A tutorial on type-based termination

Gilles Barthe[1] Benjamin Grégoire[2] Colin Riba[2]

[1] IMDEA Software, Madrid, Spain
[2] INRIA Sophia-Antipolis, France

Abstract. Type-based termination is a method to enforce termination of recursive definitions through a non-standard type system that introduces a notion of size for inhabitants of inductively defined types. The purpose of this tutorial is to provide a gentle introduction to a polymorphically typed λ-calculus with type-based termination, and to the size inference algorithm which is used to guarantee automatically termination of recursive definitions.

1 Introduction

Functional programming languages advocate the use of mathematically intuitive constructions to develop programs. In particular, functional programming languages feature mechanisms to introduce and manipulate finite datatypes such as lists and trees, and infinite datatypes such as streams and infinite trees. There are two basic ingredients to manipulate elements of a datatype: case analysis, that enables to reason by analysis on the top constructor, and fixpoints, that enable to define functions by recursion and co-recursion. Traditionally, functional programming languages allow for unrestricted fixpoints, which are subsumed by the construction

$$\frac{\Gamma, f : \tau \vdash e \ : \ \tau}{\Gamma \vdash (\mathsf{letrec}_\tau \ f = e) \ : \ \tau}$$

whose computational behavior is given by the reduction rule

$$(\mathsf{letrec}_\tau \ f = e) \quad \to \quad e[f := (\mathsf{letrec}_\tau \ f = e)]$$

Unrestricted use of fixpoints leads to typable expressions that diverge, i.e. that have an infinite reduction sequence. While non-termination is acceptable in functional programming languages, logical systems based on type theory must be terminating in order to guarantee coherence and decidability of equivalence between terms. Thus, logical systems based on type theory seek to restrict the usage of recursive definitions to enforce termination.

A standard means to enforce termination is to abandon the syntax of functional programming languages and to rely instead on combinators, known as recursors. Such recursors allow to define functions of type $d \to \sigma$, where d is an inductive datatype such as natural numbers of lists; more generally, the notion of inductive datatype captures in a type-theoretical setting the notion of least fixpoint of a monotone operator. To guarantee termination, recursors combine case

analysis and structural recursion, and their reduction rules ensure that recursive calls are applied to smaller arguments. Unfortunately, recursors are not intuitive to use. Therefore, proof assistants based on type theory, such as Coq and Agda, tend to rely on an alternative approach, that maintains the syntax of functional programming languages, but imposes instead syntactic conditions that ensure termination. Restrictions concern both the typing rule and the reduction rule. Restrictions for the typing rule impose conditions, both on the type τ and on the expression e, under which recursive definitions are well-formed. Essentially, the type τ must be of the form $d \to \sigma$, where d is an inductive datatype, as for recursors. Then, the expression e must be of the form $\lambda x : d.\ b$ where b can only make recursive calls to f on arguments that are structurally smaller than x. Finally, reductions must be restricted to the case where e is applied to an expression of the form $c\ \boldsymbol{t}$ for some constructor c. While the first and third restrictions are easily enforced, it is difficult to find appropriate criteria that enforce the second restriction. A common means to ensure that recursive calls are performed on smaller arguments is to define a syntactic check on the body b of recursive calls. However, such a syntactic approach is problematic, as shall be explained in the course of this chapter.

Type-based termination is an alternative approach to guarantee strong normalization of typable expressions through the use of a non-standard typing system in which inhabitants of inductive datatypes are given a size, which in turn is used to guarantee termination of recursive definitions. Type-based termination draws its inspiration from the set-theoretic and domain-theoretic semantics of inductive definitions, in which inductive sets are viewed as the upper limit of their approximation. In effect, type-based termination embeds these semantical intuitions into the syntax of the type theory, by letting inductive datatypes carry size annotations, and by restricting the rule for fixpoints

$$\frac{\Gamma, f : d^{\imath} \to \sigma \vdash e\ :\ d^{\widehat{\imath}} \to \sigma}{\Gamma \vdash (\mathsf{letrec}_{d^{\infty} \to \sigma}\ f = e)\ :\ d^{\infty} \to \sigma}$$

where d is an inductive datatype, $\imath$ is an arbitrary (i.e. implicitly quantified universally) size, $d^{\imath}$ denotes the $\imath$-approximation of d, and $d^{\widehat{\imath}}$ denotes the next approximation of d, and d^{∞} denotes the inductive datatype itself. As should appear from the typing rule, termination is enforced naturally by requiring that recursive calls, that correspond to occurrences of f in e, can only be made to smaller elements, as f only takes as arguments elements of type $d^{\imath}$.

Type-based termination benefits from essential characteristics that make it an attractive means to ensure termination of recursive definitions in a typed λ-calculus, both from the point of view of the users and of the designers of the type system. First and foremost, it is intuitive and easy to grasp, since the type system simply captures the idea that a recursive definition terminates whenever the size of arguments decreases in recursive calls. As a consequence, the type system is also predictable (i.e. it is possible to have *a priori* an intuition as to whether a definition is correct) and transparent (i.e. it is possible *a posteriori* to understand why a definition is incorrect) for users, which we view as essen-

tial properties of a formal system. Second, type-based termination is expressive: even for the simplest instance of type-based termination, in which the arithmetic of stages only builds on zero, successor and infinity, type-based termination is sufficiently powerful to encode many typed λ-calculi using syntactic termination criteria, and to provide precise typings for some functions that do not increase the size of their arguments (i.e. for unary functions the size of the result is smaller or equal than the size of the argument). Third, type-based termination is based on a solid theoretical foundation, namely that of approximation, which substantially simplifies in the development of realizability models. As shall be illustrated in Section 3.4, there is a good match between the syntax of the type system and its semantics, which facilitates the interpretation of recursive definitions in the realizability model. Fourth, type-based termination isolates in the design of the type system itself the components that are relevant for termination, i.e. constructors, case analysis, and fixpoint definitions, from the remaining components, whose syntax and typing rules are unaffected. Such a separation makes type-based termination robust to language extensions, and compatible with modular verification and separate compilation.

In summary, type-based termination appears as a suitable approach to guarantee strong normalization of typable terms, which in the near future may well supplant syntactic methods that are currently in use in logical systems based on type theory. On this account, the main objective of this tutorial is to provide a gentle introduction to type-based termination. For pedagogical purposes, we start with a review of mechanisms to introduce recursive definitions in typed λ-calculi, and proceed to define a type system that uses type-based termination. Then, we provide high-level proofs of the essential properties of the type system, in particular of strong normalization and of decidability of type inference; we explain the latter in great length, because of the complexity of the algorithm. We conclude with a brief examination of some possible extensions to our system, and a brief account of related work. For simplicity, we focus on a polymorphically typed λ-calculus, although all of the results that we present in this chapter scale up to dependent types.

2 Computations in polymorphic type systems

This section presents the basic framework of this tutorial and the main problem we want to address: having a convenient way for computing in type systems issued from the Curry-Howard isomorphism, while preserving crucial logical properties such as subject reduction, strong normalization and coherence.

We start in Sect. 2.1 from Girard's System F, with terms *à la* Church, as presented in [12]. This system enjoys strong normalization and coherence, and can encode every inductive datatype and every function provably total in second order Peano arithmetic [12]. However, the algorithmic behavior of System F is unsatisfactory, since basic functions such as the predecessor function on Church's numerals is not implementable in constant time [15], and it is more generally

the case of primitive recursion over all inductive datatypes, at least when the computing relation is β-reduction [16].

Hence, from a computational point of view, it is convenient to add datatypes and recursion to System F, leading to System F^{rec} presented in Sect. 2.2. However, F^{rec} lacks both strong normalization and coherence, because of general recursion and of the ability to define non well-founded datatypes.

As a first step towards a well-behaved system, we introduce in Sect. 2.3 the notion of *inductive datatype*. Then, we recall in Sect. 2.5 a syntactic termination criteria, which allows to retrieve strong normalization and coherence. The limitations of such criteria motivate the use of type-based termination, to be presented in Sect. 3.

2.1 System F

Types. We assume given a set $\mathcal{V}_{\mathcal{T}}$ of type variables. The set $\mathcal{T}$ of types is given by the abstract syntax:

$$\mathcal{T} \ ::= \ \mathcal{V}_{\mathcal{T}} \ \mid \ \mathcal{T} \to \mathcal{T} \ \mid \ \Pi \mathcal{V}_{\mathcal{T}}.\ \mathcal{T}$$

Types are denoted by lower-case Greek letters $\sigma, \tau, \theta, \dots$. Free and bound variables are defined as usual. The capture-avoiding substitution of $\boldsymbol{\tau}$ for $\boldsymbol{X}$ in σ is written $\sigma[\boldsymbol{X} := \boldsymbol{\tau}]$. We let $\mathsf{FV}_{\mathcal{T}}(e)$ be the set of type variable occurring free in τ. A type τ is *closed* if $\mathsf{FV}_{\mathcal{T}}(\tau) = \emptyset$.

Example 2.1.

(i) The type of the polymorphic identity is

$$\Pi X.\ X \to X$$

(ii) It is well-known that inductive datatypes can be coded into System F, see e.g. [12]. For instance, Peano natural numbers can be encoded as *Church's numerals*, whose type is

$$\mathsf{N}_{\mathsf{Ch}} \quad := \quad \Pi X.\ X \to (X \to X) \to X$$

From this type, we can read that Church numerals represent the free structure built from one nullary constructor (which stands for 0), and one unary constructor (which stands for the successor).

(iii) The "false" proposition is

$$\bot \quad := \quad \Pi X.\ X$$

□

Terms and reductions. We assume given a set $\mathcal{V}_{\mathcal{E}} = \{x, y, z, \dots\}$ of *(object) variables*. The set $\mathcal{E}$ of *terms* is given by the abstract syntax:

$$\mathcal{E} \ ::= \ \mathcal{V}_{\mathcal{E}} \ \mid \ \lambda \mathcal{V}_{\mathcal{E}} : \mathcal{T}.\ \mathcal{E} \ \mid \ \Lambda \mathcal{V}_{\mathcal{T}}.\ \mathcal{E} \ \mid \ \mathcal{E}\ \mathcal{E} \ \mid \ \mathcal{E}\ \mathcal{T}$$

Free and bound variables, substitution, etc. are defined as usual. The capture-avoiding substitution of e' for x in e is written $e[x := e']$. We let $\mathsf{FV}_{\mathcal{E}}(e)$ be the set of free term variables occurring in e. We say that e is *closed* when $\mathsf{FV}_{\mathcal{E}}(e) = \emptyset$.

The reduction calculus is given by β-reduction $\to_\beta$, which is defined as the compatible closure of

$$(\lambda x : \tau.\ e)\ e' \quad \succ_\beta \quad e[x := e'] \qquad \text{and} \qquad (\Lambda X.\ e)\ \tau \quad \succ_\beta \quad e[X := \tau]$$

The relation $\to_\beta$ is confluent.

Notation 2.2. *We write $e \to_\beta^n e'$ if there is $k \leq n$ such that*

$$e \quad \underbrace{\to_\beta \quad \ldots \quad \to_\beta}_{k \text{ times}} \quad e'$$

Example 2.3.
(i) The polymorphic identity is $\Lambda X.\ \lambda x : X.\ x$.
(ii) The Church's numerals are terms of the form

$$c_n \quad := \quad \Lambda X.\ \lambda x : X.\ \lambda f : X \to X.\ f^n\, x$$

The numeral c_n encodes the natural number n by computing iterations. Indeed, the expression $c_n\, p\, f$ performs n iterations of f on p:

$$c_n\ p\ f \quad \to_\beta^* \quad \underbrace{f \cdots (f}_{n \text{ times}}\ p) \quad = \quad f^n\ p$$

The constructors of Church's numerals are the terms Z and S defined as:

$$\begin{array}{lcl} \mathsf{Z} & := & \Lambda X.\ \lambda x : X.\ \lambda f : X \to X.\ x \\ \mathsf{S} & := & \lambda n : \mathsf{N_{Ch}}.\ \Lambda X.\ \lambda x : X.\ \lambda f : X \to X.\ f\,(n\, X\, x\, f) \end{array}$$

where $\mathsf{N_{Ch}}$ is the type of Church's numerals defined in Ex. 2.1.(ii). Given $\tau \in \mathcal{T}$, we can code *iteration at type* τ with the term $\mathsf{Iter}_\tau\ u\ v\ n\ :=\ n\ \tau\ u\ v$. For all $n, u, v \in \mathcal{E}$ we have

$$\begin{array}{lcll} \mathsf{Iter}_\tau\ u\ v\ \mathsf{Z} & \to_\beta^2 & (\lambda x : \tau.\ \lambda f : \tau \to \tau.\ x)\ u\ v & \to_\beta^2\ u \\ \mathsf{Iter}_\tau\ u\ v\ (\mathsf{S}\ n) & \to_\beta^2 & (\lambda x : \tau.\ \lambda f : \tau \to \tau.\ f\ (n\ \tau\ x\ f))\ u\ v & \to_\beta^2\ v\ (n\ \tau\ u\ v) \end{array}$$

Hence, $\mathsf{Iter}_\tau\, u\, v\, (\mathsf{S}\, n)$ β-reduces in four steps to $v\, (\mathsf{Iter}_\tau\, u\, v\, n)$. Using this iteration scheme, every function provably total in second order Peano arithmetic can be coded in System F [12]. □

Typing. A context is a map $\Gamma : \mathcal{V}_{\mathcal{E}} \to \mathcal{T}$ of finite domain. Given $x \notin \mathsf{dom}(\Gamma)$ we let $\Gamma, x : \tau$ be the context

$$(\Gamma, x : \tau)(y) \quad =_{\text{def}} \quad \begin{cases} \tau & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

The notation $\Gamma, x : \tau$ always implicitly assumes that $x \notin \mathsf{dom}(\Gamma)$. The typing relation of System F is defined by the rules of Fig. 1.

Example 2.4.
(i) The polymorphic identity of Ex. 2.3.(i) can be given the type of Ex. 2.1.(i):

$$\vdash \Lambda X.\ \lambda x : X.\ x\ :\ \Pi X.\ X \to X$$

(ii) Church numerals can be given the type N_{Ch}:

$$\vdash \Lambda X.\ \lambda x : X.\ \lambda f : X \to X.\ f^n\, x\ :\ \Pi X.\ X \to (X \to X) \to X$$

Moreover, $\mathsf{Z} : \mathsf{N}_{\mathsf{Ch}}$ and $\mathsf{S} : \mathsf{N}_{\mathsf{Ch}} \to \mathsf{N}_{\mathsf{Ch}}$. Iteration can be typed as follows:

$$n : \mathsf{N}_{\mathsf{Ch}}, u : \tau, v : \tau \to \tau \vdash \mathsf{Iter}_\tau\ n\ u\ v\ :\ \tau$$

□

$$\text{(var)}\ \frac{}{\Gamma, x : \sigma \vdash x\ :\ \sigma}$$

$$\text{(abs)}\ \frac{\Gamma, x : \tau \vdash e\ :\ \sigma}{\Gamma \vdash \lambda x : \tau.\ e\ :\ \tau \to \sigma} \qquad \text{(app)}\ \frac{\Gamma \vdash e\ :\ \tau \to \sigma \qquad \Gamma \vdash e'\ :\ \tau}{\Gamma \vdash e\ e'\ :\ \sigma}$$

$$\text{(T-abs)}\ \frac{\Gamma \vdash e\ :\ \sigma}{\Gamma \vdash \Lambda X.\ e\ :\ \Pi X.\ \sigma}\ \text{if } X \notin \Gamma \qquad \text{(T-app)}\ \frac{\Gamma \vdash e\ :\ \Pi X.\ \sigma}{\Gamma \vdash e\,\tau\ :\ \sigma[X := \tau]}$$

Fig. 1. Typing rules of System F

Some important properties. The most important properties of System F are subject reduction, strong normalization, and coherence.

Subject reduction states that types are closed under β-reduction.

Theorem 2.5 (Subject reduction). *If $\Gamma \vdash e\ :\ \tau$ and $e \to_\beta e'$, then also $\Gamma \vdash e'\ :\ \tau$.*

Terms typable in System F enjoy a very strong computational property: they are *strongly normalizing*. A term is strongly normalizing if every reduction sequence starting from it is finite. We can thus define the set SN_β of strongly β-normalizing terms as being the smallest set such that

$$\forall e.\quad (\forall e'.\quad e \to_\beta e' \implies e' \in \mathsf{SN}_\beta) \implies e \in \mathsf{SN}_\beta$$

Strong normalization is also useful for the implementation of the language, because it ensures that every reduction strategy (i.e. every way of reducing a term) is terminating. Strong normalization can be proved using the reducibility technique [12], which is sketched in Sect. 3.4.

Theorem 2.6 (Strong normalization). *If* $\Gamma \vdash e \ : \ \tau$ *then* $e \in \mathsf{SN}_\beta$.

We now discuss some logical properties of System F. It is easy to see that $\Gamma \vdash e \ : \ \bot$ implies $\Gamma \vdash e\,\tau \ : \ \tau$ for all type τ. According to the Curry-Howard propositions-as-types isomorphism, this means that the type $\bot$ is the false proposition: every proposition τ can be deduced from it. Therefore, having $\Gamma \vdash e \ : \ \bot$ means that everything can be deduced from Γ. From a logical perspective, it is crucial to ensure that there is no term of type $\bot$ in the empty context. This property is fortunately satisfied by System F. It can be proved by syntactical reasoning, using subject reduction and strong normalization, but a direct reducibility argument is also possible, see Sect. 3.4.

Theorem 2.7 (Coherence). *There is no term* e *such that* $\vdash e \ : \ \bot$.

2.2 A polymorphic calculus with datatypes and general recursion

It is well-known that System F has limited computational power, e.g. it is not possible to encode in System F a predecessor function that computes in constant time [15]. Therefore, programming languages and proof assistants rely on languages that extend the λ-calculus with new constants and rewrite rules. In this section, we discuss one such extension of System F *à la* Church, as presented in Sect. 2.1. This system, called F^{rec}, consists in adding *datatypes* and general recursion to System F. Before giving the formal definitions, we informally present the system with some examples of datatypes. We then recall some well-known examples showing that System F^{rec} lacks two of the most important properties of System F, namely termination and coherence.

