sqsupseteq \mathcal{F}(\mathcal{G})$. Then*

1. *$\mathcal{F}^\alpha(\mathcal{G}) \sqsupseteq \mathcal{F}^\beta(\mathcal{G})$ for $\alpha \leq \beta$, and*
2. *$\mathcal{F}^\beta(\mathcal{G}) \sqsupseteq \mathcal{F}^{\beta+1}(\mathcal{G})$.*

*Proof.* Analogously to Lemma 3.20. □

Summarizing, we can say that for a monotone $\mathcal{F}$ which is inflationary (deflationary) for $\mathcal{G}$, we get a monotone (antitone) function $f(\alpha) = \mathsf{I}_\alpha\, \mathcal{F}\, \mathcal{G}$. In both cases, the limes inferior of $f$ at some limit ordinal $\lambda$ coincides with the limes superior (lemmas 3.15 and 3.17), and, by definition of $f$, with the actual value $f(\lambda)$ of $f$ at $\lambda$, hence $f$ is continuous at all limit ordinals.

### 3.3.3 Fixed points

By the theorem of Knaster and Tarski [Tar55], each monotonic operator $\mathcal{F} \in \mathfrak{L} \to \mathfrak{L}$ in a complete lattice $\mathfrak{L}$ has a least fixed point. This fixed point can be reached "from below" by transfinite iteration of $\mathcal{F}$, starting at the bottom element $\bot$ of the lattice. By Lemma 3.20, the iterates $\mathcal{F}^\alpha(\bot)$ form an ascending chain. If for some ordinal $\gamma$

$$\mathcal{F}(\mathcal{F}^\gamma(\bot)) \sqsubseteq \mathcal{F}^\gamma(\bot),$$

i.e., the $\gamma$-iterate is a *prefixed point* of $\mathcal{F}$, then the least fixed point of $\mathcal{F}$ has been reached and we call $\gamma$ the closure ordinal of operator $\mathcal{F}$.

Analogously, the greatest fixed point does exist which can be reached "from above" by starting at the top element $\top$ of the lattice. By Lemma 3.21, the approximations $\mathcal{F}^\alpha(\bot)$ form a descending chain. For some ordinal $\gamma$ it holds that

$$\mathcal{F}(\mathcal{F}^\gamma(\top)) \sqsupseteq \mathcal{F}^\gamma(\top),$$

i.e., the $\gamma$-iterate is a *postfixed point* of $\mathcal{F}$, and the greatest fixed point has been reached.

**Cardinalities for pure kinds.** Since each $\mathcal{A} \in [\![*]\!]$ is a subset of the countable set $\top^*$ by definition, the cardinality of $[\![*]\!]$ is at most $\beth_1 = |\mathcal{P}(\mathbb{N})|$, an uncountable cardinal number. With $\beth_{n+1}$ we denote the cardinality of the power set $\mathcal{P}(\beth_n)$ of the previous cardinal $\beth_n$. Let $\kappa = \vec{p}\vec{\kappa} \to *$ be a pure kind, then $[\![\vec{p}\vec{\kappa} \to *]\!] \subset \mathcal{P}([\![\kappa_1]\!] \times \ldots \times [\![\kappa_n]\!] \times [\![*]\!])$ hence the following lemma holds:

**Lemma 3.22 (Upper bound for cardinality)** *For pure kind $\kappa$,*

$$|[\![\kappa]\!]| \leq \beth_{\mathsf{rk}(\kappa)+1}.$$

*Proof.* By induction on $\kappa$. Base case $\kappa = *$ follows by assumption. For the step case, let $n \geq 1$ and $\kappa = \vec{p}\vec{\kappa} \to *$.

$$\begin{aligned}
|[\![\vec{p}\vec{\kappa} \to *]\!]| &\leq |\mathcal{P}([\![\kappa_1]\!] \times \ldots \times [\![\kappa_n]\!] \times [\![*]\!])| && \text{set-theoretic function space}\\
&\leq |\mathcal{P}(\beth_{\mathsf{rk}(\kappa_1)+1} \times \ldots \times \beth_{\mathsf{rk}(\kappa_n)+1} \times \beth_1)| && \text{induction hypothesis}\\
&\leq |\mathcal{P}(\beth_{\max\{\mathsf{rk}(\kappa_i)+1 | 1 \leq i \leq n\}})| && \text{cardinal multiplication}\\
&\leq \beth_{1+\max_i \mathsf{rk}(\kappa_i)+1} && \text{definition of } \beth_n\\
&= \beth_{\mathsf{rk}(\kappa)+1} && \text{definition of rank}
\end{aligned}$$

□

**Closure ordinal.** At which ordinal $\gamma$ will least and greatest fixed point be surely reached for any monotone operator? We can give a "brute-force" upper bound for $\gamma$ as follows: Assume we want to construct a fixed point in lattice $\mathfrak{L} = [\![\kappa]\!]$ for some $\kappa$ by a chain $(\mathcal{F}_\alpha)$ of approximations. Since the cardinality of $\mathfrak{L}$ is bounded by $\beth_n$ where $n = \mathsf{rk}(\kappa) + 1$ (see Lemma 3.22), the chain can enumerate less then $\beth_n$ elements $\mathcal{F}_\alpha$ where $0 \leq \alpha < \beth_n$. Hence, the element $\mathcal{F}_{\beth_n}$ must be the fixed point. An upper bound $\gamma$ for the closure ordinal for all monotone operators $\mathcal{F}$ in all lattices $[\![\kappa]\!]$ with $\kappa$ pure kind is the supremum of all $\beth_n$, the ordinal $\beth_\omega$. Now we can complete the definition of $[\![\mathsf{ord}]\!]$ setting

$$\top^{\mathsf{ord}} := \beth_\omega.$$

### 3.3.4 Inductive and Coinductive Constructors

Now, we can fix the semantics of $\mu_\kappa$ and $\nu_\kappa$. Recalling that $[\![C]\!]_\theta$ was defined as $\mathsf{Sem}(C)$, we set

$$\begin{array}{lcl}
\mathsf{Sem}(\mu_\kappa)(\alpha)(\mathcal{F}) & := & \mathcal{F}^\alpha(\bot^\kappa), \text{ and}\\
\mathsf{Sem}(\nu_\kappa)(\alpha)(\mathcal{F}) & := & \mathcal{F}^\alpha(\top^\kappa).
\end{array}$$

**Lemma 3.23 (Semantics of (co)inductive constructors fulfill specification)** *We have*

$$\begin{array}{lcl}
[\![\mu_\kappa]\!] & \in & [\![\mathsf{ord}]\!] \xrightarrow{+} ([\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]) \xrightarrow{+} [\![\kappa]\!],\\
[\![\nu_\kappa]\!] & \in & [\![\mathsf{ord}]\!] \xrightarrow{-} ([\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]) \xrightarrow{+} [\![\kappa]\!].
\end{array}$$

*Proof.* By definition and lemmata 3.19, 3.20, and 3.21. □

**Theorem 3.24 (Soundness of folding and unfolding)** *Let* $\nabla_\kappa \in \{\mu_\kappa, \nu_\kappa\}$, $\mathcal{F} \in [\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]$, *and* $\alpha \in [\![\mathsf{ord}]\!]$. *We set*

$$\begin{array}{lcll}
\mathcal{G} & := & \mathcal{F}([\![\nabla_\kappa]\!]\,\alpha\,\mathcal{F}) & \textit{(unfolded)},\\
\mathcal{H} & := & [\![\nabla_\kappa]\!]([\![\mathsf{s}]\!]\alpha)\,\mathcal{F} & \textit{(folded)}.
\end{array}$$

*Then* $\mathcal{G} \sqsubseteq^\kappa \mathcal{H}$ *and* $\mathcal{H} \sqsubseteq^\kappa \mathcal{G}$.

*Proof.* We consider the inductive case $\nabla_\kappa = \mu_\kappa$, the case of coinductive constructors is proven analogously. Observe that $\mathcal{G} = \mathcal{F}^{\alpha+1}(\bot)$ and $\mathcal{H} = \mathcal{F}^{[\![\mathsf{s}]\!]\alpha}(\bot)$. Since $[\![\mathsf{s}]\!]\alpha = \alpha + 1$ for $\alpha < \top^{\mathsf{ord}}$, the only interesting case is $\alpha = \top^{\mathsf{ord}}$. Then, $[\![\mathsf{s}]\!]\alpha = \top^{\mathsf{ord}}$, but $\mathcal{G} \sqsubseteq \mathcal{H}$ still holds since $\mathcal{F}^{\top^{\mathsf{ord}}}(\bot)$ is a fixed point of $\mathcal{F}$. □

The soundness of rules TY-FOLD and TY-UNFOLD is a consequence of this theorem.

### 3.3.5 Soundness of $\lambda$-Dropping

The least fixpoint of a constructor equation

$$\begin{array}{l} X : \kappa \\ X\,\vec{A} = F\,X\,\vec{A} \end{array}$$

can mechanically be expressed as $\mu^\infty_\kappa \lambda X \lambda \vec{A}.\, F\,X\,\vec{A}$. However, if $F$ is of a regular structure, the rank of the fixed-point can be decreased. For instance, the equation

$$\mathsf{List}\,A = \mathbf{1} + A \times \mathsf{List}\,A$$

has, besides the mechanical solution $\mu^\infty_{*\xrightarrow{+}*} \lambda X \lambda A.\, \mathbf{1} + A \times X\,A$ with a rank-2 fixed-point, the rank-1 solution $\lambda A.\, \mu^\infty_* \lambda Y.\, \mathbf{1} + A \times Y$. The second constructor has been obtained from the first by *lambda dropping*. With our semantics, we can prove that the two solutions describe the same lists. More generally, assume a constructor

$$\Gamma,\, \iota : \mathsf{ord} \vdash F : \kappa_2 \xrightarrow{+} \kappa_1 \xrightarrow{p} \kappa_2$$

and a family of valuations $\theta(\alpha) := \theta[\iota \mapsto \alpha]$ for some $\theta \in [\![\Gamma]\!]$. Then for all $\alpha \in [\![\mathsf{ord}]\!]$,

$$[\![\nabla^{\iota\,p}_{*\to*} \lambda X \lambda A.\, F\,(X A)\,A]\!]_{\theta(\alpha)} = [\![\lambda A.\, \nabla^\iota_{\kappa_2}\, \lambda Y.\, F\,Y\,A]\!]_{\theta(\alpha)}$$

The proof proceeds by transfinite induction on $\alpha$ without difficulties.

## 3.4 Semantical Types

In this section, we give a generic term model for our calculus, showing that the typing rules are sound. Later we will instantiate this model to obtain a proof of strong normalization. A generic soundness proof can, for instance, be found in work of Vouillon [Vou04] and Melliès [VM04]. The idea is to consider types as *closed sets* of terms between a (minimal) set $\mathcal{N}$ of *neutral terms* and a (maximal) set $\mathcal{S}$ of *safe terms*. Informally a set is closed if for each of its terms $t$ it also contains all terms $t'$ which behave the same as $t$ (for instance, weak head reducts of $t$). The result of the soundness proof is that each typable term is safe. *Safe* can mean different things: e.g., that the term evaluates without error; in our case a safe term will be a strongly normalizing one. Although we have

taken some ideas from Vouillon and Melliès to structure our soundness proof, it needs to be stressed that their method does not directly apply, since we are interested in *strong* and not just weak normalization. For instance, we have to introduce the concepts of safe evaluation contexts and safe reduction. Matthes [Mat05] defines such safe contexts for a classical system F with sum types; safe evaluation is only implicit in his inductive definition of strongly normalizing terms.

Vaux [Vau04] and the author, in joint work with Coquand [AC05], have carried out a generic soundness proof where semantical types were not required to be closed sets. This was possible since in their model $\beta$-equal terms were identified. From such a model, however, one can only harvest a proof of weak normalization. A proof of strong normalization cannot be extracted, since the property "strongly normalizing" is not preserved under $\beta$-equality.[7] Hence, we need to fall back to closed sets; in our case, we use a form of *saturated sets* [Tai75, Luo90], which are a variant of Girard's *reducibility candidates* [GLT89].

In the following we will develop requirements for our model and exhibit them with labels with the prefix REQ.

**Function space.** Many models of the $\lambda$-calculus define the function space between sets of terms $\mathcal{A}$ and $\mathcal{B}$ as

$$\mathcal{A} \boxed{\rightarrow} \mathcal{B} := \{r \mid r\,s \in \mathcal{B} \text{ for all } s \in \mathcal{A}\}.$$

In other words, every term which behaves as a function—because it can be applied—is considered a function.[8] The function space $\mathcal{A} \boxed{\rightarrow} \mathcal{B}$ will be a closed set if $\mathcal{B}$ is closed and $\mathcal{A}$ is a set of safe terms.

**Type interval.** Each semantical type $[\![A]\!]$ must contain only safe terms and contain all neutral terms. In the following, we assume a set $\mathcal{S} \subseteq \mathsf{Tm}$ of safe terms and a set $\mathcal{N} \subseteq \mathsf{Tm}$ of neutral terms such that

| | |
|---|---|
| REQ-$\mathcal{N}$-SUB-$\mathcal{S}$ | $\mathcal{N} \subseteq \mathcal{S}$, |
| REQ-$\mathcal{S}$-VAL | if $t \in \mathcal{S} \setminus \mathcal{N}$ then $t \rhd v \in \mathsf{Val}$. |

Herein, $\rhd$ is a to-be-defined evaluation relation. One could think of $t \rhd v$ as "$t$ evaluates to $v$ without error"; then a choice for $\mathcal{S}$ would be the terms whose evaluation does not throw an exception, but might diverge, and $\mathcal{N}$ would contain the diverging terms. In our case $\rhd$ will be a variant of weak head reduction, and $\mathcal{N}$ will consist of the non-values which do not weak head reduce.

[7] If $t$ is strongly normalizing and $\Omega$ diverging, then $(\lambda\_t)\,\Omega$ is $\beta$-equal to $t$ but only weakly normalizing.

[8] Matthes [Mat00] calls this definition *elimination based*. The alternative is to define the function space *introduction based* as the closure of the set of all function *constructions*, in our case $\lambda$-abstractions, recursive and corecursive functions. The elimination-based approach has the obvious advantage of succinctness and spares us from defining a closure operator. The introduction-based approach, however, has other advantages: it provides a predicative semantics of disjoint sums [AA00] [Mat05, Sect. 4] or other constructions which have an elimination into an arbitrary type (e.g., unary sums as in the computational $\lambda$-calculus [LS05]).

Each semantical type $\mathcal{A}$ will be in the interval $[\mathcal{N}, \mathcal{S}]$, i. e., $\mathcal{N} \subseteq \mathcal{A} \subseteq \mathcal{S}$. In the following we establish the requirements for $\boxed{\to}$ to be an operator on term sets in this interval.

**Safe evaluation contexts** are isolated by the judgements $e \in \mathsf{Sframe}$ and $E \in \mathsf{Scxt}$.

$$\text{SF-APP}\ \frac{s \in \mathcal{S}}{\_\, s \in \mathsf{Sframe}} \qquad \text{SF-REC}\ \frac{s, t_1 \ldots t_n \in \mathcal{S}}{\mathsf{fix}^{\mu}_n\ s\, t_{1..n}\, \_ \in \mathsf{Sframe}}$$

$$\text{SC-ID}\ \frac{}{\mathsf{Id} \in \mathsf{Scxt}} \qquad \text{SC-PUSH}\ \frac{E \in \mathsf{Scxt} \qquad e \in \mathsf{Sframe}}{E \circ e \in \mathsf{Scxt}}$$

A *strict* function is one which is undefined for undefined arguments, or, which diverges if called with a diverging argument. We generalize "diverging" to "neutral" and call a function strict which maps neutral terms to neutral terms. Safe evaluation frames must be strict functions [Vou04], i. e.,

$$\text{REQ-STRICT} \qquad e(\mathcal{N}) \subseteq \mathcal{N} \text{ for all } e \in \mathsf{Sframe}.$$

A trivial consequence is that all safe evaluation *contexts* are strict as well.

**Lemma 3.25 (Function space is above $\mathcal{N}$)** *If $\mathcal{A} \subseteq \mathcal{S}$ and $\mathcal{N} \subseteq \mathcal{B}$ then $\mathcal{N} \subseteq \mathcal{A} \boxed{\to} \mathcal{B}$.*

*Proof.* Since REQ-STRICT can be rewritten as $\mathcal{N} \subseteq e^{-1}(\mathcal{N})$ for all $e \in \mathsf{Sframe}$, the lemma becomes obvious if rewrite the definition of function space to

$$\mathcal{A} \boxed{\to} \mathcal{B} = \bigcap_{s \in \mathcal{A}} (\_\, s)^{-1}(\mathcal{B}).$$

(By SF-APP we have $\mathcal{N}\, s \subseteq \mathcal{N} \subseteq \mathcal{B}$ for all $s \in \mathcal{A}$, hence, $\mathcal{N} \subseteq (\_\, s)^{-1}(\mathcal{B})$.) Alternatively, the lemma follows from monotonicity of function space, since REQ-STRICT implies $\mathcal{N} \subseteq \mathcal{S} \boxed{\to} \mathcal{N}$. □

Vouillon [Vou04] defines the function space as the set of *safe* terms which behave as a function. In our model instance, all functions will turn out to be safe, hence, we make this a requirement.

$$\text{REQ-FUN-SAFE} \qquad \mathcal{N} \boxed{\to} \mathcal{S} \subseteq \mathcal{S}$$

(This condition is one property of what Vaux [Vau04] calls a *stable pair* $(\mathcal{N}, \mathcal{S})$.)

**Lemma 3.26 (Functions are safe)** *If $\mathcal{N} \subseteq \mathcal{A}$ and $\mathcal{B} \subseteq \mathcal{S}$ then $\mathcal{A} \boxed{\to} \mathcal{B} \subseteq \mathcal{S}$.*

*Proof.* Since, the function space construction is contravariant on its domain and covariant on its codomain, the lemma follows directly from REQ-FUN-SAFE. □

Summing up, the requirements guarantee that function space remains within the type interval, i. e., $[\mathcal{N}, \mathcal{S}] \boxed{\to} [\mathcal{N}, \mathcal{S}] \subseteq [\mathcal{N}, \mathcal{S}]$.

### 3.4.1 Saturation

In this section, we axiomatize the evaluation relation $\rhd$ and define a generic notion of saturated set.

**Safe weak head reduction.** We assume a relation $\rhd$ on terms with the following properties.

| | | | |
|---|---|---|---|
| REQ-$\beta$ | $(\lambda x t)\, s$ | $\rhd\ [s/x]t$ | if $s \in \mathcal{S}$ |
| REQ-REC | $\mathsf{fix}^{\mu}_{n}\, s\, t_{1..n}\, v$ | $\rhd\ s\,(\mathsf{fix}^{\mu}_{n}\, s)\, t_{1..n}\, v$ | if $v \neq \mathsf{fix}^{\nu}_{n'}\, s'\, t_{1..n'}$ |
| REQ-COREC | $e(\mathsf{fix}^{\nu}_{n}\, s\, t_{1..n})$ | $\rhd\ e(s\,(\mathsf{fix}^{\nu}_{n}\, s)\, t_{1..n})$ | if $e \neq \mathsf{fix}^{\mu}_{n'}\, s'\, t_{1..n'}\, \_$ |
| REQ-ECXT | $E(t) \rhd E(t')$ | | if $t \rhd t'$ |
| REQ-TRANS | $\rhd$ is transitive | | |

The least relation satisfying these requirements would be *weak head reduction* if we omitted the side condition $s \in \mathcal{S}$ in REQ-$\beta$. In the presence of this condition, thinking of $\mathcal{S}$ as the set of strongly normalizing terms, two related terms behave equivalently with respect to strong normalization.[9] Consequently, we require that $\mathcal{S}$ is *closed* under $\rhd$ in both directions.

REQ-$\mathcal{S}$-CLOSED If $t \in \mathcal{S}$ and $t \rhd t'$ or $t \lhd t'$ then $t' \in \mathcal{S}$.

**Remark 3.27** Here, we give another justification that we need to prevent fixed-point unfolding in a term of the shape $\mathsf{fix}^{\mu}\, s\, (\mathsf{fix}^{\nu}\, s')$. Assume there were no side conditions on the reduction rules for fixed points. Let $s = \lambda\_.x$ and $e = \mathsf{fix}^{\mu}\, \mathsf{id}\, \_$. Then

$$\begin{array}{lll} e(s\,(\mathsf{fix}^{\nu}\, s)) & \longrightarrow & e(x) \text{ neutral, but} \\ e(\mathsf{fix}^{\nu}\, s) & \longrightarrow^{+} & e(\mathsf{fix}^{\nu}\, s) \text{ diverging.} \end{array}$$

Hence, the rolled and unrolled version do not behave the same, and at least the rule REQ-COREC would be unsound.

**Saturated sets.** A set $\mathcal{A}$ is *saturated*, $\mathcal{A} \in \mathsf{SAT}$, if $\mathcal{N} \subseteq \mathcal{A} \subseteq \mathcal{S}$ and $\mathcal{A}$ is closed under $\rhd$-reduction and -expansion.

**Remark 3.28** If $\mathcal{A}$ is a set of safe terms and closed under $\rhd$-reduction, then property REQ-$\mathcal{S}$-VAL is inherited from $\mathcal{S}$ to $\mathcal{A}$. This holds especially for $\mathcal{A} \in \mathsf{SAT}$.

**Lemma 3.29 (Function space is closed)** *If $\mathcal{B}$ is closed, then so is $\mathcal{A} \boxed{\to} \mathcal{B}$.*

[9] In the absence of the side condition, $[s/x]t$ could be strongly normalizing while $(\lambda x t)\, s$ is not (if $x \notin \mathsf{FV}(t)$). The side condition can also be meaningful in different contexts. E. g., if "safe" means "evaluating without error" [Vou04, VM04], and $s$ evaluates to an error, then the $\beta$-reduction $(\lambda x t)\, s \rhd [s/x]t$ is not semantics-preserving in call-by-value languages.

*Proof.* Let $r \in \mathcal{A} \boxrightarrow \mathcal{B}$. For $r' \rhd r$ we show $r' \in \mathcal{A} \boxrightarrow \mathcal{B}$ by assuming an arbitrary $s \in \mathcal{A}$ and deriving $r'\,s \in \mathcal{B}$. But this follows from $r\,s \in \mathcal{B}$, since both $r'\,s \rhd r\,s$ by REQ-ECXT and $\mathcal{B}$ is closed under $\rhd$-expansion. In the same way, closure under $\rhd$-reduction is inherited from $\mathcal{B}$ to $\mathcal{A} \boxrightarrow \mathcal{B}$. By the previous remark, we are done. □

**Corollary 3.30 (Function space is saturated)** *If $\mathcal{N} \subseteq \mathcal{A} \subseteq \mathcal{S}$ and $\mathcal{B} \in \mathsf{SAT}$ then $\mathcal{A} \boxrightarrow \mathcal{B} \in \mathsf{SAT}$.*

*Proof.* By the previous lemma, the function space $\mathcal{A} \boxrightarrow \mathcal{B}$ is closed. It is between $\mathcal{N}$ and $\mathcal{S}$ by lemmata 3.25 and 3.26. □

**Corollary 3.31 (Abstractions are functions)** *Let $\mathcal{A}, \mathcal{B} \in \mathsf{SAT}$. If $[s/x]t \in \mathcal{B}$ for all $s \in \mathcal{A}$, then $\lambda xt \in \mathcal{A} \boxrightarrow \mathcal{B}$.*

*Proof.* For $s \in \mathcal{A} \subseteq \mathcal{S}$ we have to show $(\lambda xt)\,s \in \mathcal{B}$. This follows from $(\lambda xt)\,s \rhd [s/x]t \in \mathcal{B}$, since $\mathcal{B}$ is closed under $\rhd$-expansion. □

### 3.4.2 Admissible Types for Recursion

In the last section, we have established that semantic types model abstraction and application. Now we turn our attention to the recursion combinators that extend the $\lambda$-calculus.

Let $\mathsf{O}$ be an initial segment of the ordinal numbers.

**Admissible semantic types for recursion.** The semantic type family $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ is *admissible for recursion on the $n+1$st argument* if

| | |
|---|---|
| ADM-$\mu$-SHAPE | $\mathcal{A}(\alpha) = \bigcap_{k \in K} (\mathcal{B}_{1..n}(k,\alpha) \boxrightarrow \mathcal{I}(k,\alpha) \boxrightarrow \mathcal{C}(k,\alpha))$ for some index set $K$ and $\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{I}, \mathcal{C} \in K \times \mathsf{O} \to \mathsf{SAT}$, |
| ADM-$\mu$-START | $\mathcal{I}(k,0) \subseteq \mathcal{N}$ for all $k \in K$, and |
| ADM-$\mu$-LIMIT | $\inf_{\alpha<\lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for all limits $0 \neq \lambda \in \mathsf{O}$. |

For the soundness of this criterion we need a recursive function applied to a corecursive value to be neutral.

$$\text{REQ-FIX}^{\mu}\text{FIX}^{\nu} \qquad \mathsf{fix}^{\mu}_{n}\, s\, t_{1..n}\, (\mathsf{fix}^{\nu}_{n'}\, s'\, t'_{1..n'}) \in \mathcal{N} \text{ if } s, \vec{t}, s', \vec{t'} \in \mathcal{S}$$

However, now there are *closed* neutral terms, which is quite uncommon. This means that semantically, all types are inhabited by closed terms, which in turn means that we cannot use this semantics to show that the type theory is consistent. More concretely, we cannot show that $\forall A.A$ does not have closed inhabitants. But this is not important for strong normalization.

A summary of accumulated requirements is given in Table 3.1.

| Cat. | Page | Requirement | |
|---|---|---|---|
| $\mathcal{N}$ | 62 | REQ-$\mathcal{N}$-SUB-$\mathcal{S}$ | $\mathcal{N} \subseteq \mathcal{S}$ |
| | 63 | REQ-STRICT | $\mathsf{Scxt}(\mathcal{N}) \subseteq \mathcal{N}$ |
| | 65 | REQ-FIX$^\mu$FIX$^\nu$ | $\mathsf{fix}^\mu_n\, s\, t_{1..n}(\mathsf{fix}^\nu_m\, s'\, t'_{1..m}) \in \mathcal{N}$ |
| $\mathcal{S}$ | 62 | REQ-$\mathcal{S}$-VAL | if $t \in \mathcal{S} \setminus \mathcal{N}$ then $t \rhd v$ |
| | 63 | REQ-FUN-SAFE | $\mathcal{N} \boxed{\to} \mathcal{S} \subseteq \mathcal{S}$ |
| | 64 | REQ-$\mathcal{S}$-CLOSED | $\mathcal{S}\rhd \subseteq \mathcal{S}$, $\rhd\mathcal{S} \subseteq \mathcal{S}$ |
| $\rhd$ | 64 | REQ-$\beta$ | $(\lambda x t)\, s \rhd [s/x]t$ |
| | 64 | REQ-REC | $\mathsf{fix}^\mu_n\, s\, t_{1..n}\, v \rhd s\,(\mathsf{fix}^\mu_n\, s)\, t_{1..n}\, v$ |
| | 64 | REQ-COREC | $e(\mathsf{fix}^\nu_n\, s\, t_{1..n}) \rhd e(s\,(\mathsf{fix}^\nu_n\, s)\, t_{1..n})$ |
| | 64 | REQ-ECXT | if $t \rhd t'$ then $E(t) \rhd E(t')$ |
| | 64 | REQ-TRANS | $\rhd$ is transitive |

Table 3.1: Summary of requirements.

**Lemma 3.32 (Recursion is a function)** *Let $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ be admissible for recursion on the $n+1$st argument. If $s \in \mathcal{A}(\alpha) \boxed{\to} \mathcal{A}(\alpha+1)$ for all $\alpha+1 \in \mathsf{O}$, then $\mathsf{fix}^\mu_n\, s \in \mathcal{A}(\beta)$ for all $\beta \in \mathsf{O}$.*

*Proof.* By transfinite induction on $\beta \in \mathsf{O}$.

The limit case is a direct consequence of ADM-$\mu$-LIMIT.

For the remaining cases, using ADM-$\mu$-SHAPE, assume $k \in K$, $t_i \in \mathcal{B}_i(k,\beta)$ for $1 \le i \le n$, and $r \in \mathcal{I}(k,\beta)$ and show $\mathsf{fix}^\mu_n\, s\, \vec{t}\, r \in \mathcal{C}(k,\beta)$. Since all $t_i$ are safe and $s$ is safe by assumption, $e := \mathsf{fix}^\mu_n\, s\, \vec{t}\, \_$ is a safe evaluation frame. Hence, if $r \in \mathcal{N}$, then $e(r) \in \mathcal{N} \subseteq \mathcal{C}(k,\beta)$, and in case $\beta = 0$ we are done, since, by ADM-$\mu$-START, the set $\mathcal{I}(k,0)$ contains only neutral terms.

Finally, we consider $\beta = \alpha+1$ and $r \rhd v$. If $v = \mathsf{fix}^\nu_{n'}\, s'\, t'_{1..n'}$, then $e(v) \in \mathcal{N} \subseteq \mathcal{C}(k,\alpha+1)$ by REQ-FIX$^\mu$FIX$^\nu$. Otherwise, note that $s\,(\mathsf{fix}^\mu_n\, s) \in \mathcal{A}(\alpha+1)$ by assumption and induction hypothesis. Since $e(r) \rhd e(v) \rhd s\,(\mathsf{fix}^\mu_n\, s)\, \vec{t}\, v \in \mathcal{C}(k,\alpha+1)$, we conclude by closure of $\mathcal{C}(k,\alpha+1)$ under $\rhd$-expansion. □

### 3.4.3 Refined Saturation

In this section, we will refine our notion of saturation, to prepare for the treatment of corecursion.

**Orthogonality.** An *orthogonality* relation $t \perp E$ between term $t$ and evaluation context $E$ is given by

$$t \perp E \;:\Longleftrightarrow\; E(t) \in \mathcal{S}.$$

For a set $\mathcal{A}$ of terms we can compute its *orthogonal*, the set $\mathcal{E}$ of evaluation context in which all terms $t \in A$ *behave well*, i. e., evaluate safely. In another iteration, we take the orthogonal of this set $\mathcal{E}$ of evaluation contexts, i. e., all

terms which behave well in all contexts $E \in \mathcal{E}$, and arrive at the *biorthogonal* of the original term set. The biorthogonal is closed and can serve as a semantical type—this idea is implicit in Girard's semantics of linear logic and resurfaces in Ludics [Gir01]. Girard has not written up the idea clearly himself, but it slowly spread in the (French) research community. Parigot [Par97, page 1469] defines a saturated set (which is a semantical type in the context of strong normalization proofs) as a set $\mathcal{A}$ of terms that are strongly normalizing when applied any list $\vec{s} \in \mathcal{E}$ of strongly normalizing terms contained in a certain set $\mathcal{E}$ of term lists. Applying to a list of terms is a special case of putting into an evaluation context—hence, Parigot effectively uses Girard's technique (without reference), albeit not the intuition of orthogonality. Matthes [Mat05] builds on the work of Parigot and extends it to arbitrary evaluation contexts. A nice presentation of orthogonality can be found in the recent work of Vouillon [Vou04] and Melliés [VM04]—-this is where I draw my terminology from. However, they do not show strong normalization; I had to add the concept of safe evaluation context to make it work in my case. Recently, Lindley and Stark [LS05] have used biorthogonality to show strong normalization of the computational $\lambda$-calculus and they refer to Andrew Pitts. In personal communication, Andrew Pitts claimed to have invented the technique independently of the French School.

After having given honor to our teachers, we now can define:

$$\begin{array}{lcll} \mathcal{E}^\perp & := & \{t \in \mathsf{Tm} \mid t \perp E \text{ for all } E \in \mathcal{E}\} & \text{for } \mathcal{E} \subseteq \mathsf{Ecxt} \\ \mathsf{SAT}^\perp & := & \{\mathcal{E} \mid \{\mathsf{Id}\} \subseteq \mathcal{E} \subseteq \mathsf{Scxt}\} & \\ \mathsf{SAT} & := & \{\mathcal{E}^\perp \mid \mathcal{E} \in \mathsf{SAT}^\perp\} & \\ \mathcal{N} & := & \mathsf{Scxt}^\perp & \end{array}$$

Note that $(\_)^\perp$ is an antitonic operation. Since $\mathcal{S} = \{\mathsf{Id}\}^\perp$, it is clear that $\mathcal{S}$ is the greatest and $\mathcal{N}$ the least saturated set.[10] By definition, the set of neutral terms $\mathcal{N}$ also fulfills the requirements REQ-$\mathcal{N}$-SUB-$\mathcal{S}$ and REQ-STRICT. The requirement REQ-FIX$^\mu$FIX$^\nu$ is now phrased in terms of $\mathcal{S}$:

$$\text{REQ-FIX}^\mu\text{FIX}^\nu \qquad E(\mathsf{fix}^\mu_n\, s\, t_{1..n}\, (\mathsf{fix}^\nu_{n'}\, s'\, t'_{1..n'})) \in \mathcal{S} \text{ for all } E \in \mathsf{Scxt}$$

**Lemma 3.33 ($\mathcal{E}^\perp$ is closed)** *If for $t \in \mathcal{E}^\perp$ it holds that $t \rhd t'$ or $t' \rhd t$, then $t' \in \mathcal{E}^\perp$.*

*Proof.* We need to show $E(t') \in \mathcal{S}$ for all $E \in \mathcal{E}$. This holds since $\rhd$ is closed under evaluation contexts and $\mathcal{S}$ is closed under $\rhd$. □

**Corollary 3.34** *The new notion of* SAT *is a refinement of the old one.*

We need to show that the function space is still an operation on SAT.

**Lemma 3.35 (Function space in SAT)** *If $\mathcal{N} \subseteq \mathcal{A} \subseteq \mathcal{S}$ and $\mathcal{B} \in \mathsf{SAT}$ then $\mathcal{A} \boxrightarrow \mathcal{B} \in \mathsf{SAT}$.*

[10] The converse, $\mathcal{S}^\perp = \{\mathsf{Id}\}$ is not true if we let $\mathcal{S}$ be the set SN of strongly normalizing terms: Because $r \in \mathsf{SN}$ implies $r\, x \in \mathsf{SN}$ for any variable $x$, we have $(\_\, x) \in \mathsf{SN}^\perp$.

*Proof.* Let $\mathcal{E}^\perp = \mathcal{B}$. By definition, $\mathcal{A} \boxed{\to} \mathcal{B} = \hat{\mathcal{E}}^\perp$ with $\hat{\mathcal{E}} := \{E \circ (\_\, s) \mid E \in \mathcal{E} \text{ and } s \in \mathcal{A}\} \subseteq \mathsf{Scxt}$. Since $(\mathcal{A} \boxed{\to} \mathcal{B}) \cap \mathcal{S} = \mathcal{A} \boxed{\to} \mathcal{B}$ by REQ-FUN-SAFE, we have $\mathcal{A} \boxed{\to} \mathcal{B} = \hat{\mathcal{E}}^\perp \cap \{\mathsf{Id}\}^\perp = (\hat{\mathcal{E}} \cup \{\mathsf{Id}\})^\perp$. The last step is formally proven in Lemma 3.38. □

For the semantical justification of the typing rule for corecursion, which will be given in the next section, we require $\mathsf{fix}^\nu$ to be safe, if applied to a number of safe arguments up to its arity.

$$\text{REQ-FIX}^\nu \qquad \mathsf{fix}^\nu_n \in \mathcal{S}^{n+1} \boxed{\to} \mathcal{S}.$$

The remaining requirements are displayed in Table 3.2.

| Cat. | Page | Requirement | |
|---|---|---|---|
| $\mathcal{S}$ | 67 | REQ-FIX$^\mu$FIX$^\nu$ | $\mathsf{Scxt}(\mathsf{fix}^\mu_n\, s\, t_{1..n}(\mathsf{fix}^\nu_m\, s'\, t'_{1..m})) \subseteq \mathcal{S}$ |
| | 62 | REQ-$\mathcal{S}$-VAL | if $t \in \mathcal{S} \setminus \mathcal{N}$ then $t \rhd v$ |
| | 63 | REQ-FUN-SAFE | $\mathcal{N} \boxed{\to} \mathcal{S} \subseteq \mathcal{S}$ |
| | 64 | REQ-$\mathcal{S}$-CLOSED | $\mathcal{S}\rhd \subseteq \mathcal{S}$, $\rhd\mathcal{S} \subseteq \mathcal{S}$ |
| | 68 | REQ-FIX$^\nu$ | $\mathsf{fix}^\nu_n \in \mathcal{S}^{n+1} \boxed{\to} \mathcal{S}$ |
| $\rhd$ | 64 | REQ-$\beta$ | $(\lambda x t)\, s \rhd [s/x]t$ |
| | 64 | REQ-REC | $\mathsf{fix}^\mu_n\ s\, t_{1..n}\, v \rhd s\, (\mathsf{fix}^\mu_n\ s)\, t_{1..n}\, v$ |
| | 64 | REQ-COREC | $e(\mathsf{fix}^\nu_n\ s\, t_{1..n}) \rhd e(s\, (\mathsf{fix}^\nu_n\ s)\, t_{1..n})$ |
| | 64 | REQ-ECXT | if $t \rhd t'$ then $E(t) \rhd E(t')$ |
| | 64 | REQ-TRANS | $\rhd$ is transitive |

Table 3.2: Summary of refined requirements.

A bit surprisingly, corecursive values can be found in $\mathcal{N}$:

**Lemma 3.36 (Neutral corecursive values)** *neutral corecursive values If $s, t_{1..n} \in \mathcal{S}$ and $s\, (\mathsf{fix}^\nu_n\ s)\, t_{1..n} \in \mathcal{N}$ then $v := \mathsf{fix}^\nu_n\ s\, t_{1..n} \in \mathcal{N}$.*

*Proof.* We have to show that $E(v) \in \mathcal{S}$ for all $E \in \mathsf{Scxt}$. In case $E = \mathsf{Id}$, we have $v \in \mathcal{S}$ by REQ-FIX$^\nu$. Otherwise $E = E' \circ e$. Either $e$ is a recursive function, then $E'(e(v))$ is safe by REQ-FIX$^\mu$FIX$^\nu$, or $e$ is an applicative evaluation frame and $E'(e(v)) \rhd E'(e(s\, (\mathsf{fix}^\nu_n\ s)\, t_{1..n}))$ which is neutral by assumption and REQ-STRICT, hence, $E'(e(v)) \in \mathcal{S}$ by Lemma 3.33. □

An example for such a neutral term is $\mathsf{fix}^\nu_0\, \lambda\_x$

### 3.4.4 Admissible Types for Corecursion

**Admissible semantic types for corecursion.** The semantic type family $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ is *admissible for corecursion with n arguments* if

| | |
|---|---|
| ADM-$\nu$-SHAPE | $\mathcal{A}(\alpha) = \bigcap_{k \in K}(\mathcal{B}_{1..n}(k, \alpha) \boxed{\to} \mathcal{C}(k, \alpha))$ for some index set $K$ and $\mathcal{B}_{1..n}, \mathcal{C} \in K \times \mathsf{O} \to \mathsf{SAT}$, |
| ADM-$\nu$-START | $\mathcal{S} \subseteq \mathcal{C}(k, 0)$ for all $k \in K$, and |
| ADM-$\nu$-LIMIT | $\inf_{\alpha<\lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for all limits $0 \neq \lambda \in \mathsf{O}$. |

**Lemma 3.37 (Corecursion is a function)** *Let* $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ *be admissible for corecursion with n arguments. If* $s \in \mathcal{A}(\alpha) \boxed{\to} \mathcal{A}(\alpha+1)$ *for all* $\alpha+1 \in \mathsf{O}$*, then* $\mathsf{fix}_n^\nu\, s \in \mathcal{A}(\beta)$ *for all* $\beta \in \mathsf{O}$*.*

*Proof.* By transfinite induction on $\beta \in \mathsf{O}$.

The limit case is a direct consequence of ADM-$\nu$-LIMIT.

For the remaining cases, using ADM-$\nu$-SHAPE, assume $k \in K$, $t_i \in \mathcal{B}_i(k,\beta)$ for $1 \le i \le n$ and show $v := \mathsf{fix}_n^\nu\, s\, \vec{t} \in \mathcal{C}(k,\beta)$. Note that $s$ and $t_i$ are safe, hence, $v \in \mathcal{S}$ by REQ-FIX$^\nu$. Since $\mathcal{S} \subseteq \mathcal{C}(k,0)$, we are done in case of $\beta = 0$.

For case $\beta = \alpha+1$, let $\mathcal{E} \subseteq \mathsf{Scxt}$ be a set of evaluation contexts such that $\mathcal{E}^\perp = \mathcal{C}(k,\alpha+1)$. Assume an arbitrary $E \in \mathcal{E}$ and show $E(v) \in \mathcal{S}$. If $E = \mathsf{Id}$, there is nothing new to show, otherwise $E = E' \circ e$. If $e = \mathsf{fix}_{n'}^\mu\, s'\, t_{1..n'}\, \_$, then $E(v) \in \mathcal{S}$ by REQ-FIX$^\mu$FIX$^\nu$. Otherwise, note that $s\,(\mathsf{fix}_n^\nu\, s) \in \mathcal{A}(\alpha+1)$ by assumption and induction hypothesis. Since $E(v) \rhd E(s\,(\mathsf{fix}_n^\nu\, s)\,\vec{t}) \in \mathcal{S}$, we conclude by closure of $\mathcal{S}$. □

### 3.4.5 Lattice of Saturated Sets

In this section we will establish that SAT is a complete lattice of sets. This will justify the interpretation of universal quantification as intersection and enable the choice $[\![*]\!] := \mathsf{SAT}$. Since $\mathcal{A} \subseteq \mathcal{S}$ for $\mathcal{A} \in \mathsf{SAT}$, we set $\top^* := \mathcal{S}$.

**De Morgan laws.** The orthogonality operation $\mathcal{E}^\perp$ is an intuitionistic negation. The following de Morgan laws hold:

**Lemma 3.38 (De Morgan)** *Let I some non-empty index set and* $\mathcal{E}_i \subseteq \mathsf{Ecxt}$ *for all* $i \in I$*. Then,*

$$\begin{array}{lcl} \bigcap_{i\in I} \mathcal{E}_i^\perp & = & (\bigcup_{i\in I} \mathcal{E}_i)^\perp, \text{ and} \\ \bigcup_{i\in I} \mathcal{E}_i^\perp & \subseteq & (\bigcap_{i\in I} \mathcal{E}_i)^\perp. \end{array}$$

*Proof.*

$$\begin{array}{lcll} t \in \bigcap_{i\in I} \mathcal{E}_i^\perp & \iff & \text{for all } i \in I, t \perp E \text{ for all } E \in \mathcal{E}_i \\ & \iff & t \perp E \text{ for all } E \in \bigcup_{i\in I} \mathcal{E}_i \\ & \iff & t \in (\bigcup_{i\in I} \mathcal{E}_i)^\perp \end{array}$$

$$\begin{array}{lcll} t \in \bigcup_{i\in I} \mathcal{E}_i^\perp & \implies & \text{for some } i \in I, t \perp E \text{ for all } E \in \mathcal{E}_i \\ & \implies & t \perp E \text{ for all } E \in \bigcap_{i\in I} \mathcal{E}_i \\ & \implies & t \in (\bigcap_{i\in I} \mathcal{E}_i)^\perp \end{array}$$

□

The de Morgan laws can also be read topologically: Let the closed term sets be the saturated ones. Then intersections of closed sets (even infinite intersections), are still closed, whereas this need not hold for unions. In the following we show by example that the missing de Morgan law does indeed not hold.

**Lemma 3.39 (Orthogonality not classical)** *Let the set of safe terms $\mathcal{S}$ contain exactly the strongly normalizing ones. There are $\mathcal{E}_1, \mathcal{E}_2 \subseteq \mathsf{Ecxt}$ such that $(\mathcal{E}_1 \cap \mathcal{E}_2)^\perp \not\subseteq \mathcal{E}_1^\perp \cup \mathcal{E}_2^\perp$.*

*Proof.* Let $\delta := \lambda x.\, x\, x$ and $\delta' := \lambda k.\, k\, \delta$, $\mathcal{E}_1 := \{\mathsf{Id}, (\_\, \delta)\}$, and $\mathcal{E}_2 := \{\mathsf{Id}, (\_\, \delta')\}$. Then $(\mathcal{E}_1 \cap \mathcal{E}_2)^\perp = \{\mathsf{Id}\}^\perp = \mathcal{S}$, and the normal term $\delta$ is safe. But $\delta \notin \mathcal{E}_1^\perp$, since $\delta\, \delta \longrightarrow \delta\, \delta$, and $\delta \notin \mathcal{E}_2^\perp$, since $\delta\, \delta' \longrightarrow \delta'\, \delta' \longrightarrow \delta'\, \delta \longrightarrow \delta\, \delta$. □

**Remark 3.40 (Maximal sets of contexts)** This counterexample is not completely satisfying, because $\mathcal{E}_1^\perp = \mathcal{E}_2^\perp$, hence, $\mathcal{E}_1$ and $\mathcal{E}_2$ are just incompatible representations of the same saturated set $\mathcal{A}$. This suggests that the representation of a saturated set $\mathcal{A}$ should be the biggest set of contexts $\mathcal{E}$ such that $\mathcal{E}^\perp = \mathcal{A}$. Whether such maximal sets would satisfy the missing de Morgan law, is unclear to me and left as an open question.

As a consequence of the de Morgan laws, saturated sets form a complete lattice.

**Corollary 3.41 ($\mathsf{SAT}$ closed under intersection)** *If $\mathfrak{A} \subseteq \mathsf{SAT}$, then $\bigcap \mathfrak{A} \in \mathsf{SAT}$.*

*Proof.* If $\mathfrak{A}$ is empty, we set $\bigcap \mathfrak{A} = \mathcal{S}$. Otherwise, let $\mathcal{E}_0 := \bigcup\{\mathcal{E} \mid \mathcal{E}^\perp \in \mathfrak{A}\}$. Then $\mathcal{E}_0 \in \mathsf{SAT}^\perp$ and $\mathcal{E}_0^\perp = \mathfrak{A}$. □

Since $\mathcal{N}$ is the least saturated set, $\perp^* = \bigcap \mathsf{SAT} = \mathcal{N}$.

It follows that $(\llbracket * \rrbracket, \bigsqcap^*, \top^*) := (\mathsf{SAT}, \bigcap, \mathcal{S})$ is a complete lattice of sets and can serve as basis for our model of kinds, constructors, and subtyping as developed in Section 2.3. Soundness of the subtyping rule TY-SUB is an immediate consequence of Theorem 2.27. Setting the interpretation of quantification to

$$\llbracket \forall_\kappa \rrbracket_\theta\,(\mathcal{F}) = \mathsf{Sem}(\forall_\kappa)(\mathcal{F}) := \bigcap_{\mathcal{G} \in [\kappa]} \mathcal{F}(\mathcal{G}),$$

soundness of the generalization and instantiation rules TY-GEN and TY-INST, follows immediately.