Basic features. In System F^{rec}, we represent the type natural numbers using a special type constant Nat. Furthermore, the language of λ-terms is extended by two constants $\mathsf{o} : \mathsf{Nat}$ and $\mathsf{s} : \mathsf{Nat} \to \mathsf{Nat}$ representing the two constructors of Nat. All this information is gathered in the datatype definition:

$$\textbf{Datatype} \ \ \mathsf{Nat} \ := \ \mathsf{o} : \mathsf{Nat} \ \mid \ \mathsf{s} : \mathsf{Nat} \to \mathsf{Nat}$$

This defines Nat as the least type build from the nullary constructor o and the unary constructor s.

Example 2.8. We can now represent the number n by the term $\mathsf{s}^n\,\mathsf{o}$. □

System F^{rec} provides two ways of computing on datatypes. The first one performs the destruction of constructor-headed terms, and allows to reason by case-analysis, similarly as in functional programming languages. For instance, we can define the predecessor function as follows:

$$\begin{aligned} \mathsf{pred} \ := \ \ \lambda x : \mathsf{Nat}.\ \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\ \mathsf{o} &\Rightarrow \mathsf{o} \\ \mid \mathsf{s} &\Rightarrow \lambda y : \mathsf{Nat}.\ y\ \} \end{aligned}$$

This function is evaluated as follows:

$$\begin{array}{rcl} \mathsf{pred}\,\mathsf{o} & \to & \mathsf{case}_{\mathsf{Nat}}\ \mathsf{o}\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \mid \mathsf{s} \Rightarrow \lambda y : \mathsf{Nat}.\ y\} \quad \to\ \mathsf{o} \\ \mathsf{pred}\,(\mathsf{s}\,n) & \to & \mathsf{case}_{\mathsf{Nat}}\ (\mathsf{s}\,n)\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \mid \mathsf{s} \Rightarrow \lambda y : \mathsf{Nat}.\ y\} \to (\lambda y : \mathsf{Nat}.\ y)\,n \to n \end{array}$$

and is typed using the rule

$$\frac{x : \mathsf{Nat} \vdash x\ :\ \mathsf{Nat} \qquad x : \mathsf{Nat} \vdash \mathsf{o}\ :\ \mathsf{Nat} \qquad x : \mathsf{Nat} \vdash \lambda y : \mathsf{Nat}.\ y\ :\ \mathsf{Nat} \to \mathsf{Nat}}{x : \mathsf{Nat} \vdash \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \mid \mathsf{s} \Rightarrow \lambda y : \mathsf{Nat}.\ y\}\ :\ \mathsf{Nat}}$$

Performing a case analysis over an expression e of type Nat means building an object of a given type, say σ, by reasoning by cases on the constructors of Nat. We therefore must provide a branch e_{o} for the case of o and a branch e_{s} for the case of s. If e evaluates to o, then the case-analysis evaluates to e_{o}, and if e evaluates to $\mathsf{s}\,n$, then the we get $e_{\mathsf{s}}\,n$:

$$\begin{array}{rcl} \mathsf{case}_{\sigma}\ \mathsf{o}\ \mathsf{of}\ \{\mathsf{o} \Rightarrow e_{\mathsf{o}} \mid \mathsf{s} \Rightarrow e_{\mathsf{s}}\} & \to & e_{\mathsf{o}} \\ \mathsf{case}_{\sigma}\ (\mathsf{s}\ n)\ \mathsf{of}\ \{\mathsf{o} \Rightarrow e_{\mathsf{o}} \mid \mathsf{s} \Rightarrow e_{\mathsf{s}}\} & \to & e_{\mathsf{s}}\ n \end{array}$$

Since this case-analysis must evaluate to a term of type σ, we must have $e_{\mathsf{o}} : \sigma$ and $e_{\mathsf{s}} : \mathsf{Nat} \to \sigma$. We therefore arrive at the general rule for case-analysis over natural numbers:

$$(\text{case})\ \frac{\Gamma \vdash e\ :\ \mathsf{Nat} \qquad \Gamma \vdash e_{\mathsf{o}}\ :\ \sigma \qquad \Gamma \vdash e_{\mathsf{s}}\ :\ \mathsf{Nat} \to \sigma}{\Gamma \vdash \mathsf{case}_{\sigma}\ e\ \mathsf{of}\ \{\mathsf{o} \Rightarrow e_{\mathsf{o}} \mid \mathsf{s} \Rightarrow e_{\mathsf{s}}\}\ :\ \sigma}$$

The second computing mechanism of System F^{rec} is *general recursion*. The system is equipped by a general fixpoint operator $(\mathsf{letrec}_{\tau}\ f = e)$, which is typed by the rule

$$(\text{rec})\ \frac{\Gamma, f\ :\ \tau \vdash e\ :\ \tau}{\Gamma \vdash (\mathsf{letrec}_{\tau}\ f = e)\ :\ \tau}$$

and which reduces as follows:

$$(\mathsf{letrec}_{\tau}\ f = e) \quad \to \quad e[f := (\mathsf{letrec}_{\tau}\ f = e)]$$

This allows to encode efficiently primitive recursion over natural numbers.

Example 2.9 (Gödel's System T). In F^{rec}, we can encode primitive recursion on natural numbers as follows:

$$\begin{array}{l} \mathsf{rec} := \Lambda X.\ (\mathsf{letrec}_{\mathsf{Nat}\to X}\ rec = \lambda x : \mathsf{Nat}.\ \lambda u : X.\ \lambda v : \mathsf{Nat} \to X \to X. \\ \qquad \mathsf{case}_X\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow u \\ \qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda y : \mathsf{Nat}.\ v\ y\ (rec\ y\ u\ v)\ \} \\ \qquad) : \quad \Pi X.\ \mathsf{Nat} \to (\mathsf{Nat} \to X \to X) \to X \to X \end{array}$$

Therefore, writing rec_{τ} for the head β-reduct of $\mathsf{rec}\ \tau$, we have the following reductions, which are performed in a constant number of steps:

$$\mathsf{rec}_{\tau}\ \mathsf{o}\ u\ v \quad \to^{6} \quad u \qquad \text{and} \qquad \mathsf{rec}_{\tau}\ (\mathsf{s}\ n)\ u\ v \quad \to^{6} \quad v\ n\ (\mathsf{rec}_{\tau}\ n\ u\ v)$$

□

Functions defined by primitive recursion can also be directly coded in F^{rec}. Take for instance the addition and subtraction on Nat.

Example 2.10 (Addition of two natural numbers).

$$\begin{array}{l}\mathsf{plus} := (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}}\ \mathit{plus} = \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\\ \qquad \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow y\\ \qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ \mathsf{s}\ (\mathit{plus}\ x'\ y)\ \}\\ \qquad) : \quad \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\end{array}$$

□

Example 2.11 (Subtraction of natural numbers).

$$\begin{array}{l}\mathsf{minus} := (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}}\ \mathit{ms} = \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\\ \qquad \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow x\\ \qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ \mathsf{case}_{\mathsf{Nat}}\ y\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow x\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda y' : \mathsf{Nat}.\ \mathit{ms}\ x'\ y'\}\\ \qquad\qquad \}\\ \qquad) : \quad \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\end{array}$$

□

Since general fixpoints are allowed, we can also give definitions where recursive calls are not performed on structurally smaller terms. This is the case of the Euclidean division on natural numbers. We will see in Sect. 3 that this function terminates provably with typed-based termination.

Example 2.12 (Euclidean division). This program for the Euclidean division depends on the function minus. It is not typable in systems with a syntactic guard predicate, as, syntactically, $(\mathsf{minus}\ x'\ y)$ is not properly structurally smaller than x in the program below.

$$\begin{array}{l}\mathsf{div} := (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}}\ \mathit{div} = \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\\ \qquad \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow \mathsf{o}\\ \qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ \mathsf{s}\ (\mathit{div}\ (\mathsf{minus}\ x'\ y)\ y)\ \}\\ \qquad) : \quad \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\end{array}$$

□

Polymorphic datatypes. System F^{rec} features polymorphic datatypes, such as polymorphic lists whose constructors, nil and cons, are typed as follows:

$$\mathsf{nil} : \Pi X.\ \mathsf{List}\ X \qquad\qquad \mathsf{cons} : \Pi X.\ X \to \mathsf{List}\ X \to \mathsf{List}\ X$$

Formally, the datatype of lists is defined as follows:

$$\textbf{Datatype}\ \ \mathsf{List}\ X\ :=\ \mathsf{nil} : \mathsf{List}\ X\ \mid\ \mathsf{cons} : X \to \mathsf{List}\ X \to \mathsf{List}\ X$$

List are eliminated using case-analysis, along a pattern similar to that of natural numbers. The case-analysis of polymorphic lists is performed on one particular instantiation of the datatype:

$$(\text{case})\ \frac{\Gamma \vdash e\ :\ \mathsf{List}\ \tau \qquad \Gamma \vdash e_{\mathsf{nil}}\ :\ \sigma \qquad \Gamma \vdash e_{\mathsf{cons}}\ :\ \tau \to \mathsf{List}\ \tau \to \sigma}{\Gamma \vdash \mathsf{case}_\sigma\ e\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons} \Rightarrow e_{\mathsf{cons}}\}\ :\ \sigma}$$

This means that $\mathsf{nil}\,\tau$ can be the subject of case-analysis, while nil can not. Accordingly, the branches of the case-analysis must be typable with the corresponding instantiation of the polymorphic type: e_{cons} takes an argument of type $\mathsf{List}\,\tau$, but not of type $\Pi X.\ \mathsf{List}\,X$. The reduction rules are similar to that of natural numbers:

$$\begin{aligned} \mathsf{case}_\sigma\ (\mathsf{nil}\ \tau)\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons} \Rightarrow e_{\mathsf{cons}}\} \quad &\to \quad e_{\mathsf{nil}} \\ \mathsf{case}_\sigma\ (\mathsf{cons}\ \tau\ x\ xs)\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons} \Rightarrow e_{\mathsf{cons}}\} \quad &\to \quad e_{\mathsf{cons}}\ x\ xs \end{aligned}$$

Here are two basic functions on lists, namely the concatenation of two lists and the map function.

Example 2.13 (The concatenation of two lists).

$$\begin{aligned} \mathsf{app} := \Lambda X.\ (&\mathsf{letrec}_{\mathsf{List}\,X \to \mathsf{List}\,X \to \mathsf{List}\,X}\ app = \lambda x : \mathsf{List}\ X.\ \lambda y : \mathsf{List}\ X. \\ &\mathsf{case}_{\mathsf{List}\,X}\ x\ \mathsf{of}\ \{\ \mathsf{nil} \Rightarrow y \\ &\qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z : X.\ \lambda x' : \mathsf{List}\ X.\ \ \mathsf{cons}\ X\ z\ (app\ x'\ y) \\ &\qquad\qquad \} \\ &) : \quad \Pi X.\ \mathsf{List}\ X \to \mathsf{List}\ X \to \mathsf{List}\ X \end{aligned}$$

□

Example 2.14 (The map function on a list).

$$\begin{aligned} \mathsf{map} := \Lambda X.\ \Lambda Y.\ \lambda f : X \to Y.\ \ (&\mathsf{letrec}_{\mathsf{List}\,X \to \mathsf{List}\,Y}\ map = \lambda x : \mathsf{List}\,X. \\ &\mathsf{case}_{\mathsf{List}\,Y}\ x\ \mathsf{of}\ \{\ \mathsf{nil} \Rightarrow \mathsf{nil} \\ &\qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z : X.\ \ \lambda x' : \mathsf{List}\,X.\ \mathsf{cons}\ Y\ (f\,z)\ (map\ f\ x') \\ &\qquad\qquad \} \\ &) : \quad \Pi X.\ \Pi Y.\ (X \to Y) \to \mathsf{List}\ X \to \mathsf{List}\ Y \end{aligned}$$

□

We can also define the concatenation of a list of lists. In F^{rec} the polymorphic type of lists of lists is $\Pi X.\ \mathsf{List}\ (\mathsf{List}\ X)$. The concatenation of a list of lists is therefore of type $\Pi X.\ \mathsf{List}\ (\mathsf{List}\ X) \to \mathsf{List}\ X$.

Example 2.15 (The concatenation of a list of lists).

$$\begin{aligned} \mathsf{conc} := \Lambda X.\ (&\mathsf{letrec}_{\mathsf{List}\,(\mathsf{List}\,X) \to \mathsf{List}\,X}\ conc = \lambda x : \mathsf{List}\,(\mathsf{List}\,X). \\ &\mathsf{case}_{\mathsf{List}\,X}\ x\ \mathsf{of}\ \{\ \mathsf{nil} \Rightarrow \mathsf{nil} \\ &\qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z : \mathsf{List}\,X.\ \lambda x' : \mathsf{List}\,(\mathsf{List}\,X).\ \mathsf{app}\ X\ z\ (conc\ x') \\ &\qquad\qquad \} \\ &) : \quad \Pi X.\ \mathsf{List}\ (\mathsf{List}\ X) \to \mathsf{List}\ X \end{aligned}$$

□

An other interesting polymorphic type is that of polymorphic finitely branching trees. These trees are composed of leaves, with one token of information, and of inner nodes, with one token of information and a list of successor subtrees. These two kinds of nodes are represented by the same constructor:

$$\mathsf{node} : \Pi X.\ X \to \mathsf{List}\ (\mathsf{Tree}\ X) \to \mathsf{Tree}\ X$$

For instance, the types of trees of natural numbers is $\mathsf{Tree\ Nat}$, and a leave with token n is represented by $\mathsf{node\ Nat}\ n\ (\mathsf{nil}\ (\mathsf{Tree\ Nat}))$. The important point with this type is that the recursive argument of $\mathsf{node}\,\tau$, which is a list of trees of τ, is not directly of type $\mathsf{Tree}\,\tau$ but of type $\mathsf{List}\,(\mathsf{Tree}\,\tau)$. This allows to encode trees where each node can have a different, but finite, arity.

Like natural numbers and lists, trees are eliminated by case-analysis. Since this type has only one constructor, the scheme of elimination essentially performs the projection of that constructor:

$$(\text{case})\ \frac{\Gamma \vdash e\ :\ \mathsf{List}\ \tau \qquad \Gamma \vdash e_{\mathsf{node}}\ :\ \tau \to \mathsf{List}\ (\mathsf{Tree}\ \tau) \to \sigma}{\Gamma \vdash \mathsf{case}_\sigma\ e\ \mathsf{of}\ \{\mathsf{node} \Rightarrow e_{\mathsf{node}}\}\ :\ \sigma}$$

The reduction rule is as follows:

$$\mathsf{case}_\sigma\ (\mathsf{node}\,\tau\,x\,l)\ \mathsf{of}\ \{\mathsf{node} \Rightarrow e_{\mathsf{node}}\} \quad \to \quad e_{\mathsf{node}}\,x\,l$$

For instance, the first projection is typed as

$$\frac{\Gamma \vdash e\ :\ \mathsf{List}\ \tau \qquad \Gamma \vdash \lambda x : \tau.\ \lambda l : \mathsf{List}\ (\mathsf{Tree}\ \tau).\ x\ :\ \tau \to \mathsf{List}\ (\mathsf{Tree}\ \tau) \to \tau}{\Gamma \vdash \mathsf{case}_\tau\ e\ \mathsf{of}\ \{\mathsf{node} \Rightarrow \lambda x : \tau.\ \lambda l : \mathsf{List}\ (\mathsf{Tree}\ \tau).\ x\}\ :\ \tau}$$

and we have $\mathsf{case}_\tau\ (\mathsf{node}\ \tau\ x\ l)\ \mathsf{of}\ \{\mathsf{node} \Rightarrow \lambda x : \tau.\ \lambda l : \mathsf{List}\ (\mathsf{Tree}\ \tau).\ x\}\ \to^3\ x$.

The following example treats the flattening of finitely branching trees.

Example 2.16 (Flattening of finitely branching trees). This program depends on map, defined in Ex. 2.14 and on conc, defined in Ex. 2.15. Similarly to div, it is not typable in systems with a syntactic guard predicate.

$$\begin{aligned}\mathsf{flatten} := \Lambda X.\ (&\mathsf{letrec}_{\mathsf{Tree}\,X \to \mathsf{List}\,X}\ \mathit{flat} = \lambda t : \mathsf{Tree}\,X.\ \mathsf{case}_{\mathsf{List}\ X}\ t\ \mathsf{of}\ \{\\ &\quad \mathsf{node}\ \Rightarrow\ \lambda x : X.\ \lambda l : \mathsf{List}\ (\mathsf{Tree}\ X).\ \mathsf{cons}\ x\ (\mathsf{conc}\ (\mathsf{map}\ \mathit{flat}\ l))\\ &\quad \}\\ &)\ :\quad \Pi X.\ \mathsf{Tree}\ X \to \mathsf{List}\ X\end{aligned}$$

For readability, we have left the instantiation of polymorphic types implicit at the term level. □

Higher-order datatypes. Up to now, we only have presented first order datatypes, i.e. datatypes whose inhabitants represent particular forms of finitely branching trees.

There can in fact be much more powerful datatypes, representing infinitely branching trees, that we call higher-order datatypes. One of them is the type Ord of *Brouwer ordinals*. It is defined as follows:

$$\textbf{Datatype}\ \ \mathsf{Ord}\ :=\ \mathsf{o} : \mathsf{Ord}\ \mid\ \mathsf{s} : \mathsf{Ord} \to \mathsf{Ord}\ \mid\ \mathsf{lim} : (\mathsf{Nat} \to \mathsf{Ord}) \to \mathsf{Ord}$$

Thanks to the constructors $\mathsf{o} : \mathsf{Ord}$ and $\mathsf{s} : \mathsf{Ord} \to \mathsf{Ord}$, Brouwer ordinals contain natural numbers. This is represented by the canonical injection $\mathsf{inj} : \mathsf{Nat} \to \mathsf{Ord}$ defined as follows:

$$\begin{array}{l}\mathsf{inj} := (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Ord}}\ \mathit{inj} = \lambda x : \mathsf{Nat}.\\ \qquad \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow \mathsf{o}\\ \qquad\qquad\qquad\quad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ \mathsf{s}\ (\mathit{inj}\ x')\}\\ \qquad) :\quad \mathsf{Nat} \to \mathsf{Ord}\end{array}$$

For all $p \in \mathbb{N}$, we have $\mathsf{inj}\ (\mathsf{s}^p\ \mathsf{o}) \ \to\ \mathsf{s}^p\ \mathsf{o}$. Moreover, ordinals also feature the higher-order constructor $\mathsf{lim} : (\mathsf{Nat} \to \mathsf{Ord}) \to \mathsf{Ord}$. The expression $\mathsf{lim}\ f$ represents the supremum of the countable list of ordinals represented by $f : \mathsf{Nat} \to \mathsf{Ord}$. For instance, $\mathsf{lim}\ \mathsf{inj}$ is a term-level representation of the set of natural numbers.

Addition of ordinals can easily be defined in F^{rec}.

Example 2.17 (The addition of two ordinals).

$$\begin{array}{l}\mathsf{add} := (\mathsf{letrec}_{\mathsf{Ord}\to\mathsf{Ord}\to\mathsf{Ord}}\ \mathit{add} = \lambda x : \mathsf{Ord}.\ \lambda y : \mathsf{Ord}.\\ \qquad \mathsf{case}_{\mathsf{Ord}}\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow y\\ \qquad\qquad\qquad\quad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Ord}.\ \mathsf{s}\ (\mathit{add}\ x'\ y)\\ \qquad\qquad\qquad\quad \mid \mathsf{lim} \Rightarrow \lambda f : \mathsf{Nat} \to \mathsf{Ord}.\ \ \mathsf{lim}\ (\lambda z : \mathsf{Nat}.\ \mathit{add}\ (f\ z)\ y)\\ \qquad\qquad\qquad\quad \}\\ \qquad) :\quad \mathsf{Ord} \to \mathsf{Ord} \to \mathsf{Ord}\end{array}$$

The addition of $\mathsf{lim}\ f$ and o is the limit for $n : \mathsf{Nat}$ of the addition of each $f\ n$ and o. For instance, $\mathsf{add}\ (\mathsf{lim}\ \mathsf{inj})\ \mathsf{o}\ \to^*\ \mathsf{lim}\ (\lambda x : \mathsf{Nat}.\ \mathsf{add}\ (\mathsf{inj}\ x)\ \mathsf{o})$. □

Formal definition. Now that we have presented the main features of F^{rec}, we can give its formal definition.

At the type level, F^{rec} extends System F with datatypes, which have names taken in a set $\mathcal{D}$ of datatypes identifiers. Moreover, each datatype has a fixed number of parameters. Hence we assume that each datatype identifier $d \in \mathcal{D}$ comes equipped with an arity $\mathsf{ar}(d)$.

Example 2.18. We have $\mathsf{ar}(\mathsf{Nat}) = 0$ and $\mathsf{ar}(\mathsf{List}) = \mathsf{ar}(\mathsf{Tree}) = 1$. □

Formally, the types of F^{rec} extend that of System F as follows:

$$\mathcal{T}\ ::=\ \ \ldots\ \mid\ \mathcal{D}\ \boldsymbol{\mathcal{T}}$$

where in $\mathcal{D}\ \boldsymbol{\mathcal{T}}$, it is assumed that the length of the vector $\boldsymbol{\mathcal{T}}$ is exactly the arity of the datatype.

We now turn to datatype declarations. Each datatype $d \in \mathcal{D}$ has a fixed set of *constructors* $\mathcal{C}(d)$, and each constructor $c \in \mathcal{C}(d)$ is assigned a closed type of the form

$$\Pi \boldsymbol{X}.\ \theta_1 \to \ldots \to \theta_p \to d\ \boldsymbol{X}$$

Note that the arity condition on d imposes that $\boldsymbol{X}$ has the same length for all $c \in \mathcal{C}(d)$. We let $\mathcal{C} =_{\text{def}} \bigcup\{\mathcal{C}(d) \mid d \in \mathcal{D}\}$. The declaration of a datatype, which gathers its parameters, its constructors and their types, is performed in a *datatype definition* of the form:

$$\textbf{Datatype}\ \ d\ \boldsymbol{X}\ \ :=\ \ c_1 : \sigma_1\ \ \mid\ \ \ldots\ \ \mid\ c_n : \sigma_n$$

where $\mathcal{C}(d) = \{c_1,\ \ldots\ , c_n\}$ and each σ_k is of the form $\boldsymbol{\theta}_k \to d\,\boldsymbol{X}$. Given $k \in \{1, \ldots, k\}$, we write $c_k : \Pi \boldsymbol{X}.\ \boldsymbol{\theta}_k \to d\,\boldsymbol{X}$.

Example 2.19. We review here the datatypes that we have already seen. All these datatypes represent a form of well-founded trees. We call them *inductive*, and come back on this notion in Sect. 2.3. Moreover, we give an example of a non-well founded datatype, noted D.

(i) The inductive datatype of natural number is defined as

$$\textbf{Datatype}\ \ \mathsf{Nat}\ \ :=\ \ \mathsf{o} : \mathsf{Nat}\ \ \mid\ \ \mathsf{s} : \mathsf{Nat} \to \mathsf{Nat}$$

(ii) The inductive datatype of polymorphic lists is defined as

$$\textbf{Datatype}\ \ \mathsf{List}\ X\ \ :=\ \ \mathsf{nil} : \mathsf{List}\ X\ \ \mid\ \ \mathsf{cons} : X \to \mathsf{List}\ X \to \mathsf{List}\ X$$

(iii) The inductive datatype of polymorphic finitely branching trees is

$$\textbf{Datatype}\ \ \mathsf{Tree}\ X\ \ :=\ \ \mathsf{node} : X \to \mathsf{List}\ (\mathsf{Tree}\ X) \to \mathsf{Tree}\ X$$

(iv) The inductive datatype of Brouwer ordinals is defined as

$$\textbf{Datatype}\ \ \mathsf{Ord}\ \ :=\ \ \mathsf{o} : \mathsf{Ord}\ \ \mid\ \ \mathsf{s} : \mathsf{Ord} \to \mathsf{Ord}\ \ \mid\ \ \mathsf{lim} : (\mathsf{Nat} \to \mathsf{Ord}) \to \mathsf{Ord}$$

(v) The following datatype is not well-founded. We will see in Ex. 2.22 that it allows to build a non-terminating term of type $\bot$.

$$\textbf{Datatype}\ \ \mathsf{D}\ \ :=\ \ \mathsf{c} : (\mathsf{D} \to \bot) \to \mathsf{D}$$

□

The terms of System F^{rec} extend those of the Church's style System F with constructors, case-expressions and recursive definitions:

$$\mathcal{E}\ \ ::=\ \ \ldots\ \ \mid\ \ \mathcal{C}\ \ \mid\ \ \mathsf{case}_{\mathcal{T}}\ \mathcal{E}\ \mathsf{of}\ \{\mathcal{C} \Rightarrow \mathcal{E}\}\ \ \mid\ \ (\mathsf{letrec}_{\mathcal{T}}\ \mathcal{V}_{\mathcal{E}} = \mathcal{E})$$

The reduction calculus extends β-reduction with ι-reduction for case analysis and μ-reduction for unfolding recursive definitions. Formally,

– ι-reduction $\to_\iota$ is defined as the compatible closure of

$$\mathsf{case}_\sigma\ (c_i\ \boldsymbol{\tau}\ \boldsymbol{a})\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \quad \succ_\iota \quad e_i\ \boldsymbol{a}$$

– μ-reduction $\to_\mu$ is defined as the compatible closure of

$$(\mathsf{letrec}_\tau\ f = e)\ \succ_\mu\ e[f := (\mathsf{letrec}_\tau\ f = e)]$$

Then, $\beta\iota\mu$-reduction, written $\to_{\beta\iota\mu}$, is $\to_\beta \cup \to_\iota \cup \to_\mu$. The relation $\to_{\beta\iota\mu}$ is confluent.

$$\text{(cons)}\ \frac{}{\Gamma \vdash c_k\ :\ \Pi \boldsymbol{X}.\ \boldsymbol{\theta_k} \to d\,\boldsymbol{X}} \qquad \text{(rec)}\ \frac{\Gamma, f\ :\ \tau \vdash e\ :\ \tau}{\Gamma \vdash (\mathsf{letrec}_\tau\ f = e)\ :\ \tau}$$

$$\text{(case)}\ \frac{\Gamma \vdash e\ :\ d\,\boldsymbol{\tau} \quad \Gamma \vdash e_k\ :\ \boldsymbol{\theta_k}[\boldsymbol{X} := \boldsymbol{\tau}] \to \sigma \quad (1 \le k \le n)}{\Gamma \vdash \mathsf{case}_\sigma\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \cdots \mid c_n \Rightarrow e_n\}\ :\ \sigma}$$

Fig. 2. Typing rules for F^{rec}

The type system is standard. The typing relation $\Gamma \vdash e\ :\ \tau$ extends that of System F with the rules given in Fig. 2, where in the rules for (cons) and (case) it is assumed that $\mathcal{C}(d) = \{c_1, \ldots, c_n\}$, and that the type $\boldsymbol{\theta_k} \to d\,\boldsymbol{X}$ of the constructor c_k is given by the datatype declaration.

System F^{rec} enjoys subject reduction.

Theorem 2.20 (Subject reduction). *If $\Gamma \vdash e\ :\ \tau$ and $e \to_{\beta\iota\mu} e'$, then $\Gamma \vdash e'\ :\ \tau$.*

Non-termination and incoherence. In this paragraph, we show that although convenient for computing, System F^{rec} lacks two of the most important properties of System F, namely termination and coherence. These problems are due to the presence of general recursion and non well-founded datatypes. We recall two independent examples, one involving unrestricted recursion and the other involving the non well-founded datatype D of Ex. 2.19.(v). Both examples provide a non-terminating *incoherent* term, that is, a non-terminating closed term of type $\bot$.

Example 2.21 (Recursion). The typing rule (rec) can be instantiated as follows:

$$\frac{f : \bot \vdash f\ :\ \bot}{\vdash (\mathsf{letrec}_\bot\ f = f)\ :\ \bot}$$

The closed term $(\mathsf{letrec}_\bot\ f = f)$ of type $\bot$ is non-terminating:

$$(\mathsf{letrec}_\bot\ f = f) \quad \to_\mu \quad (\mathsf{letrec}_\bot\ f = f) \quad \to_\mu \quad \ldots$$

□

The second well-known example shows how to write a non-normalizing term using case-analysis on the non well-founded datatype D of Ex. 2.19.(v). Note that it involves no recursion.

Example 2.22 (Non well-founded datatypes [14]). Consider the non well-founded datatype D of Ex. 2.19.(v). Recall that $\mathsf{c} : (\mathsf{D} \to \bot) \to \mathsf{D}$ and let

$$\mathsf{p} \;:=\; \lambda x : \mathsf{D}.\;\; \mathsf{case}_{\mathsf{D}\to\bot}\; x \;\mathsf{of}\; \{\mathsf{c} \Rightarrow \lambda y : \mathsf{D} \to \bot.\; y\}$$

We can derive

$$\dfrac{\dfrac{x : \mathsf{D} \vdash x \;:\; \mathsf{D} \qquad x : \mathsf{D} \vdash \lambda y : \mathsf{D} \to \bot.\; y \;:\; (\mathsf{D} \to \bot) \to (\mathsf{D} \to \bot)}{x : \mathsf{D} \vdash \mathsf{case}_{\mathsf{D}\to\bot}\; x \;\mathsf{of}\; \{\mathsf{c} \Rightarrow \lambda y : \mathsf{D} \to \bot.\; y\} \;:\; \mathsf{D} \to \bot}}{\vdash \lambda x : \mathsf{D}.\;\; \mathsf{case}_{\mathsf{D}\to\bot}\; x \;\mathsf{of}\; \{\mathsf{c} \Rightarrow \lambda y : \mathsf{D} \to \bot.\; y\} \;:\; \mathsf{D} \to (\mathsf{D} \to \bot)}$$

That is $p : \mathsf{D} \to (\mathsf{D} \to \bot)$. Furthermore, let $\omega_{\mathsf{D}} := \lambda x : \mathsf{D}.\; \mathsf{p}\; x\; x : \mathsf{D} \to \bot$. We then have $\mathsf{c}\;\omega_{\mathsf{D}} : \mathsf{D}$, hence $\mathsf{p}\;(\mathsf{c}\;\omega_{\mathsf{D}})\;(\mathsf{c}\;\omega_{\mathsf{D}}) : \bot$. This incoherent term is non-terminating:

$$\mathsf{p}\;(\mathsf{c}\;\omega_{\mathsf{D}})\;(\mathsf{c}\;\omega_{\mathsf{D}}) \;\to^*_{\beta\iota}\; (\lambda x : \mathsf{D}.\; \mathsf{p}\; x\; x)\;(\mathsf{c}\;\omega_{\mathsf{D}}) \;\to_{\beta\iota}\; \mathsf{p}\;(\mathsf{c}\;\omega_{\mathsf{D}})\;(\mathsf{c}\;\omega_{\mathsf{D}}) \;\to_{\beta\iota}\; \ldots$$

□

These two examples show that to achieve termination and coherence, we must restrict the formation of both recursive definitions and datatypes.

2.3 Inductive datatypes

The standard means to rule out pathological cases such as the ones above is to focus on inductive datatypes. Intuitively, inductive datatypes are datatypes that can be constructed as the least fixed point of a monotonic operator. This is formalized using the notion of positivity.

Definition 2.23 (Positivity). *Let σ nocc τ if σ does not occur in τ. The predicate σ pos τ (resp. σ neg τ), stating that all occurrences of σ in τ are positive (resp. negative), is inductively defined in Fig. 3.*

Example 2.24. In the following types, the occurrences of τ are positive and the occurrences of σ are negative:

$$\tau \qquad\qquad \sigma \to \tau \qquad\qquad (\tau \to \sigma) \to \tau$$

In particular, Nat pos Nat, $\mathsf{Tree}\; X$ pos $\mathsf{List}\;(\mathsf{Tree}\; X)$ and Ord pos $(\mathsf{Nat} \to \mathsf{Ord})$, but *not* D pos $(\mathsf{D} \to \bot)$. □

Inductive datatypes are datatypes $d \in \mathcal{D}$ in which d and its parameters occur only positively in the type of their constructors.

Definition 2.25 (Inductive datatypes).

$$\frac{}{\sigma \ \mathsf{pos}\ \sigma}$$

$$\frac{\sigma \ \mathsf{nocc}\ \tau}{\sigma \ \mathsf{pos}\ \tau} \qquad \frac{\sigma \ \mathsf{nocc}\ \tau}{\sigma \ \mathsf{neg}\ \tau}$$

$$\frac{\sigma \ \mathsf{neg}\ \tau_2 \qquad \sigma \ \mathsf{pos}\ \tau_1}{\sigma \ \mathsf{pos}\ \tau_2 \to \tau_1} \qquad \frac{\sigma \ \mathsf{pos}\ \tau_2 \qquad \sigma \ \mathsf{neg}\ \tau_1}{\sigma \ \mathsf{neg}\ \tau_2 \to \tau_1}$$

$$\frac{\sigma \ \mathsf{pos}\ \tau}{\sigma \ \mathsf{pos}\ \Pi X.\ \tau} \qquad \frac{\sigma \ \mathsf{neg}\ \tau}{\sigma \ \mathsf{neg}\ \Pi X.\ \tau}$$

$$\frac{\sigma \ \mathsf{pos}\ \tau_i \qquad (1 \leq i \leq \mathsf{ar}(d))}{\sigma \ \mathsf{pos}\ d\,\boldsymbol{\tau}} \qquad \frac{\sigma \ \mathsf{neg}\ \tau_i \qquad (1 \leq i \leq \mathsf{ar}(d))}{\sigma \ \mathsf{neg}\ d\,\boldsymbol{\tau}}$$

Fig. 3. Positivity and negativity of a type occurrence

(i) An inductive datatype definition *is a datatype declaration*

$$\textbf{Datatype} \ \ d\ \boldsymbol{X} \ := \ c_1 : \sigma_1 \ \mid \ \ldots \ \mid \ c_n : \sigma_n$$

where for all $k \in \{1, \ldots, n\}$, σ_k *is of the form* $\boldsymbol{\theta}_k \to d\ \boldsymbol{X}$ *with* $\boldsymbol{X}\ \mathsf{pos}\ \boldsymbol{\theta}_k$ *and* $d\ \boldsymbol{X}\ \mathsf{pos}\ \boldsymbol{\theta}_k$. *Inductive datatypes definitions are written*

$$\textbf{Inductive} \ \ d\ \boldsymbol{X} \ := \ c_1 : \sigma_1 \ \mid \ \ldots \ \mid \ c_n : \sigma_n$$

(ii) An environment is a sequence of datatype definitions $I_1 \ \ldots I_n$ *in which constructors of the datatype definition* I_k *only use datatypes introduced by* $I_1 \ldots I_k$.

In the remainder of this tutorial, we implicitly assume given an environment in which every $d \in \mathcal{D}$ is an inductive datatype.