**Supremum.** The supremum of the lattice $\llbracket * \rrbracket$ is formally defined as

$$\begin{aligned} \bigsqcup^*_{i \in I} \mathcal{A}_i &:= \bigsqcap^* \{\mathcal{A} \in \mathsf{SAT} \mid \mathcal{A} \sqsupseteq^* \mathcal{A}_i \text{ for all } i \in I\} \\ &= \bigcap \{\mathcal{E}^\perp \mid \mathcal{E} \in \mathsf{SAT}^\perp \text{ and } \mathcal{E}^\perp \supseteq \mathcal{A}_i \text{ for all } i \in I\} \end{aligned}$$

Vouillon [Vou04] presents another definition of the union of semantical types. Let

$$\mathcal{A}^\perp := \{E \in \mathsf{Scxt} \mid t \perp E \text{ for all } t \in \mathcal{A}\}$$

for a set of terms $\mathcal{A}$. For reasons of symmetry, the de Morgan laws hold for $\mathcal{A}^\perp$ as for $\mathcal{E}^\perp$. Since for $\mathcal{A} \subseteq \mathsf{Tm}$ and $\mathcal{E} \subseteq \mathsf{Scxt}$,

$$\mathcal{A}^\perp \supseteq \mathcal{E} \iff \mathcal{A} \subseteq \mathcal{E}^\perp,$$

the two orthogonality operations form a *Galois connection* between sets of terms $(\mathcal{P}(\mathsf{Tm}), \subseteq)$ ordered by inclusion and sets of safe evaluation contexts $(\mathsf{Scxt}, \supseteq)$ ordered by the superset relation (see Appendix C). Hence, the function which maps $\mathcal{A} \subseteq \mathsf{Tm}$ to

$$\overline{\mathcal{A}} := \mathcal{A}^{\perp\perp}$$

is a *closure operator* on sets of terms.

**Lemma 3.42** *If $\mathcal{A} \subseteq \mathcal{S}$ then $\mathcal{A}^{\perp} \in \mathsf{SAT}^{\perp}$ and $\overline{\mathcal{A}} \in \mathsf{SAT}$.*

*Proof.* If $\mathcal{A} \subseteq \mathcal{S}$ then $\mathsf{Id} \in \mathcal{A}^{\perp}$. Furthermore $\mathcal{A}^{\perp} \subseteq \mathsf{Scxt}$ by definition. □

We can now define union of term sets $\mathcal{A} \in \mathfrak{A}$ as follows:

$$\boxed{\bigcup}\mathfrak{A} := \overline{\bigcup \mathfrak{A}}$$

**Remark 3.43** The closure operation plays a crucial role when we construct the semantics of inductive types. Consider the approximations $\mathcal{N}at^{\alpha}$ of the semantical type of natural numbers. For $\alpha < \omega$, $\mathcal{N}at^{\alpha}$ contains only bounded numbers, which can be characterized by a set of *finite observations E*. But $\mathcal{N}at^{\omega}$ cannot be characterized by finite observations alone; recursive evaluation contexts (like the identity function on $\mathsf{Nat}$) are required. These come in through the closure operation that is involved in the supremum: $\mathcal{N}at^{\omega} = \boxed{\bigcup}_{\alpha<\omega} \mathcal{N}at^{\alpha}$.

**Lemma 3.44** *Let $I$ be some index set and $\mathcal{A}_i \subseteq \mathcal{S}$ for all $i \in I$. Then*

$$\bigsqcup^{*}_{i\in I} \mathcal{A}_i = \boxed{\bigcup}_{i\in I} \mathcal{A}_i.$$

*Proof.* Note that (*) $(\bigcup_{i\in I} \mathcal{A}_i)^{\perp} \in \mathsf{SAT}^{\perp}$ by Lemma 3.42.

$$\begin{array}{rcll}
\bigsqcup^{*}_{i\in I} \mathcal{A}_i & = & \bigcap\{\mathcal{E}^{\perp} \mid \mathcal{E} \in \mathsf{SAT}^{\perp} \text{ and } \mathcal{E}^{\perp} \supseteq \mathcal{A}_i \text{ for all } i \in I\} & \text{definition} \\
& = & \bigcap\{\mathcal{E}^{\perp} \mid \mathcal{E} \in \mathsf{SAT}^{\perp} \text{ and } \mathcal{E} \subseteq \mathcal{A}_i^{\perp} \text{ for all } i \in I\} & \text{Galois conn.} \\
& = & \bigcap\{\mathcal{E}^{\perp} \mid \mathcal{E} \in \mathsf{SAT}^{\perp} \text{ and } \mathcal{E} \subseteq \bigcap_{i\in I} \mathcal{A}_i^{\perp}\} & \\
& = & \big(\bigcup\{\mathcal{E} \mid \mathcal{E} \in \mathsf{SAT}^{\perp} \text{ and } \mathcal{E} \subseteq (\bigcup_{i\in I} \mathcal{A}_i)^{\perp}\}\big)^{\perp} & \text{de Morgan} \\
& = & (\bigcup_{i\in I} \mathcal{A}_i)^{\perp\perp} & (*) \\
& = & \boxed{\bigcup}_{i\in I} \mathcal{A}_i & \text{definition}
\end{array}$$

□

The supremum of saturated term sets is the closure of their set-theoretical union, but not necessarily identical to it, which is the case for the more traditional notion of saturation as found in Section 3.4.1 and in, e. g., Luo [Luo90], Altenkirch [Alt93, AA00], Matthes [Mat98, Mat00], Barthe et al. [BFG$^{+}$04], and previous work of the author [Abe03, Abe04].

We have now given a semantical justification for each typing rule. We will assemble these pieces in the next section, where we give a term model of our calculus.

## 3.5 Soundness of Typing

### 3.5.1 Admissible Types for Recursion, Syntactically

Having a semantical characterization of admissible types for recursion we can turn it into a syntactical one. For now, we give a simple characterization: A recursive function goes from an inductive type to a result type which depends *monotonically* on the ordinal index. This is the shape of types from Barthe et al. [BFG+04] and from previous work of the author [Abe04]. Examples would be $\mathsf{Nat}^\iota \to A$ and $\mathsf{List}^\iota\, A \to \mathsf{Nat}^\iota$, where $A$ does not depend on the ordinal index $\iota$. Excluded would be types like $\mathsf{Nat}^\iota \to \mathsf{Stream}^\iota\, A$ (since $\mathsf{Stream}^\iota\, A$ is antitone in $\iota$) or $\mathsf{Nat}^\iota \to \mathsf{Nat}^\iota \to \mathsf{Nat}^\iota$, the latter being the type of the maximum function. Since we also have inductive *constructors*, the general shape of the recursive argument is $\mu^\iota F \vec{H}$. Then we allow the recursive function to be a natural transformation, i. e., of the shape $\forall X : \kappa'.\, \mu^\iota F\, (H_1\, X) \ldots (H_n\, X) \to G\, X$. Of course, we can quantify over several constructor variables, arriving at the shape $\forall \vec{X} : \vec{\kappa}'.\, \mu^\iota F\, (H_1\, \vec{X}) \ldots (H_n\, \vec{X}) \to G\, \vec{X}$, or, using the abbreviation for natural transformations, at $(\mu^\iota F) \circ \vec{H} \Rightarrow G$. As a final relaxation, we do not require the inductive argument to be the first one. The function may have $n$ parameters before the inductive argument, but their types must be *antitone* in the ordinal index.

$$
\begin{aligned}
\Gamma \vdash A\ \mathsf{fix}^\mu_n\text{-}\mathsf{adm} \quad :\Longleftrightarrow \quad & \Gamma, \iota : \mathsf{ord} \vdash A\, \iota = (\vec{G}, (\mu^\iota F) \circ \vec{H} \Rightarrow G) : * \quad (\iota \notin \mathsf{FV}(A)) \\
& \text{for some } F, G, \vec{G}, \vec{H} \text{ with } |\vec{G}| = n \text{ and} \\
& \Gamma \vdash F : +\kappa \to \kappa \text{ for some pure } \kappa = \vec{p}\vec{\kappa} \to *, \\
& \Gamma, \iota : +\mathsf{ord} \vdash G : \kappa' \text{ for some } \kappa' = \circ\vec{\kappa}' \to *, \\
& \Gamma, \iota : -\mathsf{ord} \vdash G_i : \kappa' \text{ for } 1 \le i \le n, \text{ and} \\
& \Gamma \vdash H_i : \circ\vec{\kappa}' \to \kappa_i \text{ for } 1 \le i \le |\vec{\kappa}|.
\end{aligned}
$$

We now prove this criterion sound. To this end, we will make use of the semantics of declarative kinding and equality (see Section 2.3).

**Lemma 3.45 (Soundness of admissible recursion types)** *If $\Gamma \vdash A\ \mathsf{fix}^\mu_n\text{-}\mathsf{adm}$ and $\theta \in [\![\Gamma]\!]$, then $[\![A]\!]_\theta$ is admissible for recursion on the $n+1$st argument, where $\mathsf{O}$ may be any initial segment of $[\![\mathsf{ord}]\!]$.*

*Proof.* Since $\Gamma, \iota : \mathsf{ord} \vdash A\, \iota = (\vec{G}, (\mu^\iota F) \circ \vec{H} \Rightarrow G) : *$ and $\iota \notin \mathsf{FV}(A)$, we know by Thm. 2.27 that for each $\alpha \in \mathsf{O}$,

$$[\![A\, \iota]\!]_{\theta[\iota \mapsto \alpha]} = [\![A]\!]_\theta(\alpha) = [\![\vec{G}, (\mu^\iota F) \circ \vec{H} \Rightarrow G]\!]_{\theta[\iota \mapsto \alpha]} \in [\![*]\!] = \mathsf{SAT}.$$

Hence, $\mathcal{A} := [\![A]\!]_\theta \in \mathsf{O} \to \mathsf{SAT}$. In the following, we verify the conditions ADM-$\mu$-SHAPE, ADM-$\mu$-START, and ADM-$\mu$-LIMIT.

ADM-$\mu$-SHAPE Show $\mathcal{A}(\alpha) = \bigcap_{k\in K} \mathcal{B}_{1..n}(k,\alpha) \boxrightarrow \mathcal{I}(k,\alpha) \boxrightarrow \mathcal{C}(k,\alpha)$. We set

$$\begin{array}{lcll} K & := & [\![\kappa'_1]\!] \times \cdots \times [\![\kappa'_m]\!] & \text{where } m := |\vec{\kappa}'|, \\ \mathcal{B}_i(\vec{\mathcal{X}},\alpha) & := & [\![G_i]\!]_{\theta[\iota\mapsto\alpha]}\vec{\mathcal{X}} & \text{for } 1 \le i \le n, \\ \mathcal{I}(\vec{\mathcal{X}},\alpha) & := & (\mathsf{I}_\alpha\, [\![F]\!]_\theta\, \bot^\kappa)([\![H_1]\!]_\theta\vec{\mathcal{X}}) \ldots ([\![H_l]\!]_\theta\vec{\mathcal{X}}) & \text{where } l := |\vec{\kappa}|, \text{ and} \\ \mathcal{C}(\vec{\mathcal{X}},\alpha) & := & [\![G]\!]_{\theta[\iota\mapsto\alpha]}\vec{\mathcal{X}}. & \end{array}$$

ADM-$\mu$-START Show $\mathcal{I}(\vec{\mathcal{X}},0) \subseteq \mathcal{N}$. This is clear since $\mathsf{I}_0\, [\![F]\!]_\theta\, \bot^\kappa = \bot^\kappa$ and $\bot^\kappa\, \mathcal{H}_1 \ldots \mathcal{H}_l = \bot^* = \mathcal{N}$ for any $\vec{\mathcal{H}}$.

ADM-$\mu$-LIMIT Show $\inf_\lambda \mathcal{A} \subseteq \mathcal{A}(\lambda)$. We assume $f \in \inf_\lambda \mathcal{A}$ and prove

$$f \in \mathcal{B}_{1..n}(\vec{\mathcal{X}},\lambda) \boxrightarrow \mathcal{I}(\vec{\mathcal{X}},\lambda) \boxrightarrow \mathcal{C}(\vec{\mathcal{X}},\lambda).$$

To this end, fix arbitrary $t_i \in \mathcal{B}_i(\vec{\mathcal{X}},\lambda)$ and $r \in \mathcal{I}(\vec{\mathcal{X}},\lambda)$. By Lemma 3.15, the inductive type at a limit $\mathcal{I}(\vec{\mathcal{X}},\lambda) = \sup_{\alpha<\lambda} \mathcal{I}(\vec{\mathcal{X}},\alpha)$, hence $r \in \mathcal{I}(\vec{\mathcal{X}},\alpha)$ for some $\alpha < \lambda$. Since $\Gamma, \iota : -\mathsf{ord} \vdash G_i : \kappa'$, the families $\mathcal{B}_i(\vec{\mathcal{X}},\_)$ are antitone, hence $t_i \in \mathcal{B}_i(\vec{\mathcal{X}},\alpha)$. Together, $f\,\vec{t}\,r \in \mathcal{C}(\vec{\mathcal{X}},\alpha) \subseteq \mathcal{C}(\vec{\mathcal{X}},\lambda)$ by monotonicity of $\mathcal{C}(\vec{\mathcal{X}},\_)$, which we obtain from $\Gamma, \iota : +\mathsf{ord} \vdash G : \kappa'$. □

### 3.5.2 Admissible Types for Corecursion, Syntactically

Similarly to the admissible types for recursion one can motivate the ones for corecursion. The most important requirement is that the result type of a corecursive function must be coinductive. In essence, our rule is the one of Barthe et al. [BFG+04], lifted to coinductive constructors and natural transformations.

$$\begin{array}{lcll} \Gamma \vdash A\ \mathsf{fix}^\nu_n\text{-}\mathsf{adm} & :\Longleftrightarrow & \Gamma, \iota : \mathsf{ord} \vdash A\,\iota = (\vec{G} \Rightarrow (\nu^\iota F) \circ \vec{H}) : * \quad (\iota \notin \mathsf{FV}(A)) & \\ & & \text{for some } F, \vec{G}, \vec{H} \text{ with } |\vec{G}| = n \text{ and} & \\ & & \Gamma \vdash F : +\kappa \to \kappa \text{ for some pure } \kappa = \vec{p}\vec{\kappa} \to *, & \\ & & \Gamma, \iota : -\mathsf{ord} \vdash G_i : \kappa' \text{ (all } i\text{) for some } \kappa' = \circ\vec{\kappa}' \to *, \text{ and} & \\ & & \Gamma \vdash H_i : \circ\vec{\kappa}' \to \kappa_i \text{ for } 1 \le i \le |\vec{\kappa}|. & \end{array}$$

**Lemma 3.46 (Soundness of admissible corecursion types)** *If $\Gamma \vdash A\ \mathsf{fix}^\nu_n$-adm and $\theta \in [\![\Gamma]\!]$, then $[\![A]\!]_\theta$ is admissible for corecursion with n arguments (where $\mathsf{O}$ may be any initial segment of $[\![\mathsf{ord}]\!]$).*

*Proof.* Since $\Gamma, \iota : \mathsf{ord} \vdash A\,\iota = (\vec{G} \Rightarrow (\nu^\iota F) \circ \vec{H}) : *$ and $\iota \notin \mathsf{FV}(A)$, we know by Thm. 2.27 that for each $\alpha \in \mathsf{O}$,

$$[\![A\,\iota]\!]_{\theta[\iota\mapsto\alpha]} = [\![A]\!]_\theta(\alpha) = [\![\vec{G} \Rightarrow (\nu^\iota F) \circ \vec{H}]\!]_{\theta[\iota\mapsto\alpha]} \in [\![*]\!] = \mathsf{SAT}.$$

Hence, $\mathcal{A} := [\![A]\!]_\theta \in \mathsf{O} \to \mathsf{SAT}$. In the following, we verify the conditions ADM-$\nu$-SHAPE, ADM-$\nu$-START, and ADM-$\nu$-LIMIT.

ADM-$\nu$-SHAPE Show $\mathcal{A}(\alpha) = \bigcap_{k\in K} \mathcal{B}_{1..n}(k,\alpha) \boxed{\to} \mathcal{C}(k,\alpha)$. We set

$$\begin{array}{lcll} K & := & [\![\kappa'_1]\!] \times \cdots \times [\![\kappa'_m]\!] & \text{where } m := |\vec{\kappa}'|, \\ \mathcal{B}_i(\vec{\mathcal{X}},\alpha) & := & [\![G_i]\!]_{\theta[\iota\mapsto\alpha]}\vec{\mathcal{X}} & \text{for } 1 \le i \le n, \text{ and} \\ \mathcal{C}(\vec{\mathcal{X}},\alpha) & := & (\mathsf{I}_\alpha\, [\![F]\!]_\theta\, \top^\kappa)([\![H_1]\!]_\theta\vec{\mathcal{X}}) \ldots ([\![H_l]\!]_\theta\vec{\mathcal{X}}) & \text{where } l := |\vec{\kappa}|. \end{array}$$

ADM-$\nu$-START Show $\mathcal{S} \subseteq \mathcal{C}(\vec{\mathcal{X}}, 0)$. This is clear since $\mathsf{I}_0\, [\![F]\!]_\theta\, \top^\kappa = \top^\kappa$ and $\top^\kappa\, \mathcal{H}_1 \ldots \mathcal{H}_l = \top^* = \mathcal{S}$ for any $\vec{\mathcal{H}}$.

ADM-$\nu$-LIMIT Show $\inf_\lambda \mathcal{A} \subseteq \mathcal{A}(\lambda)$. We assume $f \in \inf_\lambda \mathcal{A}$ and prove

$$f \in \mathcal{B}_{1..n}(\vec{\mathcal{X}},\lambda) \boxed{\to} \mathcal{C}(\vec{\mathcal{X}},\lambda).$$

To this end, fix arbitrary $t_i \in \mathcal{B}_i(\vec{\mathcal{X}},\lambda)$ and show $f\,\vec{t} \in \mathcal{C}(\vec{\mathcal{X}},\lambda)$. By Lemma 3.17, the coinductive type at a limit $\mathcal{C}(\vec{\mathcal{X}},\lambda) = \inf_{\alpha<\lambda} \mathcal{C}(\vec{\mathcal{X}},\alpha)$, hence, we need to show $f\,\vec{t} \in \mathcal{C}(\vec{\mathcal{X}},\alpha)$ for arbitrary $\alpha < \lambda$. Since $\Gamma, \iota : -\mathsf{ord} \vdash G_i : \kappa'$, the families $\mathcal{B}_i(\vec{\mathcal{X}},\_)$ are antitone, hence $t_i \in \mathcal{B}_i(\vec{\mathcal{X}},\alpha)$. Together, $f\,\vec{t} \in \mathcal{C}(\vec{\mathcal{X}},\alpha)$ by assumption. □

### 3.5.3 Soundness Proof

**Valuations.** For the remainder of this chapter, we consider valuations $\theta \in (\mathsf{TyVar} \cup \mathsf{Var}) \to (\bigcup_\kappa [\![\kappa]\!] \cup \mathsf{Tm})$ which map constructor variables $X$ to semantical constructors $\mathcal{F} \in [\![\kappa]\!]$ for some $\kappa$ and term variables to terms. Since a constructor $F$ does not depend on term variables, it is easy to see that $[\![F]\!]_\theta = [\![F]\!]_{\theta|\mathsf{TyVar}}$. The denotation $(\!|t|\!)_\theta$ of a term $t$ under a valuation $\theta$ is defined as follows:

$$\begin{array}{lcll} (\!|c|\!)_\theta & := & c & \\ (\!|x|\!)_\theta & := & \theta(x) & \\ (\!|r\,s|\!)_\theta & := & (\!|r|\!)_\theta\ (\!|s|\!)_\theta & \\ (\!|\lambda x t|\!)_\theta & := & \lambda x.\, (\!|t|\!)_{\theta[x\mapsto x]} & \text{if } x \text{ is singular in } (\!|t|\!)_\theta \end{array}$$

By $x$ *singular* in $(\!|t|\!)_\theta$, we mean that $x \notin \mathsf{FV}(\theta(y))$ for any $y \in \mathsf{FV}(t)$ which is different from $x$.[11] Singularity is important to avoid variable capture in terms of the substitution $\theta$ by the $\lambda$-abstraction, which has moved to the outside. We can express singularity by the equivalent condition $x \notin \mathsf{FV}(\!|\lambda x t|\!)_\theta$. Note that singularity can always be achieved by renaming of the bound variable $x$, since $\mathsf{FV}(\!|t|\!)_\theta$ is finite.

**Lemma 3.47 (Substitution for singular variable)** *If $x$ is singular in $(\!|t|\!)_\theta$, then we have $[s/x](\!|t|\!)_{\theta[x\mapsto x]} = (\!|t|\!)_{\theta[x\mapsto s]}$.*

*Proof.* By induction on $t$. □

[11] This is the *variable condition* in the corresponding part of Barthe et al. [BFG+04].

**Sound valuations.** We define the proposition $\theta \in [\![\Gamma]\!]$ by recursion on $\Gamma$.

$$\begin{array}{lcl} \theta \in [\![\diamond]\!] & :\Longleftrightarrow & \text{true} \\ \theta \in [\![\Gamma, X : p\kappa]\!] & :\Longleftrightarrow & \theta \in [\![\Gamma]\!] \text{ and } \theta(X) \in [\![\kappa]\!] \\ \theta \in [\![\Gamma, x : A]\!] & :\Longleftrightarrow & \theta \in [\![\Gamma]\!] \text{ and } \theta(x) \in [\![A]\!]_\theta \end{array}$$

This notion is compatible with $\theta \in [\![\Delta]\!]$ from Section 2.3: If $\Delta$ is the restriction of $\Gamma$ to constructor variable declarations, then $\theta \in [\![\Gamma]\!]$ implies $\theta \in [\![\Delta]\!]$.

**Lemma 3.48 (Sound context lookup)** *If* $\Gamma$ cxt *and both* $(x : A) \in \Gamma$ *and* $\theta \in [\![\Gamma]\!]$ *then* $\theta(x) \in [\![A]\!]_\theta \in [\![*]\!]$.

*Proof.* By induction on $\Gamma$ cxt, using Lem. 2.20. □

**Theorem 3.49 (Soundness of typing)** *If* $\Gamma \vdash t : A$ *and* $\theta \in [\![\Gamma]\!]$ *then* $(\!|t|\!)_\theta \in [\![A]\!]_\theta$.

*Proof.* By induction on the typing derivation.

*Case*

$$\text{TY-VAR}\ \frac{(x : A) \in \Gamma \qquad \Gamma \text{ cxt}}{\Gamma \vdash x : A}$$

By Lemma 3.48, since $(\!|x|\!)_\theta = \theta(x)$.

*Case*

$$\text{TY-ABS}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x t : A \to B}$$

Let $\mathcal{A} := [\![A]\!]_\theta$ and $\mathcal{B} := [\![B]\!]_\theta$. Since, by Lemma 3.4, the context $\Gamma, x : A$ is wellformed, we have by Lemma 3.48 that $\mathcal{A} \in \mathsf{SAT}$. Likewise, $\mathcal{B} \in \mathsf{SAT}$ by Lemma 2.20, since Lemma 3.5 entails $\Gamma \vdash B : *$. W. l. o. g., $x$ is singular in $(\!|t|\!)_\theta$. To show that $(\!|\lambda x t|\!)_\theta = \lambda x.(\!|t|\!)_{\theta[x \mapsto x]} \in \mathcal{A} \boxrightarrow \mathcal{B}$, by Corollary 3.31 it is sufficient to show $[s/x](\!|t|\!)_{\theta[x \mapsto x]} \in \mathcal{B}$ for arbitrary $s \in \mathcal{A}$. This, however, follows from the induction hypothesis, since $\theta[x \mapsto s] \in [\![\Gamma, x : A]\!]$, and $[s/x](\!|t|\!)_{\theta[x \mapsto x]} = (\!|t|\!)_{\theta[x \mapsto s]}$ by Lemma 3.47.

*Case*

$$\text{TY-APP}\ \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B}$$

Let $\mathcal{A} := [\![A]\!]_\theta$ and $\mathcal{B} := [\![B]\!]_\theta$. By induction hypothesis, $(\!|r|\!)_\theta \in \mathcal{A} \boxrightarrow \mathcal{B}$ and $(\!|s|\!)_\theta \in \mathcal{A}$, thus $(\!|r\, s|\!)_\theta = (\!|r|\!)_\theta\ (\!|s|\!)_\theta \in \mathcal{B}$ by definition of the function space.

*Case*

$$\text{TY-GEN}\ \frac{\Gamma, X : \kappa \vdash t : F\, X}{\Gamma \vdash t : \forall_\kappa F}\ X \notin \mathsf{FV}(F)$$

By definition of $[\![\forall_\kappa]\!]$, we have to show $(\!|t|\!)_\theta \in [\![F]\!]_\theta(\mathcal{G})$ for all $\mathcal{G} \in [\![\kappa]\!]$. This follows by induction hypothesis, since $\theta[X \mapsto \mathcal{G}] \in [\![\Gamma, X : \kappa]\!]$, and $[\![F\, X]\!]_{\theta[X \mapsto \mathcal{G}]} = [\![F]\!]_\theta(\mathcal{G})$ because $X \notin \mathsf{FV}(F)$.

*Case*

$$\text{TY-INST}\ \frac{\Gamma \vdash t : \forall_\kappa F \qquad \Gamma \vdash G : \kappa}{\Gamma \vdash t : F\,G}$$

By induction hypothesis, $(\!|t|\!)_\theta \in \bigcap_{\mathcal{G}\in[\kappa]} [\![F]\!]_\theta(\mathcal{G})$ and $[\![G]\!]_\theta \in [\![\kappa]\!]$. Hence $(\!|t|\!)_\theta \in [\![F\,G]\!]_\theta$.

*Case*

$$\text{TY-SUB}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leq B : *}{\Gamma \vdash t : B}$$

By induction hypothesis $(\!|t|\!)_\theta \in [\![A]\!]_\theta$, which is a subset of $[\![B]\!]_\theta$ by Lemma 2.20.

*Case*

$$\text{TY-FOLD}\ \frac{\Gamma \vdash t : F\,(\nabla_\kappa\, a\, F)\,\vec{G}}{\Gamma \vdash t : \nabla_\kappa\,(a+1)\,F\,\vec{G}}$$

By Lemma 3.5 we have $\Gamma \vdash F\,(\nabla_\kappa\, a\, F)\,\vec{G} : *$, which entails $\Gamma \vdash F : +\kappa \to \kappa$ and $\Gamma \vdash a : \mathsf{ord}$, as well as $\Gamma \vdash G_i : \kappa_i$ for $1 \leq i \leq |\vec{\kappa}|$, if we define $\vec{p}\vec{\kappa} \to * := \kappa$. Hence, $\mathcal{F} := [\![F]\!]_\theta \in [\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]$ and $\alpha := [\![a]\!]_\theta \in [\![\mathsf{ord}]\!]$ and we can conclude by Theorem 3.24.

*Case*

$$\text{TY-UNFOLD}\ \frac{\Gamma \vdash r : \nabla_\kappa\,(a+1)\,F\,\vec{G}}{\Gamma \vdash r : F\,(\nabla_\kappa\, a\, F)\,\vec{G}}$$

Analogously to case TY-FOLD.

*Case*

$$\text{TY-REC}\ \frac{\Gamma \vdash A\ \mathsf{fix}^\nabla_n\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord} \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash \mathsf{fix}^\nabla_n : (\forall \imath{:}\mathsf{ord}.\, A\,\imath \to A\,(\imath+1)) \to A\,a}$$

By lemmata 3.45 and 3.46 the family $\mathcal{A} := [\![A]\!]_\theta \in [\![\mathsf{ord}]\!] \to [\![*]\!]$ is admissible for (co)recursion. We assume $s \in \bigcap_{\alpha\in[\![\mathsf{ord}]\!]} \mathcal{A}(\alpha) \boxrightarrow A([\![\mathsf{s}]\!]\alpha)$ which entails that $s \in \mathcal{A}(\alpha) \boxrightarrow \mathcal{A}(\alpha+1)$ for all $\alpha < \top^{\mathsf{ord}}$. By lemma 3.32 resp. 3.37 we conclude $\mathsf{fix}^\nabla_n\, s \in \mathcal{A}([\![a]\!]_\theta)$. □

A consequence of soundness is that all typable terms are safe, if variables are neutral.

**Corollary 3.50 (Typable terms are safe)** *Assume* $\mathsf{Var} \subseteq \mathcal{N}$. *If* $\Gamma \vdash t : B$ *then* $t \in \mathcal{S}$.

*Proof.* Let $\theta$ be a valuation with $\theta(X) = \top^\kappa$ for all $(X{:}\kappa) \in \Gamma$ and $\theta(x) = x$ for all $(x{:}A) \in \Gamma$. Since $x \in \mathcal{N} \subseteq [\![A]\!]_\theta$, the valuation $\theta$ is sound w. r. t. context $\Gamma$. Soundness of typing entails $t = (\!|t|\!)_\theta \in [\![B]\!]_\theta \subseteq \mathcal{S}$. □

## 3.6 Strong Normalization

In this section, we will instantiate the set of safe terms $\mathcal{S}$ by an inductively defined set SN of strongly normalizing terms and harvest a proof of strong normalization from the soundness proof of typing.

### 3.6.1 A Few Remarks on the Method

The usual (classical) definition of strongly normalizing terms is: *those terms, which have no infinite reduction sequences*. This definition has quite nice closure properties, which are intuitively obvious: for example, if a term has no infinite reduction sequences, then this is true for all of its subterms as well. Defining $\mathcal{S}$ as the set of (classically) strongly normalizing terms, we see immediately that REQ-FUN-SAFE, $\mathcal{N} \boxrightarrow \mathcal{S} \subseteq \mathcal{S}$, holds for non-empty $\mathcal{N}$: If $r \in \mathcal{N} \boxrightarrow \mathcal{S}$, then $r\,s$ is strongly normalizing for any $s \in \mathcal{N}$, hence, $r$ is strongly normalizing.

Barthe et al. [BFG+04] use this definition of strong normalization, albeit in a positive formulation. For us, this is not sufficient, since we also require that all non-neutral terms in $\mathcal{S}$ reduce to a value. Usually, *after* having shown strong normalization, this fact is a consequence of *type preservation* (*subject reduction*) and *progress* [Pie02]. But we need it already to show strong normalization.

Why do we require this and Barthe et al. do not? Because we are working in an equi-recursive system where an inhabitant of $\mu^a F$ can be of any shape. Hence, a recursive function is unrolled when it is applied to *any value*. In an iso-recursive system, which is used in far more normalization proofs [AA00, Abe03, Abe04, Mat05] a canonical inhabitant of $\mu^a F$ is of the form $\mathsf{in}\,t$ where $\mathsf{in}$ is a constructor. The semantics of $\mu^a F$ can be defined as those terms which are neutral or reduce to $\mathsf{in}\,t$ for some $t$. A recursive function $f : \mu^a F \to C$ is only unrolled when applied to a constructor. In the normalization proof, when looking at $f\,r$ we know that $r \in [\![\mu^{a+1} F]\!]$ is either neutral or reduces to $\mathsf{in}\,t$. Then we can unroll the recursive function and conclude the reasoning by induction hypothesis [Abe04, Thm. 5.10]. Similarly, in Lemma 3.32 we need to know that $r$ reduces to a value, but since $\mu^{a+1} F = F\,(\mu^a F)$ can be any type (product type, function type, polymorphic type etc.), we need to know that the non-neutral inhabitants of all semantical types reduce to a value. This is most directly achieved by requiring that it holds for all safe terms.

A byproduct of this decision is that we also get more in the end: We not only exclude non-termination; non-neutral terms also *do not get stuck*. Hence, we get both strong normalization and progress.

The search for a suitable definition of $\mathcal{S}$ lead us to an *inductive characterization of strongly normalizing terms*. For the lambda calculus, this was first given by van Raamsdonk et al. [vRS95, vRSSX99], but it was already implicit in Goguen's work on typed operational semantics [Gog94, Gog95, Gog99]. Joachimski and Matthes used it to give a combinatorial normalization proof for the simply-typed $\lambda$-calculus [JM03], and Matthes extended it to other calculi [Mat98, Mat00, Mat05].

### 3.6.2 Inductive Characterization

We set $\mathcal{S} := \mathsf{SN}$ and $\rhd := \longrightarrow_{\mathsf{SN}}^*$ both of which will be defined below.

**Strong(-ly normalizing) head reduction** $t \longrightarrow_{\mathsf{SN}} t'$ is defined inductively by the following rules.

$$\text{SHR-}\beta\ \frac{s \in \mathsf{SN}}{(\lambda x t)\, s \longrightarrow_{\mathsf{SN}} [s/x]t} \qquad \text{SHR-FRAME}\ \frac{r \longrightarrow_{\mathsf{SN}} r'}{e(r) \longrightarrow_{\mathsf{SN}} e(r')}$$

$$\text{SHR-REC}\ \frac{}{\mathsf{fix}^{\mu}_{n}\, s\, t_{1..n}\, v \longrightarrow_{\mathsf{SN}} s\,(\mathsf{fix}^{\mu}_{n}\, s)\, t_{1..n}\, v}\ v \neq \mathsf{fix}^{\nu}_{n'}\, s'\, t'_{1..n'}$$

$$\text{SHR-COREC}\ \frac{}{\mathsf{fix}^{\nu}_{n}\, s\, t_{1..n}\, r \longrightarrow_{\mathsf{SN}} s\,(\mathsf{fix}^{\nu}_{n}\, s)\, t_{1..n}\, r}$$

Note that due to the additional argument $r$, the last rule is just short for

$$\text{SHR-COREC}\ \frac{}{e(\mathsf{fix}^{\nu}_{n}\, s\, t_{1..n}) \longrightarrow_{\mathsf{SN}} e(s\,(\mathsf{fix}^{\nu}_{n}\, s)\, t_{1..n})}\ e \neq \mathsf{fix}^{\mu}_{n'}\, s'\, t'_{1..n'}\, \_.$$

The crucial property is that the strong head expansion of a strongly normalizing term is strongly normalizing as well. The reflexive-transitive closure of strong head reduction fulfills the desired properties of $\rhd$ from Section 3.4.1.

**Lemma 3.51 ($\longrightarrow_{\mathsf{SN}}$ is deterministic)** *If $\mathcal{D} :: r \longrightarrow_{\mathsf{SN}} r'$ and $r \longrightarrow_{\mathsf{SN}} r''$ then $r' = r''$.*

*Proof.* By induction on $\mathcal{D}$. □

**Lemma 3.52 (REQ-ECXT)** *If $t \longrightarrow_{\mathsf{SN}}^{+} t'$ then $E(t) \longrightarrow_{\mathsf{SN}}^{+} E(t')$.*

*Proof.* From SHR-FRAME by induction on $E$ and $\longrightarrow_{\mathsf{SN}}^{+}$. □

**Corollary 3.53** *The relation $\rhd := \longrightarrow_{\mathsf{SN}}^*$ fulfills the five requirements* REQ-$\beta$, REQ-REC, REQ-COREC, REQ-ECXT, *and* REQ-TRANS.

**Strongly neutral terms** $r \in \mathsf{SNe}$ are defined inductively by the following rules.

$$\text{SNE-VAR}\ \frac{}{x \in \mathsf{SNe}} \qquad \text{SNE-FRAME}\ \frac{r \in \mathsf{SNe} \qquad e \in \mathsf{Sframe}}{e(r) \in \mathsf{SNe}}$$

$$\text{SNE-FIX}^{\mu}\text{FIX}^{\nu}\ \frac{\overbrace{\mathsf{fix}^{\mu}_{n}\, s\, t_{1..n}\, \_}^{e} \in \mathsf{Sframe} \qquad \overbrace{\mathsf{fix}^{\nu}_{n'}\, s'\, t'_{1..n'}}^{v} \in \mathsf{SN}}{e(v) \in \mathsf{SNe}}$$

Since the definition of Sframe rests on the definition of $\mathcal{S} = \mathsf{SN}$, strongly neutral terms are defined simultaneously with SN below. Rule SNE-FIX$^{\mu}$FIX$^{\nu}$ fulfills requirement REQ-FIX$^{\mu}$FIX$^{\nu}$.

**Strongly normalizing terms** $t \in \mathsf{SN}$.

$$\textsc{sn-sne}\ \frac{r \in \mathsf{SNe}}{r \in \mathsf{SN}} \qquad \textsc{sn-abs}\ \frac{t \in \mathsf{SN}}{\lambda x t \in \mathsf{SN}} \qquad \textsc{sn-fix}\ \frac{\vec{t} \in \mathsf{SN}}{\mathsf{fix}_n^\nabla\, \vec{t} \in \mathsf{SN}}\ |\vec{t}| \leq n+1$$

$$\textsc{sn-exp}\ \frac{r \longrightarrow_{\mathsf{SN}} r' \qquad r' \in \mathsf{SN}}{r \in \mathsf{SN}} \qquad \textsc{sn-roll}\ \frac{s\,(\mathsf{fix}_n^\nu\, s)\,\vec{t} \in \mathsf{SN}}{\mathsf{fix}_n^\nu\, s\,\vec{t} \in \mathsf{SN}}\ |\vec{t}| \leq n$$

The set $\mathsf{SN}$ consists of strongly neutral terms (SN-SNE) and strongly normalizing values (SN-ABS, SN-FIX) and is closed under strong head expansion (SN-EXP). Rule SN-FIX fulfills requirement REQ-FIX$^\nu$. The rule SN-ROLL is admissible, but we have explicitely added it to simplify the proof of requirement REQ-FUN-SAFE (see below).

**Lemma 3.54 ($\mathsf{SN}$ is closed under $\rhd$-expansion)** *If $t \longrightarrow_{\mathsf{SN}}^* t'$ and $t' \in \mathsf{SN}$ then $t \in \mathsf{SN}$.*

*Proof.* By rule SN-EXP. □

It is clear that $\mathsf{SN}$ is also closed under $\rhd$-reduction.

**Lemma 3.55 ($\mathsf{SN}$ is closed under $\rhd$-reduction)** *If $t \in \mathsf{SN}$ and $t \longrightarrow_{\mathsf{SN}}^* t'$ then $t' \in \mathsf{SN}$.*

*Proof.* By induction on $t \longrightarrow_{\mathsf{SN}}^* t'$. If the reduction sequence is empty (in case $t = t'$), there is nothing to show. Otherwise, we perform case analysis on $t \in \mathsf{SN}$. The only rule which introduces redexes is SN-EXP.

$$\textsc{sn-exp}\ \frac{t \longrightarrow_{\mathsf{SN}} t'' \qquad t'' \in \mathsf{SN}}{t \in \mathsf{SN}}$$

Since $\longrightarrow_{\mathsf{SN}}$ is deterministic, the first reduction in the sequence $t \longrightarrow_{\mathsf{SN}}^+ t'$ is $t \longrightarrow_{\mathsf{SN}} t''$. Thus, $t' \in \mathsf{SN}$ follows by induction hypothesis. □

Thus, $\mathsf{SN}$ fulfills requirement REQ-$\mathcal{S}$-CLOSED.

In contrast to the definition of strong normalization as the absence of infinite reduction sequences, the inductive characterization of $\mathsf{SN}$ also guarantees that every $t \in \mathsf{SN}$ reduces to a weak head value, i. e., there are no stuck non-neutral terms.

**Lemma 3.56 ($\mathsf{SN}$ is weak head normalizing)** *If $r \in \mathsf{SN}$ then either $r \rhd r' \in \mathsf{SNe}$ or $r \rhd v \in \mathsf{Val} \cap \mathsf{SN}$.*

*Proof.* By induction on $r \in \mathsf{SN}$. If the last rule was SN-EXP, we proceed by induction hypothesis. All other rules introduce neutral terms or values. □

Let ${}^\rhd\mathcal{A} := \{r \mid r \rhd r' \text{ with } r' \in \mathcal{A}\}$ be the closure of a term set $\mathcal{A}$ under $\rhd$-expansion.

**Lemma 3.57 (Neutral terms)** ${}^{\triangleright}\mathsf{SNe} \subseteq \mathcal{N}$.

*Proof.* Recall that $\mathcal{N} = \mathsf{Scxt}^{\perp}$. Assume $r \triangleright r' \in \mathsf{SNe}$ and $E \in \mathsf{Scxt}$ and show $E(r) \in \mathsf{SN}$. Since $E(r') \in \mathsf{SNe}$ by iterated application of SNE-FRAME and $\mathsf{SNe} \subseteq \mathsf{SN}$ by rule SN-SNE, it follows that $E(r') \in \mathsf{SN}$. Because $E(r) \triangleright E(r')$ and $\mathsf{SN}$ is closed under $\triangleright$-expansion, $E(r) \in \mathsf{SN}$. □

In particular, all variables are in $\mathcal{N}$, which is a precondition of Cor. 3.50. Putting the last two lemmata together, we see that every term in $\mathcal{S} \setminus \mathcal{N}$ reduces to a strongly normalizing value $v \in \mathsf{SN}$ under strong head reduction. Hence, $\mathsf{SN}$ fulfills REQ-$\mathcal{S}$-VAL.

**Remark 3.58 ($\mathcal{N} \neq {}^{\triangleright}\mathsf{SNe}$)** The proposition $\mathcal{N} \subseteq {}^{\triangleright}\mathsf{SNe}$, which complements Lemma 3.57, does not hold: In Lemma 3.36 we have shown that some corecursive values inhabit $\mathcal{N}$, but they are not in $\mathsf{SNe}$. (If not for REQ-FIX$^{\mu}$FIX$^{\nu}$, this proposition would hold: We could show that no value $v$ can be safely inserted into all safe contexts by exhibiting a safe context $e = \mathsf{fix}_0^{\mu}(\lambda f \lambda y.\, f\, \lambda x x)\, \_$ in which all values would loop, even the strongly normalizing ones.)

In the following we show validity of REQ-FUN-SAFE, i. e., $r \in \mathsf{SN}$ if $r\,s \in \mathsf{SN}$ for all $s \in \mathcal{N}$.

**Lemma 3.59 (Extensionality)** *If* $\mathcal{D} :: r\,x \in \mathsf{SN}$ *or* $\mathcal{D} :: r\,x \in \mathsf{SNe}$ *then* $r \in \mathsf{SN}$.

*Proof.* By induction on $\mathcal{D}$.

*Case*

$$\textsc{sne-frame}\ \frac{r \in \mathsf{SNe}}{r\,x \in \mathsf{SNe}}$$

By SN-SNE, $r \in \mathsf{SN}$.

*Case* Let $|\vec{t}| = n$.

$$\textsc{sne-frame}\ \frac{s, \vec{t} \in \mathsf{SN}}{\mathsf{fix}_n^{\mu}\, s\, \vec{t}\, x \in \mathsf{SNe}}$$

By SN-FIX, $\mathsf{fix}_n^{\mu}\, s\, \vec{t} \in \mathsf{SN}$.

*Case*

$$\textsc{sn-sne}\ \frac{r\,x \in \mathsf{SNe}}{r\,x \in \mathsf{SN}}$$

By induction hypothesis.

*Case*

$$\textsc{sn-fix}\ \frac{\vec{t} \in \mathsf{SN}}{\mathsf{fix}_n^{\nabla}\, \vec{t}\, x \in \mathsf{SN}}\ |\vec{t}| \leq n$$

Then $\mathsf{fix}_n^{\nabla}\, \vec{t} \in \mathsf{SN}$ by SN-FIX.

*Case*

$$\text{SN-EXP}\ \frac{(\lambda x t)\, x \longrightarrow_{\mathsf{SN}} t \qquad t \in \mathsf{SN}}{(\lambda x t)\, x \in \mathsf{SN}}$$

Since $t \in \mathsf{SN}$, $\lambda x t \in \mathsf{SN}$ by SN-ABS.

*Case*

$$\text{SN-EXP}\ \frac{\mathsf{fix}_n^\nu\, s\, \vec{t}\, x \longrightarrow_{\mathsf{SN}} s\,(\mathsf{fix}_n^\nu\, s)\, \vec{t}\, x \qquad s\,(\mathsf{fix}_n^\nu\, s)\, \vec{t}\, x \in \mathsf{SN}}{\mathsf{fix}_n^\nu\, s\, \vec{t}\, x \in \mathsf{SN}}$$

By induction hypothesis, $s\,(\mathsf{fix}_n^\nu\, s)\, \vec{t} \in \mathsf{SN}$. Hence, $\mathsf{fix}_n^\nu\, s\, \vec{t} \in \mathsf{SN}$ by SN-ROLL.

*Case*

$$\dfrac{\dfrac{r \longrightarrow_{\mathsf{SN}} r'}{r\, x \longrightarrow_{\mathsf{SN}} r'\, x}\ \text{SHR-FRAME} \qquad r'\, x \in \mathsf{SN}}{r\, x \in \mathsf{SN}}\ \text{SN-EXP}$$

By induction hypothesis, $r' \in \mathsf{SN}$, hence $r \in \mathsf{SN}$ by SN-EXP.

*Case*

$$\text{SN-ROLL}\ \frac{s\,(\mathsf{fix}_n^\nu\, s)\, \vec{t}\, x \in \mathsf{SN}}{\mathsf{fix}_n^\nu\, s\, \vec{t}\, x \in \mathsf{SN}}\ |\vec{t}| < n$$

By induction hypothesis $s\,(\mathsf{fix}_n^\nu\, s)\, \vec{t} \in \mathsf{SN}$, hence, $\mathsf{fix}_n^\nu\, s\, \vec{t} \in \mathsf{SN}$ by SN-ROLL.

*Case*

$$\text{SN-ROLL}\ \frac{x\,(\mathsf{fix}_n^\nu\, x) \in \mathsf{SN}}{\mathsf{fix}_n^\nu\, x \in \mathsf{SN}}$$

Then $\mathsf{fix}_n^\nu \in \mathsf{SN}$ by SN-FIX.

□

**Corollary 3.60 (**REQ-FUN-SAFE**)** $\mathcal{N} \boxed{\rightarrow} \mathsf{SN} \subseteq \mathsf{SN}$.

*Proof.* $\{r \mid r\, s \in \mathsf{SN} \text{ for all } s \in \mathcal{N}\} \subseteq \{r \mid r\, x \in \mathsf{SN}\} \subseteq \mathsf{SN}$ by the lemma. □

By Cor. 3.50, each typable term is in SN. What remains to show that if $t \in \mathsf{SN}$ then $t$ indeed admits no infinite reduction sequences.