Example 2.26. Nat, List, Tree and Ord are inductive, but D is not. □

2.4 Guarded reduction for strong normalization

The μ-reduction is inherently non strongly normalizing. Since

$$(\mathsf{letrec}_\tau\ f = e) \quad \to_\mu \quad e[f := (\mathsf{letrec}_\tau\ f = e)]$$

there are infinite μ-reductions starting from every expression $(\mathsf{letrec}\ f = e)$ such that f occurs free in e. As a first step towards normalization, we restrict the typing and reduction rules of fixpoints. First, we require that fixpoints are only used to defined functions whose domain is a datatype, i.e. instead of the rule (rec) of Fig. 2, we will restrict to the following typing rule:

$$\frac{\Gamma, f \ : \ d\,\boldsymbol{\tau} \to \theta \vdash e \ : \ d\,\boldsymbol{\tau} \to \theta}{\Gamma \vdash (\mathsf{letrec}_{d\,\boldsymbol{\tau} \to \theta}\ f = e) \ : \ d\,\boldsymbol{\tau} \to \theta} \tag{1}$$

Note that all examples on natural numbers, lists, trees and ordinals presented in Sect. 2.2 can by typed with this rule.

Then we replace μ-reduction with a notion of *guarded* γ-reduction $\rightarrow_\gamma$ defined as the compatible closure of:

$$(\mathsf{letrec}_{d\,\boldsymbol{\tau}\to\theta}\ f = e)\ (c\ \boldsymbol{\tau}\ \boldsymbol{a}) \quad \succ_\gamma \quad e[f := (\mathsf{letrec}_{d\,\boldsymbol{\tau}\to\theta}\ f = e)]\ (c\ \boldsymbol{\tau}\ \boldsymbol{a})$$

Definition 2.27 (Guarded reduction). *The relation $\rightarrow$ is defined as*

$$\rightarrow \quad =_{def} \quad \rightarrow_\beta \ \cup\ \rightarrow_\iota \ \cup\ \rightarrow_\gamma$$

The relation $\rightarrow$ is confluent.

These restrictions do not rule out non-terminating and incoherent expressions.

Example 2.28 (Non-termination and incoherence). We can derive

$$\frac{f : \mathsf{Nat} \to \bot \vdash f\ :\ \mathsf{Nat} \to \bot}{\vdash (\mathsf{letrec}_{\mathsf{Nat}\to\bot}\ f = f)\ :\ \mathsf{Nat} \to \bot}$$

and we have $(\mathsf{letrec}_{\mathsf{Nat}\to\bot}\ f = f)\ \mathsf{o} : \bot$ with

$$(\mathsf{letrec}_{\mathsf{Nat}\to\bot}\ f = f)\ \mathsf{o} \quad \rightarrow \quad (\mathsf{letrec}_{\mathsf{Nat}\to\bot}\ f = f)\ \mathsf{o} \quad \rightarrow \quad \ldots$$

□

To obtain strong normalization, we must require that fixpoints must be functions defined by induction on an inductive datatype. This is the purpose of the criterion defined in the next section.

2.5 Syntactic termination criteria

Termination of recursive definitions can be enforced by adopting the guarded reduction rule of Definition 2.27 and by restricting the rule for recursive definitions so that e is a λ-abstraction, and the body of e is guarded by x, which stands for the recursive argument of the function. Formally, this is achieved by the rule:

$$\frac{\Gamma, f : d\,\boldsymbol{\tau} \to \sigma \vdash \lambda x : d\,\boldsymbol{\tau}.\ a\ :\ d\,\boldsymbol{\tau} \to \sigma \qquad \mathcal{G}^x_f(\emptyset, a)}{\Gamma \vdash (\mathsf{letrec}\ f = \lambda x : d\,\boldsymbol{\tau}.\ a)\ :\ d\,\boldsymbol{\tau} \to \sigma}$$

where the guard predicate $\mathcal{G}$ is defined to ensure that the calls to f are performed over expressions that are structurally smaller than x. Informally, a recursive definition is guarded by destructors, i.e. satisfies $\mathcal{G}$, if all occurrences of f in e are protected by a case analysis on x and are applied to a subcomponent of x. The notion of subcomponent is defined as the smallest transitive relation such that the variables that are introduced in the branches of a case analysis are subcomponents of the expression being matched. Barthe *et al* [5] provide a formal definition of the guard predicate for a simply typed λ-calculus and show that the resulting type system can be embedded in the simply typed fragment of System $F^\frown$ that we introduce in the next section.

Example 2.29. The addition on natural numbers, which we recall from Sect. 2.2:

$$\begin{array}{l}\mathsf{plus} := (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}}\ \mathit{plus} = \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\\ \qquad\quad \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow y\\ \qquad\qquad\qquad\quad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ \mathsf{s}\ (\mathit{plus}\ x'\ y)\ \}\\ \qquad) :\quad \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\end{array}$$

is guarded, since the only application of *plus* is protected by a case analysis on x, the formal argument of *plus*. The argument of this application is the pattern variable n, which is a component of x.

While syntactic criteria are widely used, they suffer from several weaknesses. A first weakness is that the syntactic criterion must consider all constructs of the language, and can only be applied if the body of the recursive definition is completely known. Thus, the approach is not compatible with separate compilation.

A second weakness of the approach is that the guard predicate is very sensitive to syntax; for example, the function

$$\begin{array}{l}(\mathsf{letrec}\ \mathit{always\_zero} = \lambda x : \mathsf{Nat}.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o}\\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{s} \Rightarrow \lambda n : \mathsf{Nat}.((\lambda f : \mathsf{Nat} \to \mathsf{Nat}.\ f)\ \mathit{always\_zero})\ n\ \}\\ ) :\quad \mathsf{Nat} \to \mathsf{Nat}\end{array}$$

is not accepted by the guard predicate of [5], because f is passed as an argument to the identity function. It is tempting to extend the definition of the guard predicate with the rule of the form

$$\frac{\mathcal{G}^x_f(V, a') \qquad a \mapsto a'}{\mathcal{G}^x_f(V, a)}$$

where $\mapsto$ is a subset of the reduction relation. However, checking termination of recursive definitions in large developments may become prohibitive, because of the necessity to reduce the body of recursive definitions for checking the guard condition. Worse, an inappropriate choice of $\mapsto$ may lead to allow non-terminating expressions in the type system. For example, allowing $\mapsto$ to include reductions of the form $(\lambda x : A.\, a)\ a' \to a$ when x does not appear in a leads to non-terminating expressions, because it fails to impose any condition on a' which may then contain recursive calls that are not well-founded [5].

3 The system $F^\frown$ of type-based termination

This section presents the system $F^\frown$ of type-based termination. This system has been published in [6], and is an extension of the system $\lambda^\frown$ of [5].

3.1 Semantical ideas for a type-based termination criterion

In this section, we present some intuitions underlying type-based termination. To keep things as simple as possible, we focus on *weak termination*. Recall that an expression e is weakly terminating if and only if it has a normal form. Consider the definition of a recursive function over natural numbers:

$$\frac{f : \mathsf{Nat} \to \theta \vdash e \ : \ \mathsf{Nat} \to \theta}{\vdash (\mathsf{letrec}\ f = e) \ : \ \mathsf{Nat} \to \theta}$$

This function will be computed using the evaluation rules:

$$\begin{array}{rcl} (\mathsf{letrec}\ f = e)\ \mathsf{o} & \to_\gamma & e[f := (\mathsf{letrec}\ f = e)]\ \mathsf{o} \\ (\mathsf{letrec}\ f = e)\ (\mathsf{s}\, n) & \to_\gamma & e[f := (\mathsf{letrec}\ f = e)]\ (\mathsf{s}\, n) \end{array}$$

In order to make sure that the evaluation terminates, we have to ensure that something decreases during the computation. Think of $F =_{\text{def}} (\mathsf{letrec}\ f = e)$ as being a function defined using successive approximations $F_0, \ldots, F_p, \ldots$. Now, assume that we want to evaluate $F(\mathsf{s}\, n)$. If there is some p such that the result of that evaluation can be computed using only $F_0, \ldots, F_{p+1}$, with

$$F_{k+1}(\mathsf{s}\, n) \quad \to_\gamma \quad e[f := F_k]\ (\mathsf{s}\, n) \qquad \text{for all} \quad k \leq p\ ,$$

then the evaluation of $F(\mathsf{s}\, n)$ terminates.

To express a notion of function approximation, we rely on a notion of approximation of *inductive datatype*. Roughly speaking, the type Nat of natural numbers can be drawn as

$$[\![\mathsf{Nat}]\!] \ = \ \{\mathsf{o}, \mathsf{s}\,\mathsf{o}, \ldots, \mathsf{s}^p\mathsf{o}, \ldots\}$$

Let $[\![\mathsf{Nat}]\!](0) =_{\text{def}} \{\mathsf{o}\}$ and $[\![\mathsf{Nat}]\!](p+1) =_{\text{def}} [\![\mathsf{Nat}]\!](p) \cup \{\mathsf{s}\, e \mid e \in [\![\mathsf{Nat}]\!](p)\}$ for all $p \in \mathbb{N}$. Now, the set $[\![\mathsf{Nat}]\!]$ is the limit of its approximations

$$[\![\mathsf{Nat}]\!](0) \ \subseteq \ [\![\mathsf{Nat}]\!](1) \ \subseteq \ \ldots \ \subseteq \ [\![\mathsf{Nat}]\!](p) \ \subseteq \ \ldots$$

These approximations of the type of natural numbers can be used to define functions as the limit of their approximants. More precisely, a total function $F : [\![\mathsf{Nat}]\!] \to \theta$ can be seen as the limit of its finite approximants $F_p : [\![\mathsf{Nat}]\!](p) \to \theta$ for $p \in \mathbb{N}$. Indeed, if $\mathsf{s}\, n$ is the representation of a natural number $p+1$, then $F(\mathsf{s}\, n)$ can by computed by evaluating $F_{p+1}(\mathsf{s}\, n)$. Conversely, in order to ensure that F is the limit of its approximants $(F_p)_{p \in \mathbb{N}}$, we can proceed by induction on $p \in \mathbb{N}$, and force F_{p+1} to be defined only in terms of F_p, as follows:

$$\frac{\forall p \in \mathbb{N} \qquad F_p : [\![\mathsf{Nat}]\!](p) \to \theta \vdash F_{p+1} \ : \ [\![\mathsf{Nat}]\!](p+1) \to \theta}{\vdash F \ : \ [\![\mathsf{Nat}]\!] \to \theta} \text{ if } F = \bigcup_{p \in \mathbb{N}} F_p \quad (2)$$

The basic idea of type-based termination is to use a type system to convey these notions of approximations. Each $[\![\mathsf{Nat}]\!](p)$ can be represented in the type system by an annotated type Nat^p. In such a system, the typing rule for s is

$$\frac{\vdash n \ : \ \mathsf{Nat}^p}{\vdash \mathsf{s}\, n \ : \ \mathsf{Nat}^{p+1}}$$

In addition, we introduce a type Nat^∞ to capture the datatype of natural numbers (corresponding to the datatype Nat of system F^{rec}). These types are naturally ordered by a subtyping relation, expressed by the subsumption rules:

$$\frac{\vdash n \ : \ \mathsf{Nat}^p}{\vdash n \ : \ \mathsf{Nat}^{p+1}} \qquad \frac{\vdash n \ : \ \mathsf{Nat}^p}{\vdash n \ : \ \mathsf{Nat}^\infty}$$

Now, the requirement expressed by (2) can be represented by the typing rule

$$\frac{\forall p \in \mathbb{N} \qquad f : \mathsf{Nat}^p \to \theta \vdash e \ : \ \mathsf{Nat}^{p+1} \to \theta}{\vdash (\mathsf{letrec}\ f = e) \ : \ \mathsf{Nat}^\infty \to \theta} \tag{3}$$

The only remaining issue is to type o. The obvious candidate

$$\frac{}{\vdash \mathsf{o} \ : \ \mathsf{Nat}^0}$$

is unfortunately unsound, both for termination and for coherence: Ex. 2.28 can be easily adapted.

Example 3.1. Assume that $\mathsf{o} : \mathsf{Nat}^0$. Then by subsumption we have $\mathsf{o} : \mathsf{Nat}^p$ for all $p \in \mathbb{N}$, and thus, using (3),

$$\frac{\forall p \in \mathbb{N} \qquad f : \mathsf{Nat}^p \to \bot \vdash \lambda x : \mathsf{Nat}.\ f\,\mathsf{o} \ : \ \mathsf{Nat}^{p+1} \to \bot}{\vdash (\mathsf{letrec}\ f = \lambda x : \mathsf{Nat}.\ f\,\mathsf{o}) \ : \ \mathsf{Nat}^\infty \to \bot}$$

Since $\vdash \mathsf{o} \ : \ \mathsf{Nat}^\infty$, we have a closed term $(\mathsf{letrec}\ f = \lambda x : \mathsf{Nat}.\ f\,\mathsf{o})\,\mathsf{o}$ of type $\bot$, which is moreover non-terminating:

$$(\mathsf{letrec}\ f = \lambda x : \mathsf{Nat}.\ f\,\mathsf{o})\ \mathsf{o} \ \to \ (\lambda x : \mathsf{Nat}.\ (\mathsf{letrec}\ f = \lambda x : \mathsf{Nat}.\ f\,\mathsf{o})\ \mathsf{o})\ \mathsf{o} \ \to \ \dots$$

□

A solution is to assume that o belongs to all $[\![\mathsf{Nat}]\!](p+1)$ with $p \in \mathbb{N}$, but not to $[\![\mathsf{Nat}]\!](0)$, which leads to the interpretation of inductive datatypes detailed in Sect. 3.4. This is reflected by the typing rule

$$\frac{}{\vdash \mathsf{o} \ : \ \mathsf{Nat}^{p+1}}$$

Hence, the expression $\mathsf{s}^p\mathsf{o}$ has size $p+1$.

3.2 Formal definition

Stages. Generalizing the discussion of the previous section, every datatype d is replaced by a family of approximations indexed over a set of *stages*, which are used to record a bound on the "depth" of values. Stages expression are build from a set $\mathcal{V}_\mathrm{S} = \{\imath, \jmath, \kappa, \dots\}$ of stage variables. They use the successor operation $\widehat{\cdot}$ and the constant ∞ denoting the greatest stage.

Definition 3.2 (Stages). *The set* $\mathcal{S} = \{s, r, \dots\}$ *of* stage expressions *is given by the abstract syntax:*

$$\mathcal{S} \ ::= \ \mathcal{V}_{\mathcal{S}} \ \mid \ \infty \ \mid \ \widehat{\mathcal{S}}$$

The substitution $s[\iota := r]$ *of the stage variable* ι *for* r *in* s *is defined in the obvious way.*

The inclusions $[\![\mathsf{Nat}]\!](0) \subseteq \dots \subseteq [\![\mathsf{Nat}]\!](p) \subseteq \mathsf{Nat}(p+1) \subseteq \dots \subseteq [\![\mathsf{Nat}]\!](\infty)$ will hold for each datatype $d \in \mathcal{D}$. This is reflected by a subtyping relation, which is derived from a substage relation $s \leq r$.

Definition 3.3 (Substage relation). *The* substage relation *is the smallest relation* $\leq \subseteq \mathcal{S} \times \mathcal{S}$ *closed under the rules*

$$\textit{(refl)}\ \frac{}{s \leq s} \qquad \textit{(trans)}\ \frac{s \leq r \quad r \leq p}{s \leq p} \qquad \textit{(succ)}\ \frac{}{s \leq \widehat{s}} \qquad \textit{(sup)}\ \frac{}{s \leq \infty}$$

Types. The approximations $(d^s)_{s \in \mathcal{S}}$ of datatypes are directly represented in the syntax of types. Therefore, the types of $F^{\widehat{\ }}$ are the types of F^{rec} where datatype identifiers $d \in \mathcal{D}$ are annotated by size expressions $s \in \mathcal{S}$.

Definition 3.4 (Sized types). *The set* $\overline{\mathcal{T}}$ *of* sized types *is given by the following abstract syntax:*

$$\overline{\mathcal{T}} \ ::= \ \mathcal{V}_{\mathcal{T}} \ \mid \ \overline{\mathcal{T}} \to \overline{\mathcal{T}} \ \mid \ \Pi \mathcal{V}_{\mathcal{T}}.\, \overline{\mathcal{T}} \ \mid \ \mathcal{D}^{\mathcal{S}}\, \overline{\boldsymbol{\mathcal{T}}}$$

where in the clause for datatypes, it is assumed that the length of the vector $\overline{\boldsymbol{\mathcal{T}}}$ *is exactly the arity of the datatype.*

Sized types are denoted by lower-case over lined Greek letters $\overline{\tau}, \overline{\theta}, \overline{\sigma}, \dots$.

The subtyping relation $\overline{\tau} \sqsubseteq \overline{\sigma}$ is directly inherited from the substage relation. The subtyping rule for datatypes

$$\text{(data)}\ \frac{s \leq r \qquad \overline{\boldsymbol{\tau}} \sqsubseteq \overline{\boldsymbol{\sigma}}}{d^s \overline{\boldsymbol{\tau}} \ \sqsubseteq \ d^r \overline{\boldsymbol{\sigma}}}$$

expresses two things. First, it specifies that datatypes are covariant w.r.t. their parameters (an assumption made for the sake of simplicity). For instance we have $\mathsf{List}^{\infty}\, \mathsf{Nat}^{s} \ \sqsubseteq \ \mathsf{List}^{\infty}\, \mathsf{Nat}^{\widehat{s}}$. Second, it reflects inclusions of datatypes approximations:

$$\frac{e : d^s\, \overline{\boldsymbol{\tau}} \qquad s \leq r}{e : d^r\, \overline{\boldsymbol{\tau}}}$$

The substage relation imposes that ∞ is the greatest stage of the system. Hence, we have $\mathsf{Nat}^{s} \sqsubseteq \mathsf{Nat}^{\infty}$ for all stage s. This means that the type Nat^{∞} has no information on the size of its inhabitants. Therefore, it corresponds to the type Nat of system F^{rec}.

Notation 3.5. *Given a datatype identifier* d*, we write* $d\, \overline{\boldsymbol{\tau}}$ *to mean* $d^{\infty}\, \overline{\boldsymbol{\tau}}$.

Definition 3.6 (Subtyping). *The* subtyping relation *is the smallest relation* $\overline{\tau} \sqsubseteq \overline{\sigma}$*, where* $\overline{\tau}, \overline{\sigma} \in \overline{\mathcal{T}}$*, such that*

$$\textit{(var)}\ \frac{}{X \sqsubseteq X} \qquad \textit{(func)}\ \frac{\overline{\tau'} \sqsubseteq \overline{\tau} \quad \overline{\sigma} \sqsubseteq \overline{\sigma'}}{\overline{\tau} \to \overline{\sigma} \sqsubseteq \overline{\tau'} \to \overline{\sigma'}}$$

$$\textit{(prod)}\ \frac{\overline{\tau} \sqsubseteq \overline{\sigma}}{\Pi X.\ \overline{\tau} \sqsubseteq \Pi X.\ \overline{\sigma}} \qquad \textit{(data)}\ \frac{s \leq r \quad \overline{\boldsymbol{\tau}} \sqsubseteq \overline{\boldsymbol{\sigma}}}{d^s \overline{\boldsymbol{\tau}} \sqsubseteq d^r \overline{\boldsymbol{\sigma}}}$$

We denote by $|.| : \overline{\mathcal{T}} \to \mathcal{T}$ the erasure function from sized types to types, which forgets the size information represented in a type of $F^{\widehat{\ }}$. Erasure is defined inductively as follows:

$$|X| = X \qquad |\overline{\tau} \to \overline{\theta}| = |\overline{\tau}| \to |\overline{\theta}| \qquad |\Pi X.\ \overline{\tau}| = \Pi X.\ |\overline{\tau}| \qquad |d^s\, \overline{\boldsymbol{\tau}}| = d\, |\overline{\boldsymbol{\tau}}|$$

Sized inductive datatypes. We now turn to datatype definitions. In Def. 2.25, we have defined inductive datatypes definitions for F^{rec} as declarations of the form

$$\textbf{Inductive}\ \ d\ \boldsymbol{X}\ :=\ c_1 : \sigma_1\ \mid\ \ \ldots\ \ \mid\ c_n : \sigma_n$$

where for all $k \in \{1, \ldots, n\}$, σ_k is of the form $\boldsymbol{\theta}_k \to d\ \boldsymbol{X}$ with $\boldsymbol{X}\ \mathsf{pos}\ \boldsymbol{\theta}_k$ and $d\ \boldsymbol{X}\ \mathsf{pos}\ \boldsymbol{\theta}_k$.

The inductive datatypes of $F^{\widehat{\ }}$ are annotated versions of inductive datatypes of F^{rec}. Each occurrence of $d' \neq d$ in $\overline{\boldsymbol{\theta}}_k$ is annotated with ∞, and each occurrence of d in $\boldsymbol{\theta}_k$ is annotated with the stage variable $\imath$. Then, the annotated type of c_k is $\Pi \boldsymbol{X}.\ \overline{\boldsymbol{\theta}}_k \to \widehat{d^{\imath}}\ \boldsymbol{X}$. Definitions of sized inductive datatypes are like definitions of inductive datatypes in F^{rec}, excepted that constructors are now given their sized type. For instance, sized natural numbers are declared as follows:

$$\textbf{Inductive}\ \ \mathsf{Nat}\ :=\ \mathsf{o} : \widehat{\mathsf{Nat}^{\imath}}\ \mid\ \mathsf{s} : \mathsf{Nat}^{\imath} \to \widehat{\mathsf{Nat}^{\imath}}$$

In words, the constructor o always build an expression with at least one constructor, hence of size $\widehat{0}$. Since stages record upper-bound on sizes, we have o of stage $\widehat{p}$ for all stages p. On the other hand, s turns an expression of stage p into one of stage $\widehat{p}$.

We now turn to the formal definition.

Definition 3.7 (Sized inductive datatypes).
(i) A sized inductive datatype definition *is a declaration*

$$\textbf{Inductive}\ \ d\ \boldsymbol{X}\ :=\ c_1 : \overline{\sigma}_1\ \mid\ \ \ldots\ \ \mid\ c_n : \overline{\sigma}_n$$

such that

- *its erased form* $\textbf{Inductive}\ \ d\ \boldsymbol{X}\ :=\ c_1 : |\overline{\sigma}_1|\ \mid\ \ \ldots\ \ \mid\ c_n : |\overline{\sigma}_n|$ *is an inductive datatype definition in* F^{rec}*, and*
- *for all* $k \in \{1, \ldots, n\}$*, the sized type* $\overline{\sigma}_k$ *is of the form* $\overline{\boldsymbol{\theta}}_k \to \widehat{d^{\imath}}\ \boldsymbol{X}$ *where each occurrence of* $d' \neq d$ *in* $\overline{\boldsymbol{\theta}}_k$ *is annotated with* ∞*, and each occurrence of* d *in* $\boldsymbol{\theta}_k$ *is annotated with the stage variable* $\imath$*.*

For all $k \in \{1, \dots, n\}$, we write $c_k : \Pi \boldsymbol{X}.\, \overline{\boldsymbol{\theta}}_k \to d^{\widehat{\imath}}\ \boldsymbol{X}$.

(ii) A sized environment is a sequence of sized inductive datatype definitions $I_1 \dots I_n$ in which constructors of the sized inductive datatype definition I_k only use datatypes introduced by $I_1 \dots I_k$.

Note that our definition of inductive datatypes types rules out heterogeneous and mutually inductive datatypes. This is only a matter of simplicity.

Besides, the positivity requirement for $d^{\imath}\boldsymbol{X}$ is necessary to guarantee strong normalization. Also, the positivity requirement for $\boldsymbol{X}$ is added to guarantee the soundness of the subtyping rule (data) for datatypes, and to avoid considering polarity, as in e.g. [17].

Example 3.8 (Sized datatypes definitions).

(i) The sized inductive datatype of polymorphic lists is defined as

$$\textbf{Inductive}\ \ \mathsf{List}\ X\ :=\ \mathsf{nil} : \mathsf{List}^{\widehat{\imath}}\ X\ \mid\ \mathsf{cons} : X \to \mathsf{List}^{\imath}\ X \to \mathsf{List}^{\widehat{\imath}}\ X$$

The minimal stage of a list is its length, with the nil list being of stage at least $\widehat{\imath}$. For instance, leaving implicit the type argument of constructors, we have $\mathsf{cons}\ n\ \mathsf{nil} : \mathsf{List}^{\widehat{\widehat{\imath}}}\ \mathsf{Nat}$ and $\mathsf{cons}\ n_1(\dots(\mathsf{cons}\ n_p\ \mathsf{nil})\dots) : \mathsf{List}^{\widehat{\imath}^{p+1}}\ \mathsf{Nat}$.

(ii) The sized inductive datatype of polymorphic finitely branching trees is

$$\textbf{Inductive}\ \ \mathsf{Tree}\ X\ :=\ \mathsf{node} : X \to \mathsf{List}\ (\mathsf{Tree}^{\imath}\ X) \to \mathsf{Tree}^{\widehat{\imath}}\ X$$

The minimal stage of a tree is its depth. The least tree contains just one leave $\mathsf{node}\, n\, \mathsf{nil}$ and is of stage at least $\widehat{\imath}$. Consider p trees $t_1, \dots, t_p$ of respective types $\mathsf{Tree}^{s_1}\ \mathsf{Nat}, \dots, \mathsf{Tree}^{s_p}\ \mathsf{Nat}$, and let $l\ :=\ \mathsf{cons}\ t_1(\dots(\mathsf{cons}\ t_p\ \mathsf{nil})\dots)$. For all stage s greater than each s_k, we have $l : \mathsf{List}^{\widehat{\imath}^{p+1}}\ (\mathsf{Tree}^{s}\ \mathsf{Nat})$, hence $\mathsf{node}\, n\, l : \mathsf{Tree}^{\widehat{s}}\ \mathsf{Nat}$. Therefore, the least stage of $\mathsf{node}\, n\, l$ is the strict supremum of the stages of the trees in l. Moreover, the stage of l as a list has been forgotten in the stage of $\mathsf{node}\, n\, l$.

(iii) The sized inductive datatype of Brouwer ordinals is defined as

$$\textbf{Inductive}\ \ \mathsf{Ord}\ :=\ \mathsf{o} : \mathsf{Ord}^{\widehat{\imath}}\ \mid\ \mathsf{s} : \mathsf{Ord}^{\imath} \to \mathsf{Ord}^{\widehat{\imath}}\ \mid\ \mathsf{lim} : (\mathsf{Nat} \to \mathsf{Ord}^{\imath}) \to \mathsf{Ord}^{\widehat{\imath}}$$

As with finitely branching trees, the least stage of $\mathsf{lim}\, f$ is the strict supremum of the stages of $f\, n$ for $n \in \mathsf{Nat}$. □

In the remaining of this tutorial, we implicitly assume given a sized environment in which every $d \in \mathcal{D}$ is a sized inductive datatype.

Terms and reductions. The terms of $F^{\widehat{\ }}$ are those of F^{rec}, defined in Sect. 2.2. The reduction relation of $F^{\widehat{\ }}$ is the rewrite relation $\to$ defined in Def. 2.27.

Remark 3.9 (Stages in terms). Note that the types appearing in terms are those of F^{rec}: they do not carry stage expressions. As shown in [6], subject reduction would have failed if terms conveyed stage expressions. However, it is often useful to write these annotations in examples. For instance, we may write $\lambda x : \mathsf{Nat}^{\imath}.\ x$ to denote the term $\lambda x : \mathsf{Nat}.\ x$.

Typing rules. The typing rule for fixpoints uses a predicate ι pos $\overline{\sigma}$ that is used to ensure that a stage variable occurs positively in the codomain of the type of a recursive definition. Its definition is similar to that of the predicate τ pos σ of Sect. 2.3.

Definition 3.10 (Positivity). *Given two stage expressions s and r, let s occ r (resp. s nocc r) if and only if s occurs in r (resp. does not occurs in r). Moreover, let s nocc $\overline{\tau}$ if the stage expression s does not occur in the sized type $\overline{\tau}$.*

The predicate s pos $\overline{\tau}$ (resp. s neg $\overline{\tau}$), stating that all occurrences of s in $\overline{\tau}$ are positive (resp. negative), is inductively defined in Fig. 4.

The typing rules follow [6].

Definition 3.11 (Typing). *A* sized context *is a map $\overline{\Gamma} : \mathcal{V}_{\mathcal{E}} \to \overline{\mathcal{T}}$ of finite domain. The* typing relation *is the smallest relation $\overline{\Gamma} \vdash e \ : \ \overline{\tau}$ which is closed under the rules of Fig. 5, page 30.*

All rules but (cons), (case), (rec) and (sub) do not mention stages. They are therefore the same as in F^{rec}. The rule (cons) for constructors simply says that a constructor can be given any possible stage instance of its type specified in a datatype definition.

In order to understand the rule (case), we look at it for natural numbers:

$$\frac{\overline{\Gamma} \vdash e \ : \ \mathsf{Nat}^{\widehat{s}} \qquad \overline{\Gamma} \vdash e_{\mathsf{o}} \ : \ \overline{\sigma} \qquad \overline{\Gamma} \vdash e_{\mathsf{s}} \ : \ \mathsf{Nat}^{s} \to \overline{\sigma}}{\overline{\Gamma} \vdash \mathsf{case}_{|\overline{\sigma}|} \ e \ \mathsf{of} \ \{0 \Rightarrow e_{\mathsf{o}} \mid \mathsf{s} \Rightarrow e_{\mathsf{s}}\} \ : \ \overline{\sigma}}$$

The important point, which makes the difference with the rule of F^{rec}, is that the type of the expression e subject to case analysis must have a stage of the form $\widehat{s}$. Note that this is always possible thanks to subtyping. Now, assume that e is of the form $\mathsf{s}\, n$. The rule (case) says that the term e_{s} sees n as an expression of stage s. Indeed, we have

$$\mathsf{case}_{|\overline{\sigma}|} \ (\mathsf{s}\, n) \ \mathsf{of} \ \{0 \Rightarrow e_{\mathsf{o}} \mid \mathsf{s} \Rightarrow e_{\mathsf{s}}\} \ \to_{\iota} \ e_{\mathsf{s}}\, n \qquad \text{with} \qquad \overline{\Gamma} \vdash e_{\mathsf{s}} \ : \ \mathsf{Nat}^{s} \to \overline{\sigma}$$

We now discuss the typing rule (rec) for fixpoints, in the case of natural numbers, and assuming that ι does not occur in $\overline{\theta}$:

$$(\mathrm{rec}) \ \frac{\overline{\Gamma}, f \ : \ \mathsf{Nat}^{\iota} \to \overline{\theta} \vdash e \ : \ \mathsf{Nat}^{\widehat{\iota}} \to \overline{\theta}}{\overline{\Gamma} \vdash (\mathsf{letrec}_{\mathsf{Nat} \to |\overline{\theta}|} \ f = e) \ : \ \mathsf{Nat}^{s} \to \overline{\theta}} \ \text{if } \iota \notin \overline{\Gamma}, \overline{\boldsymbol{\tau}}$$

As explained in Sect. 3.1, typing $\mathsf{fix} := (\mathsf{letrec}_{\mathsf{Nat} \to |\overline{\theta}|} \ f = e)$ with type $\mathsf{Nat}^{\infty} \to \overline{\theta}$ requires showing that the body e turns an approximation of fix of type $\mathsf{Nat}^{\iota} \to \overline{\theta}$ into its next approximation, which is of type $\mathsf{Nat}^{\widehat{\iota}} \to \overline{\theta}$. As discussed in Sect 3.4, such recursive functions are terminating and, despite its simplicity, this mechanism is powerful enough to capture course-of-value recursion.

Notation 3.12. *When writing examples of typings of fixpoints, it is convenient to write at the term level the stage annotations corresponding to fixpoint variables. For instance, given a derivation of the form*

$$(rec)\ \frac{f : \mathsf{Nat}^{\iota} \to \overline{\theta} \vdash e \ : \ \mathsf{Nat}^{\widehat{\iota}} \to \overline{\theta}}{\vdash (\mathsf{letrec}_{\mathsf{Nat}\to|\overline{\theta}|}\ f = e) \ : \ \mathsf{Nat}^{s} \to \overline{\theta}}$$

where $\iota \notin \overline{\theta}$, it is convenient to write $(\mathsf{letrec}\ f : \mathsf{Nat}^{\iota} \to \overline{\theta} = e) : \mathsf{Nat}^{s} \to \overline{\theta}$ *to mean that using $f : \mathsf{Nat}^{\iota} \to \overline{\theta}$, we must have $e : \mathsf{Nat}^{\widehat{\iota}} \to \overline{\theta}$. We use a similar notation when $\iota \in \overline{\theta}$.*

The following example, taken from [2], shows that strong normalization may fail if the positivity condition is not met. However, there are finer conditions on the occurrences of ι in $\overline{\theta}$ than positivity that nevertheless preserve strong normalization, see [3,2].

Example 3.13 (Counter-example for the positivity condition [2]). Consider the terms

$$\begin{aligned}
\mathsf{shift} &:= \lambda f : \mathsf{Nat} \to \mathsf{Nat}^{\widehat{\widehat{\iota}}}.\ \lambda x : \mathsf{Nat}.\ \ \mathsf{case}_{\mathsf{Nat}}\ \overbrace{f\,(\mathsf{s}\,x)}^{\mathsf{Nat}^{\widehat{\widehat{\iota}}}}\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow \mathsf{o} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda y : \mathsf{Nat}^{\widehat{\iota}}.\ y\ \} \\
\mathsf{plus_2} &:= \lambda x : \mathsf{Nat}.\ \mathsf{s}\,(\mathsf{s}\,x)
\end{aligned}$$

of type respectively $\mathsf{Nat} \to \mathsf{Nat}^{\widehat{\widehat{\iota}}}$ and $\mathsf{Nat} \to \mathsf{Nat}^{\widehat{\iota}}$. Note that $\mathsf{shift}\ \mathsf{plus\_2} \to^{*} \mathsf{plus\_2}$. Consider now the following fixpoint:

$$\begin{aligned}
\mathsf{loop} := (&\mathsf{letrec}\ \mathit{loop} : \mathsf{Nat}^{\iota} \to (\mathsf{Nat} \to \mathsf{Nat}^{\widehat{\iota}}) \to \mathsf{Nat} = \lambda x : \mathsf{Nat}^{\widehat{\iota}}.\ \lambda f : \mathsf{Nat} \to \mathsf{Nat}^{\widehat{\widehat{\iota}}}. \\
&\quad \mathsf{case}_{\mathsf{Nat}}\ (f\,x)\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \\
&\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}^{\widehat{\iota}}.\ \mathsf{case}\ x'\ \mathsf{of}\ \{ \\
&\qquad\qquad\qquad\qquad \mid \mathsf{o} \Rightarrow \mathsf{o} \\
&\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda y' : \mathsf{Nat}^{\iota}.\ \mathit{loop}\ y'\ (\mathsf{shift}\ f) \\
&\qquad\qquad\qquad\qquad \} \\
&\qquad\qquad \} \\
&) : \quad \mathsf{Nat}^{s} \to (\mathsf{Nat} \to \mathsf{Nat}^{\widehat{s}}) \to \mathsf{Nat}
\end{aligned}$$

The stage variable ι occurs negatively in the type $(\mathsf{Nat} \to \mathsf{Nat}^{\widehat{\iota}}) \to \mathsf{Nat}$. Therefore, the expression loop would be typable in $F^{\frown}$ without the condition $\iota\ \mathsf{pos}\ \overline{\theta}$ in the rule (rec). But then it would also be possible to type the term $\mathsf{loop}\ \mathsf{o}\ \mathsf{plus\_2}$ which is non normalizing

$$\mathsf{loop}\ \mathsf{o}\ \mathsf{plus\_2} \ \to^{*}\ \mathsf{loop}\ \mathsf{o}\ (\mathsf{shift}\ \mathsf{plus\_2}) \ \to^{*}\ \mathsf{loop}\ \mathsf{o}\ \mathsf{plus\_2} \ \to\ \ldots$$

□

$$\frac{s\ \mathsf{nocc}\ \overline{\tau}}{s\ \mathsf{pos}\ \overline{\tau}} \qquad \frac{s\ \mathsf{pos}\ \overline{\tau}_i \quad (1 \leq i \leq \mathsf{ar}(d))}{s\ \mathsf{pos}\ d^r\ \overline{\boldsymbol{\tau}}}$$

$$\frac{s\ \mathsf{pos}\ \overline{\tau}}{s\ \mathsf{pos}\ \Pi X.\ \overline{\tau}} \qquad \frac{s\ \mathsf{neg}\ \overline{\tau}_2 \qquad s\ \mathsf{pos}\ \overline{\tau}_1}{s\ \mathsf{pos}\ \overline{\tau}_2 \to \overline{\tau}_1}$$

$$\frac{s\ \mathsf{nocc}\ \overline{\tau}}{s\ \mathsf{neg}\ \overline{\tau}} \qquad \frac{s\ \mathsf{neg}\ \overline{\tau}_i \qquad (1 \leq i \leq \mathsf{ar}(d))}{s\ \mathsf{neg}\ d^r\ \overline{\boldsymbol{\tau}}}$$

$$\frac{s\ \mathsf{neg}\ \overline{\tau}}{s\ \mathsf{neg}\ \Pi X.\ \overline{\tau}} \qquad \frac{s\ \mathsf{pos}\ \overline{\tau}_2 \qquad s\ \mathsf{neg}\ \overline{\tau}_1}{s\ \mathsf{neg}\ \overline{\tau}_2 \to \overline{\tau}_1}$$

Fig. 4. Positivity and negativity of a stage occurrence

Examples. We now review some examples of functions presented in Sect. 2.2. We begin with the minus and div functions on natural numbers.