### 3.6.3 Soundness of the Inductive Characterization

Classically, a term $t$ is strongly normalizing if there are no infinite reduction sequences starting with $t$. Constructively, the set of strongly normalizing terms sn is the accessible part of Tm w. r. t. the one-step reduction relation. Hence, it can be defined as the smallest set $\mathsf{sn} \subseteq \mathsf{Tm}$ closed under the rule

$$\frac{\forall t' \longleftarrow t.\ t' \in \mathsf{sn}}{t \in \mathsf{sn}}$$

In this section, we show that $\mathsf{SN} \subseteq \mathsf{sn}$.

**Immediate closure properties of** sn**.** By analyzing the possible reductions of a term $t$ depending on its shape, we immediately obtain the following closure properties of sn:

$$\frac{}{c \in \mathsf{sn}} \qquad \frac{}{x \in \mathsf{sn}} \qquad \frac{t \in \mathsf{sn}}{\lambda x t \in \mathsf{sn}} \qquad \frac{r, s \in \mathsf{sn} \qquad \forall t \leftharpoondown r\, s.\ t \in \mathsf{sn}}{r\, s \in \mathsf{sn}}$$

Since $E(x)$ is not a value, $e(E(x))$ is never a redex. Thus, the following two closure properties are a consequence of the closure property for application:

$$\frac{E(x) \in \mathsf{sn} \qquad s \in \mathsf{sn}}{E(x)\, s \in \mathsf{sn}} \qquad \frac{E(x) \in \mathsf{sn} \qquad s, t_{1..n} \in \mathsf{sn}}{\mathsf{fix}_n^\mu\ s\, t_{1..n}\, E(x) \in \mathsf{sn}}$$

Similarly, $\mathsf{fix}_n^\nabla\ \vec{t}$ is not a redex if $|\vec{t}| \leq n+1$. Hence, we can add another closure property:

$$\frac{\vec{t} \in \mathsf{sn}}{\mathsf{fix}_n^\nabla\ \vec{t} \in \mathsf{sn}}\ |\vec{t}| \leq n+1$$

**Lemma 3.61 (Closure under subterm)** *The set of strongly normalizing terms is closed under subterm formation.*

1. *If* $\mathcal{D} :: \lambda x t \in \mathsf{sn}$ *then* $\mathcal{D}' :: t \in \mathsf{sn}$.
2. *If* $\mathcal{D} :: r\, s \in \mathsf{sn}$ *then* $\mathcal{D}' :: r \in \mathsf{sn}$.
3. *If* $\mathcal{D} :: r\, s \in \mathsf{sn}$ *then* $\mathcal{D}' :: s \in \mathsf{sn}$.

*In each case,* $\#\mathcal{D}' \leq \#\mathcal{D}$.

*Proof.* Each by induction on $\mathcal{D}$. □

**Corollary 3.62** *If* $E(r) \in sn$ *then* $E(x), r \in sn$.

**Lemma 3.63 (Closure under splitting)** *If* $\mathcal{D} :: [s/x]t \in \mathsf{sn}$ *then* $\mathcal{D}' :: t \in \mathsf{sn}$ *and* $\#\mathcal{D}' \leq \#\mathcal{D}$.

*Proof.* By induction on $\mathcal{D}$. □

**Closure under strong head expansion.** We define a relation $t \longrightarrow_{\mathsf{sn}} t'$ by the same rules as $t \longrightarrow_{\mathsf{SN}} t'$ except that we require $s \in \mathsf{sn}$ instead of SN in the rule SHR-$\beta$. Our goal is to show that sn is closed under $\longrightarrow_{\mathsf{sn}}$-expansion.

**Lemma 3.64 (Closure under strong head expansion axioms)**

1. *If* $s, [s/x]t \in \mathsf{sn}$ *then* $(\lambda x t)\, s \in \mathsf{sn}$.
2. *If* $s\, (\mathsf{fix}_n^\mu\ s)\, t_{1..n}\, v \in \mathsf{sn}$ *then* $\mathsf{fix}_n^\mu\ s\, t_{1..n}\, v \in \mathsf{sn}$.

3. *If* $E(s\,(\mathsf{fix}_n^\gamma\ s)\,t_{1..n}) \in \mathsf{sn}$ *then* $E(\mathsf{fix}_n^\gamma\ s\,t_{1..n}) \in \mathsf{sn}$.

*Proof.* 1. By Lemma 3.63 $t \in \mathsf{sn}$. We perform induction on $s, t \in \mathsf{sn}$ and analyze the possible reducts of $(\lambda x t)\,s$. 2. By induction on $s, t_{1..n}, v \in \mathsf{sn}$. 3. By induction on $s, t_{1..n}, E(x) \in \mathsf{sn}$. □

**Lemma 3.65 (**$\mathsf{sn}$ **closed under strong head expansion)**

$$\frac{t_0 \longrightarrow_{\mathsf{sn}} t_1 \qquad t_1 \in \mathsf{sn}}{t_0 \in \mathsf{sn}}$$

*Proof.* By induction on $t_0 \longrightarrow_{\mathsf{sn}} t_1$. In case of SHR-$\beta$, SHR-REC, or SHR-COREC, apply Lemma 3.64. In case $e(t_0) \longrightarrow_{\mathsf{sn}} e(t_1)$, we have $t_0 \in \mathsf{sn}$ by induction hypothesis, since $t_1 \in \mathsf{sn}$ by Cor. 3.62. We proceed by side induction on $e(x), t_0 \in \mathsf{sn}$. Since $e(t_0)$ is not a redex, each reduction must be either in $e(x)$ or $t_0$. By side induction hypothesis each reduction of $e(t_0)$ is in $\mathsf{sn}$. □

For each introduction rule of $\mathsf{SN}$ we have established a corresponding closure property of $\mathsf{sn}$. Hence, proving $\mathsf{SN} \subseteq \mathsf{sn}$ amounts to a routine induction.

**Lemma 3.66 (Inductive** $\mathsf{SN}$ **is sound)**

1. *If* $\mathcal{D} :: t \longrightarrow_{\mathsf{SN}} t'$ *then* $t \longrightarrow_{\mathsf{sn}} t'$.
2. *If* $\mathcal{D} :: t \in \mathsf{SNe}$ *then* $t = E(x) \in \mathsf{sn}$ *for some* $E$ *and* $x$.
3. *If* $\mathcal{D} :: t \in \mathsf{SN}$ *then* $t \in \mathsf{sn}$.

*Proof.* Simultaneously by induction on $\mathcal{D}$. □

**Theorem 3.67 (Strong normalization)** *If* $\Gamma \vdash t : A$*, then there are no infinite reduction sequences starting with* $t$*.*

*Proof.* By Cor. 3.50 and the previous lemma, $t \in \mathsf{sn}$. By induction on $t \in \mathsf{sn}$ we can prove that $t \longrightarrow^* t' \not\longrightarrow$, i. e., $t$ reduces to a normal form in finitely many steps. □

# Chapter 4

# Embeddings into $\mathsf{F}_{\widehat{\omega}}$

In this chapter, we consider type and reduction preserving embeddings of some total type systems into $\mathsf{F}_{\widehat{\omega}}$. Such embeddings provide a translation function $\ulcorner\cdot\urcorner$ for terms, types, and possibly kinds of the source language into the target language $\mathsf{F}_{\widehat{\omega}}$. An embedding is *type preserving* if for all $\Gamma \vdash t : A$ of the source language, it holds that $\ulcorner\Gamma\urcorner \vdash \ulcorner t\urcorner : \ulcorner A\urcorner$ in the target language. Similarly, if a source reduction step $t \longrightarrow t'$ maps to a sequence $\ulcorner t\urcorner \longrightarrow^{+} \ulcorner t'\urcorner$ of target reduction steps, we speak of a *reduction preserving embedding*.

For the purpose of embeddings, it is important that $\mathsf{F}_{\widehat{\omega}}$ is *strongly* normalizing. If $\mathsf{F}_{\widehat{\omega}}$ was only normalizing w. r. t. a very specific reduction strategy, then the reductions of the source language would not likely map to $\mathsf{F}_{\widehat{\omega}}$ reductions that are covered by the strategy. Therefore we have shown strong normalization of $\mathsf{F}_{\widehat{\omega}}$, as opposed to showing just weak normalization.

## 4.1 An Iso-Recursive Version of $\mathsf{F}_{\widehat{\omega}}$

In our presentation of $\mathsf{F}_{\widehat{\omega}}$, we decided that inductive and coinductive types should be *equi-recursive*: By the rules TY-FOLD and TY-UNFOLD, a fixed point $\nabla \infty F$ is identical to its unfolded version $F\,(\nabla \infty F)$. Alternatively, we can present the system *iso-recursively*: We drop the rules TY-FOLD and TY-UNFOLD and introduce two new term constructors:

$$\mathsf{Tm} \ni r, s, t ::= \cdots \mid \mathsf{in}\, t \mid \mathsf{out}\, t.$$

These two terms witness the isomorphism between the fixed point and its unfolded version. Hence, they are typed as follows:

$$\text{TY-IN}\ \frac{\Gamma \vdash t : F\,(\overline{\nabla}_\kappa\, a\, F)\,\vec{G}}{\Gamma \vdash \mathsf{in}\, t : \overline{\nabla}_\kappa\,(a+1)\,F\,\vec{G}}\ \overline{\nabla} \in \{\overline{\mu}, \overline{\nu}\}$$

$$\text{TY-OUT}\ \frac{\Gamma \vdash t : \overline{\nabla}_\kappa\,(a+1)\,F\,\vec{G}}{\Gamma \vdash \mathsf{out}\, t : F\,(\overline{\nabla}_\kappa\, a\, F)\,\vec{G}}\ \overline{\nabla} \in \{\overline{\mu}, \overline{\nu}\}$$

We overline $\overline{\mu}$ and $\overline{\nu}$ to distinguish them from the inductive and coinductive constructors of the equi-recursive system. The contraction rules for recursive and corecursive functions are replaced by the following ones, and we add a reduction witnessing that $(\mathsf{in}, \mathsf{out})$ is a retraction pair.

$$\begin{array}{lllll}
\text{ISO-REC} & \mathsf{fix}_n^\mu\, s\, t_{1..n}\, (\mathsf{in}\, r) & \rightarrowtail & s\, (\mathsf{fix}_n^\mu\, s)\, t_{1..n}\, (\mathsf{in}\, r) \\
\text{ISO-COREC} & \mathsf{out}\, (\mathsf{fix}_n^\nu\, s\, t_{1..n}) & \rightarrowtail & \mathsf{out}\, (s\, (\mathsf{fix}_n^\nu\, s)\, t_{1..n}) \\
\text{ISO-}\beta_\nabla & \mathsf{out}\, (\mathsf{in}\, t) & \rightarrowtail & t
\end{array}$$

The corresponding reduction relation $\longrightarrow$ is confluent in a natural way, in contrast to the equi-recursive system, where we had to introduce restrictions to remove critical pairs (see Remark 3.9).

In previous work [Abe03, Abe04], we have considered such iso-recursive systems. They seem to be easier to handle than their equi-recursive counterparts. The reduction rule ISO-REC for recursive functions is more concrete than its counterpart RED-REC in $\mathsf{F}_\omega^{\widehat{}}$. Here, recursive functions can be unfolded if applied to a *data constructor*, in this case, $\mathsf{in}$. There, the argument can be any kind of *value*. Similarly, corecursive functions are only unfolded under a destructor, $\mathsf{out}$, whereas it can be any kind of evaluation context in $\mathsf{F}_\omega^{\widehat{}}$. Critical pairs like $\mathsf{fix}_0^\mu\, s\, (\mathsf{fix}_0^\nu\, s')$ do not arise in the iso-recursive version.

Proofs of strong normalization using saturated sets are considerably simpler for iso-recursive systems than for their equi-recursive counterparts. This is because the semantics of an iso-inductive type is quite concrete: it contains all terms which evaluate to a constructor expression (plus neutral terms). In contrast, the semantics of an equi-inductive type is just its unfolding, we do not get an information on the shape of its inhabitants. We therefore require in our soundness proof in Section 3.5 that all non-neutral terms in a semantical type reduce to a value, which makes strong normalization harder to prove than in the iso-recursive case. To handle equi-coinductive types we even had to refine our notion of saturated set in Section 3.4.3, which is not necessary for iso-coinductive types [AA00, Abe03]. In the following, however, we establish strong normalization of the iso-recursive system effortlessly by embedding it into the equi-recursive one.

Let $\mathbf{1} := \forall A{:}*.\, A \to A$ and $\mathsf{id} := \lambda x x : \mathbf{1}$. We define a translation $\ulcorner \_ \urcorner$, which maps terms of the iso-recursive system to terms of $\mathsf{F}_\omega^{\widehat{}}$, constructors to constructors, and typing contexts to typing contexts. It is defined homomorphically for all constructions except the following:

$$\begin{array}{lcll}
\ulcorner \overline{\nabla}_\kappa \urcorner & = & \lambda a \lambda F.\, \nabla_\kappa\, a\, (\lambda X \lambda \vec{G}.\, \mathbf{1} \to F\, X\, \vec{G}) & \\
 & & \text{where } \kappa = \vec{p}\vec{\kappa} \to * \text{ and } |\vec{G}| = |\vec{\kappa}| & \\
\ulcorner \mathsf{in}\, t \urcorner & = & \lambda k.\, k\, \ulcorner t \urcorner & k \notin \mathsf{FV}(t) \\
\ulcorner \mathsf{out}\, r \urcorner & = & \ulcorner r \urcorner\, \mathsf{id} &
\end{array}$$

The translation preserves well-typedness. Consider the derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\ulcorner t \urcorner : F\,(\ulcorner \nabla_* \urcorner\; a\; F)}{\ulcorner t \urcorner : F\,(\nabla_*\, a\,(\lambda X.\, \mathbf{1} \to F\, X))}}{\lambda k.\, k\, \ulcorner t \urcorner : \mathbf{1} \to F\,(\nabla_*\, a\,(\lambda X.\, \mathbf{1} \to F\, X))}}{\lambda k.\, k\, \ulcorner t \urcorner : \nabla_*\,(a+1)\,(\lambda X.\, \mathbf{1} \to F\, X)}}{\ulcorner \mathsf{in}\, t \urcorner : \ulcorner \nabla_* \urcorner\;(a+1)\; F}$$

This derivation can be lifted to $\nabla_\kappa$ for higher kinds $\kappa$, and a similar derivation can be constructed for $\ulcorner \mathsf{out}\, r \urcorner$. Since all other constructions are translated homomorphically, the first part of the following theorem is a routine induction:

**Theorem 4.1 (Typing and reduction are simulated)**

1. *If $\Gamma \vdash t : A$ in the iso-recursive system, then $\ulcorner \Gamma \urcorner \vdash \ulcorner t \urcorner : \ulcorner A \urcorner$ in $\mathsf{F}^{\widehat{\ }}_\omega$.*
2. *If $t \longrightarrow t'$ in the iso-recursive system, then $\ulcorner t \urcorner \longrightarrow^+ \ulcorner t' \urcorner$ in $\mathsf{F}^{\widehat{\ }}_\omega$.*

Simulation of reduction is easy as well. We have chosen the translation such that $\ulcorner \mathsf{in}\, t \urcorner$ is a non-recursive value and $\ulcorner \mathsf{out}\, \_ \urcorner$ is a non-recursive evaluation frame. Hence,

$$\begin{array}{lcll} \mathsf{fix}^\mu_0\; s\; \ulcorner \mathsf{in}\, t \urcorner & \longrightarrow & s\,(\mathsf{fix}^\mu_0\; s)\; \ulcorner \mathsf{in}\, t \urcorner, & \text{and} \\ \ulcorner \mathsf{out}\,(\mathsf{fix}^\nu_0\; s) \urcorner & \longrightarrow & \ulcorner \mathsf{out}\,(s\,(\mathsf{fix}^\nu_0\; s)) \urcorner, & \end{array}$$

which can be lifted to (co)recursive functions with more arguments. It is also clear that $\ulcorner \mathsf{out}\,(\mathsf{in}\, t) \urcorner = (\lambda k.\, k\, \ulcorner t \urcorner)\; \mathsf{id} \longrightarrow^+ \ulcorner t \urcorner$. It follows that strong normalization of the iso-recursive system is inherited from $\mathsf{F}^{\widehat{\ }}_\omega$.

**Remark 4.2** Embedding the equi-recursive into the iso-recursive seems much harder, we need at least to translate the typing derivations. This suggests that the equi-recursive is more foundational than the iso-recursive one and justifies our choice of $\mathsf{F}^{\widehat{\ }}_\omega$.

## 4.2 Some Systems for Termination

In the following, we discuss the embedding of several systems which can be seen as precursors to $\mathsf{F}^{\widehat{\ }}_\omega$. Here, we concentrate on terminating systems with inductive types, and in the next section we switch to the dual case of productive systems with coinductive types.

**The calculus $\lambda^{\widehat{\ }}$.** The closest relative to $\mathsf{F}^{\widehat{\ }}_\omega$ is definitively $\lambda^{\widehat{\ }}$, a calculus introduced by Barthe, Frade, Giménez, Pinto, and Uustalu [BFG$^+$04] and in depth studied in Frade's Ph.D. thesis [Fra03]. It is a simply-typed $\lambda$-calculus with sized iso-inductive types and a reduction rule for recursive functions which is

similar to ISO-REC. It does not have first-class polymorphism (it features ML-polymorphism for types and sizes) nor inductive type *constructors*. The extension of admissible types we will consider in Chapter 5 is also not present in their system. We claim here that $\lambda^{\widehat{}}$ smoothly embeds into $\mathsf{F}^{\widehat{\omega}}$ which is strictly more powerful. The only technical work involved to construct an embedding is the translation of $\lambda^{\widehat{}}$'s Haskell-style inductive types into our equi-recursive types, but this is standard: use product and disjoint sum types.

**My calculus $\Lambda^+_\mu$** [Abe04]has simple iso-inductive types and their approximations, which are an notational variant of sized types with the disadvantage that they do not integrate well with polymorphism.[1] Similar problematic formulations have been devised by Amadio and Coupet-Grimal [ACG98], Giménez [Gim98], and Barras [Bar99]. The type syntax of $\lambda^{\widehat{}}$ and $\mathsf{F}^{\widehat{\omega}}$, which resembles Hughes, Pareto, and Sabry's *Synchronous Haskell* [HPS96, Par00], is a clear improvement: it is both more intuitive and more powerful since it is orthogonal to all other type constructions, like, for instance, polymorphism. $\Lambda^+_\mu$ embeds into $\lambda^{\widehat{}}$, and, even more directly, into $\mathsf{F}^{\widehat{\omega}}$.

**Some syntactic termination criteria.** Giménez proposed a termination criterion on well-typed terms called *guarded by destructors*. Barthe et al. [BFG$^+$04] formalized it in the calculus $\lambda_\mathcal{G}$, which embeds into $\lambda^{\widehat{}}$, therefore also into $\mathsf{F}^{\widehat{\omega}}$. In previous work [Abe00] I have described functional language foetus with inductive types and nested recursion, which embeds into $\Lambda^+_\mu$. Note, however, that the termination checker foetus that is described in [AA02] handles also cases of mutual recursion that cannot be expressed by nested recursion, and, thus, not be simulated in $\mathsf{F}^{\widehat{\omega}}$.

## 4.3 Some Systems For Productivity

Barthe et al. [BFG$^+$04] sketch an extension of $\lambda^{\widehat{}}$ by coinductive types. They do not extend the proofs of subject reduction and strong normalization, but no complications are to be expected there, partly because they work with iso-coinductive types. (In the case of equi-coinductive types, the definition of saturated set needs to be refined, see Section 3.4.3.) The extension of $\lambda^{\widehat{}}$ by coinduction also embeds into $\mathsf{F}^{\widehat{\omega}}$.

[1] $\Lambda^+_\mu$ uses bounded quantification to model approximations of an inductive type, whereas $\mathsf{F}^{\widehat{\omega}}$ uses size polymorphism. Consider the type of the list filter function:

$$\begin{array}{ll} \mathsf{F}^{\widehat{\omega}} & (A \to \mathsf{Bool}) \to \forall \iota.\, \mathsf{List}^\iota A \to \mathsf{List}^\iota A \\ \Lambda^+_\mu & (A \to \mathsf{Bool}) \to \forall Y \leq \mathsf{List}\, A.\, Y \to Y \end{array}$$

This worked ok, but the type of the map function $(A \to B) \to \mathsf{List}^\iota A \to \mathsf{List}^\iota B$ cannot be expressed in $\Lambda^+_\mu$.

**My calculus** $\lambda^{\mathsf{fix}\mu\nu}$ [Abe03]is an extension of $\Lambda_\mu^+$ [Abe04] by coinductive types and continuity (which will be discussed in Chapter 5). It allows corecursive functions to return pairs, which is not allowed in $\mathsf{F}_{\widehat{\omega}}$. But taking away this slightly buggy feature[2], $\lambda^{\mathsf{fix}\mu\nu}$ embeds into $\mathsf{F}_{\widehat{\omega}}$.

**The guard condition by Amadio and Coupet-Grimal [ACG98].** They present a $\lambda$-calculus with coinductive types and nested corecursion, with a more restrictive syntax than $\lambda^{\mathsf{fix}\mu\nu}$. Besides proving strong normalization with Girard's reducibility method, they also give a PER model to reason about corecursive programs. The version of their calculus with reduction rules embeds into $\lambda^{\mathsf{fix}\mu\nu}$, therefore also into $\mathsf{F}_{\widehat{\omega}}$.

**Syntactic guardedness.** Coquand [Coq93] first described a syntactic criterion for productivity: each recursive call must be guarded, i. e., occur under a constructor. This condition is easy to check, but quite restrictive; it rejects advanced corecursion schemes, as, for instance, used for the Huffman decoder in Section 3.2.4. Coquand's criterion is subsumed by Amadio and Coupet-Grimal's calculus.

## 4.4 Iteration and Primitive Recursion

There are a number of calculi without explicit recursion that are normalizing by virtue of typing: the simply-typed $\lambda$-calculus, the polymorphic $\lambda$-calculus (System $\mathsf{F}$) and the higher-order polymorphic $\lambda$-calculus (System $\mathsf{F}^\omega$), to name a few examples. In System $\mathsf{F}$, a limited form of recursion, so-called *iteration*, can be simulated, but not *primitive recursion*, a strengthening of iteration. This negative result has been proven for the $\beta$-version of System $\mathsf{F}$ by Splawski and Urzyczyn [SU99], for $\beta\eta$ it is conjectured. Primitive recursion can be simulated in the presence of positive fixed-point types [Geu92]. The emphasis is on *positive*, as opposed to *strictly positive*. Since $\mathsf{F}_{\widehat{\omega}}$ has non-strictly positive least and greatest fixed-point types, this simulation of primitive recursion can be carried out in $\mathsf{F}_{\widehat{\omega}}$, even without the use of $\mathsf{fix}$.

Using $\mathsf{fix}$, primitive recursion is even directly definable, which is not really surprising, considering the flexibility of the typing of $\mathsf{fix}$. For example, the recursor for lists can be programmed as follows:

$$
\begin{array}{lcl}
\mathsf{primRec} & : & \forall A \forall B.\, B \to (A \to \mathit{List}^\infty\, A \to B \to B) \to \mathsf{List}^\infty A \to B \\
\mathsf{primRec} & := & \lambda n \lambda c.\ \mathsf{fix}_0^\mu\, \lambda \mathit{primRec} \lambda l.\ \mathsf{match}\ l\ \mathsf{with} \\
 & & \quad \mathsf{nil} \quad\ \mapsto\ n \\
 & & \quad \mathsf{cons}\, a\, as \ \mapsto\ c\, a\, as\, (\mathit{primRec}\, as)
\end{array}
$$

(Better known is the *iterator* for lists, called `foldr` in Haskell, which we get from the recursor if we omit the argument $as : \mathsf{List}^\infty$ to $c$.)

[2] I later discovered that in $\lambda^{\mathsf{fix}\mu\nu}$ there are not enough reductions for corecursive functions [Abe05].

The type we have given to primRec above is its usual System F type. In $\mathsf{F}_{\widehat{\omega}}$, we can give it a more precise type:

$$\begin{array}{lcl} \mathsf{primRec} & : & \forall A\, \forall B : \mathsf{ord} \xrightarrow{+} *. \\ & & \quad (\forall \imath.\, B\,(\imath + 1)) \to \\ & & \quad (\forall \imath.\, A \to \mathsf{List}^{\imath}\, A \to B\,\imath \to B\,(\imath + 1)) \to \\ & & \quad \forall \imath.\, \mathsf{List}^{\imath} A \to B\,\imath \end{array}$$

**Mendler-style recursion.** Mendler was the first to present formulations of (co)iteration [Men91] and primitive (co)recursion [Men87] that resemble generalized recursion, but are in their power limited through typing. His achievements make him the "father" of type-based termination. The systems discussed in the last two sections are inspired by Mendler and tried to improve on his work. Mendler proved strong normalization of his iteration and recursion constructs by transfinite induction, similarly to how we motivate and verify our recursion rule. The semantic model behind Mendler's recursion rule

$$\mathsf{MRec} : (\forall X.\, (X \to \mu\, F) \to (X \to C) \to F\, X \to C) \to \mu\, F \to C$$

is very nicely summarized by Splawski and Urzyczyn [SU99]: the variable $X$ stands for any approximation $\mu^{\alpha} F$ of the inductive type $\mu F$, and $F\, X = F\,(\mu^{\alpha} F) = \mu^{\alpha+1} F$ for the next approximation. If one defines a function

$$\mathsf{MRec}\,(\lambda emb \lambda rec \lambda t.s)$$

by Mendler recursion, then the function body $s : C$ constructs a result from argument $t : \mu^{\alpha+1} F$ where it may invoke recursion $rec : \mu^{\alpha} F \to C$ on inductive data of the previous approximation stage. Additionally, it can convert this piece of data into the full inductive type using $emb : \mu^{\alpha} F \to \mu^{\infty} F$ (this conversion option is absent in case of Mendler iteration).

Semantically, Mendler already used sized types, but he decided to conceal them using a type variable $X$. This way, he does not extend the type syntax of polymorphic $\lambda$-calculus, which makes his solution quite elegant.[3] However, subsequent research seems to have been fixed to this *type variable* representation of approximations $\mu^{\alpha} F$ of an inductive type. The resulting type systems by Giménez [Gim98] and myself [Abe03, Abe04] were a bit clumsy. Introduction of sizes into the type system, which was done by Hughes, Pareto, and Sabry [HPS96] in the functional programming community, cleaned up this messy syntax: Barthe et al. [BFG$^+$04] first realized that these sizes can be interpreted as ordinals, and $\mathsf{F}_{\widehat{\omega}}$ features the quite intuitive syntax $\mu^{\imath} F$.

Mendler uses positive iso-inductive types with the constructor $\mathsf{in} : F\,(\mu F) \to \mu F$. His recursion scheme is directly definable in the iso-recursive version of $\mathsf{F}_{\widehat{\omega}}$:

$$\mathsf{MRec} = \lambda s.\ \mathsf{fix}_0^{\mu}\, \lambda mrec \lambda t.\, s\, \mathsf{id}\, mrec\, (\mathsf{out}\, t)$$

[3] The *syntax* seems to be a sacred cow in the logic, type theory, and programming language community: Syntax extensions and changes might not be accepted by the fellow researchers. On the other hand, new and better notation is a catalyzer for new ideas, as the history of mathematics shows.

It was later observed by Matthes [Mat98] and Uustalu and Vene [UV99] that Mendler-style inductive types do not have to be positive, since there is no destructor $\mathsf{out} : \mu F \to F(\mu F)$. In the absence of a destructor, negative types do not jeopardize normalization since for them the recursion principle $\mathsf{MRec}$ is simply useless. The non-positive version of Mendler inductive types cannot be simulated directly, but via the usual encoding into positive fixed-point types [Geu92].

**Iteration and primitive recursion for inductive constructors.** For (co)inductive constructors of higher kind, Mendler (co)iteration and (co)recursion can be defined analogously. This has been demonstrated in joint work with Matthes and Uustalu [AMU03, AMU05]. The encoding of $\mathsf{MRec}$ given above can be lifted to higher kinds and makes crucial use of inductive constructors of higher kind, which are present in $\mathsf{F}^{\widehat{\omega}}$ but not in $\lambda^{\widehat{\ }}$. For non-positive Mendler-style constructors, the recursion schemes can be encoded into positive fixed-point constructors as shown in joint work with Matthes [AM04]. Conventional iteration for inductive constructors [AM03] embeds into Mendler-style iteration [AMU05]; but conventional primitive recursion for higher kinds has not yet been studied.

We have mainly considered the inductive case, but all embeddings dualize to coinduction.

# Chapter 5

# Continuity

In this chapter, we turn our attention to the question which types are admissible for (co)recursive functions. In sections 3.4.2 and 3.4.4 we have given semantical criteria. The syntactical criteria in Section 3.1 were quite crude approximations, now we try to do better.

As described in the introduction, there are two criteria for admissible (semantical) types $\mathcal{A} : \mathsf{ord} \to *$.

1. $\mathcal{A}(0)$ must be the biggest type, the universe of terms,

2. $\inf_{\alpha<\lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$.

The first criterion is fulfilled by the general shape of the type: In case of recursion, $\mathcal{A}$ must be a function type with one domain which is an inductive type, in case of corecursion, it must be an coinductive type or a function type with a coinductive type as codomain. We will not try to relax the first criterion here, some directions of future work are hinted at in Section 7.2. Hughes, Pareto, and Sabry [HPS96] do have a wider criterion, but they only give a denotational semantics and do not have to deal with the extra complications we have by allowing reduction in any order and under binders. Their criterion is not sound in our reduction semantics.

The second criterion looks like a continuity property, if we write it as

$$\inf_{\alpha<\lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\sup_{\alpha<\lambda} \alpha).$$

However, it talks about infimum and supremum instead of limits and it is an inequation instead of an equation. In previous work [Abe03] we characterized some types that fulfill this condition syntactically. In the following, we generalize this work to polymorphism and type constructors. At the end of this chapter, we will have a syntactical derivation system for admissible types which is much less restrictive than the original one in Section 3.1.

**Supremum on $[\![*]\!]$.** In Section 3.4 we introduced two flavors of saturated sets. The second version defined a saturated set as the orthogonal of a set of evaluation contexts, and we could not establish that the supremum of saturated sets is identical to their union. However, for the development in this chapter, especially the proof of the central lemma 5.8, we require that $t \in \sup_{i\in I} \mathcal{A}_i$ implies $t \in \mathcal{A}_i$ for some $i$. It could be that this property holds for saturated sets in the refined sense, but we do not know. However, it holds for saturated sets in the original sense, therefore, we revert to the original definition. The price we have to pay is that coinductive constructors can no longer be treated in "equi" style but we have to switch to "iso" style. The necessary changes to $\mathsf{F}^{\widehat{\omega}}$ and its soundness proof are spelled out in Appendix B. In the following, we assume that the supremum on $[\![*]\!]$ is the set-theoretical union.

## 5.1 On the Necessity of Criterion 2

In a previous article [Abe04, Sec. 6.1] we proved that the type-based termination system indeed becomes unsound if we drop criterion 2: In the absence of this criterion, one can construct a looping term. In the following, we will convert this example to $\mathsf{F}^{\widehat{\omega}}$:

Assume sized type Nat with the two constructors $\mathsf{zero} : \forall\iota.\, \mathsf{Nat}^{\iota+1}$ and $\mathsf{succ} : \forall\iota.\, \mathsf{Nat}^{\iota} \to \mathsf{Nat}^{\iota+1}$. For simplicity of presentation, we assume that we have a simple pattern-matching facility. We define the following functions:

$$
\begin{array}{lcl}
\mathsf{shift} & : & \forall\iota.\, (\mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota+2}) \to \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota+1} \\
\mathsf{shift} & := & \lambda f \lambda n.\, \mathsf{match}\ f\ (\mathsf{succ}\, n)\ \mathsf{with} \\
 & & \quad \mathsf{zero} \ \mapsto\ \mathsf{zero} \\
 & & \quad \mathsf{succ}\, m \ \mapsto\ m \\
\mathsf{plus2} & : & \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\infty} \\
\mathsf{plus2} & := & \lambda n.\, \mathsf{succ}\, (\mathsf{succ}\, n)
\end{array}
$$

It is easy to see that plus2 is a fixed point of shift:

$$
\begin{array}{l}
\mathsf{shift}\ \mathsf{plus2} \ :\ \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\infty} \\
\mathsf{shift}\ \mathsf{plus2} \ \longrightarrow^{+}\ \mathsf{plus2}
\end{array}
$$

Disregarding Criterion 2, the following function is well-typed:

$$
\begin{array}{lcl}
\mathsf{loop} & : & \forall\iota.\, \mathsf{Nat}^{\iota} \to (\mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota+1}) \to \mathsf{Nat}^{\infty} \\
\mathsf{loop} & := & \mathsf{fix}^{\mu}_{0}\, \lambda \mathit{loop} \lambda\_ \lambda f.\, \mathsf{match}\ (f\, \mathsf{zero})\ \mathsf{with} \\
 & & \quad \mathsf{zero} \ \mapsto\ \mathsf{zero} \\
 & & \quad \mathsf{succ}\, n \ \mapsto\ \mathsf{match}\ n\ \mathsf{with} \\
 & & \qquad \mathsf{zero} \ \mapsto\ \mathsf{zero} \\
 & & \qquad \mathsf{succ}\, m \ \mapsto\ \mathit{loop}\, m\, (\mathsf{shift}\, f)
\end{array}
$$

To see well-typedness, here some help with the types of the bound variables

and some subterms:

$$\begin{array}{lcl} \mathit{loop} & : & \mathsf{Nat}^{\iota} \to (\mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota+1}) \to \mathsf{Nat}^{\infty} \\ \_ & : & \mathsf{Nat}^{\iota+1} \\ f & : & \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota+2} \\ n & : & \mathsf{Nat}^{\iota+1} \\ m & : & \mathsf{Nat}^{\iota} \\ \mathsf{shift}\, f & : & \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota+1} \end{array}$$

Now a certain function call loops:

$$\mathsf{loop}\ \mathsf{zero}\ \mathsf{plus2} \longrightarrow^{+} \mathsf{loop}\ \mathsf{zero}\ (\mathsf{shift}\ \mathsf{plus2}) \longrightarrow^{+} \mathsf{loop}\ \mathsf{zero}\ \mathsf{plus2}.$$

It is also not the case that the occurrence of $\iota + 1$ in the type of loop is causing the non-termination. We can modify the counterexample such that the $+1$ disappears from the type of loop. To this end, introduce a free variable $\mathit{err} : \forall A.\, A$. If we modify the definition of shift such that in the case zero the variable *err* is returned, then shift can be given type $\forall \iota.\ (\mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota+1}) \to \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota}$. If we remove one pattern matching layer from loop it gets type $\forall \iota.\, \mathsf{Nat}^{\iota} \to (\mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota}) \to \mathsf{Nat}^{\infty}$, and loop zero $(\lambda n.\, \mathsf{succ}\, n)$ loops. Thus, it is really the negative occurrence of the type $\mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\iota[+1]}$ that enables a looping term.

## 5.2 Semi-Continuity

When we define a (co-)recursive object $t$ of type $\forall \iota : \mathsf{ord}.\ A(\iota)$, we prove in the semantics that $t \in \mathcal{A}(\alpha)$ for all ordinals $\alpha \leq \top^{\mathsf{ord}}$ by transfinite induction on $\alpha$. In the case of a limit ordinal $\lambda$, we have $t \in \bigcap_{\alpha<\lambda} \mathcal{A}(\alpha) = \inf_{\alpha<\lambda} \mathcal{A}(\alpha)$ and need to show $t \in \mathcal{A}(\lambda)$. Hence, $\mathcal{A}$ must be a function over ordinals with the property

$$\inf_{\lambda} \mathcal{A} \subseteq \mathcal{A}(\lambda). \tag{5.1}$$

We are looking for a syntactical characterization of types that fulfill (5.1). For instance, what is required of $A$ and $B$ such that $C\,\iota = A\,\iota \to B\,\iota$ has this property? We would expect that $B$ has to fulfill (5.1) and $A$ some dual property. However, it is not that simple: To show $\inf_{\alpha<\lambda}(\mathcal{A}(\alpha) \to \mathcal{B}(\alpha)) \subseteq \mathcal{A}(\lambda) \to \mathcal{B}(\lambda)$ from $\inf_{\lambda} \mathcal{B} \subseteq \mathcal{B}(\lambda)$ requires that $\mathcal{A}(\lambda) \subseteq \inf_{\lambda} \mathcal{A}$, which is not even fulfilled if $\mathcal{A}(\alpha)$ denotes the natural numbers below $\alpha$. This means that not even for type $\mathsf{Nat}^{\iota} \to \mathsf{Nat}^{\iota}$ we can show property (5.1) directly.

The next idea is to strengthen the requirement to $\lim_{\lambda} \mathcal{A} \subseteq \mathcal{A}(\lambda)$ which by Lemma 3.14 entails (5.1). Here we hit the problem that many function spaces do not have limits. For example, let $\mathcal{T}(\alpha) = \mathcal{N}at(\alpha) \boxrightarrow \mathcal{N}at(\alpha)$ denote the family of endo-functions on natural numbers below $\alpha$. Consider the function $f$ which increases even numbers by one, but leaves odd numbers unchanged. This function inhabits $\mathcal{T}(\alpha)$ for infinitely many $\alpha < \omega$ (namely every second $\alpha$), hence $f \in \limsup_{\omega} \mathcal{T}$. But there is no $\alpha_0 < \omega$ such that $f \in \mathcal{T}(\alpha)$ for all

$\alpha$ between $\alpha_0$ and $\omega$, hence, $f \notin \liminf_\omega \mathcal{T}$. Since limes superior and inferior differ, $\lim_\omega \mathcal{T}$ does not exist.

Although limits might not exist, it has been observed by Hughes, Pareto and Sabry [HPS96] and the author [Abe03] that the modified requirement

$$\liminf_\lambda \mathcal{A} \subseteq \mathcal{A}(\lambda) \tag{5.2}$$

can be lifted over function types:

**Proposition 5.1** *Let* $\mathsf{O}$ *be some initial segment of the ordinal numbers,* $\lambda \in \mathsf{O}$ *a limit ordinal, and* $\mathcal{A}, \mathcal{B} \in \mathsf{O} \to \mathsf{SAT}$. *If* $\mathcal{A}(\lambda) \subseteq \liminf_\lambda \mathcal{A}$ *and* $\liminf_\lambda \mathcal{B} \subseteq \mathcal{B}(\lambda)$ *then* $\liminf_{\alpha\to\lambda} (\mathcal{A}(\alpha) \boxrightarrow \mathcal{B}(\alpha)) \subseteq \mathcal{A}(\lambda) \boxrightarrow \mathcal{B}(\lambda)$.

This proposition is true, and the condition on $\mathcal{A}$, $\mathcal{A}(\lambda) \subseteq \liminf_\lambda \mathcal{A}$, is exactly what we need. The condition on $\mathcal{B}$ is sufficient, but rather ad hoc, and we will replace it by the stronger requirement

$$\limsup_\lambda \mathcal{B} \subseteq \mathcal{B}(\lambda) \tag{5.3}$$

which is the dual to the requirement on $\mathcal{A}$. The nature of his choice is not technical but aesthetical: symmetry. Further, now the requirements on $\mathcal{A}$ and $\mathcal{B}$ correspond to already-known mathematical concepts:

**Semi-continuity.** Let $(\mathfrak{L}, \sqsubseteq, \inf, \sup)$ be a complete lattice, $\mathsf{O}$ some initial segment of the ordinal numbers and $\lambda \in \mathsf{O}$ a limit ordinal. A function $\mathcal{A} \in \mathsf{O} \to \mathfrak{L}$ is called *upper semi-continuous in* $\lambda$ if

$$\limsup_\lambda \mathcal{A} \sqsubseteq \mathcal{A}(\lambda). \tag{5.4}$$

It is called *lower semi-continuous in* $\lambda$ if

$$\mathcal{A}(\lambda) \sqsubseteq \liminf_\lambda \mathcal{A}. \tag{5.5}$$

The function $\mathcal{A}$ is *upper/lower semi-continuous*, if it is so in all limit ordinals $\lambda \in \mathsf{O}$. A function is called *$\omega$-overshooting* by Hughes, Pareto, and Sabry [HPS96], if it is lower semi-continuous in $\omega$.

The term *semi-continuity* is justified by the following observation:

**Lemma 5.2** *If* $\mathcal{A}$ *is both upper and lower semi-continuous in* $\lambda$*, then it is continuous in* $\lambda$.

*Proof.* By assumption, $\limsup_\lambda \mathcal{A} \sqsubseteq \mathcal{A}(\lambda) \sqsubseteq \liminf_\lambda \mathcal{A}$. By Lemma 3.14, the limes inferior is below the limes superior, hence both are equal and the limit $\lim_\lambda \mathcal{A}$ exists. Since $\mathcal{A}(\lambda)$ is sandwiched between the limit on both sides, it is equal to the limit. Thus, $\mathcal{A}$ is continuous in $\lambda$. □

**Remark 5.3 (Undershooting + overshooting ≠ continuous)** Property (5.2), $\liminf_\lambda \mathcal{A} \sqsubseteq \mathcal{A}(\lambda)$, which we have called *paracontinuous* in previous work [Abe03], and which would be called $\lambda$-*undershooting* following Hughes, Pareto, and Sabry [HPS96], is not strong enough to complement lower semi-continuity. For instance, consider the function $\mathcal{A} \in \mathsf{O} \to \mathfrak{L}$ which maps all odd finite ordinals to $\top$ and all others to $\bot$. It is both $\omega$-under- and -overshooting, since $\liminf_\omega \mathcal{A} = \mathcal{A}(\omega) = \bot$, but not continuous in $\omega$, since $\limsup_\omega \mathcal{A} = \top$.

**Set of semi-continuous functions.** We denote the set of upper semi-continuous functions from $\mathsf{O}$ to $\mathfrak{L}$ by $\mathsf{O} \xrightarrow{\oplus} \mathfrak{L}$, and the set of lower semi-continuous functions by $\mathsf{O} \xrightarrow{\ominus} \mathfrak{L}$.

**Lemma 5.4 (Variance and semi-continuity)** $(\mathsf{O} \xrightarrow{+} \mathfrak{L}) \subseteq (\mathsf{O} \xrightarrow{\oplus} \mathfrak{L})$ *and* $(\mathsf{O} \xrightarrow{-} \mathfrak{L}) \subseteq (\mathsf{O} \xrightarrow{\ominus} \mathfrak{L})$.

**Example 5.5 (Ordinal successor)** The ordinal successor is upper semi-continuous, since it is monotone, but not lower semi-continuous.

The concepts (5.4) and (5.5) are clearly subconcepts of continuity. Familiar subconcepts are *continuity from the left/right*, but since we are considering functions over ordinals, every function is continuous from the right.[1] But we could ask the following question: what makes a monotone function on ordinals continuous (from the left)? We find:

1. An antitone function is already continuous if it is upper semi-continuous.
2. An isotone function is already continuous if it is lower semi-continuous.

So we could say that an upper semi-continuous function is *continuous when falling* and a lower semi-continuous function is *continuous when climbing*.

## 5.3 Type Constructors and Semi-Continuity

In the following, we will analyze which type constructors preserve upper and lower semi-continuity. For the first kind, we need to push an upper limit under the type constructor, for the second kind, we need to pull a limit out from the argument of a type constructor.

**Preservation of upper semi-continuity.** We say lim sup *pushes through* $f \in \mathfrak{L} \to \mathfrak{L}'$ if for all $g \in \mathsf{O} \to \mathfrak{L}$ and all $\lambda \in \mathsf{O}$ it holds that

$$\limsup_{\alpha \to \lambda} f(g(\alpha)) \sqsubseteq f(\limsup_{\alpha \to \lambda} g(\alpha)).$$

[1] Minima always exists on the ordinals, hence, limits of falling sequences are all trivial, and since the ordinals are wellfounded, there are no infinite *strictly* falling sequences.

We also say that $f$ is a lim sup-*pushable* function.

Most of our type constructors are covariant. This allows the following reasoning: If $\mathcal{F} \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}'$ is lim sup-pushable and $\mathcal{A} \in \mathsf{O} \to \mathfrak{L}$ is upper semi-continuous, then

$$\limsup_{\alpha\to\lambda} \mathcal{F}(\mathcal{A}(\alpha)) \sqsubseteq \mathcal{F}(\limsup_{\alpha\to\lambda} \mathcal{A}(\alpha)) \sqsubseteq \mathcal{F}(\mathcal{A}(\lambda)),$$

hence, $\mathcal{F}$ preserves upper semi-continuity.

**Remark 5.6 (Covariance alone not sufficient)** In general, covariance is not sufficient to preserve upper semi-continuity. Consider the function $g : [0,1] \to [0,1]$ on the real or rational unit interval with $g(x) = 1 - x$. Then clearly $g$ is (left) upper semi-continuous at 1, since $g$ is even continuous. However, taking $f$ to be the sign function such that $f(x) = 1$ if $x > 0$ and $f(0) = 0$, we have $\limsup_{x\to 1} f(g(x)) = 1$, but $f(g(1)) = 0$.

**Preservation of lower semi-continuity.** We say lim inf can be *pulled through* $f \in \mathfrak{L} \to \mathfrak{L}'$ if for all $g \in \mathsf{O} \to \mathfrak{L}$ and all $\lambda \in \mathsf{O}$ it holds that

$$f(\liminf_{\alpha\to\lambda} g(\alpha)) \sqsubseteq \liminf_{\alpha\to\lambda} f(g(\alpha)).$$

We also say that $f$ is a lim inf-*pullable* function.

If $\mathcal{F} \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}'$ is lim inf-pullable and $\mathcal{A} \in \mathsf{O} \to \mathfrak{L}$ is lower semi-continuous, then

$$\mathcal{F}(\mathcal{A}(\lambda)) \sqsubseteq \mathcal{F}(\liminf_{\alpha\to\lambda} \mathcal{A}(\alpha)) \sqsubseteq \liminf_{\alpha\to\lambda} \mathcal{F}(\mathcal{A}(\alpha)),$$

thus, $\mathcal{F}$ preserves lower semi-continuity.

**Contravariant constructors.** If $\mathcal{F}$ is contravariant and a lim sup can be pushed into $\mathcal{F}$, becoming a lim inf, then $\mathcal{F}$ turns lower semi-continuous functions into upper semi-continuous ones. If a lim sup can be pulled out from $\mathcal{F}$ as a lim inf, then $\mathcal{F}$ maps a upper semi-continuous function to an lower semi-continuous one.