*Example 3.14 (*minus *and* div*).* In F^{rec}, minus and div are defined as follows:

$$\begin{aligned}
\mathsf{minus} := (&\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}}\ ms = \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\\
&\quad \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow x\\
&\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ \mathsf{case}\ y\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow x\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda y' : \mathsf{Nat}.\ ms\ x'\ y'\ \}\\
&\qquad\qquad \}\\
&) : \quad \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}
\end{aligned}$$

$$\begin{aligned}
\mathsf{div} := (&\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}}\ div = \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\\
&\quad \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow \mathsf{o}\\
&\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ \mathsf{s}\ (div\ (\mathsf{minus}\ x'\ y)\ y)\ \}\\
&) : \quad \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}
\end{aligned}$$

For minus, in $\overline{\Gamma'} \ =_{\mathrm{def}}\ ms : \mathsf{Nat}^{\imath} \to \mathsf{Nat} \to \mathsf{Nat}^{\imath}, x : \mathsf{Nat}^{\widehat{\imath}}, y : \mathsf{Nat}, x' : \mathsf{Nat}^{\imath}$ we have

$$\frac{\overline{\Gamma'} \vdash y\ :\ \mathsf{Nat}^{\widehat{\infty}} \qquad \overline{\Gamma'} \vdash x\ :\ \mathsf{Nat}^{\widehat{\imath}} \qquad \overline{\Gamma'} \vdash \lambda y' : \mathsf{Nat}.\ ms\,x'\,y'\ :\ \mathsf{Nat}^{\infty} \to \mathsf{Nat}^{\widehat{\imath}}}{\overline{\Gamma'} \vdash \mathsf{case}_{\mathsf{Nat}}\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow x \mid \mathsf{s} \Rightarrow \lambda y' : \mathsf{Nat}.\ ms\,x'\,y'\}\ :\ \mathsf{Nat}^{\widehat{\imath}}}$$

We deduce that

$$\frac{\overline{\Gamma} \vdash x\ :\ \mathsf{Nat}^{\widehat{\imath}} \qquad \overline{\Gamma} \vdash x\ :\ \mathsf{Nat}^{\widehat{\imath}} \qquad \overline{\Gamma} \vdash \lambda x' : \mathsf{Nat}.\ e_{\mathsf{s}}\ :\ \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\widehat{\imath}}}{\overline{\Gamma} \vdash \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow x \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ e_{\mathsf{s}}\}\ :\ \mathsf{Nat}^{\widehat{\imath}}}$$

where $e_{\mathsf{s}} := \mathsf{case}_{\mathsf{Nat}}\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow x \mid \mathsf{s} \Rightarrow \lambda y' : \mathsf{Nat}.\ ms\,x'\,y'\}$ and $\overline{\Gamma}$ is the typing context $ms : \mathsf{Nat}^{\imath} \to \mathsf{Nat} \to \mathsf{Nat}^{\imath}, x : \mathsf{Nat}^{\widehat{\imath}}, y : \mathsf{Nat}$. Using (rec), for all stages s

we get

$$\frac{ms : \mathsf{Nat}^\imath \to \mathsf{Nat} \to \mathsf{Nat}^\imath \vdash \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\ e_{\mathsf{minus}} \ : \ \mathsf{Nat}^{\widehat{\imath}} \to \mathsf{Nat} \to \mathsf{Nat}^{\widehat{\imath}}}{\vdash (\mathsf{letrec}\ ms = \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\ e_{\mathsf{minus}}) \ : \ \mathsf{Nat}^s \to \mathsf{Nat} \to \mathsf{Nat}^s}$$

where $e_{\mathsf{minus}} := \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow x \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ e_{\mathsf{s}}\}$. Hence, system $F^{\widehat{\ }}$ is powerful enough to express that the size of $(\mathsf{minus}\ n\ m)$ is at most the size of n. This information is essential for the typing of div. In the computation of $(\mathsf{div}\,(\mathsf{s}\,n)\,m)$, the recursive call to div is performed on the argument $(\mathsf{minus}\ n\ m)$ which is not a subterm of $(\mathsf{s}\,n)$. It can even be syntactically arbitrarily bigger! However, with stages we have the information that if $(\mathsf{s}\,n)$, as a natural number, is of size at most $\widehat{p}$, then $(\mathsf{minus}\ n\ m)$ is of size at most p. The termination argument relies on this decreasing from $\widehat{p}$ to p.

Formally, using (rec), for all stages s we obtain $\mathsf{div} : \mathsf{Nat}^s \to \mathsf{Nat} \to \mathsf{Nat}^s$ from the judgment

$$\begin{array}{l} \quad div : \mathsf{Nat}^\imath \to \mathsf{Nat} \to \mathsf{Nat}^\imath \vdash \\ \lambda x : \mathsf{Nat}.\ \lambda y : \mathsf{Nat}.\ \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\ \mathsf{o} \Rightarrow \mathsf{o} \\ \qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}.\ \mathsf{s}\ \underbrace{(div\ \underbrace{(\mathsf{minus}\ x'\ y)}_{\mathsf{Nat}^\imath}\ y)}_{\mathsf{Nat}^\imath} \\ \qquad\qquad\qquad\qquad \}\ :\ \mathsf{Nat}^{\widehat{\imath}} \to \mathsf{Nat} \to \mathsf{Nat}^{\widehat{\imath}} \end{array}$$

□

Example 3.15 (Ordinals). In F^{rec}, the addition on ordinals is defined as

$$\begin{array}{l} \mathsf{add} := (\mathsf{letrec}_{\mathsf{Ord}\to\mathsf{Ord}\to\mathsf{Ord}}\ add = \lambda x : \mathsf{Ord}.\ \lambda y : \mathsf{Ord}. \\ \quad \mathsf{case}_{\mathsf{Ord}}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \\ \qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Ord}.\ \mathsf{s}\ (add\ x'\ y) \\ \qquad\qquad \mid \mathsf{lim} \Rightarrow \lambda f : \mathsf{Nat} \to \mathsf{Ord}.\ \ \mathsf{lim}\ (\lambda z : \mathsf{Nat}.\ add\ (f\ z)\ y)\} \\ \quad) : \quad \mathsf{Ord} \to \mathsf{Ord} \to \mathsf{Ord} \end{array}$$

Therefore, we have $\mathsf{add}\ (\mathsf{lim}\ f)\ e \ \to^*\ \mathsf{lim}\ (\lambda z : \mathsf{Nat}.\ \mathsf{add}\ (f\ z)\ e)$. The difficulty here is that $f\,z$ is not a subterm of $\mathsf{lim}\,f$. However, this example is handled by the syntactic termination criterion described in Sect. 2.5. In $F^{\widehat{\ }}$, add is typed as follows:

$$\begin{array}{l} \mathsf{add} := (\mathsf{letrec}\ add : \mathsf{Ord}^\imath \to \mathsf{Ord} \to \mathsf{Ord} = \lambda x : \mathsf{Ord}^{\widehat{\imath}}.\ \lambda y : \mathsf{Ord}. \\ \quad \mathsf{case}_{\mathsf{Ord}}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \\ \qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Ord}^\imath.\ \mathsf{s}\ (add\ x'\ y) \\ \qquad\qquad \mid \mathsf{lim} \Rightarrow \lambda f : \mathsf{Nat} \to \mathsf{Ord}^\imath.\ \ \mathsf{lim}\ (\lambda z : \mathsf{Nat}.\ add\ \underbrace{(f\ z)}_{\mathsf{Ord}^\imath}\ y)\} \\ \quad) : \quad \mathsf{Ord}^s \to \mathsf{Ord} \to \mathsf{Ord} \end{array}$$

We now come back to the discussion of Ex. 3.8.(iii), about the stage of $\mathsf{lim}\,\mathsf{inj}$, where $\mathsf{inj} : \mathsf{Nat} \to \mathsf{Ord}$ is the canonical injection of natural numbers into ordinals.

In $F^{\widehat{}}$, it is defined as follows:

$$\begin{array}{l}\mathsf{inj} := (\mathsf{letrec}\ \mathit{inj} : \mathsf{Nat}^{\imath} \to \mathsf{Ord}^{\imath} = \lambda x : \mathsf{Nat}^{\widehat{\imath}}. \\ \qquad \mathsf{case}_{\mathsf{Nat}}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \\ \qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x' : \mathsf{Nat}^{\imath}.\ \mathsf{s}\ (\mathit{inj}\ x')\} \\ \qquad) : \quad \mathsf{Nat}^{s} \to \mathsf{Ord}^{s}\end{array}$$

Note that this definition uses the same stage variable $\imath$ to annotate both Nat and Ord. Moreover, for all $p \in \mathbb{N}$ we have $\mathsf{inj}\,(\mathsf{s}^p\,\mathsf{o}) \ \to^* \ (\mathsf{s}^p\,\mathsf{o})$. The only way to apply inj to lim is to instantiate their sized types as $\mathsf{Nat}^{\infty} \to \mathsf{Ord}^{\infty}$ and $(\mathsf{Nat}^{\infty} \to \mathsf{Ord}^{\infty}) \to \mathsf{Ord}^{\infty}$ respectively. We thus get $\mathsf{lim}\,\mathsf{inj} : \mathsf{Ord}^{\infty}$, and ∞ is the best possible approximation of the size of $\mathsf{lim}\,\mathsf{inj}$ expressible in the system. □

Example 3.16 (Concatenations of lists). The function app concatenates two lists. Therefore, if l_1 and l_2 are of respective size s_1 and s_2, then $\mathsf{app}\ l_1\,l_2$ is of size $s_1 + s_2$. But system $F^{\widehat{}}$ does not feature stage addition. Hence the precise size of $\mathsf{app}\ l_1\,l_2$ is not expressible in the system, and we have $\mathsf{app}\ l_1\,l_2 : \mathsf{List}^{\infty}\,X$. Since recursion is performed only on the first argument of app, the size of the second one is not relevant, and for all stages s we have $\mathsf{app} : \Pi X.\ \mathsf{List}^{s}\,X \to \mathsf{List}\ X \to \mathsf{List}\ X$. The function app is defined as follows:

$$\begin{array}{l}\mathsf{app} := \Lambda X.\ (\mathsf{letrec}\ \mathit{app} : \mathsf{List}^{\imath}\,X \to \mathsf{List}\,X \to \mathsf{List}\,X = \lambda x : \mathsf{List}^{\widehat{\imath}}\,X.\ \lambda y : \mathsf{List}\ X. \\ \qquad \mathsf{case}_{\mathsf{List}\,X}\ x\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow y \\ \qquad\qquad\qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z : X.\ \lambda x' : \mathsf{List}^{\imath}\,X.\ \ \mathsf{cons}\ z\ (\mathit{app}\ x'\ y) \\ \qquad\qquad\qquad\qquad \} \\ \quad) : \quad \Pi X.\ \mathsf{List}^{s}\,X \to \mathsf{List}\ X \to \mathsf{List}\ X\end{array}$$

The function conc concatenates a list of lists. As for app, we cannot express its precise typing in $F^{\widehat{}}$, and for all stage s we have $\mathsf{conc} : \Pi X.\ \mathsf{List}^{s}\,(\mathsf{List}\ X) \to \mathsf{List}\ X$. The function conc is defined as follows:

$$\begin{array}{l}\mathsf{conc} := \Lambda X.\ \ (\mathsf{letrec}\ \mathit{conc} : \mathsf{List}^{\imath}\,(\mathsf{List}\,X) \to \mathsf{List}\,X = \lambda x : \mathsf{List}^{\widehat{\imath}}\,(\mathsf{List}\,X). \\ \quad \mathsf{case}_{\mathsf{List}\,(\mathsf{List}\,X)}\ x\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil} \\ \qquad\qquad\qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z : \mathsf{List}\,X.\ \ \lambda x' : \mathsf{List}^{\imath}\,(\mathsf{List}\,X).\ \mathsf{app}\ z\ (\mathit{conc}\ x') \\ \qquad\qquad\qquad\qquad \} \\ \quad) : \quad \Pi X.\ \mathsf{List}^{s}\,(\mathsf{List}\ X) \to \mathsf{List}\ X\end{array}$$

□

Example 3.17 (The map function on a list). The function $\mathsf{map}\,f\,l$ applies the function f to each element of the list l and produces the corresponding list. Hence $\mathsf{map}\,f\,l$ evaluates to a list of the same size as l. This is expressible in $F^{\widehat{}}$ by $\mathsf{map} : \Pi X.\ \Pi Y.\ (X \to Y) \to \mathsf{List}^{s}\,X \to \mathsf{List}^{s}\,Y$. The function map is defined

as follows:

$$
\begin{array}{l}
\mathsf{map} := \Lambda X.\ \Lambda Y.\ \lambda f : X \to Y.\ \ (\mathsf{letrec}\ \mathit{map} : \mathsf{List}^{\imath}\ X \to \mathsf{List}^{\imath}\ Y = \lambda x : \mathsf{List}^{\widehat{\imath}}\ X. \\
\qquad \mathsf{case}_{\mathsf{List}\ X}\ x\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil} \\
\qquad\qquad\qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z : X.\ \ \lambda x' : \mathsf{List}^{\imath}\ X.\ \underbrace{\mathsf{cons}\ (f\ z)\ \underbrace{(\mathit{map}\ f\ x')}_{\mathsf{List}^{\imath}\ Y}}_{\mathsf{List}^{\widehat{\imath}}\ Y} \\
\qquad\qquad\qquad\qquad \} \\
\qquad) : \quad \Pi X.\ \Pi Y.\ (X \to Y) \to \mathsf{List}^{s}\ X \to \mathsf{List}^{s}\ Y
\end{array}
$$

□

Example 3.18 (Flattening of finitely branching trees). System $F^{\frown}$ is able to prove the termination of flatten, even if the recursive call is made through a call to map. However, as for app and conc, the system $F^{\frown}$ cannot express the precise typing of the flattening of finitely branching trees flatten. The function flatten is defined by induction on the depth of its argument. We thus have

$$
\begin{array}{l}
\mathsf{flatten} := \Lambda X.\ (\mathsf{letrec}\ \mathit{flat} : \mathsf{Tree}^{\imath}\ X \to \mathsf{List}\ X = \lambda t : \mathsf{Tree}^{\widehat{\imath}}\ X.\ \mathsf{case}_{\mathsf{Tree}\ X}\ t\ \mathsf{of}\ \{ \\
\qquad \mathsf{node} \Rightarrow \lambda x : X.\ \lambda xs : \mathsf{List}\ (\mathsf{Tree}^{\imath}\ X).\ \mathsf{cons}\ x\ (\underbrace{\mathsf{conc}\ (\underbrace{\mathsf{map}\ \mathit{flat}\ xs}_{\mathsf{List}\ (\mathsf{List}\ X)})}_{\mathsf{List}\ X}) \\
\qquad \} \\
) : \quad \Pi X.\ \mathsf{Tree}^{s}\ X \to \mathsf{List}\ X
\end{array}
$$

□

3.3 Some important properties

We now state some important properties of system $F^{\frown}$. They are the properties underlined in Sect. 2.1 for system F, namely subject reduction, strong normalizability of typable terms, and coherence of the type system.

Subject reduction. The proof of this property is easily adapted from the proof for $\lambda^{\frown}$ presented in [5].

Theorem 3.19 (Subject reduction). *If* $\overline{\Gamma} \vdash e\ :\ \overline{\tau}$ *and* $e \to e'$*, then also* $\overline{\Gamma} \vdash e'\ :\ \overline{\tau}$.