**Lemma 5.7** *Let $f \in \mathsf{O} \times \mathsf{O} \to \mathfrak{L}$ and $\lambda \in \mathsf{O}$ proper limit. Then*

$$\begin{aligned} \limsup_{\alpha\to\lambda} f(\alpha,\alpha) &\sqsubseteq \limsup_{\beta\to\lambda} \limsup_{\gamma\to\lambda} f(\beta,\gamma) \\ \liminf_{\beta\to\lambda} \liminf_{\gamma\to\lambda} f(\beta,\gamma) &\sqsubseteq \liminf_{\alpha\to\lambda} f(\alpha,\alpha) \end{aligned}$$

*Proof.* Let all ordinals range below $\lambda$.

$$\begin{aligned}
\sup_{\alpha\geq\alpha_0} f(\alpha,\alpha) &\sqsubseteq \sup_{\beta\geq\alpha_0}\sup_{\gamma\geq\alpha_0} f(\beta,\gamma) && \text{for all } \alpha_0\\
&= \inf_{\beta_0\leq\alpha_0}\inf_{\gamma_0\leq\alpha_0}\sup_{\beta\geq\beta_0}\sup_{\gamma\geq\gamma_0} f(\beta,\gamma)\\
\inf_{\alpha_0}\sup_{\alpha\geq\alpha_0} f(\alpha,\alpha) &\sqsubseteq \inf_{\alpha_0}\inf_{\beta_0\leq\alpha_0}\inf_{\gamma_0\leq\alpha_0}\sup_{\beta\geq\beta_0}\sup_{\gamma\geq\gamma_0} f(\beta,\gamma) && \text{taking inf on both sides}\\
&= \inf_{\beta_0}\inf_{\gamma_0}\sup_{\beta\geq\beta_0}\sup_{\gamma\geq\gamma_0} f(\beta,\gamma)\\
&= \inf_{\beta_0}\sup_{\beta\geq\beta_0}\inf_{\gamma_0}\sup_{\gamma\geq\gamma_0} f(\beta,\gamma)
\end{aligned}$$

This means that $\limsup_{\alpha\to\lambda} f(\alpha,\alpha) \sqsubseteq \limsup_{\beta\to\lambda}\limsup_{\gamma\to\lambda} f(\beta,\gamma)$. The second claim follows by dualization. □

### 5.3.1 Function Space

In this section, we show that function spaces can be upper semi-continuous, but not lower semi-continuous.[2]

**Lemma 5.8 (Pushing $\limsup$ through function space)** *Let $\mathcal{A},\mathcal{B}\in\mathsf{O}\to\mathcal{P}(\mathsf{Tm})$ and $\lambda\in\mathsf{O}$ a limit ordinal. Then*

$$\limsup_{\alpha\to\lambda}\,(\mathcal{A}(\alpha)\boxed{\to}\mathcal{B}(\alpha))\subseteq(\liminf_{\lambda}\mathcal{A})\boxed{\to}\limsup_{\lambda}\mathcal{B}.$$

*Proof.* We assume a function term $r\in\limsup_{\alpha\to\lambda}(\mathcal{A}(\alpha)\boxed{\to}\mathcal{B}(\alpha))$ and an argument term $s\in\sup_{\alpha_0<\lambda}\inf_{\alpha_0\leq\alpha\leq\lambda}\mathcal{A}(\alpha)$ and show that the application $r\,s\in\inf_{\beta_0<\lambda}\sup_{\beta_0\leq\beta<\lambda}\mathcal{B}(\beta)$. In the following, let all ordinals range below $\lambda$. We fix an arbitrary ordinal $\beta_0$. We know that there exists some $\alpha_0$ such that $s\in\mathcal{A}(\alpha)$ for all $\alpha\geq\alpha_0$. Furthermore, $r\in\mathcal{A}(\beta)\to\mathcal{B}(\beta)$ for some ordinal $\beta>\max(\alpha_0,\beta_0)$. Since $s\in\mathcal{A}(\beta)$, we conclude $r\,s\in\mathcal{B}(\beta)$. □

**Corollary 5.9** *Let $\mathcal{A}\in\mathsf{O}\xrightarrow{\ominus}\mathcal{P}(\mathsf{Tm})$, $\mathcal{B}\in\mathsf{O}\xrightarrow{\oplus}\mathcal{P}(\mathsf{Tm})$ and $\mathcal{F}(\alpha):=\mathcal{A}(\alpha)\boxed{\to}\mathcal{B}(\alpha)$. Then $\mathcal{F}\in\mathsf{O}\xrightarrow{\oplus}\mathcal{P}(\mathsf{Tm})$.*

*Proof.* Fix some limit ordinal $\lambda\in\mathsf{O}$. Since $\mathcal{A}(\lambda)\subseteq\liminf_\lambda\mathcal{A}$ and $\limsup_\lambda\mathcal{B}\subseteq\mathcal{B}(\lambda)$, the desired inequation

$$\limsup_{\alpha\to\lambda}\,(\mathcal{A}(\alpha)\boxed{\to}\mathcal{B}(\alpha))\subseteq\mathcal{A}(\lambda)\boxed{\to}\mathcal{B}(\lambda)$$

[2] An exception could be functions over finite domains, which are lower semi-continuous in their codomain. But we do not distinguish between finite and infinite here, this would require a syntactical calculus of finiteness later.

follows from the previous lemma, since the function space is contravariant on the domain and covariant on the codomain. □

The converse lemma to derive that $\mathcal{A}(\alpha) \boxrightarrow \mathcal{B}(\alpha)$ is lower semi-continuous does not hold. For instance $\mathcal{F}(\alpha) := \mathcal{N}at^{\omega} \to \mathcal{N}at^{\alpha}$ is not lower semi-continuous, although $\mathcal{N}at^{\alpha}$ is: the identity function is in $\mathcal{F}(\omega)$, but not in $\mathcal{F}(\alpha)$ for any $\alpha < \omega$.

**Remark 5.10 (Cannot pull** $\liminf$ **out of function space)** We show that not generally $(\limsup_{\alpha} \mathcal{A}(\alpha)) \boxrightarrow \mathcal{B} \subseteq \liminf_{\alpha}(\mathcal{A}(\alpha) \boxrightarrow \mathcal{B})$. Clearly, the bigger $(\liminf_{\alpha} \mathcal{A}(\alpha)) \boxrightarrow \mathcal{B}$ is then also not below $\liminf_{\alpha}(\mathcal{A}(\alpha) \boxrightarrow \mathcal{B})$ in general. Take $\mathcal{A}(n) = \mathcal{N}at^{n} \to \mathcal{N}at$ and $\mathcal{B} = \mathcal{N}at$. First, we show

$$\lambda f.\, f\,(f\,0) \in (\limsup_{n}(\mathcal{N}at^{n} \boxrightarrow \mathcal{N}at)) \boxrightarrow \mathcal{N}at.$$

We assume that for all $n_0$ there is some $n \geq n_0$ such that $f \in \mathcal{N}at^{n} \boxrightarrow \mathcal{N}at$ and show $f\,(f\,0) \in \mathcal{N}at$. Since $0 \in \mathcal{N}at^{1}$ we have $f \in \mathcal{N}at^{n} \to \mathcal{N}at$ for some $n \geq 1$ and therefore $f\,0 \in \mathcal{N}at$. Since $\mathcal{N}at = \bigcup_{n_0} \mathcal{N}at^{n_0}$, there must be a $n_0$ such that $f\,0 \in \mathcal{N}at^{n_0}$. Thus, $f\,(f\,0) \in \mathcal{N}at$ for some $n \geq n_0$. Now, we prove

$$\lambda f.\, f\,(f\,0) \notin \liminf_{n}((\mathcal{N}at^{n} \boxrightarrow \mathcal{N}at) \boxrightarrow \mathcal{N}at).$$

We show even that $\lambda f.\, f\,(f\,0)$ is in *none* of the sets $(\mathcal{N}at^{n} \boxrightarrow \mathcal{N}at) \boxrightarrow \mathcal{N}at$. Assume an arbitrary $n < \omega$. Let $f$ be the function which returns $n$ for inputs less than $n$ and diverges for all input greater or equal than $n$. Then $f \in \mathcal{N}at^{n} \boxrightarrow \mathcal{N}at$. But since $f\,0$ evaluates to $n$, $f\,(f\,0)$ diverges and, hence, cannot be in $\mathcal{N}at$.

### 5.3.2 Universal Quantification

Universal quantification is semantically interpreted by intersection. We can only show that quantification preserves upper semi-continuity.

**Lemma 5.11 (Pushing** $\limsup$ **through infimum)** *Let $I$ some index set, $f \in \mathsf{O} \times I \to \mathfrak{L}$, and $\lambda \in \mathsf{O}$ some limit ordinal. Then*

$$\limsup_{\alpha \to \lambda} \inf_{i \in I} f(\alpha, i) \sqsubseteq \inf_{i \in I} \limsup_{\alpha \to \lambda} f(\alpha, i).$$

*Proof.* For all $i \in I$ it holds that $\inf_{i \in I} f(\alpha, i) \sqsubseteq f(\alpha, i)$. Since inequation is preserved under limit formation, $\limsup_{\alpha \to \lambda} \inf_{i \in I} f(\alpha, i) \sqsubseteq \limsup_{\alpha \to \lambda} f(\alpha, i)$ for all $i \in I$, which entails our claim. □

**Corollary 5.12** *The space of upper semi-continuous functions $\mathsf{O} \overset{\oplus}{\to} \mathfrak{L}$ forms a complete lattice under pointwise ordering, pointwise infimum, and pointwise maximal element.*

*Proof.* To see that $\mathsf{O} \xrightarrow{\oplus} \mathfrak{L}$ is closed under pointwise infimum, assume a family $\mathcal{F}_i \in \mathsf{O} \to \mathfrak{L}$ (for $i \in I$) with $\limsup_{\alpha\to\lambda} \mathcal{F}_i(\alpha) \sqsubseteq \mathcal{F}_i(\lambda)$ for all $i \in I$. Minimizing both sides, we obtain

$$\limsup_{\alpha\to\lambda}(\inf_{i\in I} \mathcal{F}_i)(\alpha) \sqsubseteq \inf_{i\in I}(\limsup_{\alpha\to\lambda} \mathcal{F}_i(\alpha)) \sqsubseteq (\inf_{i\in I} \mathcal{F}_i)(\lambda)$$

by pulling the upper limit out to the left (previous lemma). □

**Example 5.13 (Supremum on $\mathsf{O} \xrightarrow{\oplus} \mathfrak{L}$ not pointwise)** The supremum of the lattice, $\sup_{i\in I} \mathcal{F}_i = \inf\{\mathcal{F} \mid \mathcal{F} \sqsupseteq \mathcal{F}_i \text{ for all } i \in I\}$, however, is *not* the pointwise one. For example, take the two element lattice $\bot \sqsubseteq \top$ and the family

$$\begin{array}{lcll} \mathcal{A}_i & : & [0;\omega] \to \{\bot,\top\} & \text{for } 0 \le i < \omega, \\ \mathcal{A}_i(\alpha) & := & \top & \text{if } i < \alpha, \\ \mathcal{A}_i(\alpha) & := & \bot & \text{else.} \end{array}$$

Then $\mathcal{A}_i(\omega) = \bot \sqsupseteq A_i(\alpha)$ for all $\alpha \ge i$, hence each $\mathcal{A}_i$ is upper semi-continuous in $\omega$. However, the pointwise supremum $\mathcal{A}(\alpha) := \sup_{i<\omega} \mathcal{A}_i(\alpha)$ takes the value $\top$ below $\omega$, hence $\limsup_\omega \mathcal{A} = \top \not\sqsubseteq \mathcal{A}(\omega) = \bot$.

On the contrary, *lower* semi-continuous functions are not closed under pointwise *infimum*. This becomes clear if we consider the same example on the dual lattice, i. e., we swap inf and sup, and $\sqsubseteq$ and $\sqsupseteq$ (which means that $\bot$ becomes the maximal element and $\top$ the minimal). With the same trick, we harvest a relationship between supremum and limes inferior:

**Lemma 5.14 (Pulling $\liminf$ out of a supremum)** *Let $I$ some index set, $f \in \mathsf{O} \times I \to \mathfrak{L}$, and $\lambda \in \mathsf{O}$ some limit ordinal. Then*

$$\sup_{i\in I} \liminf_{\alpha\to\lambda} f(\alpha, i) \sqsubseteq \liminf_{\alpha\to\lambda} \sup_{i\in I} f(\alpha, i).$$

We could now turn $\mathsf{O} \xrightarrow{\ominus} \mathfrak{L}$ into a complete lattice using the pointwise supremum. But we withstand this temptation since this would trick us into believing we could interpret universal quantification on this lattice.

**Remark 5.15** One would think that, if $A^\iota$ is lower semi-continuous, then also $\forall X.\, A^\iota$. But it is not true in our semantics. It is not clear whether it holds in another semantics, e. g., one with parametricity.

### 5.3.3 Coinductive Types

Clearly, coinductive types are not lower semi-continuous in general. For example, consider the stream of subsequent natural numbers $0, 1, \ldots$ which is an

element of $\mathsf{Stream}^{\omega}(\mathsf{Nat}^{\omega})$. Since for all $\alpha < \omega$, this stream is not contained in $\mathsf{Stream}^{\omega}(\mathsf{Nat}^{\alpha})$, the latter type family cannot be lower semi-continuous in $\omega$.

However, coinductive types map upper semi-continuous types to upper semi-continuous types under some conditions. We introduce a new symbol for *antitone iteration*, which is used to construct the semantics of coinductive types:

$$\begin{array}{lcl} \boldsymbol{\nu} & \in & \mathsf{O} \xrightarrow{-} (\mathfrak{L} \xrightarrow{+} \mathfrak{L}) \xrightarrow{+} \mathfrak{L} \\ \boldsymbol{\nu}^{\alpha}(\mathcal{F}) & = & \mathcal{F}^{\alpha}(\top) \end{array}$$

By Lemma 3.21, $\boldsymbol{\nu}$ is antitone in its first argument, hence by Lemma 3.17 we can replace the limit by an infimum at limit iterates: $\boldsymbol{\nu}^{\lambda} = \inf_{\alpha<\lambda} \boldsymbol{\nu}^{\alpha}$. This will enable us to reason about semi-continuity.

**Lemma 5.16** *Let $\phi \in \mathsf{O} \to \mathsf{O}$ and $I \subseteq \mathsf{O}$. Then*

1. $\sup_{\alpha\in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\inf_I \phi}$,
2. $\sup_{\alpha\in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsupseteq \boldsymbol{\nu}^{\inf_I \phi}$,
3. $\inf_{\alpha\in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsupseteq \boldsymbol{\nu}^{\sup_I \phi}$, *and*
4. $\inf_{\alpha\in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\sup_I \phi}$.

*Proof.* 1. and 3. follow directly from antitonicity. For 2., remember that each set of ordinals is left-closed, hence $\inf_I \phi = \phi(\alpha)$ for some $\alpha \in I$. The remaining proposition 4. is proven by cases on $\sup_I \phi$. If $\sup_I \phi$ is not a limit ordinal then $\sup_I \phi = \phi(\alpha)$ for some $\alpha \in I$. For this $\alpha$, clearly $\boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\sup_I \phi}$, which entails the lemma. Otherwise, if $\sup_I \phi$ is a limit ordinal, then by definition of $\boldsymbol{\nu}$ at limits we have to show $\inf_{\alpha\in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\beta}$ for all $\beta < \sup_I \phi$. By definition of the supremum, $\beta < \phi(\alpha)$ for some $\alpha$. Since $\boldsymbol{\nu}$ is antitone, $\boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\beta}$ from which we infer our subgoal by forming the infimum on the left hand side. □

Hence, we can push a supremum as an infimum under antitone iteration, and vice versa.

**Corollary 5.17** $\limsup_{\alpha<\lambda} \boldsymbol{\nu}^{\phi(\alpha)} = \boldsymbol{\nu}^{\liminf_{\alpha<\lambda} \phi(\alpha)}$.

**Lemma 5.18 (Coinductive types are $\limsup$-pushable)** *Let $(\mathcal{F}_{\alpha})_{\alpha\in\mathsf{O}}$ be a family of $\limsup$-pushable isotone endo-functions on a complete lattice $\mathfrak{L}$ with pointwise infimum and supremum. Then*

$$\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^{\beta} \mathcal{F}_{\alpha} \sqsubseteq \boldsymbol{\nu}^{\beta} \limsup_{\alpha\to\lambda} \mathcal{F}_{\alpha}.$$

*Proof.* By induction on $\beta$. In case $\beta = 0$, both sides become the maximum element of $\mathfrak{L}$. In the successor case we have

$$\begin{array}{lll}
\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^{\beta+1}\mathcal{F}_\alpha &= \limsup_{\alpha\to\lambda} \mathcal{F}_\alpha(\boldsymbol{\nu}^\beta \mathcal{F}_\alpha) & \\
&\sqsubseteq \limsup_{\alpha\to\lambda} \limsup_{\gamma\to\lambda} \mathcal{F}_\alpha(\boldsymbol{\nu}^\beta \mathcal{F}_\gamma) & \text{Lemma 5.7} \\
&\sqsubseteq \limsup_{\alpha\to\lambda} \mathcal{F}_\alpha(\limsup_{\gamma\to\lambda} \boldsymbol{\nu}^\beta \mathcal{F}_\gamma) & \mathcal{F}_\alpha \text{ pushable} \\
&\sqsubseteq \limsup_{\alpha\to\lambda} \mathcal{F}_\alpha(\boldsymbol{\nu}^\beta(\limsup_{\gamma\to\lambda} \mathcal{F}_\gamma)) & \mathcal{F}_\alpha \text{ isotone, i.h.} \\
&= (\limsup_{\alpha\to\lambda} \mathcal{F}_\alpha)(\boldsymbol{\nu}^\beta(\limsup_{\alpha\to\lambda} \mathcal{F}_\alpha)) & \text{inf, sup pointwise} \\
&= \boldsymbol{\nu}^{\beta+1}(\limsup_{\alpha\to\lambda} \mathcal{F}_\alpha). &
\end{array}$$

Note that in the last step we used that lim sup is defined pointwise on functions. In the remaining case $\beta = \lambda$ we exploit that lim sup pushes through infima (Lemma 5.11). □

**Corollary 5.19** *Under the assumptions of the last lemma.*

$$\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^{\phi(\alpha)}\mathcal{F}_\alpha \sqsubseteq \boldsymbol{\nu}^{\liminf_\lambda \phi} \limsup_{\alpha\to\lambda} \mathcal{F}_\alpha,$$

*Proof.* We combine Cor. 5.17 and Lemma 5.18.

$$\begin{array}{lll}
\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^{\phi(\alpha)}\mathcal{F}_\alpha &\sqsubseteq \limsup_{\alpha\to\lambda} \limsup_{\gamma\to\lambda} \boldsymbol{\nu}^{\phi(\alpha)}\mathcal{F}_\gamma & \text{Lemma 5.7} \\
&\sqsubseteq \limsup_{\alpha\to\lambda} \boldsymbol{\nu}^{\phi(\alpha)}(\limsup_{\gamma\to\lambda} \mathcal{F}_\gamma) & \text{Lemma 5.18} \\
&\sqsubseteq \boldsymbol{\nu}^{\liminf_{\alpha\to\lambda} \phi(\alpha)} \limsup_{\alpha\to\lambda} \mathcal{F}_\alpha & \text{Cor. 5.17 } \square
\end{array}$$

**Corollary 5.20 (Coinductive types preserve upper semi-continuity)** *For $\alpha \in \mathsf{O}$, let $\mathcal{F}_\alpha \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ be* lim sup*-pushable. The family $\mathcal{F}_\alpha$ be upper semi-continuous. Then*

$$\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^{\phi(\alpha)}\mathcal{F}_\alpha \sqsubseteq \boldsymbol{\nu}^{\liminf_\lambda \phi}\mathcal{F}_\lambda.$$

*Proof.* By the last corollary and upper semi-continuity of $\mathcal{F}_\alpha$. □

### 5.3.4 Inductive Types

The result of the last section can be dualized to inductive types. We define *isotone iteration*, $\boldsymbol{\mu}^\alpha \mathcal{F} = \mathcal{F}^\alpha(\bot)$.

**Lemma 5.21** *Let $\phi \in \mathsf{O} \to \mathsf{O}$ and $I \subseteq \mathsf{O}$. Then $\boldsymbol{\mu}^{\sup_I \phi} = \sup_{\alpha\in I} \boldsymbol{\mu}^{\phi(\alpha)}$ and $\boldsymbol{\mu}^{\inf_I \phi} = \inf_{\alpha\in I} \boldsymbol{\mu}^{\phi(\alpha)}$.*

**Corollary 5.22** $\boldsymbol{\mu}^{\liminf_\lambda \phi} = \liminf_{\alpha\to\lambda} \boldsymbol{\mu}^{\phi(\alpha)}$.

**Lemma 5.23 (Inductive types are $\liminf$-pullable)** *For $\alpha \in \mathsf{O}$, let $\mathcal{F}_\alpha \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ be $\liminf$-pullable. Then isotone iteration $\boldsymbol{\mu}^\beta \mathcal{F}_\alpha$ is $\liminf$-pullable, i. e.,*

$$\boldsymbol{\mu}^\beta \liminf_{\alpha\to\lambda} \mathcal{F}_\alpha \sqsubseteq \liminf_{\alpha\to\lambda} \boldsymbol{\mu}^\beta \mathcal{F}_\alpha.$$

**Corollary 5.24** *Let $\phi$ be a lower semi-continuous function on ordinals. In the context of the lemma,*

$$\boldsymbol{\mu}^{\phi(\lambda)} \liminf_{\alpha\to\lambda} \mathcal{F}_\alpha \sqsubseteq \liminf_{\alpha\to\lambda} \boldsymbol{\mu}^{\phi(\alpha)} \mathcal{F}_\alpha.$$

**Corollary 5.25 (Inductive types preserve lower semi-continuity)** *Under the assumptions of the last corollary isotone iteration $\boldsymbol{\mu}^{\phi(\alpha)} \mathcal{F}_\alpha$ is lower semi-continuous, i. e.,*

$$\boldsymbol{\mu}^{\phi(\lambda)} \mathcal{F}_\lambda \sqsubseteq \liminf_{\alpha\to\lambda} \boldsymbol{\mu}^{\phi(\alpha)} \mathcal{F}_\alpha,$$

*if the family $\mathcal{F}_\alpha$ is lower semi-continuous.*

Since isotone types are upper semi-continuous, many inductive types are trivially upper semi-continuous. First, every type $\boldsymbol{\mu}^\alpha \mathcal{F}$ where $\mathcal{F}$ does not depend on $\alpha$, but also, for example $\mathcal{L}ist^\alpha(\mathcal{N}at^\alpha)$ and $\mathcal{T}ree^\alpha(\mathcal{N}at, \mathcal{N}at^\alpha)$.

But there are other types which are upper semi-continuous as well, for instance $\mathcal{L}ist^\alpha(\mathcal{S}tream^\alpha(\mathcal{N}at))$. This type is acceptable since it's generator $\mathcal{F}_\alpha(\mathcal{X}) = \mathcal{U}nit \boxplus \mathcal{S}tream^\alpha(\mathcal{N}at) \boxtimes \mathcal{X}$ is *infimum continuous*[3], which means that

$$\mathcal{F}_\alpha(\mathcal{X}) \cap \mathcal{F}_\beta(\mathcal{Y}) \subseteq \mathcal{U}nit \boxplus (\mathcal{S}tream^\alpha(\mathcal{N}at) \cap \mathcal{S}tream^\beta(\mathcal{N}at)) \boxtimes (\mathcal{X} \cap \mathcal{Y}).$$

Pareto calls such generators *lenient* [Par00, p. 122], recasting the conditions as $\mathcal{F}_\alpha(\mathcal{X}) \cap \mathcal{F}_\beta(\mathcal{Y}) \subseteq \mathcal{F}_\alpha(\mathcal{X} \cap \mathcal{Y})$. Since Pareto's type generators are infimum continuous by definition ([HPS96, Restriction 3.2]), they are also lenient, hence, each inductive type is upper semi-continuous.

On the contrary, our type iterators are not lenient. This is because we allow function spaces in data types. Consider $\mathcal{F}_i(\mathcal{X}) = \mathcal{N}at^i \to \mathcal{X}$ which constructs $\mathcal{N}at^i$-hungry functions. Then the identity function is in $\mathcal{F}_{j+1}(\mathcal{N}at^{j+1})$ and in $\mathcal{F}_j(\mathcal{N}at^j)$, but not in $\mathcal{F}_{j+1}(\mathcal{N}at^{j+1} \cap \mathcal{N}at^j) = \mathcal{N}at^{j+1} \to \mathcal{N}at^j$.

If all inductive types would preserve upper semi-continuity, we would have the chain

$$\limsup_{i\to\omega} \boldsymbol{\mu}^\beta \mathcal{F}_i \subseteq \boldsymbol{\mu}^\beta \limsup_{i\to\omega} \mathcal{F}_i \subseteq \boldsymbol{\mu}^\beta \mathcal{F}_\omega \tag{5.6}$$

for a isotone, upper semi-continuous type constructor which is lim sup-pushable. In the following, we will construct a counterexample to our hypothesis.

[3] A function $f$ is infimum continuous if $\inf_{i\in I} f(x_i) = f(\inf_{i\in I} x_i)$ for all sequences $(x_i)_{i\in I}$. An infimum continuous function is isotone. An isotone function is infimum continuous if $\inf_{i\in I} f(x_i) \leq f(\inf_{i\in I} x_i)$. This last condition is what I called "continuous" in earlier work [Abe03].

**An inductive type which is not upper semi-continuous.** A minor variant of the last example, $\mathcal{F}_i(\mathcal{X}) = \mathbf{1} \boxplus (\mathcal{N}at^i \boxrightarrow \mathcal{X})$ constructs the family of trees which have at least $i$ "good" branches at each node. First, we observe that, using upper semi-continuity of sums and function space,

$$\begin{array}{lll} \limsup_{i\to\omega} \mathcal{F}_i(\mathcal{X}_i) & \subseteq & \mathbf{1} \boxplus (\limsup_{i\to\omega}(\mathcal{N}at^i \boxrightarrow \mathcal{X}_i)) \\ & \subseteq & \mathbf{1} \boxplus ((\liminf_{i\to\omega} \mathcal{N}at^i) \boxrightarrow (\limsup_{i\to\omega} \mathcal{X}_i)) \\ & = & \mathbf{1} \boxplus (\mathcal{N}at^\omega \boxrightarrow (\limsup_{i\to\omega} \mathcal{X}_i)) \\ & = & \mathcal{F}_\omega(\limsup_{i\to\omega} \mathcal{X}_i). \end{array}$$

Hence, the functor family $\mathcal{F}$ is lim sup-pushable, but, as we will show, isotone iteration of $\mathcal{F}$ is upper semi-continuous.

The $\omega$th iteration $\boldsymbol{\mu}^\omega \mathcal{F}_i$ contains the $i$-branching trees of finite height. Let $t_i$ be a $\omega$-branching tree of height $i$ for all $i < \omega$. Consider the tree $t(n) = t_n$, whose root has infinitely many branches, containing the $t_i$ in order. The first 0 branches are good and of height $< 0$ (empty quantification), the first 1 branches are good and of height $< 1$, the first 2 branches are good and of height $< 2$ etc. This implies that

$$t \in \inf_{i<\omega} \sup_{j<\omega} \boldsymbol{\mu}^j \mathcal{F}_i = \limsup_{i\to\omega} \mu^\omega \mathcal{F}_i. \tag{5.7}$$

But it is not true that $t$ is an infinitely branching tree of finite height, i. e.,

$$t \notin \sup_{j<\omega} \boldsymbol{\mu}^j \mathcal{F}_\omega = \boldsymbol{\mu}^\omega \mathcal{F}_\omega. \tag{5.8}$$

This invalidates the inequation 5.6 for ordinal $\beta = \omega$; inductive types do not generally preserve upper semi-continuity.

Could we nevertheless accept all inductive types as result types of recursive functions without jeopardizing termination? After all, our semantics might be defective, ruling out a class of functions for internal reasons? The answer is no. In the following, we show how to construct a looping term if the inductive type of Nat-hungry functions $\mathsf{Hungry}^\iota(\mathsf{Nat}^\iota) = \mu^\iota X.\, \mathsf{Nat}^\iota \to X$ is accepted as result of a recursive function.

**Constructing a looping term.** We assume that the natural numbers are constructed by the two closed normal forms zero and succ and that there is a elimination $\mathsf{case}_{\mathsf{Nat}}$. We postulate the following typings and reductions.

$$\begin{array}{lll} \mathsf{zero} & : & \forall \iota.\, \mathsf{Nat}^{\iota+1} \\ \mathsf{succ} & : & \forall \iota.\, \mathsf{Nat}^\iota \to \mathsf{Nat}^{\iota+1} \\ \\ \mathsf{case}_{\mathsf{Nat}} & : & \forall X \forall \iota.\, X \to (\mathsf{Nat}^\iota \to X) \to \mathsf{Nat}^{\iota+1} \to X \\ \mathsf{case}_{\mathsf{Nat}}\ z\ s\ \mathsf{zero} & \longrightarrow^+ & z \\ \mathsf{case}_{\mathsf{Nat}}\ z\ s\ (\mathsf{succ}\, n) & \longrightarrow^+ & s\, n \end{array}$$

Since zero and succ are closed and normal, they must be weak-head values, hence, trigger unfolding of recursive functions. We can define a predecessor

function on positive numbers as follows:

$$\begin{array}{lcl} \mathit{err} & : & \forall X.X \\ \mathsf{pred} & : & \forall \imath.\, \mathsf{Nat}^{\imath+1} \to \mathsf{Nat}^{\imath} \\ \mathsf{pred} & := & \mathsf{case}_{\mathsf{Nat}}\ \mathit{err}\ \mathsf{id} \end{array}$$

Herein, we had to assume a variable *err* which inhabits every type. Note that there cannot be a *closed* term pred of type $\forall \imath.\, \mathsf{Nat}^{\imath+1} \to \mathsf{Nat}^{\imath}$, since then pred zero would be a closed normal form in the type $\forall \imath.\, \mathsf{Nat}^{\imath}$ whose semantics $\mathcal{N}at^0$ contains only neutral, i. e., open normal forms. The assumption of variable *err* is unproblematic, since we are refuting strong normalization in this example, and strong normalization treats also reduction of open terms. It is clear that $\mathsf{pred}\ (\mathsf{succ}\ t) \longrightarrow^+ t$, hence by induction on $n$, also

$$\mathsf{pred}^n\ (\mathsf{succ}^n\ t) \longrightarrow^+ t$$

for any term $t$.

Next, we define a function which transforms a $\mathsf{Nat}^{\jmath}$-hungry function into a $\mathsf{Nat}^{\jmath+1}$-hungry function. In the visualization of $A$-hungry functions as $A$-branching trees, this means that we insert a left-most branch at each node.

$$\begin{array}{l} \mathsf{s} \ : \ \forall \jmath \forall \imath.\, \mathsf{Hungry}^{\imath}(\mathsf{Nat}^{\jmath}) \to \mathsf{Hungry}^{\imath}(\mathsf{Nat}^{\jmath+1}) \\ \mathsf{s} \ := \ \mathsf{fix}_0^{\mu}\, \lambda s \lambda h.\ s \circ h \circ \mathsf{pred} \\ \mathsf{s}^n\ v \longrightarrow^+ \mathsf{s}^n \circ v \circ \mathsf{pred}^n \end{array}$$

At this, point we can construct a hungry function h recursively, *if we disregard that* $\mathsf{Hungry}^{\imath}(\mathsf{Nat}^{\imath})$ *does not denote a upper semi-continuous function*. Since the type of hungry functions does not contain values semantically, this is clearly pathological already. Later we will exploit h to construct a non-normalizing term.

$$\begin{array}{l} \mathsf{h} \ : \ \forall \imath.\, \mathsf{Nat}^{\imath} \to \mathsf{Hungry}^{\imath}(\mathsf{Nat}^{\imath}) \\ \mathsf{h} \ := \ \mathsf{fix}_0^{\mu}\, \lambda h \lambda \_.\ \mathsf{s} \circ h \circ \mathsf{pred} \\ \mathsf{h}\ v \ \longrightarrow^+ \mathsf{s} \circ \mathsf{h} \circ \mathsf{pred} \end{array}$$

To implement a diverging term, we need the inverse p of s. Intuitively, it cuts the left-most branch of each node.

$$\begin{array}{l} \mathsf{p} \ : \ \forall \jmath \forall \imath.\, \mathsf{Hungry}^{\imath}(\mathsf{Nat}^{\jmath+1}) \to \mathsf{Hungry}^{\imath}(\mathsf{Nat}^{\jmath}) \\ \mathsf{p} \ := \ \mathsf{fix}_0^{\mu}\, \lambda p \lambda h.\ p \circ h \circ \mathsf{succ} \\ \mathsf{p}^n\ v \longrightarrow^+ \mathsf{p}^n \circ v \circ \mathsf{succ}^n \end{array}$$

Finally, we define a "traversal" function tr which diverges on $\mathsf{h}\ v$.

$$\begin{array}{l} \mathsf{tr} \ : \ \mathsf{Hungry}^{\infty}(\mathsf{Nat}^{\infty}) \to \mathbf{0} \\ \mathsf{tr} \ := \ \mathsf{fix}_0^{\mu}\, \lambda tr \lambda h.\ tr\ ((p \circ h \circ \mathsf{succ})\ \mathsf{zero}) \\ \mathsf{tr}\ v \longrightarrow^+ \mathsf{tr}\ ((\mathsf{p} \circ v \circ \mathsf{succ})\ \mathsf{zero}) \end{array}$$

Now, we construct the following reduction sequence for any $n \in \mathbb{N}$.

$$\begin{array}{lll}
\mathsf{tr}\ (\mathsf{p}^n\ (\mathsf{s}^n\ (\mathsf{h}\ \mathsf{zero}))) & \longrightarrow^+ & \mathsf{tr}\ (\mathsf{p}^n\ (\mathsf{s}^n\ (\mathsf{s} \circ \mathsf{h} \circ \mathsf{pred}))) \\
& \longrightarrow^* & \mathsf{tr}\ (\mathsf{p}^n\ (\mathsf{s}^{n+1} \circ \mathsf{h} \circ \mathsf{pred}^{n+1})) \\
& \longrightarrow^* & \mathsf{tr}\ (\mathsf{p}^n \circ \mathsf{s}^{n+1} \circ \mathsf{h} \circ \mathsf{pred}^{n+1} \circ \mathsf{succ}^n) \\
& \longrightarrow^+ & \mathsf{tr}\ ((\mathsf{p}^{n+1} \circ \mathsf{s}^{n+1} \circ \mathsf{h} \circ \mathsf{pred}^{n+1} \circ \mathsf{succ}^{n+1})\ \mathsf{zero}) \\
& \longrightarrow^+ & \mathsf{tr}\ (\mathsf{p}^{n+1}\ (\mathsf{s}^{n+1}\ (\mathsf{h}\ \mathsf{zero})))
\end{array}$$

It is clear that $\mathsf{tr}\ (\mathsf{h}\ \mathsf{zero})$ diverges.

### 5.3.5 Product and Sum Types

Product and sum type are both upper and lower semi-continuous. This has been shown by Pareto [Par00] for a denotational semantics, and by myself [Abe03] for a SN semantics for a type system with iso-(co)inductive types. However, I do not know whether this can be shown for the impredicative encodings of sum and product from Example 3.3. In the following, we assume we have the continuity property for sum and product. This might require to add them as primitive notions to $\mathsf{F}^{\widehat{}}_{\omega}$.

**Lemma 5.26** *Let $\mathcal{A}, \mathcal{B} \in \mathsf{O} \to \mathcal{P}(\mathsf{Tm})$. Then for all limit ordinals $\lambda \in \mathsf{O}$,*

$$\begin{array}{lll}
(a) & \limsup_{\alpha\to\lambda}(\mathcal{A}(\alpha) \boxed{\times} \mathcal{B}(\alpha)) & \subseteq\ (\limsup_\lambda \mathcal{A}) \boxed{\times} (\limsup_\lambda \mathcal{B}), \\
(b) & \limsup_{\alpha\to\lambda}(\mathcal{A}(\alpha) \boxed{+} \mathcal{B}(\alpha)) & \subseteq\ (\limsup_\lambda \mathcal{A}) \boxed{+} (\limsup_\lambda \mathcal{B}), \\
(c) & (\liminf_\lambda \mathcal{A}) \boxed{\times} (\liminf_\lambda \mathcal{B}) & \subseteq\ \liminf_{\alpha\to\lambda}(\mathcal{A}(\alpha) \boxed{\times} \mathcal{B}(\alpha)), \textit{ and} \\
(d) & (\liminf_\lambda \mathcal{A}) \boxed{+} (\liminf_\lambda \mathcal{B}) & \subseteq\ \liminf_{\alpha\to\lambda}(\mathcal{A}(\alpha) \boxed{+} \mathcal{B}(\alpha)).
\end{array}$$

**Corollary 5.27 (Product and sum preserve semi-continuity)** *Let $q \in \{\oplus, \ominus\}$ and $\mathcal{A}, \mathcal{B} \in \mathsf{O} \xrightarrow{q} \mathcal{P}(\mathsf{Tm})$. Then $(\alpha \mapsto \mathcal{A}(\alpha) \boxed{+} \mathcal{B}(\alpha)), (\alpha \mapsto \mathcal{A}(\alpha) \boxed{\times} \mathcal{B}(\alpha)) \in \mathsf{O} \xrightarrow{q} \mathcal{P}(\mathsf{Tm})$.*

## 5.4 A Kinding System for Semi-Continuity

In this section we define a calculus to derive semi-continuous constructors. This calculus will be part of the new criterion for admissible types for (co)recursion.

**Polarities for semi-continuous functions.** Similar to the polarities $+$, $-$, and $\circ$ for co-, contra-, and non-variant constructors we introduce two new polarities for semi-continuous constructors:

$$\begin{array}{rcll}
\mathsf{HPol} \ni q & ::= & \oplus & \text{upper semi-continuity} \\
& \mid & \ominus & \text{lower semi-continuity}
\end{array}$$

**Positive contexts.** We define contexts $\Pi \in \mathsf{Cxt}^+ \subseteq \mathsf{PCxt}$ in which each variable has positive polarity and each kind is pure:

$$\begin{array}{rcll} \mathsf{Cxt}^+ \ni \Pi & ::= & \diamond & \text{empty context} \\ & | & \Pi, X : +\kappa_* & \text{positive constructor variable} \end{array}$$

**Well-formed size expressions.** The judgement $\Delta \vdash a\ ord$ singles out well-formed $\lambda$-free constructors of kind ord. With this separate judgement, it is very easy to show that every well-formed $a$ is either an affine function in one size variable, or equal to $\top^{\mathsf{ord}}$.

$$\textsc{ord-}\infty\ \frac{}{\Delta \vdash \infty\ ord} \qquad \textsc{ord-var}\ \frac{(\iota : p\mathsf{ord}) \in \Delta \qquad p \leq +}{\Delta \vdash \iota\ ord} \qquad \textsc{ord-s}\ \frac{\Delta \vdash a\ ord}{\Delta \vdash \mathsf{s}\, a\ ord}$$

**Derivation system for kinds and semi-continuity.** Let $\Delta$ be a polarized context with $(\iota : p\mathsf{ord}) \in \Delta$ for some $p$. We define the judgement $\Delta; \vec{X} : +\vec{\kappa}_* \vdash^{\iota q} F : \kappa$ inductively by the following rules. Variables $\vec{X}$ can only appear *strictly positive* in $F$, and they are disjoint from the variables in $\Delta$. The intended semantics of the judgement depends on $q$: If $q = \oplus$ then $F$ is a well-kinded, upper semi-continuous constructor in ordinal variable $\iota$, and a lim sup can be pushed into its arguments $\vec{X}$. If $q = \ominus$ then $F$ is lower semi-continuous and a lim inf can be pulled out from its arguments $\vec{X}$.

The first three rules enable us to derive semi-continuity from an ordinary kinding judgement $\Delta \vdash F : \kappa$.

$$\textsc{cont-in}\ \frac{\Delta \vdash F : \kappa}{\Delta, \iota : p\mathsf{ord}; \Pi \vdash^{\iota q} F : \kappa}$$

$$\textsc{cont-co}\ \frac{\Delta, \iota : +\mathsf{ord} \vdash F : \kappa \qquad p \leq +}{\Delta, \iota : p\mathsf{ord}; \Pi \vdash^{\iota \oplus} F : \kappa}$$

$$\textsc{cont-contra}\ \frac{\Delta, \iota : -\mathsf{ord} \vdash F : \kappa \qquad p \leq -}{\Delta, \iota : p\mathsf{ord}; \Pi \vdash^{\iota \ominus} F : \kappa}$$

These rules are justified by the fact that each invariant function is continuous, each covariant function is upper semi-continuous and each contravariant function is lower semi-continuous. The next rules are for the pure $\lambda$-calculus part of the constructor grammar.

$$\textsc{cont-var}\ \frac{X : p\kappa \in \Delta, \Pi \qquad p \leq +}{\Delta; \Pi \vdash^{\iota q} X : \kappa}$$

$$\textsc{cont-abs}\ \frac{\Delta, X : p\kappa; \Pi \vdash^{\iota q} F : \kappa'}{\Delta; \Pi \vdash^{\iota q} \lambda X F : p\kappa \to \kappa'}\ X \neq \iota$$

$$\textsc{cont-app}\ \frac{\Delta, \iota : p'\mathsf{ord}; \Pi \vdash^{\iota q} F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta, \iota : p'\mathsf{ord}; \Pi \vdash^{\iota q} F\, G : \kappa'}$$

Note that in the application rule, we require the argument $G$ to be constant in $\iota$. The background is that the composition of $F$ and $G$ has trivially all the continuity and variance properties of $F$ if $G$ is constant. Of course, with this rule we could never derive, e. g., that $A \to B$ is upper semi-continuous in $\iota$ if $B$ is upper semi-continuous and $A$ is lower semi-continuous. Hence, we need specialized rules for the constants, which implement the observations from the last section.

$$\text{CONT-SUM}\ \frac{\Delta;\Pi \vdash^{\iota q} A, B : *}{\Delta;\Pi \vdash^{\iota q} A + B : *} \qquad \text{CONT-PROD}\ \frac{\Delta;\Pi \vdash^{\iota q} A, B : *}{\Delta;\Pi \vdash^{\iota q} A \times B : *}$$

$$\text{CONT-ARR}\ \frac{-\Delta;\diamond \vdash^{\iota\ominus} A : * \qquad \Delta;\Pi \vdash^{\iota\oplus} B : *}{\Delta;\Pi \vdash^{\iota\oplus} A \to B : *}$$

$$\text{CONT-}\forall\ \frac{\Delta;\Pi \vdash^{\iota\oplus} F : \circ\kappa \to *}{\Delta;\Pi \vdash^{\iota\oplus} \forall_\kappa F : *}$$

$$\text{CONT-MU}\ \frac{\Delta;\Pi, X : +\kappa_* \vdash^{\iota\ominus} F : \kappa_* \qquad \Delta \vdash^{\iota\ominus} a : \mathsf{ord}}{\Delta;\Pi \vdash^{\iota\ominus} \mu^a X F : \kappa_*}$$

$$\text{CONT-NU}\ \frac{\Delta;\Pi, X : +\kappa_* \vdash^{\iota\oplus} F : \kappa_* \qquad \Delta \vdash a\ ord}{\Delta;\Pi \vdash^{\iota\oplus} \nu^a X F : \kappa_*}$$

We abbreviate $\Delta;\diamond \vdash^{\iota q} F : \kappa$ by $\Delta \vdash^{\iota q} F : \kappa$.

**Example 5.28 (Type of minimum function)** Recall that $\mathsf{Nat}^\iota = \mu^\iota X.\, \mathbf{1} + X$. First, we derive that Nat is lower semi-continuous:

$$\dfrac{\dfrac{\dfrac{}{\iota : \circ\mathsf{ord}; X : +* \vdash^{\iota\ominus} X : *}\ \text{CONT-VAR}}{\iota : \circ\mathsf{ord}; X : +* \vdash^{\iota\ominus} \mathbf{1} + X : *}\ \text{CONT-SUM} \qquad \dfrac{}{\iota : \circ\mathsf{ord} \vdash^{\iota\ominus} \iota : \mathsf{ord}}\ \text{CONT-VAR}}{\iota : \circ\mathsf{ord}; \diamond \vdash^{\iota\ominus} \mathsf{Nat}^\iota : *}\ \text{CONT-MU}$$

Since $\mathsf{Nat} : \mathsf{ord} \xrightarrow{+} *$ is also upper semi-continuous, we can derive that the type of the minimum and maximum functions is upper semi-continuous. We omit $: *$ and empty positive contexts $\Pi$ in the derivation:

$$\dfrac{\iota : \circ\mathsf{ord} \vdash^{\iota\ominus} \mathsf{Nat}^\iota \qquad \dfrac{\iota : \circ\mathsf{ord} \vdash^{\iota\ominus} \mathsf{Nat}^\iota \qquad \dfrac{\iota : +\mathsf{ord} \vdash \mathsf{Nat}^\iota}{\iota : \circ\mathsf{ord} \vdash^{\iota\oplus} \mathsf{Nat}^\iota}\ \text{CONT-CO}}{\iota : \circ\mathsf{ord} \vdash^{\iota\oplus} \mathsf{Nat}^\iota \to \mathsf{Nat}^\iota}\ \text{CONT-ARR}}{\iota : \circ\mathsf{ord} \vdash^{\iota\oplus} \mathsf{Nat}^\iota \to \mathsf{Nat}^\iota \to \mathsf{Nat}^\iota}\ \text{CONT-ARR}$$

**Example 5.29 (Stream of natural numbers)** Let $\mathsf{Stream}^\iota\, A = \nu^\iota X.\, A \times X$, as de-

fined in Example 2.9. The type $\mathsf{Stream}^\iota\ \mathsf{Nat}^\iota$ is upper semi-continuous.

$$\dfrac{\dfrac{\dfrac{\iota : +\mathsf{ord} \vdash \mathsf{Nat}^\iota}{\iota : \circ\mathsf{ord}; X : +* \vdash^{\iota\oplus} \mathsf{Nat}^\iota}\,\text{CONT-CO} \qquad \dfrac{}{\iota : \circ\mathsf{ord}; X : +* \vdash^{\iota\oplus} X}\,\text{CONT-VAR}}{\iota : \circ\mathsf{ord}; X : +* \vdash^{\iota\oplus} \mathsf{Nat}^\iota \times X}\,\text{CONT-PROD}}{\iota : \circ\mathsf{ord} \vdash^{\iota\oplus} \mathsf{Stream}^\iota\ \mathsf{Nat}^\iota}\,\text{CONT-NU}$$

**Example 5.30 (Finitely branching trees)** Node-labeled forests of height $< \iota$ with less than $\jmath$ branches can be implemented as $\mu^\iota X.\, \mathsf{List}^\jmath\,(A \times X)$. If we let the two indices coincide, we obtain $\mu^\iota X.\, \mathsf{List}^\iota\,(A \times X)$, which is upper semi-continuous (since isotone), but also lower semi-continuous. This is an example with two strictly positive variables:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\ldots X, Y : +* \vdash^{\iota\ominus} X}\,\text{CONT-VAR}}{\ldots X, Y : +* \vdash^{\iota\ominus} (A \times X)}\,\text{CONT-PROD} \qquad \dfrac{}{\ldots X, Y : +* \vdash^{\iota\ominus} Y}\,\text{CONT-VAR}}{\ldots X, Y : +* \vdash^{\iota\ominus} (A \times X) \times Y}\,\text{CONT-PROD}}{\iota : +\mathsf{ord}; X, Y : +* \vdash^{\iota\ominus} \mathbf{1} + (A \times X) \times Y}\,\text{CONT-SUM}}{\iota : +\mathsf{ord}; X : +* \vdash^{\iota\ominus} \mathsf{List}^\iota\,(A \times X)}\,\text{CONT-MU}}{\iota : +\mathsf{ord}; \diamond \vdash^{\iota\ominus} \mu^\iota X.\, \mathsf{List}^\iota\,(A \times X)}\,\text{CONT-MU}$$

**Lemma 5.31 (Soundness of judgement for size expressions)** *If $\mathcal{D} :: \Delta \vdash a\ \mathsf{ord}$ then $\Delta \vdash a : \mathsf{ord}$.*

*Proof.* By induction on $\mathcal{D}$. □

**Theorem 5.32 (Erasure of continuity)** *If $\mathcal{D} :: \Delta; \Pi \vdash^{\iota q} F : \kappa$ then $\Delta, \Pi \vdash F : \kappa$.*

*Proof.* By induction on $\mathcal{D}$. □

A consequence of the erasure theorem is that all derivations of semi-continuous constructors produce only well-kinded constructors.