With respect to stage annotations, subject reduction says that the size approximations represented by stages are preserved by reduction, and moreover that they can be retrieved by the type system after a reduction step.

$$\text{(var)}\ \frac{}{\overline{\Gamma}, x : \overline{\sigma} \vdash x : \overline{\sigma}} \qquad \text{(sub)}\ \frac{\overline{\Gamma} \vdash e : \overline{\sigma} \qquad \overline{\sigma} \sqsubseteq \overline{\tau}}{\overline{\Gamma} \vdash e : \overline{\tau}}$$

$$\text{(abs)}\ \frac{\overline{\Gamma}, x : \overline{\tau} \vdash e : \overline{\sigma}}{\overline{\Gamma} \vdash \lambda x : |\overline{\tau}|.\ e : \overline{\tau} \to \overline{\sigma}} \qquad \text{(app)}\ \frac{\overline{\Gamma} \vdash e : \overline{\tau} \to \overline{\sigma} \qquad \overline{\Gamma} \vdash e' : \overline{\tau}}{\overline{\Gamma} \vdash e\ e' : \overline{\sigma}}$$

$$\text{(T-abs)}\ \frac{\overline{\Gamma} \vdash e : \overline{\sigma}}{\overline{\Gamma} \vdash \Lambda X.\ e : \Pi X.\ \overline{\sigma}}\ \text{if } X \notin \overline{\Gamma} \qquad \text{(T-app)}\ \frac{\overline{\Gamma} \vdash e : \Pi X.\ \overline{\sigma}}{\overline{\Gamma} \vdash e\ |\overline{\tau}| : \overline{\sigma}[X := \overline{\tau}]}$$

$$\text{(cons)}\ \frac{}{\overline{\Gamma} \vdash c_k : \Pi \boldsymbol{X}.\ \overline{\boldsymbol{\theta}_k} \to d^{\widehat{i}}\ \boldsymbol{X}}\ \text{if } c_k \in \mathcal{C}(d) \text{ for some } d$$

$$\text{(case)}\ \frac{\begin{array}{c} c_k : \Pi \boldsymbol{X}.\ \overline{\boldsymbol{\theta}_k} \to d^{\widehat{i}}\ \boldsymbol{X} \\ \overline{\Gamma} \vdash e : d^{\widehat{s}}\ \overline{\boldsymbol{\tau}} \qquad \overline{\Gamma} \vdash e_k : \overline{\boldsymbol{\theta}_k}[\boldsymbol{X} := \overline{\boldsymbol{\tau}}, \imath := s] \to \overline{\sigma} \qquad (1 \le k \le n) \end{array}}{\begin{array}{l} \overline{\Gamma} \vdash \mathsf{case}_{|\overline{\sigma}|}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \cdots \mid c_n \Rightarrow e_n\} : \overline{\sigma} \\ \text{if } \mathcal{C}(d) = \{c_1, \ldots, c_n\} \end{array}}$$

$$\text{(rec)}\ \frac{\overline{\Gamma}, f : d^{\imath}\overline{\boldsymbol{\tau}} \to \overline{\theta} \vdash e : d^{\widehat{\imath}}\overline{\boldsymbol{\tau}} \to \overline{\theta}[\imath := \widehat{\imath}] \qquad \imath\ \mathsf{pos}\ \overline{\theta}}{\overline{\Gamma} \vdash (\mathsf{letrec}_{d|\overline{\boldsymbol{\tau}}| \to |\overline{\theta}|}\ f = e) : d^{s}\overline{\boldsymbol{\tau}} \to \overline{\theta}[\imath := s]}\ \text{if } \imath \notin \overline{\Gamma}, \overline{\boldsymbol{\tau}}$$

Fig. 5. Typing rules for $F^{\widehat{\ }}$

Strong normalization and coherence. System $F^{\widehat{\ }}$ enjoys the two crucial properties that fail for F^{rec}, namely strong normalizability of typable terms and coherence. Proofs are sketched in the next section, respectively in Cor. 3.29 and in Cor. 3.30. They both rely on a reducibility interpretation of $F^{\widehat{\ }}$ by saturated sets [11,18]. Let SN be the set of strongly normalizing terms. Hence SN is the least set of terms such that

$$\forall e.\quad (\forall e'.\quad e \to_{\beta\iota\gamma} e' \quad\Longrightarrow\quad e' \in \mathsf{SN}) \quad\Longrightarrow\quad e \in \mathsf{SN}$$

Theorem 3.20 (Strong normalization). *If* $\overline{\Gamma} \vdash e : \overline{\tau}$ *then* $e \in \mathsf{SN}$.

Theorem 3.21 (Coherence). *There is no term* e *such that* $\vdash e : \bot$.

3.4 A reducibility interpretation

In this section, we sketch the correctness proof of a reducibility semantics for $F^{\widehat{\ }}$. Our semantics is based on a variant of reducibility [12] called Tait's saturated sets [18], and will be used to prove both the strong normalization of typable terms and the coherence of the type system. We begin by the interpretation of stages, and turn to the model construction. We then state its correctness, from which we deduce strong normalization and coherence.

In the whole section, if f is a map from A to B, $a \in A$ and $b \in B$, then $f(a := b) : A \to B$ maps a to b and is equal to f everywhere else.

The stage model. Stages are interpreted by the ordinals used to build the interpretation of inductive types. While first-order inductive types can be interpreted by induction on $\mathbb{N}$, higher-order inductive types may require an induction on countable ordinals. Recall that $(\Omega, \leq_\Omega)$ denote the well-ordered set of countable ordinals and by $+_\Omega$ the usual ordinal addition on Ω.

Let $\widehat{\Omega} =_{\text{def}} \Omega \cup \{\Omega\}$. For all $\alpha \in \Omega$ and all $\beta \in \widehat{\Omega}$, let $\alpha < \beta$ iff ($\beta = \Omega$ or $\alpha <_\Omega \beta$).

Definition 3.22 (Interpretation of stages). *A* stage valuation *is a map π from $\mathcal{V}_{\mathcal{S}}$ to $\widehat{\Omega}$, and is extended to a* stage interpretation $(\!|.|\!)_\pi : \mathcal{S} \to \widehat{\Omega}$ *as follows:*

$$(\!|\iota|\!)_\pi = \pi(\iota) \qquad (\!|0|\!)_\pi = 0 \qquad (\!|\infty|\!)_\pi = \Omega \qquad (\!|\widehat{s}|\!)_\pi = \begin{cases} (\!|s|\!)_\pi + 1 & \text{if } (\!|s|\!)_\pi < \Omega \\ \Omega & \text{if } (\!|s|\!)_\pi = \Omega \end{cases}$$

Type interpretation. In this section, we define the type interpretation and prove its correctness. Our proof follows the pattern of [1]. We interpret types by saturated sets. It is convenient to define them by means of *elimination contexts*:

$$E[\,] \;::=\; [\,] \mid E[\,]\; e \mid E[\,]\;|\tau| \mid \mathsf{case}_{|\tau|}\; E[\,]\; \mathsf{of}\; \{\boldsymbol{c} \Rightarrow \boldsymbol{e}\}$$

Note that the hole $[\,]$ of $E[\,]$ never occurs under a binder. Thus $E[\,]$ can be seen as a term with one occurrence of a special variable $[\,]$. Therefore, we can define $E[e]$ as $E[\,][[\,] := e]$. The relation of *weak head $\beta\iota\gamma$-reduction* is defined as $E[e] \to_{\text{wh}} E[e']$ if and only if $e \succ_{\beta\iota\gamma} e'$.

Definition 3.23 (Saturated sets).
A set $S \subseteq \mathsf{SN}$ is saturated *($S \in \mathsf{SAT}$) if*

(SAT1) *$E[x] \in S$ for all $E[\,] \in \mathsf{SN}$ and all $x \in \mathcal{V}_{\mathcal{E}}$,*
(SAT2) *if $e \in \mathsf{SN}$ and $e \to_{wh} e'$ for some $e' \in S$ then $e \in S$.*

One can easily show that $\mathsf{SN} \in \mathsf{SAT}$ and that $\bigcap \mathcal{Y}, \bigcup \mathcal{Y} \in \mathsf{SAT}$ for all non-empty $\mathcal{Y} \subseteq \mathsf{SAT}$. One can also check that the *function space* on SAT, defined for $X, Y \in \mathsf{SAT}$ as:

$$X \to Y \;=_{\text{def}}\; \{e \mid \forall e'.\; e' \in X \implies e\, e' \in Y\}$$

returns a saturated set.

Because saturated sets are closed under non-empty intersections, one can define for each $X \subseteq \mathsf{SN}$ the smallest saturated set containing X, written $\overline{X}$. We let $\bot =_{\text{def}} \overline{\emptyset}$; it is easy to show that $\bot$ is the smallest element of SAT. The following properties precisely characterizes the membership of an expression to a saturated set.

Lemma 3.24.
(i) $\bot = \{e \in \mathsf{SN} \mid \exists E[\,], x.\quad e \to_{wh}^* E[x]\}$.
(ii) If $X \subseteq \mathsf{SN}$ then $\overline{X} = \bot \cup \{e \in \mathsf{SN} \mid e \to_{wh}^* X\}$.

The interpretation of types is defined in two steps. We first define the interpretation scheme of types, given an interpretation of datatypes. We then define the interpretation of datatypes.

Definition 3.25. *An* interpretation of datatypes *is a family* $(\mathcal{I}_d)_{d\in\mathcal{D}}$ *of functions* $\mathcal{I}_d : \mathsf{SAT}^{\mathsf{ar}(d)} \times \widehat{\Omega} \to \mathsf{SAT}$ *for each* $d \in \mathcal{D}$. *Given an interpretation of datatypes* $\mathcal{I}$, *a stage valuation* π *and a* type valuation $\xi : \mathcal{V}_{\mathcal{T}} \to \mathsf{SAT}$, *the* type interpretation $[\![.]\!]^{\mathcal{I}}_{\pi,\xi} : \underline{\mathcal{T}} \to \mathsf{SAT}$ *is defined by induction on types as follows*

$$\begin{aligned}
[\![X]\!]^{\mathcal{I}}_{\pi,\xi} &= \xi(X)\\
[\![\tau \to \sigma]\!]^{\mathcal{I}}_{\pi,\xi} &= [\![\tau]\!]^{\mathcal{I}}_{\pi,\xi} \to [\![\sigma]\!]^{\mathcal{I}}_{\pi,\xi}\\
[\![\Pi X.\ \tau]\!]^{\mathcal{I}}_{\pi,\xi} &= \left\{ e \ \mid\ \forall |\sigma| \in |\mathcal{T}|,\ \forall S \in \mathsf{SAT},\quad e\ |\sigma| \in [\![\tau]\!]^{\mathcal{I}}_{\pi,\xi(X:=S)} \right\}\\
[\![d^s \boldsymbol{\tau}]\!]^{\mathcal{I}}_{\pi,\xi} &= \mathcal{I}_d([\![\boldsymbol{\tau}]\!]^{\mathcal{I}}_{\pi,\xi}, (\!|s|\!)_\pi)
\end{aligned}$$

We now define the interpretation of inductive datatypes. Recall that they are defined in an ordered list $I_1, \dots, I_n$ of declarations (see Def. 2.25). Let us say that k is the *rank* of d if d is defined in I_k. The interpretation $(\mathcal{I}_d)_{d\in\mathcal{D}}$ is defined by induction on the rank, and for each $d \in \mathcal{D}$, the map $\mathcal{I}_d : \mathsf{SAT}^{\mathsf{ar}(d)} \times \widehat{\Omega} \to \mathsf{SAT}$ is defined by induction on $\widehat{\Omega}$.

Definition 3.26. *For all* $d \in \mathcal{D}$, *all* $\boldsymbol{S} \in \mathsf{SAT}^{\mathsf{ar}(d)}$ *and all* $\alpha \in \widehat{\Omega}$, *we define* $\mathcal{I}_d(\boldsymbol{S}, \alpha)$ *by induction on pairs* (k, α) *ordered by* $(<,<)_{lex}$, *where* k *is the rank of* d, *as follows:*

$$\begin{aligned}
\mathcal{I}_d(\boldsymbol{S}, 0) &= \bot\\
\mathcal{I}_d(\boldsymbol{S}, \alpha+1) &= \bigcup\{ c\,[\![\overline{\boldsymbol{\theta}}]\!]^{\mathcal{I}}_{\iota:=\alpha, \boldsymbol{X}:=\boldsymbol{S}} \mid c \in \mathcal{C}(d) \wedge \mathsf{Type}(c) = \Pi \boldsymbol{X}.\ \overline{\boldsymbol{\theta}} \to d^{\widehat{\iota}}\boldsymbol{X} \}\\
\mathcal{I}_d(\boldsymbol{S}, \lambda) &= \bigcup\{ \mathcal{I}_d(\boldsymbol{S}, \alpha) \mid \alpha < \lambda \} \qquad \textit{if } \lambda \textit{ is a limit ordinal}
\end{aligned}$$

where $c\,\boldsymbol{S} =_{def} \overline{\{ c\,|\boldsymbol{\tau}|\,\boldsymbol{a} \mid \boldsymbol{a} \in \boldsymbol{S} \ \wedge\ |\boldsymbol{\tau}| \in |\mathcal{T}| \}}$ *for all* $\boldsymbol{S} \in \mathsf{SAT}$.

Note that $\mathcal{I}_d(\boldsymbol{S}, \alpha+1)$ only uses $c[\![\overline{\boldsymbol{\theta}}]\!]^{\mathcal{I}}_{\iota:=\alpha,\boldsymbol{X}:=\boldsymbol{S}}$ with $c \in \mathcal{C}(d)$, which in turn only uses $\mathcal{I}_d(\boldsymbol{U}, \beta)$ with $(p,\beta)\,(<,<)_{\mathrm{lex}}\,(k, \alpha+1)$, where k (resp. p) is the rank of d (resp. d').

Now that we have an interpretation of inductive datatypes $(\mathcal{I}_d)_{d\in\mathcal{D}}$, we can interpret types as in Def. 3.25 using this interpretation of datatypes. It is convenient to denote $[\![.]\!]^{\mathcal{I}}_{\pi,\xi}$ by $[\![.]\!]_{\pi,\xi}$.

We gather in Fig. 6 some properties of $(\!|.|\!)_\pi$ and $[\![.]\!]_{\pi,\xi}$. The following Proposition states that each inductive datatype can be interpreted by a countable ordinal. This is crucial in order to deal with the rule (cons) in the proof of Thm. 3.28. The key-point is that for every *countable* $S \subseteq \Omega$, there is $\beta \in \Omega$ such that $\alpha < \beta$ for all $\alpha \in S$ [9].

Proposition 3.27. *For all* $d \in \mathcal{D}$ *and all* $\boldsymbol{S} \in \mathsf{SAT}^{\mathsf{ar}(d)}$, *there is an ordinal* $\alpha < \Omega$ *such that* $\mathcal{I}_d(\boldsymbol{S}, \alpha) = \mathcal{I}_d(\boldsymbol{S}, \beta)$ *for all* β *such that* $\alpha \le \beta \le \Omega$.

Substitution	$(\!\|p[\iota := s]\|\!)_\pi$	$=$	$(\!\|p\|\!)_{\pi(\iota:=(\!\|s\|\!)_\pi)}$
	$[\![\overline{\tau}[\iota := s]]\!]_{\pi,\xi}$	$=$	$[\![\overline{\tau}]\!]_{\pi(\iota:=(\!\|s\|\!)_\pi),\xi}$
	$[\![\overline{\tau}[X := \overline{\sigma}]]\!]_{\pi,\xi}$	$=$	$[\![\overline{\tau}]\!]_{\pi,\xi(X:=[\![\overline{\sigma}]\!]_{\pi,\xi})}$
Stage monotony	$\alpha \leq \beta$	$\Rightarrow$	$\mathcal{I}_d(\boldsymbol{S}, \alpha) \subseteq \mathcal{I}_d(\boldsymbol{S}, \beta)$
	$\alpha \leq \beta \ \wedge \ \iota \ \mathsf{pos}\ \theta$	$\Rightarrow$	$[\![\theta]\!]_{\pi(\iota:=\alpha),\xi} \subseteq [\![\theta]\!]_{\pi(\iota:=\beta),\xi}$
	$\alpha \leq \beta \ \wedge \ \iota \ \mathsf{neg}\ \theta$	$\Rightarrow$	$[\![\theta]\!]_{\pi(\iota:=\beta),\xi} \subseteq [\![\theta]\!]_{\pi(\iota:=\alpha),\xi}$
Substage soundness	$s \leq p$	$\Rightarrow$	$(\!\|s\|\!)_\pi \leq (\!\|p\|\!)_\pi$
Subtyping soundness	$\overline{\tau} \sqsubseteq \overline{\sigma}$	$\Rightarrow$	$[\![\overline{\tau}]\!]_{\pi,\xi} \subseteq [\![\overline{\sigma}]\!]_{\pi,\xi}$

Fig. 6. Properties of the type interpretation

Correctness of the interpretation. As usual, soundness is shown by induction on typing derivations. Given $\pi : \mathcal{V}_\mathcal{S} \to \widehat{\Omega}$, $\xi : \mathcal{V}_\mathcal{T} \to \mathsf{SAT}$ and $\rho : (\mathcal{V}_\mathcal{E} \to \mathcal{E}) \uplus (\mathcal{V}_\mathcal{T} \to |\mathcal{T}|)$, we let $(\pi, \xi, \rho) \models \overline{\Gamma}$ if and only if $\rho(x) \in [\![\overline{\Gamma}(x)]\!]_{\pi,\xi}$ for all $x \in \mathsf{dom}(\overline{\Gamma})$.

Theorem 3.28 (Typing soundness). *If $\overline{\Gamma} \vdash e \ : \ \overline{\tau}$, then $e\rho \in [\![\overline{\tau}]\!]_{\pi,\xi}$ for all π, ξ, ρ such that $(\pi, \xi, \rho) \models \overline{\Gamma}$.*

We deduce the strong normalization of typable terms and the coherence of the system.