## 5.5 Semantical Soundness of Continuity Derivations

In the following, we will show that all types that are kindable by the calculus of the last section indeed have the postulated semi-continuity properties. We will make use of all the semantical propositions proven in Section 5.3.

**Lemma 5.33** *If $\mathcal{D} :: \Delta \vdash a\ \mathsf{ord}$, then for all $\theta \in [\![\Delta]\!]$ and ordinal variables $\iota$,*

$$\phi := (\alpha \mapsto [\![a]\!]_{\theta[\iota \mapsto \alpha]}) \in [\![\mathsf{ord}]\!] \to [\![\mathsf{ord}]\!]$$

*is either the constant function $\phi(\alpha) = \top^{\mathsf{ord}}$ or a function of the shape $\phi(\alpha) = \min\{\alpha + n, \top^{\mathsf{ord}}\}$ for some natural number $n$.*

*Proof.* By induction on $\mathcal{D}$. □

**Theorem 5.34 (Semantical soundness)** *Let* $\theta \in [\![\Delta, \Pi]\!]$, $(X : +\kappa') \in \Pi$, $\mathcal{G} \in [\![\mathsf{ord}]\!] \to [\![\kappa']\!]$, *and* $\lambda \in [\![\mathsf{ord}]\!]$ *a limit ordinal.*

1. *If* $\mathcal{D} :: \Delta; \Pi \vdash^{\iota \oplus} F : \kappa$ *then*
   (a) $\limsup_{\alpha \to \lambda} [\![F]\!]_{\theta[\iota \mapsto \alpha]} \sqsubseteq^{\kappa} [\![F]\!]_{\theta[\iota \mapsto \lambda]}$, *and*
   (b) $\limsup_{\alpha \to \lambda} [\![F]\!]_{\theta[X \mapsto \mathcal{G}(\alpha)]} \sqsubseteq^{\kappa} [\![F]\!]_{\theta[X \mapsto \limsup_{\lambda} \mathcal{G}]}$.
2. *If* $\mathcal{D} :: \Delta; \Pi \vdash^{\iota \ominus} F : \kappa$ *then*
   (a) $[\![F]\!]_{\theta[\iota \mapsto \lambda]} \sqsubseteq^{\kappa} \liminf_{\alpha \to \lambda} [\![F]\!]_{\theta[\iota \mapsto \alpha]}$, *and*
   (b) $[\![F]\!]_{\theta[X \mapsto \liminf_{\lambda} \mathcal{G}]} \sqsubseteq^{\kappa} \liminf_{\alpha \to \lambda} [\![F]\!]_{\theta[X \mapsto \mathcal{G}(\alpha)]}$.

*Proof.* By induction on $\mathcal{D}$.

*Case* CONT-IN. A constant function is trivially continuous.

*Case* CONT-CO. Isotone functions are upper semi-continuous (Lemma 5.4).

*Case* CONT-CONTRA. Analogously.

*Case* CONT-VAR. By assumption.

*Case* CONT-ABS.

$$\frac{\Delta, X : p\kappa; \Pi \vdash^{\iota q} F : \kappa'}{\Delta; \Pi \vdash^{\iota q} \lambda X F : p\kappa \to \kappa'} \; X \neq \iota$$

For subcase $q = \oplus$, we need to show that (a),

$$\limsup_{\alpha \to \lambda} ([\![\lambda X F]\!]_{\theta[\iota \mapsto \alpha]}) \sqsubseteq^{p\kappa \to \kappa'} [\![\lambda X F]\!]_{\theta[\iota \mapsto \lambda]}.$$

It is sufficient that for all $\mathcal{G} \in [\![\kappa]\!]$,

$$\begin{array}{lcl} (\limsup_{\alpha \to \lambda} [\![\lambda X F]\!]_{\theta[\iota \mapsto \alpha]})\, \mathcal{G} & = & \limsup_{\alpha \to \lambda} ([\![\lambda X F]\!]_{\theta[\iota \mapsto \alpha]}\, \mathcal{G}) \\ & = & \limsup_{\alpha \to \lambda} ([\![F]\!]_{\theta[\iota \mapsto \alpha][X \mapsto \mathcal{G}]}) \\ & = & \limsup_{\alpha \to \lambda} ([\![F]\!]_{\theta[X \mapsto \mathcal{G}][\iota \mapsto \alpha]}) \\ & \sqsubseteq^{\kappa'} & [\![F]\!]_{\theta[X \mapsto \mathcal{G}][\iota \mapsto \lambda]} \\ & = & [\![\lambda X F]\!]_{\theta[\iota \mapsto \lambda]}\, \mathcal{G}, \end{array}$$

which follows by induction hypothesis. Goal (b) follows similarly, and case $q = \ominus$ analogously. We have used that infimum and supremum are defined pointwise on lattices $[\![\kappa]\!]$, if $\kappa$ is an ordinary polarized kind without continuity restrictions (Lemma 2.16).[4]

[4] On lattices with continuity, e. g., $\mathsf{O} \xrightarrow{\oplus} \mathfrak{L}$, the supremum might not be the pointwise one (Example 5.13!).

*Case* CONT-APP. Similarly, by pointwise infimum and supremum.

*Case* CONT-SUM and CONT-PROD. By Cor. 5.27.

*Case* CONT-ARR. By Cor. 5.9.

*Case* CONT-$\forall$. By Cor. 5.12.

*Case* CONT-MU.

$$\frac{\Delta; \Pi, X{:}{+}\kappa_* \vdash^{\iota\ominus} F : \kappa_* \qquad \Delta \vdash^{\iota\ominus} a : \mathsf{ord}}{\Delta; \Pi \vdash^{\iota\ominus} \mu^a XF : \kappa_*}$$

By induction hypothesis, $\phi := \alpha \mapsto [\![a]\!]_{\theta[\iota\mapsto\alpha]}$ is a lower semi-continuous function on ordinals. Let $\mathcal{F}_\alpha(\mathcal{H}) = [\![F]\!]_{\theta[X\mapsto\mathcal{H}][\iota\mapsto\alpha]}$. By induction hypothesis we have for all limits $\lambda \in [\![\mathsf{ord}]\!]$, $\mathcal{H} \in [\![\kappa_*]\!]$, $\beta \in [\![\mathsf{ord}]\!]$, $\mathcal{G} \in [\![\mathsf{ord}]\!] \to [\![\kappa_*]\!]$ that

(a) $[\![F]\!]_{\theta[X\mapsto\mathcal{H}][\iota\mapsto\lambda]} = \mathcal{F}_\lambda(\mathcal{H}) \sqsubseteq \liminf_{\alpha\to\lambda} \mathcal{F}_\alpha(\mathcal{H})$,
(b) $[\![F]\!]_{\theta[X\mapsto\liminf_\lambda \mathcal{G}][\iota\mapsto\beta]} = \mathcal{F}_\beta(\liminf_\lambda \mathcal{G}) \sqsubseteq \liminf_{\alpha\to\lambda} \mathcal{F}_\beta(\mathcal{G}(\alpha))$,

thus, $\mathcal{F}$ is both lower semi-continuous and lim inf-pullable. Goal (a),

$$\boldsymbol{\mu}^{\phi(\lambda)} \mathcal{F}_\lambda \sqsubseteq \liminf_{\alpha\to\lambda} \boldsymbol{\mu}^{\phi(\alpha)} \mathcal{F}_\alpha$$

follows by Cor 5.25.

To show (b) for some variable $(Y{:}{+}\kappa') \in \Pi$, let $\beta = [\![a]\!]_\theta$ and $\mathcal{F}(\mathcal{G})(\mathcal{H}) = [\![F]\!]_{\theta[Y\mapsto\mathcal{G}][X\mapsto\mathcal{H}]}$. Then $[\![\mu^a XF]\!]_{\theta[Y\mapsto\mathcal{G}]} = \boldsymbol{\mu}^\beta \mathcal{F}(\mathcal{G})$. By induction hypothesis (b), for all $\mathcal{G}' \in [\![\kappa']\!]$, $\mathcal{H} \in [\![\kappa_*]\!]$, $\mathcal{G} \in \mathsf{ord} \to [\![\kappa']\!]$, and $\lambda$ limit

(b.1) $\mathcal{F}(\mathcal{G}')(\liminf_\lambda \mathcal{H}) \sqsubseteq \liminf_{\beta\to\lambda} \mathcal{F}(\mathcal{G}')(\mathcal{H}(\beta))$ and
(b.2) $\mathcal{F}(\liminf_\lambda \mathcal{G}) \sqsubseteq \liminf_{\alpha\to\lambda} \mathcal{F}(\mathcal{G}(\alpha))$.

Since by (b.1) $\mathcal{T}_\alpha = \mathcal{F}(\mathcal{G}(\alpha))$ is lim inf-pullable, we can apply Lemma 5.23, we have for all $\mathcal{G}$,

$$\boldsymbol{\mu}^\beta \liminf_{\alpha\to\lambda} \mathcal{F}(\mathcal{G}(\alpha)) \sqsubseteq \liminf_{\alpha\to\lambda} \boldsymbol{\mu}^\beta \mathcal{F}(\mathcal{G}(\alpha)).$$

Combining this with (b.2), we can prove our goal

$$\boldsymbol{\mu}^\beta \mathcal{F}(\liminf_\lambda \mathcal{G}) \sqsubseteq \liminf_{\alpha\to\lambda} \boldsymbol{\mu}^\beta \mathcal{F}(\mathcal{G}(\alpha)).$$

*Case* CONT-NU.

$$\frac{\Delta; \Pi, X{:}{+}\kappa_* \vdash^{\iota\oplus} F : \kappa_* \qquad \Delta \vdash a\ ord}{\Delta; \Pi \vdash^{\iota\oplus} \nu^a XF : \kappa_*}$$

Let $\mathcal{F}_\alpha(\mathcal{H}) = [\![F]\!]_{\theta[X\mapsto\mathcal{H}][\iota\mapsto\alpha]}$. By induction hypothesis we have for all limits $\lambda \in [\![\mathsf{ord}]\!]$, $\mathcal{H} \in [\![\kappa_*]\!]$, $\beta \in [\![\mathsf{ord}]\!]$, $\mathcal{G} \in [\![\mathsf{ord}]\!] \to [\![\kappa_*]\!]$ that

(a) $\limsup_{\alpha\to\lambda} \mathcal{F}_\alpha(\mathcal{H}) \sqsubseteq \mathcal{F}_\lambda(\mathcal{H}) = [\![F]\!]_{\theta[X\mapsto\mathcal{H}][\iota\mapsto\lambda]}$,
(b) $\limsup_{\alpha\to\lambda} \mathcal{F}_\beta(\mathcal{G}(\alpha)) \sqsubseteq \mathcal{F}_\beta(\limsup_\lambda \mathcal{G}) = [\![F]\!]_{\theta[X\mapsto\limsup_\lambda \mathcal{G}][\iota\mapsto\beta]}$,

thus, $\mathcal{F}$ is both upper semi-continuous and lim sup-pushable. First, we observe that the greatest fixed point of $\mathcal{F}_\beta$ is already reached at iterate $\omega$, because

$$\begin{array}{rcl} \boldsymbol{\nu}^\omega \mathcal{F}_\beta & = & \limsup_{\alpha\to\omega} \mathcal{F}_\beta^\alpha \\ & = & \limsup_{\alpha\to\omega} \mathcal{F}_\beta(\mathcal{F}_\beta^\alpha) \\ & \sqsubseteq & \mathcal{F}_\beta(\limsup_{\alpha\to\omega} \mathcal{F}_\beta^\alpha) \\ & = & \mathcal{F}_\beta(\boldsymbol{\nu}^\omega \mathcal{F}_\beta) \\ & = & \boldsymbol{\nu}^{\omega+1} \mathcal{F}_\beta. \end{array}$$

(It is well-known that strictly positive coinductive types close at $\omega$.) By Cor. 5.20, we have

$$\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^{\phi(\alpha)} \mathcal{F}_\alpha \sqsubseteq \boldsymbol{\nu}^{\liminf_\lambda \phi} \mathcal{F}_\lambda.$$

Now, by Lemma 5.33, $\phi := \alpha \mapsto [\![a]\!]_{\theta[\iota\mapsto\alpha]}$ is either affine or constantly $\top^{\mathsf{ord}}$. In both cases, $\phi(\lambda) \geq \omega$ and $\liminf_\lambda \phi \geq \omega$, hence, goal (a),

$$\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^{\phi(\alpha)} \mathcal{F}_\alpha \sqsubseteq \boldsymbol{\nu}^{\phi(\lambda)} \mathcal{F}_\lambda$$

follows since the fixed-point is reached at $\omega$.

To show (b) for some variable $(Y : +\kappa') \in \Pi$, let $\beta = [\![a]\!]_\theta$ and $\mathcal{F}(\mathcal{G})(\mathcal{H}) = [\![F]\!]_{\theta[Y\mapsto\mathcal{G}][X\mapsto\mathcal{H}]}$. Then $[\![\nu^a XF]\!]_{\theta[Y\mapsto\mathcal{G}]} = \boldsymbol{\nu}^\beta \mathcal{F}(\mathcal{G})$. By induction hypothesis (b), for all $\mathcal{G}' \in [\![\kappa']\!]$, $\mathcal{H} \in [\![\kappa_*]\!]$, $\mathcal{G} \in \mathsf{ord} \to [\![\kappa']\!]$, and $\lambda$ limit

$$\begin{array}{ll} \text{(b.1)} & \limsup_{\beta\to\lambda} \mathcal{F}(\mathcal{G}')(\mathcal{H}(\beta)) \sqsubseteq \mathcal{F}(\mathcal{G}')(\limsup_\lambda \mathcal{H}) \text{ and} \\ \text{(b.2)} & \limsup_{\alpha\to\lambda} \mathcal{F}(\mathcal{G}(\alpha)) \sqsubseteq \mathcal{F}(\limsup_\lambda \mathcal{G}). \end{array}$$

Since by (b.1) $\mathcal{T}_\alpha = \mathcal{F}(\mathcal{G}(\alpha))$ is lim sup-pushable, we can apply Lemma 5.18, yielding for all $\mathcal{G}$,

$$\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^\beta \mathcal{F}(\mathcal{G}(\alpha)) \sqsubseteq \boldsymbol{\nu}^\beta \limsup_{\alpha\to\lambda} \mathcal{F}(\mathcal{G}(\alpha)).$$

Combining this with (b.2), we can prove our goal

$$\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^\beta \mathcal{F}(\mathcal{G}(\alpha)) \sqsubseteq \boldsymbol{\nu}^\beta \mathcal{F}(\limsup_\lambda \mathcal{G}).$$

Uff! □

## 5.6 Type-Based Termination with Continuous Types

We replace the conditions for admissible (co)recursion types from Section 3.1.2 by the following, which use the new judgments for semi-continuity:

$$\begin{array}{rl} \Gamma \vdash A\ \mathsf{fix}^{\mu}_{n}\text{-}\mathsf{adm} & \text{iff } \Gamma, \iota{:}\mathsf{ord} \vdash A\,\iota = (\vec{G}, (\mu^{\iota} F) \circ \vec{H} \Rightarrow G) : * \\ & \text{and } \Gamma, \iota{:}\mathsf{ord} \vdash^{\iota\oplus} \vec{G}, (\mu^{\iota} F) \circ \vec{H} \Rightarrow G : * \\ & \text{for some } F, G, \vec{G}, \vec{H} \text{ with } |\vec{G}| = n \end{array}$$

$$\begin{array}{rl} \Gamma \vdash A\ \mathsf{fix}^{\nu}_{n}\text{-}\mathsf{adm} & \text{iff } \Gamma, \iota{:}\mathsf{ord} \vdash A\,\iota = (\vec{G} \Rightarrow (\nu^{\iota} F) \circ \vec{H}) : * \\ & \text{and } \Gamma, \iota{:}\mathsf{ord} \vdash^{\iota\oplus} \vec{G} \Rightarrow (\nu^{\iota} F) \circ \vec{H} : * \\ & \text{for some } F, \vec{G}, \vec{H} \text{ with } |\vec{G}| = n \end{array}$$

**Example 5.35 (Stream of natural numbers)** We can now assign a precise type to the stream of natural numbers 0,1,....

$$\begin{array}{lcl} \mathsf{mapStream} & : & \forall A \forall B.\, (A \to B) \to \forall \iota.\, \mathsf{Stream}^{\iota} A \to \mathsf{Stream}^{\iota} B \\ \mathsf{mapStream} & := & \lambda f.\ \mathsf{fix}^{\nu}_{0}\, \lambda \mathit{mapStream} \lambda s.\, \langle f\,(\mathsf{fst}\, s),\ \mathit{mapStream}\,(\mathsf{snd}\, s)\rangle \end{array}$$

$$\begin{array}{lcl} \mathsf{nats} & : & \forall \iota.\, \mathsf{Stream}^{\iota}\, \mathsf{Nat}^{\iota} \\ \mathsf{nats} & := & \mathsf{fix}^{\nu}_{0}\, \lambda \mathit{nats}.\, \langle \mathsf{zero},\ \mathsf{mapStream}\ \mathsf{succ}\ \mathit{nats}\rangle \end{array}$$

Here are the types of some subexpressions of nats:

$$\begin{array}{lcl} \mathit{nats} & : & \mathsf{Stream}^{\iota}\, \mathsf{Nat}^{\iota} \\ \mathsf{succ} & : & \mathsf{Nat}^{\iota} \to \mathsf{Nat}^{\iota+1} \\ \mathsf{mapStream}\ \mathsf{succ}\ \mathit{nats} & : & \mathsf{Stream}^{\iota}\, \mathsf{Nat}^{\iota+1} \\ \mathsf{zero} & : & \mathsf{Nat}^{\iota+1} \\ \langle \mathsf{zero},\ \mathsf{mapStream}\ \mathsf{succ}\ \mathit{nats}\rangle & : & \mathsf{Stream}^{\iota+1} \mathsf{Nat}^{\iota+1} \end{array}$$

Since $\mathsf{Stream}^{\iota}\, \mathsf{Nat}^{\iota}$ is upper semi-continuous (Example 5.29) it is admissible for corecursion. Note that it was not admissible according to the previous criterion (Section 3.1.2).

More examples for definitions that require the new criterion will be given in sections 6.1 (breath-first traversal of finitely branching trees), 6.4, (equality for monadic lists), 6.5 (a generic merge function), and 6.7 (transitivity of simple subtyping derivations).

## 5.7 Related and Future Work

The author [Abe03] has investigated admissible recursion types on the basis of slightly different concept. Roughly, lower semi-continuity of $f$ is replaced by the weaker condition (1) $f(\lambda) \sqsubseteq \sup_{\lambda} f$ and upper semi-continuity by the weaker condition (2) $\liminf_{\lambda} f \sqsubseteq f(\lambda)$. He shows the following main lemma: If $\mathcal{A}(\lambda) \sqsubseteq \sup_{\lambda} \mathcal{A}$ and $\liminf_{\lambda} \mathcal{B} \sqsubseteq \mathcal{B}(\lambda)$, then $\liminf_{\alpha\to\lambda}(\mathcal{A}(\alpha) \boxrightarrow \mathcal{B}(\alpha)) \sqsubseteq$

$\mathcal{A}(\lambda) \boxrightarrow \mathcal{B}(\lambda)$. One could think that, since the conditions are weaker, more types are admissible. But the opposite is the case. Since (1) distributes only over a product $A(\alpha) \boxtimes \mathcal{B}(\alpha)$ if $\mathcal{A}$ and $\mathcal{B}$ are monotonic, products that involve an antitonic component are rejected by the syntactic calculus "cocont" which incarnates (1). However, in this thesis such types, for instance, $\mathsf{List}^\iota(\mathsf{Nat}^\iota) \times (\mathsf{Nat}^\iota \to \mathsf{Bool})$, are accepted as lower semi-continuous.

Pareto [Par00] has investigated properties of types similar to semi-continuity. His setting differs considerably from ours: He considers only fixed-points of type constructors that are reached at iteration $\omega$. These are inductive types without embedded function spaces, and strictly positive coinductive types. Size variables range over natural numbers and, consequently, $\forall\iota.\, A(\iota)$ quantifies only over natural numbers. However, for certain $A$ the variable $\iota$ may be instantiated by $\omega$: Pareto describes a class of valid types which he calls $\omega$-*undershooting*. These types have the property $\liminf_{\iota\to\omega} A(\iota) \subseteq A(\omega)$. Function types are $\omega$-undershooting if their codomain is so and if their domain is $\omega$-*overshooting*, $A(\omega) \subseteq \liminf_{\iota\to\omega} A(\iota)$. We have generalized the term $\omega$-overshooting to arbitrary limit ordinals and called it lower semi-continuous. Also, we have demonstrated that our notion of upper semi-continuity, which uses lim sup, is as useful as Pareto's notion of $\omega$-undershooting, which uses lim inf.

We have shown that, in contrast to Pareto, we rightfully refuse inductive types as preservers of upper semi-continuity, since we allow embedded function spaces in inductive types.

We have given a *syntax-directed* derivation system for semi-continuous types. This was possible since we also created the new concepts of lim sup-*pushable* and lim inf-*pullable* type constructors on the semantical side. Our calculus is still weak on the higher-order features, and this is an area of future research. Ideally, we could extend our annotation of function kinds to express how a lim sup could be pushed into or a lim inf could be pulled out of a constructor. Then we would not need special rules for $\to$, $\forall$, etc, but the kinding of these constants would take care of continuity properties—as it does currently with variance properties (polarities $+$, $-$, and $\circ$).

# Chapter 6

# Examples

In this chapter, we present several examples, some of which exhibit an interesting recursion scheme (breadth-first traversal in Section 6.1, normalization procedures in sections 6.2 and 6.3, generic programming in Section 6.5) and others that are interesting for theoretical reasons (impredicative datatypes in Section 6.6, inductive proofs in Section 6.7). Pareto provided many functions on natural numbers, lists, and streams in his thesis [Par00]. Most of these are also typable in $\mathsf{F}_{\widehat{\omega}}$, except those that use addition of size variables, what we do not support.

In the examples we use syntactic sugar like pattern matching, which is a conservative extension of $\mathsf{F}_{\widehat{\omega}}$. It can be reduced to the primitives as described in section 2.4 of the article [AMU05].

## 6.1 Breadth-First Tree Traversal

Breadth-first traversal of a tree is not a structurally recursive program. In the following, we will refine the standard definition such that it is typable in $\mathsf{F}_{\widehat{\omega}}$.

Recall that in Haskell, `[a]` is the type of lists with constructors `nil :: [a]` and `(:) :: a -> [a] -> [a]` and a function `(++) :: [a] -> [a] -> [a]`, which concatenates two lists. Since in Haskell least and greatest solutions of type equations coincide, there is no distinction between inductive and coinductive types, and `[a]` contains finite and infinite lists. For this example, we mean only finite lists. *Rose trees* are finitely-branching labeled trees with one constructor `Rose :: a -> [Rose a] -> Rose a`. A *rose forest* is a list of rose trees. Breadth-first flattening of a rose forest can be implemented as follows:

```
data Rose a = Rose a [Rose a]

bf0 :: [Rose a] -> [a]
bf0 [] = []
bf0 (Rose a rs : rs') = a : bf0 (rs' ++ rs)
```

Other breadth-first operations like breadth-first left or right folding can be expressed similarly. The definition of `bf0` uses general recursion, since the recursive argument `rs' ++ rs` is not a subterm of `Rose a rs : rs'`. Termination could be shown by a decreasing measure: the number of `Rose` constructors in the forest. We show termination by massaging the definition until it fits our type system.

Looking a bit closer, we see that the traversal of the forest can be separated into phases: In the first phase, we process the roots of all rose trees. In the next phase, we do the same for the subtrees which emerged by cutting the roots in the first phase. Etc.

```
step :: [Rose a] -> ([a], [Rose a])
step [] = ([], [])
step (Rose a rs' : rs) =
    let (as, rs'') = step rs
        in  (a : as, rs' ++ rs'')
```

The `step` function makes such a run over the forest, returning a list of roots and the subforest. It is a simple iterative function and returns rose trees which are strictly less tall than the input trees. This is recognized by its typing in $\mathsf{F}_{\widehat{\omega}}$:

$$
\begin{array}{lcl}
\mathsf{Rose} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Rose} & := & \lambda\iota\lambda A.\ \mu^{\iota}_{*}\lambda X.\ A \times \mathsf{List}^{\infty} X
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{step} & : & \forall A \forall \jmath \forall \iota.\ \mathsf{List}^{\jmath}(\mathsf{Rose}^{\iota+1}\ A) \to \mathsf{List}^{\jmath}\ A \times \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota}\ A) \\
\mathsf{step} & := & \mathsf{fix}^{\mu}_{0}\ \lambda \mathit{step} \lambda l.\ \mathsf{match}\ l\ \mathsf{with} \\
 & & \quad \mathsf{nil} \mapsto \langle \mathsf{nil}, \mathsf{nil} \rangle \\
 & & \quad \mathsf{cons}\ \langle a, rs' \rangle\ rs \mapsto \mathsf{match}\ \mathit{step}\ rs\ \mathsf{with} \\
 & & \qquad \langle as,\ rs'' \rangle \mapsto \langle \mathsf{cons}\ a\ as,\ \mathsf{append}\ rs'\ rs'' \rangle
\end{array}
$$

The type $B(\jmath) = \mathsf{List}^{\jmath}(\mathsf{Rose}^{\iota+1}\ A) \to \mathsf{List}^{\jmath}\ A \times \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota}\ A)$ of the recursion is admissible, the second size variable $\iota$ does not interfere with admissibility.

Iterating `step`, we can perform a complete breadth-first traversal.

```
bf1 :: [Rose a] -> [a]
bf1 rs =
    let (as, rs') = step rs
        in as ++ bf1 rs'
```

However, its type $C(\iota) = \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota}\ A) \to \mathsf{List}^{\infty} A$ is *not* admissible for recursion. And indeed, `bf1` is not terminating on the empty forest, which means that it eventually loops on every input. If we introduce a special clause for the empty forest, we regain termination.

```
bf2 :: [Rose a] -> [a]
bf2 [] = []
bf2 rs =
    let (as, rs') = step rs
        in as ++ bf2 rs'
```

But since we have not improved on the type of `bf2`, our type system will still reject it. We need to make explicit that we deal with non-empty forests in the recursion.

```
bf3 :: [Rose a] -> [a]
bf3 []     = []
bf3 (r:rs) = bf3' r rs

bf3' :: Rose a -> [Rose a] -> [a]
bf3' r rs =
    case step (r:rs) of
        (as, [])       -> as
        (as, r' : rs') -> as ++ bf3' r' rs'
```

The recursion in function `bf3'` can be given the type

$$C(\iota) = \mathsf{Rose}^{\iota}\, A \to \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota}\, A) \to \mathsf{List}^{\infty}\, A$$

which is admissible in the extension of $\mathsf{F}_{\widehat{\omega}}$ introduced in Chapter 5. For this, we have to show

$$A : \circ *, \iota : \circ \mathsf{ord}; \diamond \vdash^{\iota \oplus} \mathsf{Rose}^{\iota}\, A \to \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota}\, A) \to \mathsf{List}^{\infty}\, A : *.$$

The non-obvious part of this goal is to show that $\mathsf{List}^{\infty}(\mathsf{Rose}^{\iota}\, A)$ is lower semi-continuous. In the following derivation, let $\Gamma := A : \circ *, \iota : \circ \mathsf{ord}$. We further save space by dropping inessential parts of the derivation and its judgements.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; X, Y, Z \vdash^{\iota \ominus} 1 + Y \times Z}}{\Gamma; X, Y \vdash^{\iota \ominus} \mathsf{List}^{\infty} Y}}{\Gamma; X, Y \vdash^{\iota \ominus} A \times \mathsf{List}^{\infty} Y} \qquad \dfrac{}{\Gamma; X \vdash^{\iota \ominus} \iota : \mathsf{ord}}}{\Gamma; X \vdash^{\iota \ominus} \mathsf{Rose}^{\iota}\, A} \qquad \dfrac{}{\Gamma; X \vdash^{\iota \ominus} X}}{\dfrac{\Gamma; X \vdash^{\iota \ominus} 1 + \mathsf{Rose}^{\iota}\, A \times X}{\Gamma; \diamond \vdash^{\iota \ominus} \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota}\, A)}}$$

As a thumb rule, a type is lower semi-continuous, if its composed only of constant types, sum, product, and strictly positive inductive types.

This is the encoding of `bf3'` in $\mathsf{F}_{\widehat{\omega}}$:

$$\begin{array}{lcl} \mathsf{bf}_3' & : & \forall A \forall \iota.\, \mathsf{Rose}^{\iota}\, A \to \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota}\, A) \to \mathsf{List}^{\infty} A \\ \mathsf{bf}_3' & := & \mathsf{fix}_0^{\mu}\, \lambda bf \lambda r \lambda rs.\, \mathsf{match}\ \mathsf{step}\ (\mathsf{cons}\, r\, rs)\ \mathsf{with} \\ & & \quad \langle as,\ \mathsf{nil} \rangle \mapsto as \\ & & \quad \langle as,\ \mathsf{cons}\, r'\, rs' \rangle \mapsto \mathsf{append}\, as\, (bf\, r'\, rs') \end{array}$$

We can assign the following types:

$$
\begin{array}{lcl}
bf & : & \mathsf{Rose}^{\iota}\, A \to \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota}\, A) \to \mathsf{List}^{\infty} A \\
r & : & \mathsf{Rose}^{\iota+1} A \\
rs & : & \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota+1} A) \\
\mathsf{cons}\, r\, rs & : & \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota+1} A) \\
\mathsf{step}\, (\mathsf{cons}\, r\, rs) & : & \mathsf{List}^{\infty} A \times \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota} A) \\
r' & : & \mathsf{Rose}^{\iota} A \\
rs' & : & \mathsf{List}^{\infty}(\mathsf{Rose}^{\iota} A) \\
bf\ r'\ rs' & : & \mathsf{List}^{\infty} A
\end{array}
$$

Looking at `bf3` we recognize that `step` is only called with non-empty forests. This enables the following optimization:

```
step4 :: Rose a -> [Rose a] -> ([a], [Rose a])
step4 (Rose a rs') []     = ([a], rs')
step4 (Rose a rs') (r:rs) =
    let (as, rs'') = step4 r rs
        in  (a : as, rs' ++ rs'')

bf4' :: Rose a -> [Rose a] -> [a]
bf4' r rs =
    case step4 r rs of
        (as, [])       -> as
        (as, r' : rs') -> as ++ bf4' r' rs'
```

We have successfully turned breadth-first traversal into a program which is accepted by our type system, by exhibiting a structure that was implicit in the original program. Well, the resulting program is much longer than the original, have we gained anything? Indeed, the refined program runs even faster than the original! For a literal translation of these programs into SML/NJ, Version 110.0.7, I have obtained the following running times on a SuSE Linux 9.1 system with an Intel® Pentium® III 1066MHz Mobile CPU and 256 MB system RAM.

| Depth | Size | Rep | bf0 | bf2 | bf4 |
|---|---|---|---|---|---|
| 1 | 2 | 1000000 | 0.112 | 0.128 | 0.107 |
| 2 | 5 | 318641 | 0.119 | 0.136 | 0.105 |
| 3 | 16 | 52006 | 0.092 | 0.081 | 0.075 |
| 4 | 65 | 5281 | 0.109 | 0.034 | 0.032 |
| 5 | 326 | 371 | 0.176 | 0.013 | 0.012 |
| 6 | 1957 | 20 | 0.380 | 0.006 | 0.005 |
| 7 | 13700 | 1 | 1.984 | 0.006 | 0.005 |

The first two columns display quantitative information about traversed rose trees and the third column the number of repetitions. The remaining columns list the running times of the three programs in seconds. The speed-up can be

explained by a more economic use of concatenation: The original program *appends* each subforest immediately to the (possibly very long) traversed forest, which can be a quite expensive operation. The refined programs only *prepend* (relatively short) subforests to an intermediate forest, which will be the traversed forest in the next phase. This is, in comparison, a cheap operation.

## 6.2 Continuous Normalization of Infinite De Bruijn Terms

Whether a normalizer/evaluator terminates depends on the object language and on the object expression that is to be evaluated. Usually, an evaluator cannot be implemented in a total meta language. In Section 6.3 we will present an exception: normalization of simply typed lambda-terms can actually be implemented in $\mathsf{F}_{\widehat{\omega}}$. But even evaluators for partial object languages, which might be invoked on non-terminating object programs, can be implemented in $\mathsf{F}_{\widehat{\omega}}$ using coinduction. The trick is to produce the result step by step, and if no new piece of output can be guaranteed in the next step, instead produce a *tick*, which means *wait for more output*. (Of course, nothing guarantees that there will not be an infinite succession of ticks.) Such evaluation techniques are known under the slogan *continuous normalization* in proof theory and have been introduced by Mints [Min78] for the sequent calculus and adopted for natural deduction style term systems by Ruckert [Ruc85] and Schwichtenberg [Sch98], who built upon Buchholz's infinite notations of sequent proofs [Buc91].

Aehlig and Joachimski [AJ05] describe continuous normalization of infinite $\lambda$-terms by guarded corecursion in the sense of Coquand [Coq93]. They give the following grammar for infinite de Bruijn terms with two "repetition" symbols $\mathcal{R}$ and $\beta$ [AJ05, page 43] .

$$\Lambda^{\mathsf{co}}_{\mathcal{R}} \ni r, s ::=^{\mathsf{co}} k \mid \lambda r \mid r\, s \mid \mathcal{R} r \mid \beta r$$

(Herein, $k$ denotes a de Bruijn index.) After defining substitution $r[s]$ of $s$ for the 0th variable in $r$, they give the following recursive definition of a function $r@\vec{s}$, which continuously $\beta$-normalizes the term $r\,\vec{s}$ (where $\vec{s}$ is a possibly empty, finite list of terms)[AJ05, page 46].

$$\begin{aligned}
k @ (s_1, \ldots, s_n) &= k\ (s_1@()) \ldots (s_n@()) \\
(\lambda r) @ (s, \vec{s}) &= \beta(r[s]@\vec{s}) \\
(\lambda r) @ () &= \lambda(r@()) \\
(r\, s) @ \vec{s} &= \mathcal{R}(r@(s, \vec{s})) \\
(\mathcal{R} r) @ \vec{s} &= \mathcal{R}(r@\vec{s}) \\
(\beta r) @ \vec{s} &= \beta(r@\vec{s})
\end{aligned}$$

Since each recursive call is under a term constructor, these equations define a corecursive function according to the guarded-by-constructors principle [Coq93, Gim95]. The constructors $\mathcal{R}$ and $\beta$ do not contribute to the semantics of a

lambda-term, but act as a notification that computation is still ongoing, but the outermost constructor of the lambda-term is not yet known. The symbol $\mathcal{R}$ is produced whenever an application is evaluated (fourth equation), the symbol $\beta$ when a $\beta$-reduction has been performed. It is remarked that continuous normalization could be defined without the $\beta$-constructor. Then, the recursive call in the second equation is no longer under a constructor, thus, violates the guardedness condition. But the recursive call is justified since the length of the second argument has been decreased. Hence, the "termination" argument is lexicographic: Either the definedness of the output is increased (the second argument might be increased, as in the case of application), or the definedness of the output is not increased, but the size of the second argument is decreased. To make this precise, we express the normalization algorithm in $\mathsf{F}_{\widehat{\omega}}$.

In $\mathsf{F}_{\widehat{\omega}}$, we can define a coinductive type $\mathsf{dB} : \mathsf{ord} \to *$ with the following constructors:

$$\begin{array}{lcl} \mathsf{var} & : & \forall \iota.\, \mathsf{Nat} \to \mathsf{dB}^{\iota+1} \\ \mathsf{abs} & : & \forall \iota.\, \mathsf{dB}^{\iota} \to \mathsf{dB}^{\iota+1} \\ \mathsf{app} & : & \forall \iota.\, \mathsf{dB}^{\iota} \to \mathsf{dB}^{\iota} \to \mathsf{dB}^{\iota+1} \\ \mathsf{rep} & : & \forall \iota.\, \mathsf{dB}^{\iota} \to \mathsf{dB}^{\iota+1} \end{array}$$

We assume we have already defined substitution $\mathsf{subst} : \mathsf{dB}^{\infty} \to \mathsf{dB}^{\infty} \to \mathsf{dB}^{\infty}$ (such that $\mathsf{subst}\, r\, s$ returns $r[s]$) and the library function for lists,

$$\begin{array}{l} \mathsf{foldl} : \forall A \forall B.\, (B \to A \to B) \to B \to \mathsf{List}^{\infty} A \to B \\ \mathsf{foldl}\, o\, e\, [a_1, \ldots, a_n] = e\, o\, a_1\, o\, \ldots\, o\, a_n \end{array}$$

where $[a_1, \ldots, a_n]$ denotes, as in ML and Haskell, the list $\mathsf{cons}\, a_1\, (\ldots \mathsf{cons}\, a_n \mathsf{nil})$, and $o$ is a binary operation written infix. Then we can implement the normalization function @ by the following definition:

$$\begin{array}{lcl} \mathsf{napp} & : & \forall \iota \forall \jmath.\, \mathsf{dB}^{\infty} \to \mathsf{List}^{\jmath}\, \mathsf{dB}^{\infty} \to \mathsf{dB}^{\iota} \\ \mathsf{napp} & := & \mathsf{fix}_2^{\nu}\, \lambda napp_0. \\ & & \mathsf{fix}_1^{\mu}\, \lambda napp_1. \\ & & \quad \lambda t \lambda l.\ \mathsf{match}\ t\ \mathsf{with} \\ & & \qquad \mathsf{var}\, k \mapsto \mathsf{foldl}\, (\lambda r' \lambda s.\, \mathsf{app}\, r'\, (napp_0\, s\, \mathsf{nil}))\ t\ l \\ & & \qquad \mathsf{abs}\, r \mapsto \mathsf{match}\ l\ \mathsf{with} \\ & & \qquad\qquad \mathsf{nil} \mapsto \mathsf{abs}\, (napp_0\, r\, \mathsf{nil}) \\ & & \qquad\qquad \mathsf{cons}\, s\, l' \mapsto napp_1\, (\mathsf{subst}\, r\, s)\, l' \\ & & \qquad \mathsf{app}\, r\, s \mapsto \mathsf{rep}\, (napp_0\, r\, (\mathsf{cons}\, s\, l)) \\ & & \qquad \mathsf{rep}\, r \mapsto \mathsf{rep}\, (napp_0\, r\, l) \end{array}$$

This definition is well-typed, since we can assign the following types to vari-

ables and subexpressions:

$$\begin{array}{lcl}
napp_0 & : & \forall \jmath.\, \mathsf{dB}^\infty \to \mathsf{List}^\jmath \mathsf{dB}^\infty \to \mathsf{dB}^\iota \\
napp_1 & : & \mathsf{dB}^\infty \to \mathsf{List}^\jmath \mathsf{dB}^\infty \to \mathsf{dB}^{\iota+1} \\
r, s, t & : & \mathsf{dB}^\infty \\
l & : & \mathsf{List}^{\jmath+1} \mathsf{dB}^\infty \\
r' & : & \mathsf{dB}^{\iota+1} \leq \mathsf{dB}^\iota \\
\mathsf{app}\, r'\, (napp_0\, s\, \mathsf{nil}) & : & \mathsf{dB}^{\iota+1} \\
\mathsf{foldl}\, (\dots)\, t\, l & : & \mathsf{dB}^{\iota+1} \\
\mathsf{abs}\, (napp_0\, r\, \mathsf{nil}) & : & \mathsf{dB}^{\iota+1} \\
l' & : & \mathsf{List}^\jmath \mathsf{dB}^\infty \\
napp_1\, (\mathsf{subst}\, r\, s)\, l' & : & \mathsf{dB}^{\iota+1} \\
\mathsf{rep}\, (napp_0\, \dots) & : & \mathsf{dB}^{\iota+1}
\end{array}$$

An equivalently precise type of napp is the instantiation $\mathsf{dB}^\infty \to \mathsf{List}^\infty \mathsf{dB}^\infty \to \mathsf{dB}^\infty$, but we used the size variables $\iota$ and $\jmath$ to indicate that napp is defined by lexicographic induction over $(\iota, \jmath)$.

**Remark 6.1** The function napp's modulus of continuity is not the identity; we cannot assign to it the more precise type $\forall \iota.\, \mathsf{dB}^\iota \to \mathsf{List}^\infty \mathsf{dB}^\iota \to \mathsf{dB}^\iota$. This would require subst to be of type $\forall \iota.\, \mathsf{dB}^\iota \to \mathsf{dB}^{\iota+1} \to \mathsf{dB}^{\iota+1}$ which is clearly invalid: if in the call subst $r\, s$, the first argument $r$ does not contain the 0th variable, then the result is $r$ which only has type $\mathsf{dB}^\iota$. Hence, the most precise type for subst is $\forall \iota.\, \mathsf{dB}^\iota \to \mathsf{dB}^\iota \to \mathsf{dB}^\iota$.

## 6.3 Normalization of Simply-Typed De Bruijn Terms

We continue the example of Section 3.2.7. Simple types over a fixed type Atom of base types can be represented by the following constructors, which are easily definable $\mathsf{F}_{\widehat{\omega}}$:

$$\begin{array}{lcl}
\mathsf{Ty} & : & \mathsf{ord} \xrightarrow{+} * \\
\mathsf{atom} & : & \forall \iota.\, \mathsf{Atom} \to \mathsf{Ty}^{\iota+1} \\
\mathsf{arr} & : & \forall \iota.\, \mathsf{Ty}^\iota \to \mathsf{Ty}^\iota \to \mathsf{Ty}^{\iota+1}
\end{array}$$

Joachimski and Matthes [JM03] describe a normalization procedure for simply-typed $\lambda$-terms; a similar algorithm has been found by Watkins, Cervesato, Pfenning, and Walker [WCPW03] for a term language of intuitionistic linear logic. At its heart lies an operation $r@s^A$ which produces a the $\beta$-normal form of $r\, s$ where $r$ and $s$ are normal and $s$ is of type $A$. It uses a normalizing substitution function $[s/x : A]r$ which returns the normal form of the substitution $[s/x]r$ if $r$ and $s$ are normal and $s$ is of type $A$. The termination order of these

mutual recursive functions is the lexicographic product of $A$ and $r$.

$$
\begin{array}{lcll}
(\lambda x{:}A.\, r)@s^A & = & [s/x{:}A]r & \\
(x\,\vec{r})@s^A & = & x\,\vec{r}\,s & \\
{[s/x{:}A](\lambda y{:}B.\, r)} & = & \lambda y{:}B.\, [s/x{:}A]r & \text{w.l.o.g., } y \notin \mathsf{FV}(s) \text{ and } x \neq y \\
{[s/x{:}A](r\,t)} & = & r'\,([s/x{:}A]t) & \text{if } r' := [s/x{:}A]r \text{ neutral} \\
 & = & r'@([s/x{:}A]t)^C & \text{otherwise (then } C \text{ is smaller than } A) \\
{[s/x{:}A]y} & = & s^A & \text{if } x = y \\
 & = & y & \text{otherwise}
\end{array}
$$

The result of substitution into a neutral term is either a neutral term or a normal term plus its type. We encode these alternatives in the type $\mathsf{Res}^\iota\, A$, where $\iota$ is an upper bound on the size of the type and $A$ is the set of free variables which might occur in the result term.

$$
\begin{array}{lcl}
\mathsf{Res} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Res} & := & \lambda\iota\lambda A.\, (\mathbf{1} + \mathsf{Ty}^\iota) \times \mathsf{TLam}^\infty\, A \\
\mathsf{res}_{\mathsf{Ne}} & : & \forall\iota.\, \mathsf{TLam}^\infty\, A \to \mathsf{Res}^\iota\, A \\
\mathsf{res}_{\mathsf{Ne}} & := & \lambda r.\, \langle \mathsf{inl}\, \langle\rangle,\ r\rangle \\
\mathsf{res}_{\mathsf{Nf}} & : & \forall\iota.\, \mathsf{Ty}^\iota \to \mathsf{TLam}^\infty\, A \to \mathsf{Res}^\iota\, A \\
\mathsf{res}_{\mathsf{Nf}} & := & \lambda a\lambda r.\, \langle \mathsf{inr}\, a,\ r\rangle \\
\mathsf{weak}_{\mathsf{Res}} & : & \forall\iota\forall A.\, \mathsf{Res}^\iota\, A \to \mathsf{Res}^\iota\, (\mathbf{1} + A) \\
\mathsf{weak}_{\mathsf{Res}} & := & \lambda\langle m, r\rangle.\, \langle m,\ \mathsf{map}_{\mathsf{TLam}}\, \mathsf{inr}\, r\rangle
\end{array}
$$

The function $\mathsf{weak}_{\mathsf{Res}}$ lifts the free variables in a result term by one (the function $\mathsf{map}_{\mathsf{TLam}}$ is defined in Section 3.2.7).

The implementation of $[s/x : A]r$ is problematic, since for our encoding of de Bruijn terms only $r\,\rho$ of all free variables is directly implementable. Substitution of a single variable is then implemented as a special case of parallel substitution. But for the termination of normalizing substitution, the size of type $A$ is important, so we expose it in the mapping $\rho$.

$$
\begin{array}{lcll}
(\lambda x{:}A.\, r)@s^A & = & r(\rho_0[x \mapsto s^A]) & \\
(x\,\vec{r})@s^A & = & x\,\vec{r}\,s & \\
(\lambda y{:}B.\, r)\rho^A & = & \lambda y{:}B.\, r(\rho[y \mapsto y])^A & \text{w.l.o.g., } y \text{ singular in } \rho \\
(r\,s)\rho^A & = & r'\,(s\rho^A) & \text{if } r' := r\rho^A \text{ neutral} \\
 & = & r'@(s\rho^A)^C & \text{otherwise (then } C \text{ is smaller than } A) \\
y\rho^A & = & \rho(y) &
\end{array}
$$

In the first line, $\rho_0$ denotes the identity substitution. A sharp look reveals that the mapping $\rho$ assigns exactly to one variable a non-trivial term, namely $s$ to $x$, all other variables are mapped to themselves. A generalization of this invariant

can be expressed in the type $\mathsf{Subst}^\iota\ A\ B$ of $\rho$:

$$
\begin{array}{lcl}
\mathsf{Subst} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{-} * \xrightarrow{+} * \\
\mathsf{Subst} & := & \lambda\iota\lambda A\lambda B.\ A \to \mathsf{Res}^\iota B \\
\\
\mathsf{sg}_{\mathsf{Subst}} & : & \forall\iota\forall A.\ \mathsf{TLam}^\infty A \to \mathsf{Ty}^\iota \to \mathsf{Subst}^\iota\ (\mathbf{1}+A)\ A \\
\mathsf{sg}_{\mathsf{Subst}} & := & \lambda s\lambda a\lambda mx.\ \mathsf{match}\ mx\ \mathsf{with} \\
 & & \quad \mathsf{inl}\ \langle\rangle \ \mapsto\ \mathsf{res}_{\mathsf{Nf}}\ a\ s \\
 & & \quad \mathsf{inr}\ x \ \mapsto\ \mathsf{res}_{\mathsf{Ne}}\ (\mathsf{var}\ x) \\
\\
\mathsf{lift}_{\mathsf{Subst}} & : & \forall\iota\forall A\forall B.\ \mathsf{Subst}^\iota\ A\ B \to \mathsf{Subst}^\iota\ (\mathbf{1}+A)\ (\mathbf{1}+B) \\
\mathsf{lift}_{\mathsf{Subst}} & := & \lambda rho\lambda mx.\ \mathsf{match}\ mx\ \mathsf{with} \\
 & & \quad \mathsf{inl}\ \langle\rangle \ \mapsto\ \mathsf{res}_{\mathsf{Ne}}\ (\mathsf{var}\ (\mathsf{inl}\ \langle\rangle)) \\
 & & \quad \mathsf{inr}\ x \ \mapsto\ \mathsf{weak}_{\mathsf{Res}}\ (rho\ x)
\end{array}
$$

The call $\mathsf{sg}_{\mathsf{Subst}}\ s\ a : \mathsf{Subst}^\iota\ (\mathbf{1}+A)\ A$ corresponds to $\rho_0[x \mapsto s^A]$; it generates a substitution which maps the variable $x$ in $\mathbf{1}$ to $\mathsf{res}_{\mathsf{Nf}}\ a\ s$ and the variables $y$ in $A$ to $\mathsf{res}_{\mathsf{Ne}}\ (\mathsf{var}\ y)$. The extension $\rho[y \mapsto y]$ of a substitution $\rho$ is implemented by $\mathsf{lift}_{\mathsf{Subst}}\ rho$.

$$
\begin{array}{lcl}
\mathsf{tm} & : & \forall A.\ \mathsf{Res}^\infty A \to \mathsf{TLam}^\infty A \\
\mathsf{tm} & := & \lambda\langle m, r\rangle.\ r \\
\\
\mathsf{abs}_{\mathsf{Res}} & : & \forall\iota\forall A.\ \mathsf{Ty}^\infty \to \mathsf{Res}^\infty(\mathbf{1}+A) \to \mathsf{Res}^\iota A \\
\mathsf{abs}_{\mathsf{Res}} & := & \lambda a\lambda p.\ \langle \mathsf{inl}\ \langle\rangle,\ \mathsf{abs}\ a\ (\mathsf{tm}\ p)\rangle \\
\\
\mathsf{app}_{\mathsf{Nf}} & : & \forall A.\ \mathsf{Ty}^\infty \to \mathsf{TLam}^\infty\ A \to \mathsf{TLam}^\infty\ A \to \mathsf{TLam}^\infty\ A \\
\mathsf{app}_{\mathsf{Nf}} & := & \mathsf{fix}_0^\mu\ \lambda app. \\
 & & \quad \mathsf{let}\ \ \mathsf{subst} = \\
 & & \qquad \mathsf{fix}_0^\mu\ \lambda subst\lambda t\lambda rho.\ \mathsf{match}\ t\ \mathsf{with} \\
 & & \qquad\quad \mathsf{abs}\ c\ u \ \mapsto\ \mathsf{abs}_{\mathsf{Res}}\ c\ (subst\ u\ (\mathsf{lift}_{\mathsf{Subst}}\ rho)) \\
 & & \qquad\quad \mathsf{var}\ y \ \mapsto\ rho\ y \\
 & & \qquad\quad \mathsf{app}\ r\ s \ \mapsto\ \mathsf{match}\ subst\ r\ rho\ \mathsf{with} \\
 & & \qquad\qquad \mathsf{res}_{\mathsf{Ne}}\ r' \ \mapsto\ \mathsf{res}_{\mathsf{Ne}}\ (\mathsf{app}\ r'\ (\mathsf{tm}\ (subst\ s\ rho))) \\
 & & \qquad\qquad \mathsf{res}_{\mathsf{Nf}}\ (\mathsf{arr}\ a\ b)\ r' \ \mapsto\ \mathsf{res}_{\mathsf{Nf}}\ b\ (app\ a\ r'\ (\mathsf{tm}\ (subst\ s\ rho))) \\
 & & \quad \mathsf{in}\ \ \lambda a\lambda r\lambda s.\ \mathsf{match}\ r\ \mathsf{with} \\
 & & \qquad \mathsf{abs}\ \_\ t \ \mapsto\ \mathsf{tm}\ (\mathsf{subst}\ t\ (\mathsf{sg}_{\mathsf{Subst}}\ s\ a)) \\
 & & \qquad \mathsf{var}\ \_ \ \mapsto\ \mathsf{app}\ r\ s \\
 & & \qquad \mathsf{app}\ \_\ \_ \ \mapsto\ \mathsf{app}\ r\ s
\end{array}
$$

The matching of $p := subst\ r\ rho$ is not complete, we have omitted the case $\mathsf{res}_{\mathsf{Nf}}\ (\mathsf{atom}a)\ r'$. This case is excluded by an invariant we cannot express in our type system: that $r$, and also the parallel substitution of $r$ have function type. In practice, this clause is never needed, so we could return anything suitable.

The body of $\mathsf{app}_{\mathsf{Nf}}$ is quite complex; to increase readability we have used a $\mathsf{let}\ x = s\ \mathsf{in}\ t$ construct as syntactic sugar for $(\lambda x t)\ s$. First we consider the

typing of subst:

$$
\begin{array}{lcl}
\mathit{app} & : & \mathsf{Ty}^{\iota} \to \forall A.\, \mathsf{TLam}^{\infty} A \to \mathsf{TLam}^{\infty} A \to \mathsf{TLam}^{\infty} A \\
\mathit{subst} & : & \forall A \forall B.\, \mathsf{TLam}^{\jmath} A \to \mathsf{Subst}^{\iota+1}\, A\, B \to \mathsf{Res}^{\iota+1} B \\
t & : & \mathsf{TLam}^{\jmath+1} A \\
\mathit{rho} & : & \mathsf{Subst}^{\iota+1}\, A\, B \\
c & : & \mathsf{Ty}^{\infty} \\
u & : & \mathsf{TLam}^{\jmath}(\mathbf{1} + A) \\
\mathsf{lift}_{\mathsf{Subst}}\, \mathit{rho} & : & \mathsf{Subst}^{\iota+1}\, (\mathbf{1} + A)\, (\mathbf{1} + B) \\
\mathit{subst}\, u\, (\mathsf{lift}_{\mathsf{Subst}}\, \mathit{rho}) & : & \mathsf{Res}^{\iota+1} (\mathbf{1} + B) \\
\mathsf{abs}_{\mathsf{Res}}\, c\, (\dots) & : & \mathsf{Res}^{\iota+1} B \\
y & : & A \\
\mathit{rho}\, y & : & \mathsf{Res}^{\iota+1} B \\
r, s & : & \mathsf{TLam}^{\jmath}\, A \\
\mathit{subst}\, r\, \mathit{rho} & : & \mathsf{Res}^{\iota+1}\, B \\
r',\ \mathsf{tm}\, (\mathit{subst}\, s\, \mathit{vrho}) & : & \mathsf{TLam}^{\infty} B \\
\mathsf{res}_{\mathsf{Ne}}\, (\mathsf{app}\, r'\, (\dots)) & : & \mathsf{Res}^{\iota+1}\, B \\
\mathsf{arr}\, a\, b & : & \mathsf{Ty}^{\iota+1} \\
a, b & : & \mathsf{Ty}^{\iota} \\
\mathit{app}\, a\, r'\, (\dots) & : & \mathsf{TLam}^{\infty} B \\
\mathsf{res}_{\mathsf{Nf}}\, b\, (\dots) & : & \mathsf{Res}^{\iota} B \leq \mathsf{Res}^{\iota+1} B \\
\mathsf{subst} & : & \forall A \forall B.\, \mathsf{TLam}^{\infty} A \to \mathsf{Subst}^{\iota+1}\, A\, B \to \mathsf{Res}^{\iota+1} B
\end{array}
$$

Well-typedness of $\mathsf{app}_{\mathsf{Nf}}$ now follows:

$$
\begin{array}{lcl}
\mathit{app} & : & \mathsf{Ty}^{\iota} \to \forall A.\, \mathsf{TLam}^{\infty} A \to \mathsf{TLam}^{\infty} A \to \mathsf{TLam}^{\infty} A \\
\mathsf{subst} & : & \forall A \forall B.\, \mathsf{TLam}^{\infty} A \to \mathsf{Subst}^{\iota+1}\, A\, B \to \mathsf{Res}^{\iota+1} B \\
a & : & \mathsf{Ty}^{\iota+1} \\
r, s & : & \mathsf{TLam}^{\infty}\, A \\
t & : & \mathsf{TLam}^{\infty}\, (\mathbf{1} + A) \\
\mathsf{sg}_{\mathsf{Subst}}\, s\, a & : & \mathsf{Subst}^{\iota+1}\, (\mathbf{1} + A)\, A \\
\mathsf{subst}\, t\, (\dots) & : & \mathsf{Res}^{\iota+1}\, A \\
\mathsf{tm}\, (\dots),\ \mathsf{app}\, r\, s & : & \mathsf{TLam}^{\infty} A
\end{array}
$$

## 6.4 Data Types with Higher-Order Parameters

Type-based termination, unlike termination using structural term orderings, scales effortlessly to data types with higher-order parameters. For example, consider the type of monadic lists

$$
\begin{array}{lcl}
\mathsf{MList} & : & \mathsf{ord} \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{MList} & := & \lambda\iota\lambda M\lambda A.\, \mu^{\iota}\lambda X.\, 1 + M\, A \times M\, X
\end{array}
$$

with constructors

$$
\begin{array}{lcl}
\mathsf{mnil} & : & \forall \iota.\, \forall M\!:\!(* \xrightarrow{+} *).\, \forall A. \\
 & & \quad \mathsf{MList}^{\iota+1}\, M\, A \\
\mathsf{mnil} & := & \mathsf{inl}\, \langle\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{mcons} & : & \forall \iota.\, \forall M\!:\!(* \xrightarrow{+} *).\, \forall A. \\
 & & \quad M\, A \to M\, (\mathsf{MList}^{\iota+1}\, M\, A) \to \mathsf{MList}^{\iota+1}\, M\, A \\
\mathsf{mcons} & := & \lambda ma \lambda mas.\, \mathsf{inr}\, \langle ma,\, mas \rangle.
\end{array}
$$

Equality-test for monadic lists $\mathsf{MList}^{\iota}\, M\, A$ must necessarily be parameterized by an equality for $A$ and an *equality transformer* for $M$, meaning a function which turns an equality for an arbitrary type $A$ into an equality for type $M\, A$.

$$
\begin{array}{lcl}
\mathsf{Eq} & : & * \to * \\
\mathsf{Eq} & := & \lambda A.\, A \to A \to \mathsf{Bool}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{eqMList} & : & \forall M.\, (\forall A.\, \mathsf{Eq}\, A \to \mathsf{Eq}\, (M\, A)) \to \forall A.\, \mathsf{Eq}\, A \to \mathsf{Eq}\, (\mathsf{MList}^{\infty} M\, A) \\
\mathsf{eqMList} & := & \lambda meq \lambda eq.\ \mathsf{fix}_0^{\mu}\, \lambda eqMList. \\
 & & \quad \lambda k \lambda l.\mathsf{match}\, \langle k, l \rangle\, \mathsf{with} \\
 & & \qquad \langle \mathsf{mnil},\, \mathsf{mnil} \rangle \mapsto \mathsf{true} \\
 & & \qquad \langle \mathsf{mcons}\, ma\, mas, \\
 & & \qquad\ \mathsf{mcons}\, mb\, mbs \rangle \mapsto meq\ eq\ ma\ mb\ \ \mathsf{and} \\
 & & \qquad\qquad\qquad\qquad\ \ meq\ eqMList\ mas\ mbs \\
 & & \qquad \_ \mapsto \mathsf{false}
\end{array}
$$

The function eqMList exhibits a funny recursion pattern: instead of having a recursive call in which the function is applied to some structurally smaller arguments, in the "recursive call" *meq eqMList*, *it is passed itself as an argument* to one of the function arguments. Now this behavior is surely problematic: If we do not know how *meq* handles its argument, there is no way we could justify termination. For instance, the execution of *meq eqMList* could involve an application of *eqMList* to some non-empty constant lists. In this case eqMList would clearly diverge. Fortunately, the parametric type $\forall A.\, \mathsf{Eq}\, A \to \mathsf{Eq}\, (M\, A)$ of *meq* prevents such a use of *eqMList*. And indeed, eqMList is terminates on all inputs, since it is well-typed in $\mathsf{F}_{\widehat{\omega}}$:

$$
\begin{array}{lcl}
eqMList & : & \mathsf{Eq}\, (\mathsf{MList}^{\iota}\, M\, A) \\
k,\, l & : & \mathsf{MList}^{\iota+1}\, M\, A \\
mas,\, mbs & : & M\, (\mathsf{MList}^{\iota}\, M\, A) \\
meq & : & \forall A.\, \mathsf{Eq}\, A \to \mathsf{Eq}\, (M\, A) \\
meq\ eqMList & : & \mathsf{Eq}\, (M\, (\mathsf{MList}^{\iota}\, M\, A)) \\
meq\ eqMList\ mas\ mbs & : & \mathsf{Bool}
\end{array}
$$

This was a first instance of a non-standard recursion behavior; in the next section we will see more such examples.

## 6.5 Generic Programming

Jansson and Jeuring [JJ97] and Hinze [Hin02] describe frameworks for polytypic and generic programming. In these, both types and values can be constructed by recursion on some index type. A common feature is that the behavior is only specified for the type and constructor constants like $\mathsf{Nat}$, $\mathbf{1}$, $+$ and $\times$, and this uniquely defines the constructed type or value. In the following we propose an ad-hoc extension by sized types, *sized polytypic programming*. This framework is good enough to model Hinze's generalized tries [Hin00b], but whether it scales to other examples requires more research.

**Type-indexed types** are of kind-indexed kinds. A type-indexed constructor $\mathsf{Type}\langle F : \kappa\rangle$ has kind-indexed kind $\mathsf{TYPE}\langle\kappa\rangle$. In the polytypic framework, each such constructor $\mathsf{Type}\langle F\rangle$ and each such kind $\mathsf{TYPE}\langle\kappa\rangle$ must obey the following laws:

$$\begin{array}{lcl}
\mathsf{TYPE}\langle\mathsf{ord}\rangle & = & \mathsf{ord} \\
\mathsf{TYPE}\langle\kappa_1 \xrightarrow{p} \kappa_2\rangle & = & \mathsf{TYPE}\langle\kappa_1\rangle \xrightarrow{p} \mathsf{TYPE}\langle\kappa_2\rangle \\
 & & \\
\mathsf{Type}\langle X\rangle & = & X \\
\mathsf{Type}\langle\lambda X F\rangle & = & \lambda X.\, \mathsf{Type}\langle F\rangle \\
\mathsf{Type}\langle F\, G\rangle & = & \mathsf{Type}\langle F\rangle\ \mathsf{Type}\langle G\rangle \\
\mathsf{Type}\langle\nabla_\kappa\rangle & = & \nabla_{\mathsf{TYPE}\langle\kappa\rangle} \\
\mathsf{Type}\langle\mathsf{s}\rangle & = & \mathsf{s} \\
\mathsf{Type}\langle\infty\rangle & = & \infty
\end{array}$$

This means, function kinds are always mapped to function kinds, application to application, fixed point to fixed point etc. Hence, a kind-indexed kind $\mathsf{TYPE}\langle\kappa\rangle$ is determined by the value of $\mathsf{TYPE}\langle*\rangle$, and a type-indexed type $\mathsf{Type}\langle F\rangle$ by the value of $\mathsf{Type}\langle C\rangle$ for the constants $C$ which appear in $F$.

**Proposition 6.2 (Well-kindedness of type-indexed types)** *Let $\Sigma$ a signature of constructor constants. If $\mathsf{Type}\langle C\rangle : \mathsf{TYPE}\langle\kappa\rangle$ for all $(C : \kappa) \in \Sigma$, and $\mathcal{D} :: X_1 : p_1\kappa_1, \ldots, X_n : p_n\kappa_n \vdash F : \kappa$, then $X_1 : p_1\mathsf{TYPE}\langle\kappa_1\rangle, \ldots, X_n : p_n\mathsf{TYPE}\langle\kappa_n\rangle \vdash \mathsf{Type}\langle F\rangle : \mathsf{TYPE}\langle\kappa\rangle$.*

*Proof.* By induction on $\mathcal{D}$. □

**Remark 6.3** Note that the presence of polarities restricts the choices for $\mathsf{Type}\langle C\rangle$. However, if index types are constructed in a signature without polymorphism and function space, as it is usual in the generic programming community, all function kinds are covariant and we do not have to worry about polarities.

**Example: finite maps via generalized tries.** Hinze [Hin00b] defines generalized tries $\mathsf{Map}\langle F\rangle$ by recursion on $F$. In particular, $\mathsf{Map}\langle K : *\rangle\, V$ is the type of

finite maps from domain $K$ to codomain $V$. The following representation using type-level $\lambda$ can be found in his article on type-indexed data types [HJL04, page 139].

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

Well-kindedness of these definitions is immediate, except maybe for $\mathsf{Map}\langle\times\rangle$ which must be of kind $(* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *)$. For $\mathsf{Map}\langle+\rangle$ we have used the variant of *spotted products* (or lifted products) which Hinze mentions in section 4.1 of his article [Hin00b]. This way we avoid that certain empty tries have a infinite normal form (see [Hin00b, page 341]) which requires lazy evaluation. The constructor for finite maps over strings can now be computed as follows:

$$
\begin{array}{ll}
 & \mathsf{Map}\langle \lambda\imath.\, \mathsf{List}'\, \mathsf{Char} \rangle \\
= & \mathsf{Map}\langle \lambda\imath.\, \mu_*\, \imath\, \lambda X.\, \mathbf{1} + \mathsf{Char} \times X \rangle \\
= & \lambda\imath.\, \mu_{*\xrightarrow{+}*}\, \imath\, \lambda X.\, \mathsf{Map}\langle + \rangle\, \mathsf{Map}\langle \mathbf{1} \rangle\, (\mathsf{Map}\langle \times \rangle\, \mathsf{Map}\langle \mathsf{Char} \rangle\, X) \\
= & \lambda\imath.\, \mu_{*\xrightarrow{+}*}\, \imath\, \lambda X \lambda V.\, \mathbf{1} + (\mathbf{1} + V) \times \mathsf{Map}\langle \mathsf{Char} \rangle\, (X\, V) \\
(= & \lambda\imath \lambda V.\, \mu_*\, \imath\, \lambda Y.\, \mathbf{1} + (\mathbf{1} + V) \times \mathsf{Map}\langle \mathsf{Char} \rangle\, Y)
\end{array}
$$

For the last (and optional) step, we have applied $\lambda$-dropping (see Sec. 3.3.5), to turn the second-order fixed point into a first-order one. The matching kind is

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

**Type-index values** $\mathsf{poly}\langle F : \kappa \rangle$ are of kind-indexed types. These types $\mathsf{Poly}\langle F : \kappa \rangle$ are defined by recursion on $\kappa$, and $F$ is just a parameter. The kind $\kappa$ must fit into the grammar

$$\kappa ::= * \mid \mathsf{ord} \xrightarrow{p} \kappa \mid \kappa_1 \xrightarrow{p} \kappa_2.$$

Note that Hinze, Jeuring, and Löh [HJL04, page 142] allow kind-indexed types $\mathsf{Poly}\langle \ldots \rangle$ with several constructor parameters. For our examples, however, a single parameter is sufficient.

The following laws hold for all constructor-indexed values $\mathsf{poly}\langle F : \kappa \rangle$ and

kind-indexed types $\mathsf{Poly}\langle F : \kappa \rangle$ in the framework:

$$
\begin{array}{lcll}
\mathsf{Poly}\langle F : \mathsf{ord} \xrightarrow{p} \kappa \rangle & = & \forall \iota : \mathsf{ord}.\, \mathsf{Poly}\langle F\, \iota : \kappa \rangle & \\
\mathsf{Poly}\langle F : \kappa_1 \xrightarrow{p} \kappa_2 \rangle & = & \forall G : \kappa_1.\, \mathsf{Poly}\langle G : \kappa_1 \rangle \to \mathsf{Poly}\langle F\, G : \kappa_2 \rangle & \\
 & & & \\
\mathsf{poly}\langle X \rangle & = & x & \\
\mathsf{poly}\langle \lambda \iota F : \mathsf{ord} \to \kappa \rangle & = & \mathsf{poly}\langle F \rangle & \\
\mathsf{poly}\langle \lambda X F : \kappa_1 \to \kappa_2 \rangle & = & \lambda x.\, \mathsf{poly}\langle F \rangle & \text{where } \kappa_1 \neq \mathsf{ord} \\
\mathsf{poly}\langle \nabla_\kappa\, a \rangle & = & \mathsf{fix}_n^{\nabla} & \text{for some } n \\
\mathsf{poly}\langle F\, G \rangle & = & \mathsf{poly}\langle F \rangle\, \mathsf{poly}\langle G \rangle & \text{where } F \neq \nabla
\end{array}
$$

In order to ignore size expressions, which do not contribute to the computational behavior of the generic value $\mathsf{poly}\langle F \rangle$, we ignore size abstractions and size parameters to fixed points. To make this work, the constructor parameter $F$ must be in normal form, and constants $C$ of the signature should not take a size argument. A polytypic value of an (co)inductive constructor is (co)recursive. The parameter $n$ in $\mathsf{fix}_n^{\nabla}$ must be given appropriately.

Note that kind-indexed types need not be compositional, i. e., in general $\mathsf{Poly}\langle F\, G \rangle = \mathsf{Poly}\langle F \rangle\, \mathsf{Poly}\langle G \rangle$ does *not* hold. On the other hand, they are parametric in the constructor argument, since only a single equation $\mathsf{Poly}\langle A : * \rangle = \ldots$ is given by the user, which does not analyze the structure of $A$.

**Example: finite map lookup.** We give an adaption of Hinze's generic lookup function to our setting. Herein, we use the bind operation $\gg\!=$ for the *Maybe* monad $\lambda V.\, \mathbf{1} + V$. It obeys the laws $\mathsf{inl}() \gg\!= f \longrightarrow^+ \mathsf{inl}()$ and $\mathsf{inr}\, v \gg\!= f \longrightarrow^+ f\, v$. Note that in this section, we will write pairs as $(r, s)$ instead of $\langle r, s \rangle$ and the empty tuple as $()$ instead of $\langle\rangle$, to avoid confusions with the notation for type indices.

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

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

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

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

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

Note that the type of $\mathsf{lookup}\langle\mathsf{List}\rangle$ mentions the size variable $\iota$ twice, as index to both inductive arguments. This makes sense, since the length of the search keys determines the depth of the trie. Welltypedness can be ensured on an abstract level:

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

Finally, the type $\mathsf{Lookup}\langle\mathsf{List}^\iota K\rangle$ is admissible for recursion on the first argument, since the first argument is of shape $\mu^\iota F$ and the whole type $\mathsf{Lookup}\langle\mathsf{List}^\iota K\rangle$ is upper semi-continuous in $\iota$.

**Trie merging.** Hinze [Hin00b] presents three elementary operations to construct finite tries: empty, single, and merge. In the following we replay the construction of merge in our framework, since it exhibits a most interesting recursion scheme.

First we define the type $\mathsf{Bin}\, V$ for binary operations on $V$ and a function comb which lifts a merging function for $V$ to a merging function for $\mathbf{1} + V$.

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

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

The following definitions determine a generic merging function.

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

The instance for list tries can be computed as follows:

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

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

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

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

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

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

$$\begin{array}{lcl}
\mathsf{Bush} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Bush} & := & \lambda\iota.\,\mu_{*\xrightarrow{+}*}\,\iota\,\lambda X\lambda K.\,\mathbf{1} + K \times X\,(X\,K) \\
 & & \\
\mathsf{Map}\langle\mathsf{Bush}\rangle & : & \mathsf{ord} \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *) \\
\mathsf{Map}\langle\mathsf{Bush}\rangle & = & \lambda\iota.\,\mu_{(*\xrightarrow{+}*)\xrightarrow{+}(*\xrightarrow{+}*)}\,\iota\,\lambda X\lambda F\lambda V.\,\mathbf{1} + (\mathbf{1} + V) \times F\,(X\,(X\,F)\,V)
\end{array}$$

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

$$\begin{array}{l}
\mathsf{merge}\langle\mathsf{Bush}\rangle \\
\quad = \quad \mathsf{merge}\langle\lambda\iota.\,\mu^\iota\,\lambda X\lambda K.\,\mathbf{1} + K \times X\,(X\,K)\rangle \\
\quad = \quad \mathsf{fix}_2^\mu\,\lambda merge\lambda merge_K. \\
\qquad\quad \mathsf{merge}\langle+\rangle\,\mathsf{merge}\langle\mathbf{1}\rangle\,(\mathsf{merge}\langle\times\rangle\,merge_K\,(merge\,(merge\,merge_K))) \\
\quad = \quad \mathsf{fix}_2^\mu\,\lambda merge\lambda merge_K \\
\qquad\quad \lambda c.\,\mathsf{comb}\,\lambda(mv_1, t_1)\lambda(mv_2, t_2). \\
\qquad\qquad (\mathsf{comb}\;c\;mv_1\;mv_2,\;\; merge_K\,(merge\,(merge\,merge_K)\,c)\,t_1\,t_2)
\end{array}$$

The recursion pattern of $\mathsf{merge}\langle\mathsf{Bush}\rangle$ is adventurous. Not only is the recursive instance *merge* passed to an argument to the function $merge_K$, but also this function is modified during recursion: it is replaced by $(merge\,merge_K)$, which involves the recursive instance again! All these complications are "miraculously" resolved by typing!

**Related work.** Hinze, Jeuring, and Löh [HJL04] present generic programming with type-indexed datatypes and type-indexed value both in Haskell and the higher-order polymorphic $\lambda$-calculus with a standard model. Altenkirch [Alt01] investigates the representation of *total* functions over inductive types as higher-order coinductive types. He establishes isomorphisms in a categorical semantics.

## 6.6 Impredicative Data Types

Some kinds of structural orderings for termination do not scale to impredicativity. On such ordering is described by Coquand [Coq92] and implemented in the termination checker foetus [AA02, Abe99]. It rests on the axioms

$$\begin{array}{lcll}
t & < & \mathsf{c}\,t, & \text{if c is a data constructor, and} \\
f\,t & \leq & f. &
\end{array}$$

In the presence of datatypes with impredicativity, this ordering is not well-founded. Coquand demonstrates this using a type $\mathsf{V}$ with a single constructor

$$\mathsf{c} : (\forall A.\, A \to A) \to \mathsf{V}.$$

With $\mathsf{id} = \lambda x x$ we obtain the cycle $\mathsf{c}\,\mathsf{id} > \mathsf{id} \geq \mathsf{id}\,(\mathsf{c}\,\mathsf{id}) = \mathsf{c}\,\mathsf{id}$. Hence, a termination checker based on this ordering would accept the recursive program $f\,(\mathsf{c}\,g) = f\,(g\,(\mathsf{c}\,g))$. Let us analyze this paradox with sized types:

$$\begin{array}{lcl} \mathsf{V} & : & \mathsf{ord} \xrightarrow{+} * \\ \mathsf{V} & := & \lambda\iota.\, \mu^{\iota}_{*} \lambda\_.\, \forall A.\, A \to A \\ \\ \mathsf{loop} & : & \forall\iota.\, \mathsf{V}^{\iota} \to \mathbf{0} \\ \mathsf{loop} & := & \mathsf{fix}^{\mu}_{0}\, \lambda f \lambda g.\, f\,(g\,g) \end{array}$$

Type-based termination rejects this program: We have

$$\begin{array}{lcl} g & : & \forall A.\, A \to A \\ g & : & \mathsf{V}^{\iota+1} \\ g\,g & : & \mathsf{V}^{\iota+1}, \end{array}$$

but $f : \mathsf{V}^{\iota} \to \mathbf{0}$ does not accept this argument.

## 6.7 Inductive Proofs as Recursive Functions

In this section, we will demonstrate how $\mathsf{F}^{\widehat{}}_{\omega}$ can be used to certify termination of recursive functions which correspond to proofs by induction in some dependent type theory.

As example theorem, we consider transitivity for a simple inductively defined subtyping relation. A similar example has been given by Wahlstedt [Wah04], who uses size-change termination [LJBA01].

We consider a language of simple types $\sigma, \tau$ with a least type $\bot$ and a greatest type $\top$.

$$\mathsf{Ty} \ni \sigma, \tau ::= \bot \mid \top \mid \sigma \times \tau \mid \sigma \Rightarrow \tau$$

Subtyping is defined inductively by the following rules:

$$\mathsf{SBot}\ \frac{}{\bot <: \tau} \qquad \mathsf{STop}\ \frac{}{\tau <: \top}$$

$$\mathsf{SProd}\ \frac{\sigma_1 <: \sigma_2 \qquad \tau_1 <: \tau_2}{\sigma_1 \times \tau_1 <: \sigma_2 \times \tau_2} \qquad \mathsf{SArr}\ \frac{\sigma_2 <: \sigma_1 \qquad \tau_1 <: \tau_2}{\sigma_1 \Rightarrow \tau_1 <: \sigma_2 \Rightarrow \tau_2}$$

We represent this subtyping relation by an inductive family $\mathsf{Sub} : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Prop}$ with four constructors:

$$\begin{array}{lcl} \mathsf{SBot} & : & \mathsf{Sub}\,\bot\,\tau \\ \mathsf{STop} & : & \mathsf{Sub}\,\tau\,\top \\ \mathsf{SProd} & : & \mathsf{Sub}\,\sigma_1\,\sigma_2 \to \mathsf{Sub}\,\tau_1\,\tau_2 \to \mathsf{Sub}\,(\sigma_1 \times \tau_1)\,(\sigma_2 \times \tau_2) \\ \mathsf{SArr} & : & \mathsf{Sub}\,\sigma_2\,\sigma_1 \to \mathsf{Sub}\,\tau_1\,\tau_2 \to \mathsf{Sub}\,(\sigma_1 \Rightarrow \tau_1)\,(\sigma_2 \Rightarrow \tau_2) \end{array}$$

We have suppressed the arguments of type Ty for all constructors: SBot and SBot take an additional hidden argument $\tau$, and SProd and SArr take four hidden arguments $\sigma_1$, $\sigma_2$, $\tau_1$, and $\tau_2$.

The relation $<:$ is transitive, and it can be shown by induction on the sum of the heights of the two input derivations. In type theory, the proof can be implemented as a recursive function

$$\mathsf{trans} : \mathsf{Sub}\,\tau_1\,\tau_2 \to \mathsf{Sub}\,\tau_2\,\tau_3 \to \mathsf{Sub}\,\tau_1\,\tau_3.$$

Again, we consider $\tau_1$, $\tau_2$, and $\tau_3$ hidden arguments of trans. Each case in the proof of transitivity corresponds to one pattern matching clause of trans. Using the type dependencies, the following matching is complete:

$$\begin{array}{llll}
\mathsf{trans} & \mathsf{SBot} & _ & = \mathsf{SBot} \\
\mathsf{trans} & _ & \mathsf{STop} & = \mathsf{STop} \\
\mathsf{trans} & (\mathsf{SProd}\,d_1\,d_1') & (\mathsf{SProd}\,d_2\,d_2') & = \mathsf{SProd}\,(\mathsf{trans}\,d_1\,d_2)\,(\mathsf{trans}\,d_1'\,d_2') \\
\mathsf{trans} & (\mathsf{SArr}\,d_1\,d_1') & (\mathsf{SArr}\,d_2\,d_2') & = \mathsf{SArr}\,(\mathsf{trans}\,d_2\,d_1)\,(\mathsf{trans}\,d_1'\,d_2')
\end{array}$$

This function is not just defined by structural recursion on the first argument or by lexicographic recursion on both arguments, since in the last line, a subterm ($d_2$) of the second argument appears as first argument to a recursive call ($\mathsf{trans}\,d_2\,d_1$) and vice versa. A valid termination measure would be the sum of the size of both arguments. We can also certify its termination by typing it in $\mathsf{F}_{\widehat{\omega}}$. To this end, we have to erase all dependencies.

In $\mathsf{F}_{\widehat{\omega}}$, we set $\mathsf{Sub}^\iota := \mu^\iota\,\lambda X.\,\mathbf{1} + \mathbf{1} + X \times X + X \times X$. The four constructors are now definable:

$$\begin{array}{lll}
\mathsf{SBot} & : & \forall \imath.\,\mathsf{Sub}^{\iota+1} \\
\mathsf{STop} & : & \forall \imath.\,\mathsf{Sub}^{\iota+1} \\
\mathsf{SProd} & : & \forall \imath.\,\mathsf{Sub}^\iota \to \mathsf{Sub}^\iota \to \mathsf{Sub}^{\iota+1} \\
\mathsf{SArr} & : & \forall \imath.\,\mathsf{Sub}^\iota \to \mathsf{Sub}^\iota \to \mathsf{Sub}^{\iota+1}
\end{array}$$

The recursive function can now be given the type

$$\mathsf{trans} : \forall \imath.\,\mathsf{Sub}^\iota \to \mathsf{Sub}^\iota \to \mathsf{Sub}^\iota.$$

One problem remains: In $\mathsf{F}_{\widehat{\omega}}$, we have no partiality, but without dependencies, the four patterns defining trans do not cover the full value space. This problem can be mended by adding a catch-all clause $\mathsf{trans}\ \_\ \_ = \mathsf{SBot}$, or by extending $\mathsf{F}_{\widehat{\omega}}$ by partiality. For instance, one could have *four* eliminations of the disjoint sum type, $\mathsf{case}_S$, for $S \subseteq \{\mathsf{l}, \mathsf{r}\}$, with typing

$$\begin{array}{lll}
\mathsf{case}_{\{\}} & : & \forall A \forall B \forall C.\, A + B \to C \\
\mathsf{case}_{\{\mathsf{l}\}} & : & \forall A \forall B \forall C.\, A + B \to (A \to C) \to C \\
\mathsf{case}_{\{\mathsf{r}\}} & : & \forall A \forall B \forall C.\, A + B \to (B \to C) \to C \\
\mathsf{case}_{\{\mathsf{l},\mathsf{r}\}} & : & \forall A \forall B \forall C.\, A + B \to (A \to C) \to (B \to C) \to C
\end{array}$$

and the reduction rules

$$\begin{array}{lcl}
\mathsf{case}_{\{\mathsf{l}\}}\,(\mathsf{in}_{\mathsf{l}}\,r)\,s & \rightarrowtail & s\,r \\
\mathsf{case}_{\{\mathsf{r}\}}\,(\mathsf{in}_{\mathsf{r}}\,r)\,t & \rightarrowtail & t\,r \\
\mathsf{case}_{\{\mathsf{l},\mathsf{r}\}}\,(\mathsf{in}_{\mathsf{l}}\,r)\,s\,t & \rightarrowtail & s\,r \\
\mathsf{case}_{\{\mathsf{l},\mathsf{r}\}}\,(\mathsf{in}_{\mathsf{r}}\,r)\,s\,t & \rightarrowtail & t\,r.
\end{array}$$

A term $\mathsf{case}_S\,(\mathsf{in}_k\,r)$ with $k \notin S$ would be considered neutral.

# Chapter 7

# Extensions

In this chapter, we sketch some useful extensions of $\mathsf{F}\widehat{\omega}$.

## 7.1 Mutual Recursion

Normal de Bruijn terms can be defined by two mutually recursive type constructors in Haskell:

```
data Ne a = Var a
          | App (Ne a) (Nf a)

data Nf a = Ne (Ne a)
          | Abs (Nf (Maybe a))
```

In $\mathsf{F}\widehat{\omega}$, there is no notion of mutual recursion, but it can be simulated with nested recursion.[1] For mutual type constructors, the standard encoding into nested inductive type constructors works smoothly:

$$
\begin{array}{lcl}
\mathsf{NeF}, \mathsf{NfF} & : & (* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{NeF} & := & \lambda X \lambda Y \lambda A.\, A + Y\, A \times X\, A \\
\mathsf{NfF} & := & \lambda X \lambda Y \lambda Y.\, Y\, A + X\, (\mathbf{1} + A) \\
 & & \\
\mathsf{Ne}, \mathsf{Nf} & : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Ne} & := & \lambda \imath.\, \mu^{\imath}_{*\xrightarrow{+}*}\, \lambda Y.\, \mathsf{NeF}\, (\mu_{*\xrightarrow{+}*}\, \lambda X.\, \mathsf{NfF}\, X\, Y)\, Y \\
\mathsf{Nf} & := & \lambda \imath.\, \mu^{\imath}_{*\xrightarrow{+}*}\, \lambda X.\, \mathsf{NfF}\, X\, (\mu_{*\xrightarrow{+}*}\, \lambda Y.\, \mathsf{NeF}\, X\, Y)
\end{array}
$$

[1] Nested is also called *interleaving* [AA00].

As usual, we can define the following data constructors:

$$\begin{array}{lcl}
\mathsf{var} & : & \forall A \forall \imath.\, A \to \mathsf{Ne}^{\imath+1}\, A \\
\mathsf{var} & := & \lambda x.\, \mathsf{inl}\, x \\
\mathsf{app} & : & \forall A \forall \imath.\, \mathsf{Ne}^{\imath}\, A \to \mathsf{Nf}^{\imath} A \to \mathsf{Ne}^{\imath+1}\, A \\
\mathsf{app} & := & \lambda r \lambda s.\, \mathsf{inr}\, \langle r, s \rangle \\
\mathsf{ne} & : & \forall A \forall \imath.\, \mathsf{Ne}^{\imath} A \to \mathsf{Nf}^{\imath+1}\, A \\
\mathsf{ne} & := & \lambda r.\, \mathsf{inl}\, r \\
\mathsf{abs} & : & \forall A \forall \imath.\, \mathsf{Nf}^{\imath} A \to \mathsf{Nf}^{\imath+1}\, A \\
\mathsf{abs} & := & \lambda r.\, \mathsf{inr}\, r
\end{array}$$

To encode mutual recursion on the term level, we need to extend the type system of $\mathsf{F}\widehat{\omega}$. We basically need a *well-founded recursion* rule and a *bounded* recursion rule:

$$\textsc{ty-wf-rec}\ \frac{\Gamma \vdash A\ \mathsf{fix}_n^{\nabla}\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}_n^{\nabla} : (\forall \imath{:}\mathsf{ord}.\, (\forall \jmath \leq \imath.\, A\, \jmath) \to A\, (\imath+1)) \to A\, a}$$

$$\textsc{ty-bd-rec}\ \frac{\Gamma \vdash A\ \mathsf{fix}_n^{\nabla}\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}_n^{\nabla} : (\forall \imath \leq a.\, A\, \imath \to A\, (\imath+1)) \to A\, a'}\ a' \in \{a, \mathsf{s}\, a\}$$

Bounded quantification $\forall X \leq G : \kappa.\, A$ is to be understood as usual. It has been studied extensively in the last decade, e. g., by Pierce and Steffen [PS97, Ste98] and Duggan and Compagnoni [DC99]. In our calculus, we could add it through a constant and some defined notation:

$$\begin{array}{lcl}
\forall_\kappa^{\leq} & : & \kappa \xrightarrow{-} (\kappa \xrightarrow{\circ} *) \xrightarrow{+} * \\
\forall X \leq G{:}\kappa.\, A & := & \forall_\kappa^{\leq}\, G\, (\lambda X A)
\end{array}$$

Adding a top kind $\mathsf{Top}_\kappa : \kappa$ for all kinds (for ord it is $\infty$), we can then define usual quantification by $\forall_\kappa := \forall_\kappa^{\leq}\, \mathsf{Top}_\kappa$. We replace the generalization and instantiation rules by the following:

$$\textsc{ty-gen}\ \frac{\Gamma, X \leq G{:}\kappa \vdash t : F\, X}{\Gamma \vdash t : \forall_\kappa^{\leq} G\, F} \qquad \textsc{ty-inst}\ \frac{\Gamma \vdash t\ :\ \forall_\kappa^{\leq} H\, F \qquad \Gamma \vdash G \leq H : \kappa}{\Gamma \vdash t : F\, G}$$

Figure 7.1 displays the mapping terms of normal and neutral de Bruijn terms. The term $\mathsf{map}_{\mathsf{Nf}}$ reuses the lifting operation from Section 3.2.7. We have used the standard encoding of mutual recursion into nested recursion, and it is compatible with our extension of $\mathsf{F}\widehat{\omega}$as the following type assignment for $\mathsf{map}_{\mathsf{Nf}}$ in Figure 7.1 shows.

Bounded and well-founded recursion can be combined into bounded well-founded recursion.

$$\textsc{ty-bd-wf-rec}\ \frac{\Gamma \vdash A\ \mathsf{fix}_n^{\nabla}\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}_n^{\nabla} : (\forall \imath \leq a.\, (\forall \jmath \leq \imath.\, A\, \jmath) \to A\, (\imath+1)) \to A\, a'}\ a' \in \{a, \mathsf{s}\, a\}$$

Mapping terms.

$$
\begin{array}{lcl}
\mathsf{Map}_{\mathsf{Ne}}, \mathsf{Map}_{\mathsf{Nf}} & : & \mathsf{ord} \xrightarrow{\circ} * \\
\mathsf{Map}_{\mathsf{Ne}} & := & \lambda\imath.\, \forall A \forall B.\, (A \to B) \to \mathsf{Ne}^{\imath} A \to \mathsf{Ne}^{\imath} B \\
\mathsf{Map}_{\mathsf{Nf}} & := & \lambda\imath.\, \forall A \forall B.\, (A \to B) \to \mathsf{Ne}^{\imath} A \to \mathsf{Ne}^{\imath} B \\
\\
\mathsf{map}_{\mathsf{NeF}} & : & \forall\imath.\, \mathsf{Map}_{\mathsf{Nf}}\imath \to \mathsf{Map}_{\mathsf{Ne}}\imath \to \mathsf{Map}_{\mathsf{Ne}}(\imath+1) \\
\mathsf{map}_{\mathsf{NeF}} & := & \lambda map_{\mathsf{Nf}} \lambda map_{\mathsf{Ne}} \lambda f \lambda t.\, \mathsf{match}\ t\ \mathsf{with} \\
 & & \quad \mathsf{var}\, x \quad \mapsto \quad \mathsf{var}\, (f\, x) \\
 & & \quad \mathsf{app}\, r\, s \quad \mapsto \quad \mathsf{app}\, (map_{\mathsf{Ne}}\, f\, r)\, (map_{\mathsf{Nf}}\, f\, s) \\
\\
\mathsf{map}_{\mathsf{NfF}} & : & \forall\imath.\, \mathsf{Map}_{\mathsf{Nf}}\imath \to \mathsf{Map}_{\mathsf{Ne}}\imath \to \mathsf{Map}_{\mathsf{Nf}}(\imath+1) \\
\mathsf{map}_{\mathsf{NfF}} & := & \lambda map_{\mathsf{Nf}} \lambda map_{\mathsf{Ne}} \lambda f \lambda t.\, \mathsf{match}\ t\ \mathsf{with} \\
 & & \quad \mathsf{ne}\, r \quad \mapsto \quad \mathsf{ne}\, (map_{\mathsf{Ne}}\, f\, r) \\
 & & \quad \mathsf{abs}\, r \quad \mapsto \quad \mathsf{abs}\, (map_{\mathsf{Nf}}\, (\mathsf{lift}\, f)\, r) \\
\\
\mathsf{map}_{\mathsf{Ne}} & : & \forall\imath.\, \mathsf{Map}_{\mathsf{Ne}}\imath \\
\mathsf{map}_{\mathsf{Ne}} & := & \mathsf{fix}_1^{\mu}\, \lambda map_{\mathsf{Ne}}.\, \mathsf{map}_{\mathsf{NeF}}\, (\mathsf{fix}_1^{\mu}\, \lambda map_{\mathsf{Nf}}.\, \mathsf{map}_{\mathsf{NfF}}\, map_{\mathsf{Nf}}\, map_{\mathsf{Ne}})\, map_{\mathsf{Ne}} \\
\\
\mathsf{map}_{\mathsf{Nf}} & : & \forall\imath.\, \mathsf{Map}_{\mathsf{Nf}}\imath \\
\mathsf{map}_{\mathsf{Nf}} & := & \mathsf{fix}_1^{\mu}\, \lambda map_{\mathsf{Nf}}.\, \mathsf{map}_{\mathsf{NfF}}\, map_{\mathsf{Nf}}\, (\mathsf{fix}_1^{\mu}\, \lambda map_{\mathsf{Ne}}.\, \mathsf{map}_{\mathsf{NeF}}\, map_{\mathsf{Nf}}\, map_{\mathsf{Ne}})
\end{array}
$$

Type assignment for $\mathsf{map}_{\mathsf{Nf}}$.

$$
\begin{array}{lcll}
map_{\mathsf{Nf}} & : & \forall\jmath \leq \imath.\, \mathsf{Map}_{\mathsf{Nf}}\jmath & \text{assumption} \\
\jmath \leq \imath & & & \text{assumption} \\
map_{\mathsf{Ne}} & : & \mathsf{Map}_{\mathsf{Nf}}\jmath & \text{assumption} \\
map_{\mathsf{Nf}} & : & \mathsf{Map}_{\mathsf{Nf}}\imath & \text{instantiation} \\
\mathsf{map}_{\mathsf{NeF}}\, map_{\mathsf{Nf}}\, map_{\mathsf{Ne}} & : & \mathsf{Map}_{\mathsf{Ne}}(\jmath+1) & \\
(\mathsf{fix}_0^{\mu}\, \lambda map_{\mathsf{Ne}} \ldots) & : & \mathsf{Map}_{\mathsf{Ne}}\imath & \text{TY-BD-REC} \\
map_{\mathsf{Nf}} & : & \mathsf{Map}_{\mathsf{Nf}}\imath & \text{instantiation} \\
\mathsf{map}_{\mathsf{NfF}}\, map_{\mathsf{Nf}}\, (\ldots) & : & \mathsf{Map}_{\mathsf{Nf}}(\imath+1) & \text{TY-WF-REC}
\end{array}
$$

Figure 7.1: Functoriality for normal de Bruijn terms.

The presence of bounded type assumptions $X \leq G : \kappa$ in $\Gamma$, especially $\iota \leq a$ for ordinals, does not jeopardize our normalization result. The induction principle behind TY-BD-WF-REC can easily justified by transfinite induction. A more natural formulation of this rule would one using strict inequality of ordinals:

$$\frac{\Gamma \vdash A\ \mathsf{fix}_n^{\nabla}\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}_n^{\nabla} : (\forall \iota{<}a.\, (\forall \jmath{<}\iota.\, A\,\jmath) \to A\,\iota) \to A\,a}$$

But how to prove normalization in presence of strict inequality assumptions $\iota < a$ in $\Gamma$ requires further thought.

**Mutual Recursion Via Products.** Alternatively, mutual recursion can be added as a new principle, realized via products. This has been carried out by Pareto [Par00, p. 152] and Xi [Xi02]. To implement mutual recursion on the constructor level we need product kinds.

## 7.2 More Admissible Types

Our criterion for types of recursive functions admits the type $\forall \iota.\, \mathsf{Nat}^\iota \to \mathsf{Nat}^\iota \to \mathsf{Nat}^\iota$, e. g., as type of the minimum or maximum function for natural numbers, but not the isomorphic type $\forall \iota.\, \mathsf{Nat}^\iota \times \mathsf{Nat}^\iota \to \mathsf{Nat}^\iota$. Using this type, the minimum function would be coded as follows:

$$\begin{array}{lcl}
\mathsf{min} & : & \forall \iota.\, \mathsf{Nat}^\iota \times \mathsf{Nat}^\iota \to \mathsf{Nat}^\iota \\
\mathsf{min} & := & \mathsf{fix}_0^\mu\, \lambda min \lambda \langle x, y \rangle.\, \mathsf{match}\ \langle x, y \rangle\ \mathsf{with} \\
 & & \quad \langle \mathsf{zero},\ _ \rangle \mapsto \mathsf{zero} \\
 & & \quad \langle _,\ \mathsf{zero} \rangle \mapsto \mathsf{zero} \\
 & & \quad \langle \mathsf{succ}\, x',\ \mathsf{succ}\, y' \rangle \mapsto \mathsf{succ}\, (min\, \langle x', y' \rangle)
\end{array}$$

However, in our reduction semantics, min is not strongly normalizing, since the pair $\langle x', y' \rangle$ counts as a value and triggers unfolding of recursion:

$$\begin{array}{lcl}
\mathsf{min}\, \langle x, y \rangle & \longrightarrow^+ & \mathsf{match} \ldots \mathsf{succ}\, (\mathsf{min}\, \langle x', y' \rangle) \\
 & \longrightarrow^+ & \mathsf{match} \ldots \mathsf{succ}\, (\mathsf{match} \ldots \mathsf{succ}\, (\mathsf{min}\, \langle x'', y'' \rangle)) \\
 & \longrightarrow^+ & \ldots
\end{array}$$

The most promising solution to this problem is add patterns as a primitive language construct and to tie unfolding recursion to pattern matching:

$$\begin{array}{lcl}
\mathsf{min} & : & \forall \iota.\, \mathsf{Nat}^\iota \times \mathsf{Nat}^\iota \to \mathsf{Nat}^\iota \\
\mathsf{min} & := & \mathsf{fix}_0^\mu\, \lambda min \lambda \left( \begin{array}{lcl}
\langle \mathsf{zero},\ _ \rangle & \mapsto & \mathsf{zero} \\
\langle _,\ \mathsf{zero} \rangle & \mapsto & \mathsf{zero} \\
\langle \mathsf{succ}\, x',\ \mathsf{succ}\, y' \rangle & \mapsto & \mathsf{succ}\, (min\, \langle x', y' \rangle)
\end{array} \right)
\end{array}$$

The reduction semantics has to be adapted such that $\mathsf{min}\, \langle x, y \rangle$ is no longer a redex, only $\mathsf{min}\, \langle \mathsf{zero}, y \rangle$ and $\mathsf{min}\, \langle x, \mathsf{zero} \rangle$, which both reduce to zero, and $\mathsf{min}\, \langle \mathsf{succ}\, x',\ \mathsf{succ}\, y' \rangle$, which reduces to $\mathsf{succ}\, (\mathsf{min}\, \langle x', y' \rangle)$ in one step.

Since products of the form $A \times A$ are isomorphic to $\mathsf{Bool} \to A$, one could imagine a type system which also accept the following, a little artificial, variant of the minimum function:

$$
\begin{array}{lcl}
\mathsf{min} & : & \forall \iota.\, (\mathsf{Bool} \to \mathsf{Nat}^{\iota}) \to \mathsf{Nat}^{\iota} \\
\mathsf{min} & := & \mathsf{fix}_0^{\mu}\, \lambda min \lambda f.\, \mathsf{match}\ \langle f\ \mathsf{true},\ f\ \mathsf{false} \rangle\ \mathsf{with} \\
 & & \quad \langle \mathsf{zero},\ \_ \rangle \quad\quad \mapsto \quad \mathsf{zero} \\
 & & \quad \langle \_,\ \mathsf{zero} \rangle \quad\quad \mapsto \quad \mathsf{zero} \\
 & & \quad \langle \mathsf{succ}\, x',\ \mathsf{succ}\, y' \rangle \quad \mapsto \quad \mathsf{succ}\, (min\, (\lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ x'\ \mathsf{else}\ y'))
\end{array}
$$

Since a $\lambda$-abstraction is a value, this definition of min is not strongly normalizing for similar reasons as the first definition with products. However, in this case, it seems impossible to salvage strong normalization. But if one is only interested in the termination of closed programs, this definition is fine, and indeed, the type system of Hughes, Pareto, and Sabry [HPS96] accepts it.

# Chapter 8

# Conclusion

In this thesis, I have presented a simple but powerful type system which certifies termination and productivity for higher-order functional programs with higher-rank polymorphism and higher-order and heterogeneous data types that can contain both infinitely deep and infinitely branching trees. Its practical applicability has been demonstrated by numerous non-trivial examples. It has been shown that the marriage of types with termination analysis can effortlessly treat the case of higher-order functional programs which are hard for termination analyses based on term orderings. The *types* paradigm has again proven an effective tool to structure program analysis.

Some problems have been left open in this thesis. For one thing, more work has to be done on models of $\mathsf{F}_{\widehat{\omega}}$. The current model proves strong normalization, but it does not give much information about the impredicative encodings of disjoint sum and product. It would be desirable to have a model where the impredicative encodings have semi-continuity properties, but it is unclear to me whether this can be achieved.[1]

Also, the types-as-sets-of-evaluation-contexts model needs refinement. One would like that $t \perp \bigcap_{i\in I} \mathcal{E}_i$ implies $t \perp \mathcal{E}_i$ for some $i \in I$, if the $\mathcal{E}_i$ are sets of evaluation contexts with certain closure properties and form a sequence of a certain form, e.g., a chain or a directed sequence. With that property, one could prove semi-continuity properties for the refined saturation model which was necessary for equi-coinductive types. Again, this is future research.

Finally, I am looking for a model of type-based termination and productivity that is simpler than the ones known to me: (1) types as sets of strongly normalizing terms (this thesis, Barthe et al. [BFG$^+$04], Blanqui [Bla04]) and (2) types as upward closed subsets of a domain with $\perp$-element (Pareto [HPS96, Par00]). It seems funny that in order to reason about terminating and productive functional programs, which never produce an undefined result, one either has to generalize evaluation to symbolic computation and speak about strong

[1] Christine Paulin-Mohring has told me that the impredicative encodings are not adequate—they contain junk.

normalization, or one has to add a $\perp$-element and show that it is never a result of a well-typed program. A promising direction is Xi's work [Xi02] which gives a much simpler call-by-value semantics. However, he cannot handle infinite objects like streams. Maybe coinductive types can be modeled as sets of evaluation contexts, as in this thesis, but on the basis of a simpler operational semantics, e. g., a big-step call-by-name semantics for closed programs.

## 8.1 Related Work

There is an abundant literature on termination. We will only consider a few recent publications. We divide them into two categories: papers which deal with fully automated termination *checking*, and those which presents methods how to (partially) automate termination *proofs*.

### 8.1.1 Termination Proofs

A very general method to define function is by well-founded recursion. Nordström has demonstrated its use in Martin-Löf Type Theory [Nor88]. In the Calculus of Inductive Constructions, this recursion scheme is an instance of primitive recursion over the accessibility predicate, which is the constructive version of well-foundedness and can be defined as an infinitely-branching inductive data type. Balaa and Bertot [BB00] demonstrate how to recover the fixed-point equation from well-founded recursion in Coq.

From a recursive function definition, an induction scheme can be extracted [Wal92, Hut92]. Each function clause corresponds to one case, and each recursive call in this clause to one induction hypothesis available in this case. One way to prove termination of the function is to first establish the induction scheme and use it to show termination. The second part can often be fully mechanized. In the following, we describe some works which follow roughly this technique.

In Higher-Order Logic, as implemented in the Isabelle [Pau90] interactive theorem prover, a fixed-point combinator WFREC for well-founded recursion is definable. Slind [Sli96] wrote the TFL package which translates recursive functions defined by pattern matching into WFREC-expressions, and extracts termination conditions as proof obligations. Special tactics attempt to discharge these obligations with the help of a user-supplied measure. This approach is very flexible and works quite well in practice. Unfortunately, it cannot be used to construct *proofs* by induction, since derivations are not first class objects in HOL and cannot be manipulated by functions (the Curry-Howard isomorphism breaks down).

A similar mechanism is implemented in PVS [ORS92]. With a recursive definition the user has to supply a termination measure and prove the arising termination conditions. PVS has powerful proof automation, hence, discharging the conditions causes in most cases not much trouble.

Bove and Capretta [BC05a] take a recursive definition and generate its inductive domain predicate such that the induction scheme associated to this predicate corresponds to the above describe induction scheme associated with the recursive function. The recursive definition is translated into Type Theory as a partial function which is well-defined on the inductive domain. To show that a function is total one has to prove that all values inhabit the inductive domain. Recently Bove and Capretta have extended their approach to higher-order domains [BC05b].

Bertot [Ber05] uses domain predicates to define corecursive stream filtering functions. He requires the selection predicate to hold infinitely often on the input streams to obtain productive output streams. Since "infinitely often" means at each point "eventually" and he uses an inductive definition of "eventually", he can define the filtering function using recursion inside corecursion. With his technique he manages to define the Sieve of Eratosthenes by corecursion in Coq. Mixed recursion/corecursion is also possible in $\mathsf{F}\widehat{\omega}$ (see Huffman decoding example, Section 3.2.4), but to see that the Sieve of Eratosthenes is productive requires mathematical knowledge, which is beyond our system.

**Ultrametric spaces and Banach's fixed-point theorem.** Buchholz [Buc05] investigates recursive and corecursive definitions as fixed-points of functionals on ultrametric spaces. For instance, streams of natural numbers $\mathcal{S}$ form an ultrametric space with the equivalence relations $\approx_n \subseteq \mathcal{S} \times \mathcal{S}$ for $n \in \mathbb{N}$ defined by

$$\begin{array}{lcl} s \approx_0 s' & \iff & \mathsf{true} \\ s \approx_{n+1} s' & \iff & \mathsf{hd}\, s = \mathsf{hd}\, s' \text{ and } \mathsf{tl}\, s \approx_n \mathsf{tl}\, s'. \end{array}$$

Banach's fixed-point theorem guarantees such definitions to be well-defined if the underlying functional is contractive. Buchholz turns this principle into a system for "type-based termination" by decorating function arrows with moduli $\phi : \mathbb{N} \to \mathbb{N}$. The modulus of a function expresses how well its arguments must be related for its results to be related, or more precisely,

$$f \in A \xrightarrow{\phi} B \iff \forall l \in \mathbb{N} \forall a, a' \in A.\, a \approx_{\phi(l)} a' \implies f\, a \approx_l f\, a' \in B.$$

Then, the tail function $\mathsf{tl}$ can be given type $\mathcal{S} \xrightarrow{+1} \mathcal{S}$ and stream construction $\mathsf{cons} : \mathbb{N} \to \mathcal{S} \xrightarrow{-1} \mathcal{S}$. In terms of Buchholz' systems, $\mathsf{F}\widehat{\omega}$ is restricted to moduli of the form $\pm k$ and uses a fixed ultrametric arising from the approximation of (co)inductive fixed points. I think Buchholz can simulate the simply-typed strictly-positive fragment of $\mathsf{F}\widehat{\omega}$, but I am not sure whether he can define recursive functions over inductive datatypes with a closure ordinal $> \omega$.

A conceptual difference between rewriting in $\mathsf{F}\widehat{\omega}$ and Buchholz' system is that in $\mathsf{F}\widehat{\omega}$, fixed points are unrolled on demand, whereas in Buchholz' system a term is given an amount of "fuel" and each fixed-point unrolling costs one unit. The amount of initial fuel depends on the modulus of the term and on how precise the result of the term should be approximated.

Buchholz does not treat polymorphism.

Gianantonio and Miculan [GM03] generalize ultrametric spaces further to *ordered families of equivalences* (OFE). Basically, there is a collection of equivalence relations $\equiv$ indexed by any well-ordered set (in Buchholz' case, this was always $\mathbb{N}$). Otherwise, their method is very similar to Buchholz'. Their main selling point is that they can treat mixed recursive/corecursive definitions; as the Huffman-decoder examples of Huffman decoding and prime numbers show (see sections 3.2.4 and 3.2.5), this is also possible in $\mathsf{F}_{\widehat{\omega}}$.

Finally, Matthews [Mat99] has explored the use of *converging equivalence relations* (CERs), which are a variant of OFEs to define corecursive objects in Isabelle. His results are similar to Gianantonio and Miculan's.

### 8.1.2 Termination Checking

**Size-change principle.** Lee, Jones, and Ben-Amram [LJBA01] have coined the term *size-change principle* for termination: *A program terminates on all inputs if every infinite call sequence ... would cause an infinite descent in some data values.* Checking size-change termination is in general PSPACE-complete, but with some restrictions it becomes polynomial [Lee02].

Jones and Bohr [JB04] apply *size-change termination* to closed lambda-terms under deterministic call-by-value evaluation. One lambda-expression *calls* another if the evaluation of the first depends on the evaluation of the second. Using an operational semantics with environments (explicit substitutions), they maintain the invariant that the set of subexpressions is not increased under evaluation and call. Since the initial lambda-expression has only a finite number of subexpressions, a call graph can be constructed on these using abstract interpretation. Sereni [Ser04], also with Jones [SJ05], refines the abstract interpretation and refutes the conjecture that the size-change principle certifies termination for all simply-typed closed lambda-terms [Ser05]. This suggests that (simple) typing and size-change are complementary principles.

The size-change termination analysis for closed lambda-expressions cannot directly be extended to open expressions since it hinges on the subexpression-property, which fails for open expressions: Let $\overline{n}$ denote the $n$th Church numeral. Then $\overline{n}\,(\overline{m}\,f)\,x$ has $O(n+m)$ subexpressions, but its evaluation, $f^{nm}\,x$, has $O(nm)$ subexpressions.

In own work with Altenkirch [AA02], simple typing has been combined with a termination criterion based on call graphs which likens the one of Jones et al.In essence, it does not accept functions which swap their parameters in recursive calls, otherwise, it is isomorphic to Jones et al.. The tool `foetus` [Abe98] detects lexicographic termination orderings for simply-typed functional programs and inductive types.

Wahlstedt [Wah04] combines size-change termination with dependent types and first-order inductive datatypes. His normalization proof uses a reducibility semantics and Ramsey's theorem.

**Abstract interpretation and types.** Telford and Turner [TT00, TT97] check termination and productivity of recursive functions by an abstract interpretation. They track size-change on an abstract domain and can handle first-order functional programs quite well. However, type-based termination seems the more promising approach, since then type analysis and size analysis are integrated into one language and can benefit from each other.

Amadio and Coupet-Grimal [ACG98] extend the simply-typed $\lambda$-calculus by *non-nested* coinductive types and describe a type system for well-defined corecursion via a fixed-point combinator. The typing rule for fixed-points is motivated by transfinite induction, but no closure ordinal is provided. Soundness of the system is proven through a model of partial equivalence relations (PER). The PER model validates an equality consisting of $\beta$, $\eta$, and fixed-point unfolding axioms, plus a uniqueness theorem for fixed points. They also give confluent $\beta$ and fixed-point reduction rules, whose termination is shown via a model of reducibility candidates. Their system subsumes Coquand's guard condition [Coq93], but since it does not have size polymorphism, it is outmatched by Barthe et al. [BFG$^+$04] and my system $\lambda^{\mathsf{fix}\mu\nu}$ [Abe03], which in turn are both subsumed by this thesis.

McBride, the principal implementor of Epigram [MM04] seems to follow the slogan *every total program is structurally recursive*, i. e., he tries to unveil hidden structures and defuse functional programs to arrive at components which are all primitive recursive. For instance, as already noted by Turner [Tur95], quick-sort is just a deforested version of tree-sort. Altenkirch, McBride, and McKinna [AMM05] defuse also merge-sort by introducing some kind of balanced tree as intermediate structure. Sized types, as presented in this thesis, can elegantly capture structural recursion, but can do more than that. By expressing size relations between input and output of functions, e. g., for the subtraction function, recursive functions such as Euclidean division that are not structurally recursive in the strict sense are acceptable as strongly normalizing by the type system. While it may be a intellectually satisfying and understanding-heightening enterprise to exhibit deeper mathematical structures in generally recursive programs and turn them into primitive recursive ones, it is certainly desirable to have a type system that accepts as many terminating recursive programs as possible.

Altenkirch has suggested to me to represent type-based termination *within* dependent type theory, since sized types are in some sense "poor mans dependent types". However, this would require the implementation of ordinal notation systems in type theory [CHS97], a non-trivial task. It seems impossible to implement the ordinal $\infty$ of $\mathsf{F}_{\widehat{\omega}}$ in type theory, since it serves a the closure ordinal of *all* inductive definitions of the theory, hence, it should be inaccessible within the theory.

### 8.1.3 Sized Types

The work of Hughes, Pareto and Sabry [HPS96] and Pareto's thesis [Par00], as well as the work of Giménez [Gim98], Frade [Fra03], and Barthe et al. [BFG+04], which are closest to this thesis, have already been related to our approach (e. g., see Section 1.5).

Barthe, Gregoire, and Pastawski [BGP05] have extended System F à la Church with sized inductive inductive types, arriving at System $\mathsf{F}^{\widehat{}}$. It is roughly the Church-version of $\lambda^{\widehat{}}$ with quantification over types. For this system they have implemented a type inference algorithm with computes and generalizes size constraints, and proven its soundness and completeness. Introducing dummy abstractions and applications to model type abstraction and application, $\mathsf{F}^{\widehat{}}$ can be embedded into our calculus $\mathsf{F}^{\widehat{}}_{\omega}$, which is in Curry-style, in a reduction preserving way. Data types in $\mathsf{F}^{\widehat{}}_{\omega}$ can be contravariant, interleaved, heterogeneous, and coinductive, which is not possible for data types in $\mathsf{F}^{\widehat{}}$.

Blanqui [Bla04, Bla05] decorates the inductive types of his Calculus of Algebraic Constructions, which is an extension of the Calculus of Inductive Constructions underlying Coq [BC04], with sizes of the same expressiveness as in $\mathsf{F}^{\widehat{}}_{\omega}$ and $\lambda^{\widehat{}}$. He proves that reduction rules which adhere to the type-based termination criterion are strongly normalizing and presents a constraint-based algorithm for checking sized types. His system is quite powerful, featuring dependent types and reduction rules that go beyond computation ($\beta$). However, our system is not subsumed since he does not feature size polymorphism or, since his positivity condition on data types is syntactical and not kind-based as in our approach, interesting heterogeneous data types.

Chin and Khoo [CK01], also with Xu [CKX01], extend the approach of Hughes, Pareto, and Sabry on sized types. They describe an algorithm how to infer lower and upper size bounds for a strict functional language with algebraic data types. Sizes are constrained by Presburger arithmetic formulas. They do not treat sizes $> \omega$ that arise for data types with embedded function spaces.

Zenger [Zen98], introduces *indexed types*, which are a shallow form of dependent types, in the sense that they can express data invariants over decidable constraint domains, but cannot type more programs than a non-dependent language like Haskell. Zenger's approach is similar to the one by Chin and Khoo. He cannot certify termination with his type system.

Portillo, Hammond, Loidl, and Vasconcelos [PHLV02] describe a *cost inference* algorithm for higher-order and polymorphic, but non-recursive functional programs. To estimated computation costs, they also need sized types. They treat the special case of sized natural numbers and sized lists, but their size language is more expressive than ours, featuring sum, difference, product, and maximum. Polymorphic combinators such as folds are problematic in their approach because all size information is lost. They could benefit from our idea to make size expressions first-class citizens in the type language, then list folds could be assigned a more precise type (see Section 1.5). Their approach has been extended to recursive functions by Vasconcelos and Hammond [VH04].

**Rich size languages.** Crary and Weirich [CW99] present a intermediate language, LX, for a typed compiler with a rich constructor level: Constructors form a strongly normalizing purely functional language on their own, with data structures and primitive recursion. Consequently, besides function kinds, they have product, sum, and inductive kinds, with a natural number kind as a special case. Data structures on the level of programs can be connected to their abstraction on the level of constructors, since recursive types can be parameterized by shapes, which are elements of inductive kinds. For example, a type of trees parameterized by a perfectly balanced constructor-level tree of size $n$ contains only perfectly balanced object-level trees of size $n$. This way, structure invariants of data types can be represented in LX. To certify execution time bounds, the type system of LX is extended by a virtual clock [CW00]. The new language, LXres, can determine primitive-recursive execution costs, which are represented as constructors of kind `Nat`. To specify cost-functions, Crary and Weirich use a variant of Mendler-style primitive recursion (see Section 4.4) on the constructor level. It is not unthinkable to transfer our approach to terminating recursion to the constructor level of LXres. Our whole development would then occur one level higher; we would require sized inductive kinds, subkinding, and a *sort* of ordinals, sort then being the fourth syntactic level (objects, constructors, kinds, sorts).

Hongwei Xi [Xi02] presents a framework for type-based termination within DML, recently extended to ATS, which are a form of dependently typed programming languages, only that the program and type level expressions are syntactically separated (as in LX), but put into relation through singleton types. His sizes for types can be exact (as opposed to our system, where they are only upper bounds), and he allows termination on measures which are lexicographic products of natural numbers. He can treat many practical first- and higher-order functional programs. However, infinitely branching trees and coinductive structures are not covered by his approach.

# Appendix A

# Summary of $\mathsf{F}\widehat{\omega}$

In the following, we summarize the syntactic rules of $\mathsf{F}\widehat{\omega}$.

## A.1 Kinds and Constructors

**Polarities.** Order given by $\circ \leq p$ and $p \leq p$.

$$\begin{array}{rcll} p & ::= & + & \text{covariant} \\ & | & - & \text{contravariant} \\ & | & \circ & \text{non-variant} \end{array}$$

**Kinds.** Pure kinds $\kappa_*$ do not mention $\mathsf{ord}$.

$$\begin{array}{rcll} \kappa & ::= & * & \text{types} \\ & | & \mathsf{ord} & \text{ordinals} \\ & | & p\kappa_1 \to \kappa_2 & \text{co-/contra-/non-variant constructor transformers} \end{array}$$

**Constructors** are given by the following Curry-style type-level lambda-calculus with some constants.

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

**Signature.** The constructor constants $C$ are taken from a fixed signature $\Sigma$ which contains at least the following constants together with their kinding.

$$\begin{array}{lll} \to & : * \xrightarrow{-} * \xrightarrow{+} * & \text{function space} \\ \forall_\kappa & : (\kappa \xrightarrow{\circ} *) \xrightarrow{+} * & \text{quantification} \\ \mu_{\kappa_*} & : \mathsf{ord} \xrightarrow{+} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_* & \text{inductive constructors} \\ \nu_{\kappa_*} & : \mathsf{ord} \xrightarrow{-} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_* & \text{coinductive constructors} \\ \mathsf{s} & : \mathsf{ord} \xrightarrow{+} \mathsf{ord} & \text{successor of ordinal} \\ \infty & : \mathsf{ord} & \text{infinity ordinal} \end{array}$$

**Polarized contexts.**

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

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

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

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

$$\textsc{kind-c}\ \frac{C : \kappa \in \Sigma}{\Delta \vdash C : \kappa} \qquad \textsc{kind-var}\ \frac{X : p\kappa \in \Delta \qquad p \leq +}{\Delta \vdash X : \kappa}$$

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

**Constructor equality.** Computation axioms.

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

$$\textsc{eq-}\eta\ \frac{\Delta \vdash F : p\kappa \to \kappa'}{\Delta \vdash (\lambda X.\, F\,X) = F : p\kappa \to \kappa'}$$

$$\textsc{eq-}\infty\ \frac{}{\Delta \vdash \mathsf{s}\,\infty = \infty : \mathsf{ord}}$$

Congruences.

$$\textsc{eq-c}\ \frac{C : \kappa \in \Sigma}{\Delta \vdash C = C : \kappa} \qquad \textsc{eq-var}\ \frac{X : p\kappa \in \Delta \qquad p \leq +}{\Delta \vdash X = X : \kappa}$$

$$\textsc{eq-app}\ \frac{\Delta \vdash F = F' : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G = G' : \kappa}{\Delta \vdash F\,G = F'\,G' : \kappa'}$$

$$\textsc{eq-}\lambda\ \frac{\Delta, X : p\kappa \vdash F = F' : \kappa'}{\Delta \vdash \lambda X F = \lambda X F' : p\kappa \to \kappa'}$$

Symmetry and transitivity.

$$\text{EQ-SYM}\ \frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F' = F : \kappa}$$

$$\text{EQ-TRANS}\ \frac{\Delta \vdash F_1 = F_2 : \kappa \qquad \Delta \vdash F_2 = F_3 : \kappa}{\Delta \vdash F_1 = F_3 : \kappa}$$

**Declarative Higher-Order Subtyping.** Reflexivity, transitivity, antisymmetry.

$$\text{LEQ-REFL}\ \frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F \leq F' : \kappa}$$

$$\text{LEQ-TRANS}\ \frac{\Delta \vdash F_1 \leq F_2 : \kappa \qquad \Delta \vdash F_2 \leq F_3 : \kappa}{\Delta \vdash F_1 \leq F_3 : \kappa}$$

$$\text{LEQ-ANTISYM}\ \frac{\Delta \vdash F \leq F' : \kappa \qquad \Delta \vdash F' \leq F : \kappa}{\Delta \vdash F = F' : \kappa}$$

Abstraction and application.

$$\text{LEQ-}\lambda\ \frac{\Delta,\, X{:}p\kappa \vdash F \leq F' : \kappa'}{\Delta \vdash \lambda X F \leq \lambda X F' : p\kappa \to \kappa'}$$

$$\text{LEQ-APP}\ \frac{\Delta \vdash F \leq F' : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G \leq F'\,G : \kappa'}$$

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

$$\text{LEQ-APP}-\ \frac{\Delta \vdash F : -\kappa \to \kappa' \qquad -^{-1}\Delta \vdash G' \leq G : \kappa}{\Delta \vdash F\,G \leq F\,G' : \kappa'}$$

Successor and infinity.

$$\text{LEQ-S-R}\ \frac{\Delta \vdash a : \mathsf{ord}}{\Delta \vdash a \leq \mathsf{s}\,a : \mathsf{ord}} \qquad \text{LEQ-}\infty\ \frac{\Delta \vdash a : \mathsf{ord}}{\Delta \vdash a \leq \infty : \mathsf{ord}}$$

## A.2 Terms, Typing and Reduction

**Terms.**

$$r, s, t \quad ::= \quad x \mid \lambda x t \mid r\,s \mid \mathsf{fix}_n^{\mu} \mid \mathsf{fix}_n^{\nu}$$

### A.2.1 Static Semantics

**Typing contexts.**

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

Wellformed contexts.

$$\textsc{cxt-empty}\ \frac{}{\diamond\ \mathsf{cxt}} \qquad \textsc{cxt-tyvar}\ \frac{\Gamma\ \mathsf{cxt}}{\Gamma, X : \circ\kappa\ \mathsf{cxt}} \qquad \textsc{cxt-var}\ \frac{\Gamma\ \mathsf{cxt} \qquad \Gamma \vdash A : *}{\Gamma, x : A\ \mathsf{cxt}}$$

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

$$\textsc{ty-var}\ \frac{(x : A) \in \Gamma \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash x : A} \qquad \textsc{ty-abs}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x t : A \to B}$$

$$\textsc{ty-app}\ \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B}$$

Quantification.

$$\textsc{ty-gen}\ \frac{\Gamma, X : \kappa \vdash t : F\, X}{\Gamma \vdash t : \forall_\kappa F}\ X \notin \mathsf{FV}(F) \qquad \textsc{ty-inst}\ \frac{\Gamma \vdash t\ :\ \forall_\kappa F \qquad \Gamma \vdash G : \kappa}{\Gamma \vdash t : F\, G}$$

Subsumption.

$$\textsc{ty-sub}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leq B : *}{\Gamma \vdash t : B}$$

Folding and unfolding for (co)inductive types ($\nabla \in \{\mu, \nu\}$).

$$\textsc{ty-fold}\ \frac{\Gamma \vdash t : F\,(\nabla_\kappa\, a\, F)\,\vec{G}}{\Gamma \vdash t : \nabla_\kappa\,(a+1)\,F\,\vec{G}} \qquad \textsc{ty-unfold}\ \frac{\Gamma \vdash r : \nabla_\kappa\,(a+1)\,F\,\vec{G}}{\Gamma \vdash r : F\,(\nabla_\kappa\, a\, F)\,\vec{G}}$$

Recursion ($\nabla = \mu$) and corecursion ($\nabla = \nu$).

$$\textsc{ty-rec}\ \frac{\Gamma \vdash A\ \mathsf{fix}_n^\nabla\text{-}\mathsf{adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}_n^\nabla : (\forall \iota : \mathsf{ord}.\, A\,\iota \to A\,(\iota+1)) \to A\, a}$$

### A.2.2 Admissible Recursion Types

**Notation for size index.** We sometimes write the size index superscript, e. g., $\mu^\iota$ instead of $\mu\,\iota$, or $\nu^\iota$ instead of $\nu\,\iota$.

**Natural transformations.** For constructors $\vec{F}, G : \vec{p\kappa} \to *$, let

$$\vec{F} \Rightarrow G \ :\Longleftrightarrow\ \forall \vec{X} : \vec{\kappa}.\, F_1\,\vec{X} \to \ldots F_n\,\vec{X} \to G\,\vec{X}.$$

Also, we abbreviate $\lambda \vec{X}.\, F(H_1\,\vec{X}) \ldots (H_n\,\vec{X})$ by $F \circ \vec{H}$.

**Admissible types for recursion and corecursion.**

$$\begin{aligned}\Gamma \vdash A\ \mathsf{fix}^{\mu}_{n}\text{-}\mathsf{adm} \quad :\Longleftrightarrow \quad & \Gamma, \iota\!:\!\mathsf{ord} \vdash A\,\iota = (\vec{G}, \mu^{\iota} F \circ \vec{H} \Rightarrow G) : * \quad (\iota \notin \mathsf{FV}(A)) \\ & \text{for some } F, G, \vec{G}, \vec{H} \text{ with } |\vec{G}| = n \text{ and} \\ & \Gamma \vdash F : +\kappa \to \kappa \text{ for some pure } \kappa = \vec{p}\vec{\kappa} \to *, \\ & \Gamma, \iota\!:\!+\mathsf{ord} \vdash G : \kappa' \text{ for some } \kappa' = \circ\vec{\kappa}' \to *, \\ & \Gamma, \iota\!:\!-\mathsf{ord} \vdash G_i : \kappa' \text{ for } 1 \le i \le n, \text{ and} \\ & \Gamma \vdash H_i : \circ\vec{\kappa}' \to \kappa_i \text{ for } 1 \le i \le |\vec{\kappa}|.\end{aligned}$$

$$\begin{aligned}\Gamma \vdash A\ \mathsf{fix}^{\nu}_{n}\text{-}\mathsf{adm} \quad :\Longleftrightarrow \quad & \Gamma, \iota\!:\!\mathsf{ord} \vdash A\,\iota = (\vec{G} \Rightarrow \nu^{\iota} F \circ \vec{H}) : * \quad (\iota \notin \mathsf{FV}(A)) \\ & \text{for some } F, \vec{G}, \vec{H} \text{ with } |\vec{G}| = n \text{ and} \\ & \Gamma \vdash F : +\kappa \to \kappa \text{ for some pure } \kappa = \vec{p}\vec{\kappa} \to *, \\ & \Gamma, \iota\!:\!-\mathsf{ord} \vdash G_i : \kappa' \text{ (all } i\text{) for some } \kappa' = \circ\vec{\kappa}' \to *, \text{ and} \\ & \Gamma \vdash H_i : \circ\vec{\kappa}' \to \kappa_i \text{ for } 1 \le i \le |\vec{\kappa}|.\end{aligned}$$

### A.2.3 Dynamic Semantics

**(Lazy) Values.**

$$\begin{array}{rcll} \mathsf{Val} \ni v & ::= & \lambda x t & \\ & \mid & \mathsf{fix}^{\nabla}_{n}\,\vec{t} & \text{where } |\vec{t}| \le n+1 \end{array}$$

**Evaluation frames.**

$$e(\_) ::= \_\,s \mid \mathsf{fix}^{\mu}_{n}\,s\,t_{1..n}\,\_$$

**Reduction.** $t \longrightarrow t'$

$$\begin{array}{lcll} (\lambda x t)\,s & \longrightarrow_{\beta} & [s/x]t & \\ \mathsf{fix}^{\mu}_{n}\,s\,t_{1..n}\,v & \longrightarrow_{\beta} & s\,(\mathsf{fix}^{\mu}_{n}\,s)\,t_{1..n}\,v & \text{if } v \neq \mathsf{fix}^{\nu}_{n'}\,s'\,t_{1..n'} \\ e(\mathsf{fix}^{\nu}_{n}\,s\,t_{1..n}) & \longrightarrow_{\beta} & e(s\,(\mathsf{fix}^{\nu}_{n}\,s)\,t_{1..n}) & \text{if } e \neq \mathsf{fix}^{\mu}_{n'}\,s'\,t_{1..n'}\,\_ \end{array}$$

plus congruences for all term constructors.

## A.3 Semi-Continuous Types for Recursion

**Positive contexts.**

$$\mathsf{Cxt}^{+} \ni \Pi \quad ::= \quad \diamond \mid \Pi, X\!:\!+\kappa_{*}$$

**Derivation system for semi-continuity.** $\Delta; \Pi \vdash^{\iota q} F : \kappa$ for $q \in \{\oplus, \ominus\}$
Converting ordinary derivations.

$$\textsc{cont-in}\ \frac{\Delta \vdash F : \kappa}{\Delta, \iota : p\mathsf{ord}; \Pi \vdash^{\iota q} F : \kappa} \qquad \textsc{cont-co}\ \frac{\Delta, \iota : +\mathsf{ord} \vdash F : \kappa \qquad p \leq +}{\Delta, \iota : p\mathsf{ord}; \Pi \vdash^{\iota \oplus} F : \kappa}$$

$$\textsc{cont-contra}\ \frac{\Delta, \iota : -\mathsf{ord} \vdash F : \kappa \qquad p \leq -}{\Delta, \iota : p\mathsf{ord}; \Pi \vdash^{\iota \ominus} F : \kappa}$$

$\lambda$-calculus part.

$$\textsc{cont-var}\ \frac{X : p\kappa \in \Delta, \Pi \qquad p \leq +}{\Delta; \Pi \vdash^{\iota q} X : \kappa}$$

$$\textsc{cont-abs}\ \frac{\Delta, X : p\kappa; \Pi \vdash^{\iota q} F : \kappa'}{\Delta; \Pi \vdash^{\iota q} \lambda X F : p\kappa \to \kappa'}\ X \neq \iota$$

$$\textsc{cont-app}\ \frac{\Delta, \iota : p'\mathsf{ord}; \Pi \vdash^{\iota q} F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta, \iota : p'\mathsf{ord}; \Pi \vdash^{\iota q} F\,G : \kappa'}$$

Built-in constructors:

$$\textsc{cont-sum}\ \frac{\Delta; \Pi \vdash^{\iota q} A, B : *}{\Delta; \Pi \vdash^{\iota q} A + B : *} \qquad \textsc{cont-prod}\ \frac{\Delta; \Pi \vdash^{\iota q} A, B : *}{\Delta; \Pi \vdash^{\iota q} A \times B : *}$$

$$\textsc{cont-arr}\ \frac{-\Delta; \diamond \vdash^{\iota \ominus} A : * \qquad \Delta; \Pi \vdash^{\iota \oplus} B : *}{\Delta; \Pi \vdash^{\iota \oplus} A \to B : *}$$

$$\textsc{cont-}\forall\ \frac{\Delta; \Pi \vdash^{\iota \oplus} F : \circ\kappa \to *}{\Delta; \Pi \vdash^{\iota \oplus} \forall_\kappa F : *}$$

$$\textsc{cont-mu}\ \frac{\Delta; \Pi, X : +\kappa_* \vdash^{\iota \ominus} F : \kappa_* \qquad \Delta \vdash^{\iota \ominus} a : \mathsf{ord}}{\Delta; \Pi \vdash^{\iota \ominus} \mu^a X F : \kappa_*}$$

$$\textsc{cont-nu}\ \frac{\Delta; \Pi, X : +\kappa_* \vdash^{\iota \oplus} F : \kappa_* \qquad \Delta \vdash a\ ord}{\Delta; \Pi \vdash^{\iota \oplus} \nu^a X F : \kappa_*}$$

**Improved criterion for admissible types.**

$$\begin{aligned} \Gamma \vdash A\ \mathsf{fix}^{\mu}_n\text{-}\mathsf{adm} \quad &\text{iff } \Gamma, \iota : \circ\mathsf{ord} \vdash A\,\iota = (G_{1..n}, \mu^\iota F \circ \vec{H} \Rightarrow G) : * \\ &\text{and } \Gamma, \iota : \circ\mathsf{ord}; \diamond \vdash^{\iota \oplus} G_{1..n}, \mu^\iota F \circ \vec{H} \Rightarrow G : * \end{aligned}$$

$$\begin{aligned} \Gamma \vdash A\ \mathsf{fix}^{\nu}_n\text{-}\mathsf{adm} \quad &\text{iff } \Gamma, \iota : \circ\mathsf{ord} \vdash A\,\iota = (G_{1..n} \Rightarrow \nu^\iota F \circ \vec{H}) : * \\ &\text{and } \Gamma, \iota : \circ\mathsf{ord}; \diamond \vdash^{\iota \oplus} G_{1..n} \Rightarrow \nu^\iota F \circ \vec{H} : * \end{aligned}$$

# Appendix B

# Iso-Coinductive Constructors

In this section, we present a variant of $\mathsf{F}_{\widehat{\omega}}$ which still has equi-inductive constructors, but *iso-coinductive* constructors, i. e., folding and unfolding coinductive types is no longer silent on the term level. It is clear that the iso-version of a system is strongly normalizing if the equi-version is (see Section 4.1). However, the normalization proof is easier: it can be carried out with the original, more constructive definition of saturated term sets. The main advantage is that saturated sets in the original sense are closed under unions. Of course, now the symmetry between induction and coinduction is somewhat broken. But this has also an advantage: The pathological neutral terms, recursive functions applied to corecursive values, disappear. Now, every neutral term is bound to have a free variable, so no closed term will get stuck.

In the following, we summarize the changes to system definition and soundness proof.

## B.1 Syntax

**Terms.** We extend the language of terms by two new constants:

$$\begin{array}{rcrcll} \mathsf{Const} & \ni & c & ::= & \mathsf{in}^\nu & \text{codata constructor} \\ & & & \mid & \mathsf{out}^\nu & \text{codata destructor} \end{array}$$

**Typing.** The folding and unfolding rules are replaced by the following:

$$\text{TY-FOLD}^\mu\ \frac{\Gamma \vdash t : F\,(\mu_\kappa\, a\, F)\,\vec{G}}{\Gamma \vdash t : \mu_\kappa\,(a+1)\,F\,\vec{G}} \qquad \text{TY-UNFOLD}^\mu\ \frac{\Gamma \vdash r : \mu_\kappa\,(a+1)\,F\,\vec{G}}{\Gamma \vdash r : F\,(\mu_\kappa\, a\, F)\,\vec{G}}$$

$$\text{TY-FOLD}^\nu\ \frac{\Gamma \vdash t : F\,(\nu_\kappa\, a\, F)\,\vec{G}}{\Gamma \vdash \mathsf{in}^\nu\, t : \nu_\kappa\,(a+1)\,F\,\vec{G}} \qquad \text{TY-UNFOLD}^\nu\ \frac{\Gamma \vdash r : \nu_\kappa\,(a+1)\,F\,\vec{G}}{\Gamma \vdash \mathsf{out}^\nu\, r : F\,(\nu_\kappa\, a\, F)\,\vec{G}}$$

The rules for $\lambda$-terms, quantification, subsumption, recursion and corecursion stay in place.

**Example B.1 (Repeat function, revisited)** In the new system, the body of repeat is wrapped into a codata constructor.

$$\begin{array}{rcl} \mathsf{repeat}\ a & := & \mathsf{fix}_0^{\nu}\ \lambda \mathit{repeat}.\ \mathsf{in}^{\nu}\ (\mathsf{pair}\ a\ \mathit{repeat}) \\ \lambda a.\ \mathsf{repeat}\ a & : & \forall A.\ A \to \mathsf{Stream}^{\infty}\ A \end{array}$$

**Evaluation frames.** The codata destructor is a new atomic evaluation context:

$$\begin{array}{rcrcll} \mathsf{Eframe} & \ni & e & ::= & \_\ s & \text{application} \\ & & & \mid & \mathsf{fix}_n^{\mu}\ s\ t_{1..n}\ \_ & \text{recursive function call} \\ & & & \mid & \mathsf{out}^{\nu}\ \_ & \text{codata destruction} \end{array}$$

Evaluation context are compositions of evaluation frames, as before.

**Values.** The new constants give rise to new values. As before, each under-applied constant is a value, but also codata construction $\mathsf{in}^{\nu}\ t$.

$$\begin{array}{rcll} \mathsf{Val} \ni v & ::= & \lambda x t & \\ & \mid & \mathsf{fix}_n^{\nabla} & \\ & \mid & \mathsf{fix}_n^{\nabla}\ s\ \vec{t} & \text{where } 0 \le |\vec{t}| \le n \\ & \mid & \mathsf{out}^{\nu} \mid \mathsf{in}^{\nu} \mid \mathsf{in}^{\nu}\ t & \end{array}$$

**Reduction.** Corecursive functions are only unrolled under a codata destructor. We get the following contractions:

$$\begin{array}{lrcl} \text{RED-}\beta & (\lambda x t)\ s & \rightarrowtail & [s/x]t \\ \text{RED-REC} & \mathsf{fix}_n^{\mu}\ s\ t_{1..n}\ v & \rightarrowtail & s\ (\mathsf{fix}_n^{\mu}\ s)\ t_{1..n}\ v \\ \text{RED-COREC} & \mathsf{out}^{\nu}\ (\mathsf{fix}_n^{\nu}\ s\ t_{1..n}) & \rightarrowtail & \mathsf{out}^{\nu}\ (s\ (\mathsf{fix}_n^{\nu}\ s)\ t_{1..n}) \\ \text{RED-}\beta_{\nu} & \mathsf{out}^{\nu}\ (\mathsf{in}^{\nu}\ r) & \rightarrowtail & r \end{array}$$

By requiring explicit folding and unfolding for codata, we have syntactically removed the critical application of recursive functions to corecursive values. The term $\mathsf{fix}_n^{\mu}\ s\ t_{1..n}\ (\mathsf{fix}_{n'}^{\nu}\ s'\ t_{1..n'})$ now has only one sensible reduction: unrolling recursion; corecursion requires a destructor to be unrolled.

**Example B.2 (Reduction for repeat)** The codata destructor triggers one unrolling and then vanishes with the freshly created codata constructor.

$$\begin{array}{rcl} \mathsf{out}^{\nu}\ (\mathsf{repeat}\ a) & \longrightarrow & \mathsf{out}^{\nu}\ ((\lambda \mathit{repeat}.\ \mathsf{in}^{\nu}\ (\mathsf{pair}\ a\ \mathit{repeat}))\ (\mathsf{repeat}\ a)) \\ & \longrightarrow & \mathsf{out}^{\nu}\ (\mathsf{in}^{\nu}\ (\mathsf{pair}\ a\ (\mathsf{repeat}\ a)) \\ & \longrightarrow & \mathsf{pair}\ a\ (\mathsf{repeat}\ a) \end{array}$$

## B.2 Soundness

**Safe evaluation contexts.** Codata destruction is an additional safe evaluation context.

$$\text{SF-OUT}\ \frac{}{\mathsf{out}^{\nu}\ \_ \in \mathsf{Sframe}}$$

**Safe weak head reduction** now also accounts for $\beta_\nu$-contractions.

| | | | |
|---|---|---|---|
| REQ-$\beta$ | $(\lambda x t)\, s$ | $\rhd\ [s/x]t$ | if $s \in \mathcal{S}$ |
| REQ-REC | $\mathsf{fix}^\mu_n\, s\, t_{1..n}\, v$ | $\rhd\ s\,(\mathsf{fix}^\mu_n\, s)\, t_{1..n}\, v$ | |
| REQ-COREC | $\mathsf{out}^\nu\,(\mathsf{fix}^\nu_n\, s\, t_{1..n})$ | $\rhd\ \mathsf{out}^\nu\,(s\,(\mathsf{fix}^\nu_n\, s)\, t_{1..n})$ | |
| REQ-$\beta_\nu$ | $\mathsf{out}^\nu\,(\mathsf{in}^\nu\, r)$ | $\rhd\ r$ | |
| REQ-ECXT | $E(t) \rhd E(t')$ | | if $t \rhd t'$ |
| REQ-TRANS | $\rhd$ is transitive | | |

The requirement

$$\text{REQ-FIX}^\mu\text{FIX}^\nu \qquad \mathsf{fix}^\mu_n\, s\, t_{1..n}\,(\mathsf{fix}^\nu_{n'}\, s'\, t'_{1..n'}) \in \mathcal{N} \text{ if } s, \vec{t}, s', \vec{t}' \in \mathcal{S}$$

can be dropped, since REQ-REC now covers also the application of a recursive function to a corecursive value.

We still require

$$\text{REQ-FIX}^\nu \qquad \mathsf{fix}^\nu_n \in \mathcal{S}^{n+1} \boxed{\longrightarrow} \mathcal{S}.$$

**Saturated sets.** We revert to our old definition: A set $\mathcal{A}$ is *saturated*, $\mathcal{A} \in \mathsf{SAT}$, if $\mathcal{N} \subseteq \mathcal{A} \subseteq \mathcal{S}$ and $\mathcal{A}$ is closed under $\rhd$-reduction and -expansion.

**Lemma B.3** *If $\mathcal{A}$ is saturated, then also $\mathcal{A}' := \{r \mid \mathsf{out}^\nu\, r \in \mathcal{A}\}$.*

*Proof.* To show $\mathcal{N} \subseteq \mathcal{A}'$, we require $\mathsf{out}^\nu(\mathcal{N}) \subseteq \mathcal{A}$. Since $\mathcal{N} \subseteq \mathcal{A}$, by assumption, we conclude with REQ-STRICT. The next goal, $\mathcal{A}' \subseteq \mathcal{S}$ follows from the new requirement

$$\text{REQ-OUT}^\nu \qquad \mathsf{out}^\nu\, r \in \mathcal{S} \text{ implies } r \in \mathcal{S}.$$

Finally, we need to show that $\mathcal{A}'$ is closed under $\rhd$-reduction and -expansion. This property is inherited from $\mathcal{A}$ by REQ-ECXT. □

**Lattice of saturated sets.** SAT forms a complete lattice under the set-theoretic intersection *and union*. That we now have $\sup \mathfrak{A} = \boxed{\bigcup}\mathfrak{A} = \bigcup \mathfrak{A}$ for $\mathfrak{A} \subseteq \mathsf{SAT}$, is the main gain of this development.

**Lemma B.4** *Let $\mathfrak{A} \subseteq \mathsf{SAT}$. Then $\bigcap \mathfrak{A} \in \mathsf{SAT}$ and $\bigcup \mathfrak{A} \in \mathsf{SAT}$.*

*Proof.* If $\mathfrak{A}$ is empty, then $\bigcap \mathfrak{A} = \mathcal{S}$ and $\bigcup \mathfrak{A} = \mathcal{N}$. Otherwise, let $t \in \bigcup \mathfrak{A}$. Then there exists some $\mathcal{A} \in \mathfrak{A}$ such that $t \in \mathcal{A}$. Now if $t \rhd t'$ or $t \lhd t'$, then $t' \in \mathcal{A} \subseteq \bigcup \mathfrak{A}$, hence, $\bigcup \mathfrak{A}$ is closed. If $t \in \bigcap \mathfrak{A}$ then $t \in \mathcal{A}$ for all $\mathcal{A} \in \mathfrak{A}$. Since all $\mathcal{A}$ are closed, each $\rhd$-reduction or -expansion $t'$ of $t$ is in every $\mathcal{A}$, hence, also in $\bigcap \mathfrak{A}$. □

**Semantics of $\mu$ and $\nu$.** Let $\kappa = \vec{p}\vec{\kappa} \to *$ and $|\vec{\mathcal{H}}| = |\vec{\kappa}|$.

$$\begin{array}{lcl} \mathsf{out}^{-1}_{\nu_\kappa} & : & (\llbracket\kappa\rrbracket \xrightarrow{+} \llbracket\kappa\rrbracket) \xrightarrow{+} \llbracket\kappa\rrbracket \xrightarrow{+} \llbracket\kappa\rrbracket \\ \mathsf{out}^{-1}_{\nu_\kappa}(\mathcal{F})(\mathcal{G})(\vec{\mathcal{H}}) & := & \{r \mid \mathsf{out}^\nu\, r \in \mathcal{F}(\mathcal{G})(\vec{\mathcal{H}})\} \\ \\ \mathsf{Sem}(\mu_\kappa)(\alpha)(\mathcal{F}) & := & \mathcal{F}^\alpha(\bot^\kappa) \\ \mathsf{Sem}(\nu_\kappa)(\alpha)(\mathcal{F}) & := & (\mathsf{out}^{-1}_{\nu_\kappa}(\mathcal{F}))^\alpha(\top^\kappa). \end{array}$$

Since $\llbracket * \rrbracket = \mathsf{SAT}$, we have $\llbracket\nu_\kappa\rrbracket = \mathsf{Sem}(\nu_\kappa) \in \llbracket\mathsf{ord}\rrbracket \xrightarrow{-} (\llbracket\kappa\rrbracket \xrightarrow{+} \llbracket\kappa\rrbracket) \xrightarrow{+} \llbracket\kappa\rrbracket$, hence, the semantics of coinductive types is sound.

**Lemma B.5 (Soundness of codata construction)** *Let $\mathcal{A} \in \mathsf{SAT}$. If $t \in \mathcal{A}$ then $\mathsf{in}^\nu\, t \in \{r \mid \mathsf{out}^\nu\, r \in \mathcal{A}\}$.*

*Proof.* $\mathsf{out}^\nu\,(\mathsf{in}^\nu\, t) \rhd t$, and $\mathcal{A}$ is closed under $\rhd$-expansion. □

**Lemma B.6 (Soundness of (un)folding for coinductive constructors)** *Let $\kappa = \vec{p}\vec{\kappa} \to *$, $\mathcal{H}_i \in \llbracket\kappa_i\rrbracket$ $\mathcal{F} \in \llbracket\kappa\rrbracket \xrightarrow{+} \llbracket\kappa\rrbracket$, and $\alpha \in \llbracket\mathsf{ord}\rrbracket$. We set*

$$\begin{array}{lcll} \mathcal{A} & := & \mathcal{F}(\llbracket\nu_\kappa\rrbracket\,\alpha\,\mathcal{F})(\vec{\mathcal{H}}) & \textit{(unfolded)} \\ \mathcal{B} & := & (\llbracket\nu_\kappa\rrbracket(\llbracket\mathsf{s}\rrbracket\alpha)\,\mathcal{F})(\vec{\mathcal{H}}) & \textit{(folded)} \end{array}$$

*Then $\mathsf{in}^\nu(\mathcal{A}) \subseteq \mathcal{B}$ and $\mathsf{out}^\nu(\mathcal{B}) \subseteq \mathcal{A}$.*

*Proof.* If $\alpha < \top^{\mathsf{ord}}$, then

$$\mathcal{B} = \mathsf{out}^{-1}_{\nu_\kappa}(\mathcal{F})(\llbracket\nu_\kappa\rrbracket\alpha\,\mathcal{F})(\vec{\mathcal{H}}),$$

which means that $t \in \mathcal{B} \iff \mathsf{out}^\nu\, t \in \mathcal{A}$. With Lemma B.5, our claim follows. In case $\alpha = \top^{\mathsf{ord}}$ the equation for $\mathcal{B}$ is still valid, since the fixed point is reached at the closure ordinal. □

The soundness of rules TY-FOLD$^\nu$ and TY-UNFOLD$^\nu$ is a consequence of this theorem.

**Admissible semantic types for corecursion.** We need to prove soundness of corecursion again, now for our old definition of SAT. The reasoning power we have lost by reverting to the old definition of SAT is compensated by a new requirement ADM-$\nu$-STEP.

The semantic type family $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ is *admissible for corecursion with n arguments* if the following *four* conditions are met:

| | |
|---|---|
| ADM-$\nu$-SHAPE | $\mathcal{A}(\alpha) = \bigcap_{k\in K}(\mathcal{B}_{1..n}(k,\alpha) \boxrightarrow \mathcal{C}(k,\alpha))$ for some index set $K$ and $\mathcal{B}_{1..n}, \mathcal{C} \in K \times \mathsf{O} \to \mathsf{SAT}$, |
| ADM-$\nu$-START | $\mathcal{S} \subseteq \mathcal{C}(k,0)$ for all $k \in K$, |
| ADM-$\nu$-STEP | $t' \in \mathcal{C}(k,\alpha+1)$ and $\mathsf{out}^\nu\, t \rhd \mathsf{out}^\nu\, t'$ imply $t \in \mathcal{C}(k,\alpha+1)$, |
| ADM-$\nu$-LIMIT | $\inf_{\alpha<\lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for all limits $0 \neq \lambda \in \mathsf{O}$. |

**Lemma B.7 (Corecursion is a function)** *Let $\mathcal{A} \in \mathsf{O} \to \mathsf{SAT}$ be admissible for corecursion with n arguments. If $s \in \mathcal{A}(\alpha) \boxrightarrow \mathcal{A}(\alpha+1)$ for all $\alpha+1 \in \mathsf{O}$, then $\mathsf{fix}_n^\nu\ s \in \mathcal{A}(\beta)$ for all $\beta \in \mathsf{O}$.*

*Proof.* By transfinite induction on $\beta \in \mathsf{O}$. The new proof differs from the old one in the step case.

The limit case is a direct consequence of ADM-$\nu$-LIMIT.

For the remaining cases, using ADM-$\nu$-SHAPE, assume $k \in K$, $t_i \in \mathcal{B}_i(k,\beta)$ for $1 \le i \le n$ and show $v := \mathsf{fix}_n^\nu\, s\, \vec{t} \in \mathcal{C}(k,\beta)$. Note that $s$ and $t_i$ are safe, hence, $v \in \mathcal{S}$ by REQ-FIX$^\nu$. Since $\mathcal{S} \subseteq \mathcal{C}(k,0)$, we are done in case of $\beta = 0$.

For case $\beta = \alpha+1$, we have $s\,(\mathsf{fix}^\nu\ s) \in \mathcal{A}(\alpha+1)$ by induction hypothesis and assumption, hence, $s\,(\mathsf{fix}^\nu\ s)\,\vec{t} \in \mathcal{C}(k,\alpha+1)$. Since $\mathsf{out}^\nu\, v \rhd \mathsf{out}^\nu\,(s\,(\mathsf{fix}^\nu\ s)\,\vec{t})$, by ADM-$\nu$-STEP it follows that $v \in \mathcal{C}(k,\alpha+1)$. □

**Admissible types for corecursion, syntactically.** We recapitulate the criterion on types for corecursion:

$$\begin{aligned}\Gamma \vdash A\ \mathsf{fix}_n^\nu\text{-}\mathsf{adm} \quad :\Longleftrightarrow\quad & \Gamma, \iota\colon \mathsf{ord} \vdash A\,\iota = (\vec{G} \Rightarrow (\nu^\iota F) \circ \vec{H}) : * \quad (\iota \notin \mathsf{FV}(A)) \\ & \text{for some } F, \vec{G}, \vec{H} \text{ with } |\vec{G}| = n \text{ and} \\ & \Gamma \vdash F : +\kappa \to \kappa \text{ for some pure } \kappa = \vec{p}\vec{\kappa} \to *, \\ & \Gamma, \iota\colon -\mathsf{ord} \vdash G_i : \kappa' \text{ (all } i\text{) for some } \kappa' = \circ\vec{\kappa}' \to *, \text{ and} \\ & \Gamma \vdash H_i : \circ\vec{\kappa}' \to \kappa_i \text{ for } 1 \le i \le |\vec{\kappa}|.\end{aligned}$$

**Lemma B.8 (Soundness of admissible corecursion types)** *If $\Gamma \vdash A\ \mathsf{fix}_n^\nu\text{-}\mathsf{adm}$ and $\theta \in [\![\Gamma]\!]$, then $[\![A]\!]_\theta$ is admissible for corecursion with n arguments.*

*Proof.* As in proof of the corresponding lemma 3.46, we set

$$\mathcal{A}(\alpha) = [\![A]\!]_\theta(\alpha) = [\![\vec{G} \Rightarrow (\nu^\iota F) \circ \vec{H}]\!]_{\theta[\iota \mapsto \alpha]}$$

In the following, we verify the conditions ADM-$\nu$-SHAPE and ADM-$\nu$-STEP; ADM-$\nu$-START and ADM-$\nu$-LIMIT are proven as before.

ADM-$\nu$-SHAPE Show $\mathcal{A}(\alpha) = \bigcap_{k\in K} \mathcal{B}_{1..n}(k,\alpha) \boxrightarrow \mathcal{C}(k,\alpha)$. We set

$$\begin{array}{lcll} K & := & [\![\kappa'_1]\!] \times \cdots \times [\![\kappa'_m]\!] & \text{where } m := |\vec{\kappa}'|, \\ \mathcal{B}_i(\vec{\mathcal{X}},\alpha) & := & [\![G_i]\!]_{\theta[\iota\mapsto\alpha]}\vec{\mathcal{X}} & \text{for } 1 \le i \le n, \text{ and} \\ \mathcal{F} & := & \mathsf{out}_{\nu_\kappa}^{-1}([\![F]\!]_\theta) & \\ \mathcal{H}_i(\vec{\mathcal{X}}) & := & [\![H_i]\!]_\theta\vec{\mathcal{X}} & \text{for } 1 \le i \le |\vec{\kappa}| \\ \mathcal{C}(\vec{\mathcal{X}},\alpha) & := & (\mathsf{I}_\alpha\,\mathcal{F}\,\top^\kappa)(\vec{\mathcal{H}}(\vec{\mathcal{X}})). & \end{array}$$

ADM-$\nu$-STEP Assume $t' \in \mathcal{C}(\vec{\mathcal{X}},\alpha+1) = \mathcal{F}(\mathsf{I}_\alpha\,\mathcal{F}\,\top)(\vec{\mathcal{H}}(\vec{\mathcal{X}}))$ and $\mathsf{out}^\nu\, t \rhd \mathsf{out}^\nu\, t'$. By assumption, $\mathsf{out}^\nu\, t' \in [\![F]\!]_\theta(\mathsf{I}_\alpha\,\mathcal{F}\,\top)(\vec{\mathcal{H}}(\vec{\mathcal{X}}))$, and since this set is closed by $\rhd$-expansion, $\mathsf{out}^\nu\, t$ inhabits it as well. But this means that $t \in \mathcal{C}(\vec{\mathcal{X}},\alpha+1)$, as required. □

**Theorem B.9 (Soundness of typing)** *If* $\Gamma \vdash t : A$ *and* $\theta \in [\![\Gamma]\!]$ *then* $(\!|t|\!)_\theta \in [\![A]\!]_\theta$.

*Proof.* By induction on the typing derivation. We do the new cases. For inductive constructors, nothing changes; TY-FOLD$^\mu$ can be handled as TY-FOLD before, and the same holds for unfolding. The cases for coinductive constructors are:

*Case*

$$\text{TY-FOLD}^\nu\ \frac{\Gamma \vdash t : F\,(\nu_\kappa\, a\, F)\,\vec{G}}{\Gamma \vdash \mathsf{in}^\nu\, t : \nu_\kappa\,(a+1)\,F\,\vec{G}}$$

By Lemma 3.5 we have $\Gamma \vdash F\,(\nu_\kappa\, a\, F)\,\vec{G} : *$, which entails $\Gamma \vdash F : +\kappa \to \kappa$ and $\Gamma \vdash a : \mathsf{ord}$, as well as $\Gamma \vdash G_i : \kappa_i$ for $1 \le i \le |\vec{\kappa}|$, if we define $\vec{p}\vec{\kappa} \to * := \kappa$. Hence, $\mathcal{F} := [\![F]\!]_\theta \in [\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]$ and $\alpha := [\![a]\!]_\theta \in [\![\mathsf{ord}]\!]$ and we can conclude by Lemma B.6.

*Case*

$$\text{TY-UNFOLD}^\nu\ \frac{\Gamma \vdash r : \nu_\kappa\,(a+1)\,F\,\vec{G}}{\Gamma \vdash \mathsf{out}^\nu\, r : F\,(\nu_\kappa\, a\, F)\,\vec{G}}$$

Analogously to case TY-FOLD$^\nu$.

□

## B.3 Strong Normalization

**Strong(-ly normalising) head reduction** $t \longrightarrow_{\mathsf{SN}} t'$ is defined inductively by the following rules.

$$\text{SHR-}\beta\ \frac{s \in \mathsf{SN}}{(\lambda x t)\, s \longrightarrow_{\mathsf{SN}} [s/x]t} \qquad \text{SHR-FRAME}\ \frac{r \longrightarrow_{\mathsf{SN}} r'}{e(r) \longrightarrow_{\mathsf{SN}} e(r')}$$

$$\text{SHR-REC}\ \frac{}{\mathsf{fix}^\mu_n\, s\, t_{1..n}\, v \longrightarrow_{\mathsf{SN}} s\,(\mathsf{fix}^\mu_n\, s)\, t_{1..n}\, v}$$

$$\text{SHR-COREC}\ \frac{}{\mathsf{out}^\nu\,(\mathsf{fix}^\nu_n\, s\, t_{1..n}) \longrightarrow_{\mathsf{SN}} \mathsf{out}^\nu\,(s\,(\mathsf{fix}^\nu_n\, s)\, t_{1..n})}$$

$$\text{SHR-}\beta_\nu\ \frac{}{\mathsf{out}^\nu(\mathsf{in}^\nu\, t) \longrightarrow_{\mathsf{SN}} t}$$

It is easy to see that $\longrightarrow_{\mathsf{SN}}$ is deterministic and closed under evaluation contexts, hence we can set its reflexive-transitive extension $\longrightarrow^*_{\mathsf{SN}}$ to be $\rhd$.

**Strongly neutral terms** $r \in \mathsf{SNe}$ are defined inductively by the following rules. The pathological case SNE-FIX$^\mu$FIX$^\nu$ has disappeared.

$$\text{SNE-VAR}\ \frac{}{x \in \mathsf{SNe}} \qquad \text{SNE-FRAME}\ \frac{r \in \mathsf{SNe} \qquad e \in \mathsf{Sframe}}{e(r) \in \mathsf{SNe}}$$

**Strongly normalizing terms** $t \in \mathsf{SN}$. The last three rules account for the new constants.

$$\textsc{sn-sne}\ \frac{r \in \mathsf{SNe}}{r \in \mathsf{SN}} \qquad \textsc{sn-abs}\ \frac{t \in \mathsf{SN}}{\lambda x t \in \mathsf{SN}} \qquad \textsc{sn-fix}\ \frac{\vec{t} \in \mathsf{SN}}{\mathsf{fix}_n^{\nabla}\, \vec{t} \in \mathsf{SN}}\ |\vec{t}| \leq n+1$$

$$\textsc{sn-exp}\ \frac{t \longrightarrow_{\mathsf{SN}} t' \qquad t' \in \mathsf{SN}}{t \in \mathsf{SN}} \qquad \textsc{sn-roll}\ \frac{s\,(\mathsf{fix}_n^{\nu}\, s)\, \vec{t} \in \mathsf{SN}}{\mathsf{fix}_n^{\nu}\, s\, \vec{t} \in \mathsf{SN}}\ |\vec{t}| \leq n$$

$$\textsc{sn-in}\ \frac{t \in \mathsf{SN}}{\mathsf{in}^{\nu}\, t \in \mathsf{SN}} \qquad \textsc{sn-in}\ \frac{}{\mathsf{in}^{\nu} \in \mathsf{SN}} \qquad \textsc{sn-out}\ \frac{}{\mathsf{out}^{\nu} \in \mathsf{SN}}$$

(We have reused the rule name SN-IN in the same sense as we have reused the name SN-FIX.)

It is clear that SN is closed under $\triangleright$-expansion (rule SN-EXP) and -reduction (each head redex must have been introduced by SN-EXP). Hence, we can set $\mathcal{S} = \mathsf{SN}$, and REQ-$\mathcal{S}$-CLOSED is fulfilled. We also see that each term in SN strong head reduces either to a value or to a strongly neutral term. Setting $\mathcal{N} = {}^{\triangleright}\mathsf{SNe}$, the requirement REQ-$\mathcal{S}$-VAL becomes true: each term $t \in \mathcal{S} \setminus \mathcal{N}$ reduces to a value, $t \triangleright v$. Of course $\mathcal{N} \subseteq \mathcal{S}$ (REQ-$\mathcal{N}$-SUB-$\mathcal{S}$) and $e(\mathcal{N}) \subseteq \mathcal{N}$ (REQ-STRICT) are validated immediately as well. The requirement REQ-FIX$^{\nu}$ is an instances of the rules SN-FIX; the requirement REQ-OUT$^{\nu}$ is fulfilled by the following lemma.

**Lemma B.10 (**REQ-OUT$^{\nu}$**)** *If* $\mathcal{D} :: \mathsf{out}^{\nu}\, r \in \mathsf{SN}$ *then* $r \in \mathsf{SN}$.

*Proof.* By induction on $\mathcal{D}$.

*Case*

$$\textsc{sn-sne}\ \frac{\textsc{sne-frame}\dfrac{r \in \mathsf{SN}}{\mathsf{out}^{\nu}\, r \in \mathsf{SN}}}{\mathsf{out}^{\nu}\, r \in \mathsf{SN}}$$

By assumption $r \in \mathsf{SN}$.

*Case*

$$\textsc{sn-exp}\ \frac{\mathsf{out}^{\nu}\,(\mathsf{fix}_n^{\nu}\, s\, t_{1..n}) \longrightarrow_{\mathsf{sn}} \mathsf{out}^{\nu}\,(s\,(\mathsf{fix}_n^{\nu}\, s)\, t_{1..n}) \in \mathsf{SN}}{\mathsf{out}^{\nu}\,(\mathsf{fix}_n^{\nu}\, s\, t_{1..n}) \in \mathsf{SN}}$$

By induction hypothesis, $s\,(\mathsf{fix}_n^{\nu}\, s)\, t_{1..n} \in \mathsf{SN}$. Then by SN-ROLL, $\mathsf{fix}_n^{\nu}\, s\, t_{1..n} \in \mathsf{SN}$.

*Case*

$$\textsc{sn-exp}\ \frac{\mathsf{out}^{\nu}\,(\mathsf{in}^{\nu}\, t) \longrightarrow_{\mathsf{sn}} t \qquad t \in \mathsf{SN}}{\mathsf{out}^{\nu}\,(\mathsf{in}^{\nu}\, t) \in \mathsf{SN}}$$

By SN-IN, $\mathsf{in}^{\nu}\, t \in \mathsf{SN}$. □

**Proof of** REQ-FUN-SAFE $\mathcal{N} \boxed{\to} \mathsf{SN} \subseteq \mathsf{SN}$. It is sufficient to show that $r\,x \in \mathsf{SN}$ implies $r \in \mathsf{SN}$.

The following lemma holds in the new setting since corecursion is no longer unrolled under application.

**Lemma B.11** *If $\mathcal{D} :: r\,x \longrightarrow_{\mathsf{SN}} t$ then either $r = \lambda x t$ or $t = r'\,x$ with $r \longrightarrow_{\mathsf{SN}} r'$.*

*Proof.* By induction on $\mathcal{D}$. □

**Lemma B.12 (Extensionality)** *If $\mathcal{D} :: r\,x \in \mathsf{SN}$ or $\mathcal{D} :: r\,x \in \mathsf{SNe}$ then $r \in \mathsf{SN}$.*

*Proof.* By induction on $\mathcal{D}$.

*Case*

$$\textsc{sne-frame}\ \frac{r \in \mathsf{SNe}}{r\,x \in \mathsf{SNe}}$$

By SN-SNE, $r \in \mathsf{SN}$.

*Case* Let $|\vec{t}| = n$.

$$\textsc{sne-frame}\ \frac{s, \vec{t} \in \mathsf{SN}}{\mathsf{fix}^{\mu}_{n}\,s\,\vec{t}\,x \in \mathsf{SNe}}$$

By SN-FIX, $\mathsf{fix}^{\mu}_{n}\,s\,\vec{t} \in \mathsf{SN}$.

*Case*

$$\textsc{sn-sne}\ \frac{r\,x \in \mathsf{SNe}}{r\,x \in \mathsf{SN}}$$

By induction hypothesis.

*Case*

$$\textsc{sn-fix}\ \frac{\vec{t} \in \mathsf{SN}}{\mathsf{fix}^{\nabla}_{n}\,\vec{t}\,x \in \mathsf{SN}}\ |\vec{t}| \leq n$$

Then $\mathsf{fix}^{\nabla}_{n}\,\vec{t} \in \mathsf{SN}$ by SN-FIX.

*Case*

$$\textsc{sn-roll}\ \frac{s\,(\mathsf{fix}^{\nu}_{n}\,s)\,\vec{t}\,x \in \mathsf{SN}}{\mathsf{fix}^{\nu}_{n}\,s\,\vec{t}\,x \in \mathsf{SN}}\ |\vec{t}| < n$$

By induction hypothesis $s\,(\mathsf{fix}^{\nu}_{n}\,s)\,\vec{t} \in \mathsf{SN}$, hence, $\mathsf{fix}^{\nu}_{n}\,s\,\vec{t} \in \mathsf{SN}$ by SN-ROLL.

*Case*

$$\textsc{sn-roll}\ \frac{x\,(\mathsf{fix}^{\nu}_{n}\,x) \in \mathsf{SN}}{\mathsf{fix}^{\nu}_{n}\,x \in \mathsf{SN}}$$

Then $\mathsf{fix}^{\nu}_{n} \in \mathsf{SN}$ by SN-FIX.

*Case*

$$\text{SN-EXP}\ \frac{r\,x \longrightarrow_{\mathsf{SN}} t \qquad t \in \mathsf{SN}}{r\,x \in \mathsf{SN}}$$

By Lemma B.11, we can distinguish two cases. If $r = \lambda x t$, then $r \in \mathsf{SN}$ by SN-ABS. Otherwise, $t = r'\,x$ and $r \longrightarrow_{\mathsf{SN}} r'$. By induction hypothesis, $r' \in \mathsf{SN}$, hence, $r \in \mathsf{SN}$ by SN-EXP.

*Case*

$$\text{SN-IN}\ \frac{x \in \mathsf{SN}}{\mathsf{in}^{\nu}\,x \in \mathsf{SN}}$$

By SN-IN we have $\mathsf{in}^{\nu} \in \mathsf{SN}$. □

**Corollary B.13 (**REQ-FUN-SAFE**)** $\mathcal{N} \boxed{\rightarrow} \mathsf{SN} \subseteq \mathsf{SN}$.

*Proof.* $\{r \mid r\,s \in \mathsf{SN} \text{ for all } s \in \mathcal{N}\} \subseteq \{r \mid r\,x \in \mathsf{SN}\} \subseteq \mathsf{SN}$ by the lemma. □

Soundness for the inductive characterization, $\mathsf{SN} \subseteq \mathsf{sn}$, can be proven as before.

# Appendix C

# Galois Connections

We follow Vouillon [Vou04].

**Definition C.1 (Galois connection)** Let $(X, \leq_X)$ and $(Y, \leq_Y)$ be partially ordered sets. A pair of functions $(f, g) \in (X \to Y) \times (Y \to X)$ is called a *Galois connection* between $X$ and $Y$ iff

$$f(x) \leq_Y y \iff x \leq_X g(y).$$

**Lemma C.2** *The pair $(f, g)$ is a Galois connection iff*

1. *$f \circ g$ is contractive, $g \circ f$ is extensive, and*
2. *$f$ and $g$ are isotone.*

*Proof.* 1. Since $g(y) \leq g(y)$, we have $f(g(y)) \leq y$. Likewise, $x \leq g(f(x))$ follows from $f(x) \leq f(x)$. 2. The assumption $x \leq x'$ implies $x \leq g(f(x')$ since $g \circ f$ is extensive, hence $f(x) \leq f(x')$. Isotonicity of $g$ is proven analogously.

For the converse direction, first assume $f(x) \leq y$. This entails $x \leq g(f(x)) \leq g(y)$. The reverse implication is equally trivial. □

**Corollary C.3**

1. *Both $f \circ g \circ f = f$ and $g \circ f \circ g = g$.*
2. *$f \circ g$ and $g \circ f$ are idempotent.*

**Definition C.4 (Closure operator)** A function $x \mapsto \overline{x}$ is a *closure operator* on $(X, \leq)$ if it is isotone, extensive, and idempotent.

**Corollary C.5** *If $(f, g)$ is a Galois connection between $X$ and $Y$, then $g \circ f$ is a closure operator on $X$.*

**Lemma C.6 (Polarity)** *Let $R \subseteq X \times Y$ be a relation and*

$$\begin{array}{lcl} f & \in & \mathcal{P}(X) \to \mathcal{P}(Y) \\ f(A) & := & A^R := \{y \in Y \mid x\ R\ y \text{ for all } x \in A\} \end{array}$$

$$\begin{array}{lcl} g & \in & \mathcal{P}(Y) \to \mathcal{P}(X) \\ g(B) & := & B^R := \{x \in X \mid x\ R\ y \text{ for all } y \in B\} \end{array}$$

*Then $(f, g)$ is a Galois connection between $(\mathcal{P}(X), \subseteq)$ and $(\mathcal{P}(Y), \supseteq)$, called the* polarity.

*Proof.*

$$\begin{array}{rcl} f(A) \supseteq B & \iff & (\forall y \in B \forall x \in A.\ x\ R\ y) \\ & \iff & (\forall x \in A \forall y \in B.\ x\ R\ y) \iff A \subseteq g(B) \end{array}$$

□

**Example C.7 (Orthogonality)** Consider a set of safe terms $\mathcal{S} \subseteq \mathsf{Tm}$. The *orthogonality* relation $\perp \subseteq \mathsf{Tm} \times \mathsf{Ecxt}$ between terms and evaluation contexts is given by

$$t \perp E \iff E(t) \in \mathcal{S}.$$

Then the function $\mathcal{A} \mapsto \overline{\mathcal{A}} := \mathcal{A}^{\perp\perp}$ is a closure operator on sets of terms, and for $\mathcal{E} \subseteq \mathsf{Ecxt}$ the set of terms $\mathcal{E}^{\perp}$ is closed.

# Bibliography


[AA00] Andreas Abel and Thorsten Altenkirch. A predicative strong normalisation proof for a $\lambda$-calculus with interleaving inductive types. In T. Coquand, P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proof and Programs, International Workshop, TYPES '99*, volume 1956 of *Lecture Notes in Computer Science*, pages 21–40. Springer-Verlag, 2000.

[AA02] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.

[Abe98] Andreas Abel. foetus – termination checker for simple functional programs. Programming Lab Report, 1998.

[Abe99] Andreas Abel. A semantic analysis of structural recursion. Master's thesis, Ludwig-Maximilians-University Munich, 1999.

[Abe00] Andreas Abel. Specification and verification of a formal system for structurally recursive functions. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proof and Programs, International Workshop, TYPES '99*, volume 1956 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2000.

[Abe03] Andreas Abel. Termination and guardedness checking with continuous types. In M. Hofmann, editor, *Typed Lambda Calculi and Applications (TLCA 2003), Valencia, Spain*, volume 2701 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, June 2003.

[Abe04] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS'03).

[Abe05] Andreas Abel. Termination and productivity checking with continuous types—second thoughts. http://www.tcs.ifi.lmu.de/~abel/errata-tlca03.pdf, January 2005.

[AC05] Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*, pages 23–38. Springer-Verlag, April 2005.

[ACG98] Roberto M. Amadio and Solange Coupet-Grimal. Analysis of a guard condition in type theory. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structures, First International Conference, FoSSaCS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 48–62. Springer-Verlag, 1998.

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[AJ05] Klaus Aehlig and Felix Joachimski. Continuous normalization for the lambda-calculus and Gödel's T. *Annals of Pure and Applied Logic*, 133:39–71, 2005.

[Alt93] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, November 1993.

[Alt99] Thorsten Altenkirch. Logical relations and inductive/coinductive types. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings*, Lecture Notes in Computer Science, pages 343–354. Springer-Verlag, 1999.

[Alt01] Thorsten Altenkirch. Representations of first order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 8 – 21. Springer-Verlag, 2001.

[AM03] Andreas Abel and Ralph Matthes. (Co-)iteration for higher-order nested datatypes. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 1–20, Berg en Dal, The Netherlands, 2003. Springer-Verlag.

[AM04] Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, CSL'04*, volume 3210 of *Lecture Notes in Computer Science*, pages 190–204. Springer-Verlag, 2004.

[AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[AMU03] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Generalized iteration and coiteration for higher-order nested datatypes. In A. Gordon, editor, *Foundations of Software Science and Computation Structures (FoSSaCS 2003)*, volume 2620 of *Lecture Notes in Computer Science*, pages 54–69, Warsaw, Poland, April 2003. Springer-Verlag.

[AMU05] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1–2):3–66, 2005.

[AR99] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer-Verlag, 1999.

[Bar99] Bruno Barras. *Auto-validation d'un sysème de preuves avec familles inductives*. PhD thesis, Universit'e Paris 7, 1999.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed $\lambda$-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BB00] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, May 2004.

[BC05a] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[BC05b] Ana Bove and Venanzio Capretta. Recursive functions with higher order domains. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*, pages 116–130. Springer-Verlag, 2005.

[Bee04] Michael Beeson. Lambda logic. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning, 2nd International Joint Conference, IJCAR 2004*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 440–474, Cork, Ireland, July 2004. Springer-Verlag.

[Ber05] Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*, pages 102–115. Springer-Verlag, 2005.

[BFG+04] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):1–45, 2004.

[BGJ00] Richard Bird, Jeremy Gibbons, and Geraint Jones. Program optimisation, naturally. In *Millenial Perspectives in Computer Science*, Palgrave, 2000.

[BGP05] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for type-based termination in a polymorphic setting. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*, pages 71–85. Springer-Verlag, 2005.

[BJO01] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive data type systems. *Theoretical Computer Science*, 277, 2001.

[Bla04] Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3 – 5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 24–39. Springer-Verlag, 2004.

[Bla05] Frédéric Blanqui. Decidability of type-checking in the Calculus of Algebraic Constructions with size annotations. In C.-H. Luke Ong, editor, *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3634 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 2005.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 1998.

[BP99a] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[BP99b] Richard S. Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[Buc91] Wilfried Buchholz. Notation systems for infinite derivations. *Archive of Mathematical Logic*, 30:277–296, 1991.

[Buc05] Wilfried Buchholz. A term calculus for (co-)recursive definitions on streamlike data-structures. *Annals of Pure and Applied Logic*, 136(1–2):75–90, 2005.

[Bur69] Rod Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[Cap05] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 2005. To appear.

[CC99] Catarina Coquand and Thierry Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages (LFM'99)*, Paris, France, September 1999.

[CHS97] Thierry Coquand, Peter Hancock, and Anton Setzer. Ordinals in type theory. Slides, August 1997. Aarhus.

[CK01] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2–3):261–300, 2001.

[CKX01] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. Higher-order polymorphic sized types for safety checks. In *The Second Asian Workshop on Programming Languages and Systems, APLAS'01, Korea Advanced Institute of Science and Technology, Daejeon, Korea, December 17-18, 2001, Proceedings*, pages 117–131, 2001.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In Bengt Nordström, Kent Pettersson, and Gordon Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, pages 71–83, 1992.

[Coq93] Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs (TYPES '93)*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France*, volume 34 of *SIGPLAN Notices*, pages 233–248. ACM Press, 1999.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–198, Boston, Massachusetts, USA, January 2000.

[Dan99] Norman Danner. Transfinite iteration functionals and ordinal arithmetic. arXiv.org e-Print archive, August 1999.

[DC99] Dominic Duggan and Adriana Compagnoni. Subtyping for object type constructors, January 1999. Presented at FOOL 6.

[Dyb94] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[Far04] William M. Farmer. Formalizing undefinedness arising in calculus. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning, 2nd International Joint Conference, IJCAR 2004*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 475–489, Cork, Ireland, July 2004. Springer-Verlag.

[Fra03] Maria João Frade. *Type-Based Termination of Recursive Definitions and Constructor Subtyping in Typed Lambda Calculi*. PhD thesis, Universidade do Minho, Departamento de Informática, 2003.

[Geu92] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In Bengt Nordström, Kent Pettersson, and Gordon Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, pages 193–217, 1992.

[Gim95] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs, International Workshop TYPES´94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *LNCS*, pages 39–59. Springer, 1995.

[Gim98] Eduardo Giménez. Structural recursive definitions in type theory. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1998.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII, 1972.

[Gir01] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[GM03] Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Gog94] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, August 1994. Available as LFCS Report ECS-LFCS-94-304.

[Gog95] Healfdene Goguen. Typed operational semantics. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculi and Applications (TLCA 1995)*, volume 902 of *Lecture Notes in Computer Science*, pages 186–200. Springer-Verlag, 1995.

[Gog99] Healfdene Goguen. Soundness of the logical framework for its typed operational semantics. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, TLCA 1999*, volume 1581 of *Lecture Notes in Computer Science*, L'Aquila, Italy, 1999. Springer-Verlag.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993.

[Hin98] Ralf Hinze. Numerical representations as higher-order nested datatypes. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn, December 1998.

[Hin99] Ralf Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics & Theoretical Computer Science*, 3(4):193–214, 1999.

[Hin00a] Ralf Hinze. Efficient generalized folds. In Johan Jeuring, editor, *Proceedings of the Second Workshop on Generic Programming, WGP 2000*, Ponte de Lima, Portugal, July 2000.

[Hin00b] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, July 2000.

[Hin01] Ralf Hinze. Manufacturing datatypes. *Journal of Functional Programming*, 11(5):493–524, 2001.

[Hin02] Ralf Hinze. Polytypic values possess polykinded types. *MPC Special Issue, Science of Computer Programming*, 43:129–159, 2002.

[HJL04] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *MPC Special Issue, Science of Computer Programming*, 51:117–151, 2004.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *23rd Symposium on Principles of Programming Languages, POPL'96*, pages 410–423, 1996.

[Huf52] D. Huffman. My famous paper on Huffman trees. In *Proc. IRE*, volume 40, pages 1098–1101, 1952.

[Hut92] Dieter Hutter. Automatisierung der vollständigen Induktion. Oldenbourgh Verlag, 1992.

[JB04] Neil Jones and Nina Bohr. Termination analysis of the untyped $\lambda$-calculus. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3 – 5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 2004.

[JG02] Neil D. Jones and Arne J. Glenstrup. Abstract and conclusions of PLI invited paper: program generation, termination, and binding-time analysis. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 1–1. ACM Press, 2002.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming extension. In *24th Symposium on Principles of Programming Languages, POPL'97, Paris, France*, pages 470–482. ACM Press, 1997.

[JM03] Felix Joachimski and Ralph Matthes. Short proofs of normalization. *Archive of Mathematical Logic*, 42(1):59–87, 2003.

[KTU93] Assaf Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):290–311, 1993.

[Lee02] Chin Soon Lee. Program termination analysis in polynomial time. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 218–235. Springer-Verlag, 2002.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages (POPL'01)*, London, UK, January 2001. ACM Press.

[LS05] Sam Lindley and Ian Stark. Reducibility and $\top\top$-lifting for computation types. In Paweł Urzyczyn, editor, *Typed Lambda Calculi*

*and Applications (TLCA 2005), Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[Luo90] Zhaohui Luo. *ECC: An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[Mat98] Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians-University, May 1998.

[Mat99] John Matthews. Recursive function definition over coinductive types. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99, Nice, France*, volume 1690 of *Lecture Notes in Computer Science*, pages 73–90. Springer-Verlag, 1999.

[Mat00] Ralph Matthes. Characterizing strongly normalizing terms of a calculus with generalized applications via intersection types. In *ITRS 00, ICALP Satellite Workshops 2000*, pages 339–354, 2000.

[Mat01] Ralph Matthes. Monotone inductive and coinductive constructors of rank 2. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 600–614. Springer-Verlag, 2001.

[Mat05] Ralph Matthes. Non-strictly positive fixed-points for classical natural deduction. *APAL*, 2005. To appear.

[McB06] Conor McBride. Type-preserving renaming and substitution. *Journal of Functional Programming*, 2006. Functional Pearl. To appear.

[Men87] Nax P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, N.Y.*, pages 30–36. IEEE Computer Society Press, 1987.

[Men91] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[MG01] Clare Martin and Jeremy Gibbons. On the semantics of nested datatypes. *Information Processing Letters*, 80(5):233–238, December 2001.

[MGB04] Clare Martin, Jeremy Gibbons, and Ian Bayley. Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, 16(1):19–35, 2004.

[Min78] Grigori Mints. Finite investigations of transfinite derivations. *Journal of Soviet Mathematics*, 10:548–596, 1978. Translated from: Zap. Nauchn. Semin. LOMI 49 (1975).

[MM04] Connor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 2004.

[Nor88] Bengt Nordström. Terminating general recursion. *BIT*, 28(3):605–619, 1988.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Par97] Michel Parigot. Proofs of strong normalization for second order classical natural deduction. *Journal of Symbolic Logic*, 62(4):1461–1479, 1997.

[Par00] Lars Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, 2000.

[Pau90] Lawrence Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[PDM89] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical report, Carnegie Mellon University, 1989.

[PHLV02] A.J. Rebon Portillo, K. Hammond, H-W. Loidl, and P. Vasconcelos. Cost analysis using automatic size and time inference. In *IFL'02 International Workshop on the Implementation of Functional Languages Madrid, Spain, September 16-18, 2002*, volume 2670 of *Lecture Notes in Computer Science*, pages 232–247. Springer-Verlag, 2002.

[Pie01] Brigitte Pientka. Termination and reduction checking for higher-order logic programs. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 401–415. Springer-Verlag, 2001.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PL89] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In *TAPSOFT, Vol.2*, volume 352 of *Lecture Notes in Computer Science*, pages 345–359. Springer-Verlag, 1989.

[Pol94] Randy Pollack. *The Theory of LEGO*. PhD thesis, University of Edinburgh, 1994.

[PS97] Benjamin C. Pierce and Martin Steffen. Higher order subtyping. *Theoretical Computer Science*, 176(1,2):235–282, 1997.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.

[RP93] J. C. Reynolds and G. D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 105:1–29, 1993.

[Ruc85] Martin Ruckert. *Church-Rosser Theorem und Normalisierung für Termkalküle mit unendlichen Termen unter Einschluß permutativer Reduktionen*. PhD thesis, Mathematisches Institut der LMU München, 1985.

[Sch98] Helmut Schwichtenberg. Finite notations for infinite terms. *Annals of Pure and Applied Logic*, 94(1-3):201–222, 1998.

[Ser04] Damien Sereni. Size-change termination for higher-order functional programs. Technical report, Oxford University Computing Laboratory, 2004.

[Ser05] Damien Sereni. Simply-typed $\lambda$-calculus and SCT. Unpublished note, 2005.

[SJ05] Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2005.

[Sli96] Konrad Slind. Function definition in higher order logic. In *Proceedings of TPHOLs 96*, volume 1125 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Ste95] Joachim Steinbach. Simplification orderings: History of results. *Fundamenta Informaticae*, 24(1/2):47–87, 1995.

[Ste98] Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Technische Fakultät, Universität Erlangen, 1998.

[SU99] Zdzisław Spławski and Paweł Urzyczyn. Type fixpoints: Iteration vs. recursion. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France*, volume 34 of *SIGPLAN Notices*, pages 102–113. ACM Press, 1999.

[Tai75] William W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Logic Colloquium Boston 1971/72*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer-Verlag, 1975.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TT97] Alastair J. Telford and David A. Turner. Ensuring streams flow. In *Algebraic Methodology and Software Technology (AMAST '97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 509–523. Springer-Verlag, 1997.

[TT00] Alastair J. Telford and David A. Turner. Ensuring termination in ESFP. *Journal of Universal Computer Science*, 6(4):474–488, April 2000. Proceedings of BCTCS 15 (1999).

[Tur95] David Turner. Elementary strong functional programming. In *Programming Languages in Education, First International Symposium*, volume 1022 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[UV99] Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of Computing*, 6(3):343–361, 1999.

[Vau04] Lionel Vaux. A type system with implicit types. English version of his mémoire de maîtrise, June 2004.

[VH04] Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In Philip W. Trinder, Greg Michaelson, and Ricardo Pena, editors, *Implementation of Functional Languages, 15th International Workshop, IFL 2003, Edinburgh, UK, September 8-11, 2003, Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, 2004.

[VM04] Jérôme Vouillon and Paul-André Melliès. Semantic types: A fresh look at the ideal model for types. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 52–63. ACM Press, 2004.

[Vou04] Jérôme Vouillon. Subtyping union types. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, CSL'04*, volume 3210 of *Lecture Notes in Computer Science*, pages 415–429. Springer-Verlag, 2004.

[vRS95] Femke van Raamsdonk and Paula Severi. On normalisation. Technical Report CS-R9545, CWI, 1995.

[vRSSX99] Femke van Raamsdonk, Paula Severi, Morten Heine Sørensen, and Hongwei Xi. Perpetual reductions in lambda calculus. *Information and Computation*, 149(2):173–225, March 1999.

[Wah04] David Wahlstedt. Type theory with first-order data types and size-change termination. Licentiate Thesis, Chalmers University of Technology, September 2004.

[Wal92] Christoph Walther. Computing induction axioms. In *International Conference on Logic Programming and Automated Reasoning – LPAR 92*, volume 624 of *Lecture Notes in Artificial Intelligence*, St. Petersburg, 1992. Springer-Verlag.

[WCPW03] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgements and properties. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2003.

[Wei05] Eric W. Weisstein. Goldbach conjecture. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GoldbachConjecture.html, 2005.

[Xi01] Hongwei Xi. Dependent types for program termination verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, Boston, USA, June 2001.

[Xi02] Hongwei Xi. Dependent types for program termination verification. *Journal of Higher-Order and Symbolic Computation*, 15(1):91–131, October 2002.

[Zen98] Christoph Zenger. *Indizierte Typen*. PhD thesis, Fakultät für Informatik der Universität Kalsruhe, 1998.

# Index