Corollary 3.29 (Strong normalization). *If $\overline{\Gamma} \vdash e \ : \ \overline{\tau}$ then $e \in \mathsf{SN}$.*

Proof. Apply Thm. 3.28 with any π and ξ, and with the identity substitution for ρ. We thus have $(\pi, \xi, \rho) \models \overline{\Gamma}$, hence $e = e\rho \in [\![\overline{\tau}]\!]_{\pi,\xi} \subseteq \mathsf{SN}$. □

Corollary 3.30 (Coherence). *There is no term e such that $\vdash e \ : \ \Pi X.\ X$.*

Proof. Assume that $\vdash e \ : \ \Pi X.\ X$. Note that e must be a closed term, i.e. $\mathsf{FV}_\mathcal{E}(e) = \emptyset$. By Thm. 3.28, we have $e \in [\![\Pi X.\ X]\!]$. Therefore, for all $\tau \in \mathcal{T}$, we have $e\,\tau \in \bot$. By Lem. 3.24.(i), $e\,\tau$ reduces to a term of the form $E[x]$ for some $x \in \mathcal{V}_\mathcal{E}$. But $E[x]$ is an open term, which contradicts the fact that $e\,\tau$ is closed. □

4 Type inference

The purpose of this section is to present a sound and complete algorithm that infers size annotations for $F^\frown$. One particularity of our algorithm is to return concise results, in the form of constrained types (C, τ) where τ is a sized type and C is a set of stage inequalities. Restricting such constrained types is beneficial for two reasons: first of all, sets of stage inequalities are always satisfiable (by mapping all stage variables to ∞), hence a term e is typable whenever the inference algorithm does not return an error. Second of all, the algorithm avoids the use of disjunction, which makes satisfiability of constraints complex. Disjunctive typings are avoided by requiring recursive definitions to carry tags that

identify which positions are meant to carry a size annotation related to the size of the recursive argument. Consider the following expression:

$$(\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{o})$$

It may be given the types $\mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\imath}$ and $\mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\widehat{\jmath}}$. If we restrict to conjunctive constrained types as discussed above, it is impossible to obtain a more general type that subsumes both types. In order to achieve more general types without using disjunctive constrained types, we tag positions whose size must use the same base size variable as the recursive argument with a special symbol $\star$. These tags will be used by the inference algorithm to separate between stage variables that must pertain to the same hierarchy as the stage variable of the recursive arguments, and those that must not. In effect, the inference algorithm will produce the following results:

$$(\mathsf{letrec}_{\mathsf{Nat}^\star\to\mathsf{Nat}^\star}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{o}) : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\imath}$$
$$(\mathsf{letrec}_{\mathsf{Nat}^\star\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{o}) : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\widehat{\jmath}}$$

For clarity, the inference algorithm is defined together with a checking algorithm, that takes as additional argument a candidate type and verifies that it is a correct instance of the computed type. Since we start from terms that do not carry size annotations, both algorithms must generate size variables that are used to build the size annotations that decorate the inferred or checked types. In order to guarantee that they only introduce fresh size variables, the algorithms take an auxiliary parameter V, that represents the set of size variables that have been used elsewhere, and return an extended set V' that includes V and the new stage variables that were used for the expression under evaluation. Therefore,

- the type inference algorithm $\texttt{Infer}(V, \overline{\Gamma}, e)$ takes as input a context $\overline{\Gamma}$, an expression e and a set of size variables V s.t. $\mathsf{FV}(\overline{\Gamma}) \subseteq V$, and returns a tuple $(V', C, \overline{\tau})$ where $\overline{\tau}$ is an annotated type, C is a constraint, and V' is a set of size variables s.t. $\mathsf{FV}(C, \overline{\tau}) \cup V \subseteq V'$;
- the type checking algorithm takes as additional input a candidate type $\overline{\tau}$; then $\texttt{Check}(V, \overline{\Gamma}, e, \overline{\tau})$ returns a pair (V', C) s.t. $\mathsf{FV}(C, \overline{\tau}) \cup V \subseteq V'$ and ensuring that e has type $\rho\overline{\tau}$ in environment $\overline{\Gamma}$ provided that ρ is a solution for C.

The algorithm is sound and complete.

Proposition 4.1 (Soundness and completeness of `Check` and `Infer`).

- *Soundness:*
  (i) If $\texttt{Check}(V, \overline{\Gamma}, e, \overline{\tau}) = (V', C)$ *then* $\rho(\overline{\Gamma}) \vdash e : \rho(\overline{\tau})$ *for all* ρ *s.t.* $\rho \models C$.
  (ii) If $\texttt{Infer}(V, \overline{\Gamma}, e) = (V', C, \overline{\tau})$ *then* $\rho(\overline{\Gamma}) \vdash e : \rho(\overline{\tau})$ *for all* ρ *s.t.* $\rho \models C$.
- *Completeness:*
  (i) If $\rho(\overline{\Gamma}) \vdash e : \rho\overline{\tau}$ *and* $\mathsf{FV}_{\mathcal{I}}(\overline{\Gamma}, \overline{\tau}) \subseteq V$ *then there exist* V', C, ρ' *such that* $\texttt{Check}(V, \overline{\Gamma}, e, \overline{\tau}) = (V', C)$ *and* $\rho' \models C$ *and* $\rho =_V \rho'$.

(ii) If $\rho(\overline{\Gamma}) \vdash e \ : \ \overline{\theta}$ *and* $\mathsf{FV}_{\mathcal{J}}(\overline{\Gamma}) \subseteq V$ *there exist* $V', C, \overline{\tau}, \rho'$ *such that* $\mathtt{Infer}(V, \overline{\Gamma}, e) = (V', C, \overline{\tau})$ *and* $\rho' \models C$ *and* $\rho'(\overline{\tau}) \sqsubseteq \overline{\theta}$ *and* $\rho' =_V \rho$
where $\rho =_V \rho'$ *means that* $\rho(\alpha) = \rho'(\alpha)$ *for all* $\alpha \in V$.

Note that every conjunctive constraint has a solution. Therefore, if the inference algorithm is successful on input $(\overline{\Gamma}, e)$, i.e. $\mathtt{Infer}(\overline{\Gamma}, e) = (C, \tau)$, one can find ρ such that $\rho \models C$. Therefore, by soundness $\rho\overline{\Gamma} \vdash e \ : \ \rho\tau$.

The crux of the algorithm is the rule for recursive definitions, which must check the existence of solutions for more elaborate constraints, in which one can also declare that a stage variable $\imath$ can only be interpreted as itself (in effect it amounts to restrict ourselves to substitutions ρ such that $\rho(\imath) = \imath$), and that a stage s cannot be in the same hierarchy as a fixed stage variable $\imath$. For such systems, the existence of a solution is not always guaranteed, and we shall therefore device a dedicated algorithm to verify whether a solution exists.

Outline. Type inference is presented step by step. We begin by recalling the straightforward type inference algorithm of system F in Sect. 4.1. Then, in Sect. 4.2, we discuss the effect of adding sized inductive datatypes, with subtyping and case analysis but without recursion. We concentrate on the subtyping relation and the necessity to infer stage annotations for λ-abstractions, type applications and case analysis. At this level, we do not need to be precise about freshness conditions. Its is sufficient to work with

- the judgment $C \ ; \ \overline{\Gamma} \vdash e \uparrow \overline{\tau}$, which stands for $\mathtt{Infer}(\overline{\Gamma}, e) = (C, \overline{\tau})$, and
- the judgment $C \ ; \ \overline{\Gamma} \vdash e \downarrow \overline{\tau}$, which stands for $\mathtt{Check}(\overline{\Gamma}, e, \overline{\tau}) = C$.

In Sect. 4.3, we informally discuss the way we handle recursive definitions. The main point is an auxiliary algorithm called `RecCheck`, which is informally presented and justified. Finally, in Sect. 4.4 we discuss the full type inference algorithm of $F\widehat{\ }$, as presented in [6] and using the functions $\mathtt{Infer}(V, \overline{\Gamma}, e)$ and $\mathtt{Check}(V, \overline{\Gamma}, e, \overline{\tau})$.

4.1 Preliminaries: Type inference in system *F*

In the Church style system F, as presented in Sect. 2.1, type inference is trivial, because the typing derivation of a term is uniquely determined by the shape of a term. Hence, the type inference algorithm is directly given by the typing rules read bottom-up:

- The type of x in the context Γ is $\Gamma(x)$ if and only if $x \in \mathsf{dom}(\Gamma)$.
- The type of $\lambda x : \tau.\ e$ in Γ is $\tau \to \sigma$ if and only if the type of e in $\Gamma, x \ : \ \tau$ is σ;
- The type of $e\, e'$ in Γ is σ if and only if there is a (necessarily unique) type τ such that the type of e (resp. e') in Γ is $\tau \to \sigma$ (resp. τ).
- The type of $\Lambda X.\ e$ in Γ is $\Pi X.\ \tau$ if and only if the type of e in Γ is τ and $X \notin \Gamma$.
- The type of $e\,\tau$ in Γ is $\sigma[X := \tau]$ if and only if the type of e in Γ is $\Pi X.\ \sigma$.

4.2 Adding sized inductive datatypes

We present type inference in system F enriched with the sized typing rules for constructors, case analysis, and subtyping. Recall that sized types, defined in Def. 3.4, are given by the abstract syntax:

$$\overline{\mathcal{T}} \;::=\; \mathcal{V}_{\mathcal{T}} \;\mid\; \overline{\mathcal{T}} \to \overline{\mathcal{T}} \;\mid\; \Pi \mathcal{V}_{\mathcal{T}}.\, \overline{\mathcal{T}} \;\mid\; \mathcal{D}^{\mathcal{S}}\, \overline{\boldsymbol{\mathcal{T}}}$$

and that the typing rule (cons), (case) and (sub) are the following:

$$\text{(cons)}\ \frac{}{\overline{\Gamma} \vdash c_k \;:\; \Pi \boldsymbol{X}.\, \overline{\boldsymbol{\theta}}_{\boldsymbol{k}} \to d^{\hat{\imath}}\, \boldsymbol{X}} \quad \text{if } c_k \in \mathcal{C}(d) \text{ for some } d \qquad \text{(sub)}\ \frac{\overline{\Gamma} \vdash e \;:\; \overline{\sigma} \qquad \overline{\sigma} \sqsubseteq \overline{\tau}}{\overline{\Gamma} \vdash e \;:\; \overline{\tau}}$$

$$\text{(case)}\ \frac{\begin{array}{c} c_k : \Pi \boldsymbol{X}.\, \overline{\boldsymbol{\theta}}_{\boldsymbol{k}} \to d^{\hat{\imath}}\, \boldsymbol{X} \\ \overline{\Gamma} \vdash e \;:\; d^{\hat{s}}\, \overline{\boldsymbol{\tau}} \qquad \overline{\Gamma} \vdash e_k \;:\; \overline{\boldsymbol{\theta}}_{\boldsymbol{k}}[\boldsymbol{X} := \overline{\boldsymbol{\tau}}, \imath := s] \to \overline{\sigma} \qquad (1 \le k \le n) \end{array}}{\overline{\Gamma} \vdash \mathsf{case}_{|\overline{\sigma}|}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \cdots \mid c_n \Rightarrow e_n\} \;:\; \overline{\sigma}} \quad \text{if } \mathcal{C}(d) = \{c_1, \ldots, c_n\}$$

In this section, it is not mandatory for the informal discussion to be very precise on freshness conditions. Moreover, we work step by step, and progressively introduce the features of the type inference algorithm. Therefore, instead of using $\texttt{Infer}(\overline{\Gamma}, e)$ and $\texttt{Check}(\overline{\Gamma}, e, \overline{\tau})$, we start with two simple unconstrained judgments $\overline{\Gamma} \vdash e \uparrow \overline{\tau}$ for type inference and $\overline{\Gamma} \vdash e \downarrow \overline{\tau}$ for type-checking. we will then introduce constraints, which lead us to

- the judgment $C \;;\; \overline{\Gamma} \vdash e \uparrow \overline{\tau}$, which stands for $\texttt{Infer}(\overline{\Gamma}, e) = (C, \overline{\tau})$, and
- the judgment $C \;;\; \overline{\Gamma} \vdash e \downarrow \overline{\tau}$, which stands for $\texttt{Check}(\overline{\Gamma}, e, \overline{\tau}) = C$.

Inference of size annotations. We now discuss the case of λ-abstractions. The natural inference rule would be

$$\frac{\overline{\Gamma}, x \;:\; \overline{\tau} \vdash e \uparrow \overline{\sigma}}{\overline{\Gamma} \vdash \lambda x : |\overline{\tau}|.\; e \uparrow \overline{\tau} \to \overline{\sigma}} \tag{4}$$

That is, we infer the type $\overline{\tau} \to \overline{\sigma}$ for $\lambda x : |\overline{\tau}|.\; e$ if we can infer the type $\overline{\sigma}$ for e in a context where the variable x is given the type $\overline{\tau}$. In other words, we have to infer the type $\overline{\tau}$ from its erasure $|\overline{\tau}|$. The difficulty is that the type $\overline{\tau}$ may depend on e. For instance, with

$$\overline{\Gamma} =_{\text{def}} f : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\imath}, g : \mathsf{Nat}^{j} \to \mathsf{Nat}^{j}$$

we have

$$\frac{\overline{\Gamma}, x \;:\; \mathsf{Nat}^{\imath} \vdash f\, x \;:\; \mathsf{Nat}^{\imath}}{\overline{\Gamma} \vdash \lambda x : \mathsf{Nat}.\;\; f\, x \;:\; \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\imath}} \quad \text{and} \quad \frac{\overline{\Gamma}, x \;:\; \mathsf{Nat}^{j} \vdash g\, x \;:\; \mathsf{Nat}^{\imath}}{\overline{\Gamma} \vdash \lambda x : \mathsf{Nat}.\;\; g\, x \;:\; \mathsf{Nat}^{j} \to \mathsf{Nat}^{\imath}}$$

where the typing of $\lambda x : \mathsf{Nat}.\;\; f\, x$ and $\lambda x : \mathsf{Nat}.\;\; g\, x$ require two different annotations of Nat. A solution is to proceed similarly as in Hindley-Milner type inference for ML-like languages. We perform type inference in a system whose stage expressions feature *inference stage variables* $\mathcal{V}_{\mathcal{I}} = \{\alpha, \beta, \ldots\}$.

Definition 4.2 (Inference stages). *The set* $\mathcal{S}_\mathcal{J} = \{s, r, \dots\}$ *of* inference stage expressions *is given by the abstract syntax:*

$$\mathcal{S}_\mathcal{J} \ ::= \ \mathcal{V}_\mathcal{S} \ \mid \ \mathcal{V}_\mathcal{J} \ \mid \ \widehat{\mathcal{S}_\mathcal{J}} \ \mid \ \infty$$

The substitution $s[\alpha := r]$ *of the inference stage variable* α *for* r *in* s *is defined in the obvious way.*

Then, each type $\sigma \in \mathcal{T}$ can be systematically annotated with inference variables. This is performed by a function $\mathtt{Annot}$ satisfying the following specification. If V is a set of inference stage variables and $\sigma \in \mathcal{T}$ is a type, then $\mathtt{Annot}(\sigma, V)$ returns a pair $(\overline{\sigma}, V')$ such that

- $|\overline{\sigma}| = \sigma$,
- each occurrence of an inductive datatype in $\overline{\sigma}$ is annotated with a distinct inference stage variable $\alpha \notin V$ (so that α occurs at most once in $\overline{\sigma}$),
- $V' = V \cup \mathsf{FV}_\mathcal{J}(\overline{\sigma})$.

For instance, $\mathtt{Annot}(\mathsf{Nat} \to \mathsf{Nat}, \{\alpha_1, \alpha_2\}) = (\mathsf{Nat}^{\alpha_3} \to \mathsf{Nat}^{\alpha_4}, \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\})$. For the moment, we do not use the set of variables V' produced by $\mathtt{Annot}$. If we add these systematic annotations to the rule (4), we obtain

$$\text{(abs)} \ \frac{(\overline{\tau}, V) := \mathtt{Annot}(\tau, \mathsf{FV}_\mathcal{J}(\overline{\Gamma})) \qquad \overline{\Gamma}, x \ : \ \overline{\tau} \vdash e \uparrow \overline{\sigma}}{\overline{\Gamma} \vdash \lambda x : \tau.\ e \uparrow \overline{\tau} \to \overline{\sigma}}$$

In such system, the intended semantics of type inference and type checking can be phrased by the property that for all substitution $\rho : \mathcal{V}_\mathcal{J} \to \mathcal{S}_\mathcal{J}$ we have

$$\begin{array}{lcl} \Gamma \vdash e \uparrow \overline{\tau} & \Longrightarrow & \Gamma\rho \vdash e \ : \ \overline{\tau}\rho \\ \Gamma \vdash e \downarrow \overline{\tau} & \Longrightarrow & \Gamma\rho \vdash e \ : \ \overline{\tau}\rho \end{array}$$

Checking subtyping derivations. Type inference with a rule like the above does not work directly: we have to take subtyping into account more seriously. Let us look at the type inference derivation of $\lambda x : \mathsf{Nat}.\ f\,x$ in the context $f : \mathsf{Nat}^\imath \to \mathsf{Nat}^\infty$. We would have a derivation of the form

$$\frac{f : \mathsf{Nat}^\imath \to \mathsf{Nat}^\infty, x : \mathsf{Nat}^\alpha \vdash f\,x \uparrow ? \qquad (\mathsf{Nat}^\alpha, V) = \mathtt{Annot}(\mathsf{Nat}, \emptyset)}{f : \mathsf{Nat}^\imath \to \mathsf{Nat}^\infty \vdash \lambda x : \mathsf{Nat}.\ f\,x \uparrow ?} \tag{5}$$

but we get stuck because this would require $\mathsf{Nat}^\alpha \sqsubseteq \mathsf{Nat}^\imath$, which does not hold.

In other words, once we generate sized types featuring stage inference variables, we have to adapt our way of handling subtyping. In Sect. 3.2, we have seen that the substage relation $s \leq r$ (defined in Def. 3.3) leads to the subtyping relation $\overline{\tau} \sqsubseteq \overline{\sigma}$ (defined in Def. 3.6). For type checking, we go the other way: starting from a subtyping assertion $\overline{\tau} \sqsubseteq \overline{\sigma}$, we generate a conjunction of substage assertions $s_1 \leq r_1, \dots, s_n \leq r_n$, which holds if and only if $\overline{\tau} \sqsubseteq \overline{\sigma}$ is derivable.

Definition 4.3 (Constraints).

(i) A constraint *is either the false constraint* $\bot$ *or a set of inference stage expressions inequalities* $\{s_1 \leq r_1, \ldots, s_n \leq r_n\}$.

(ii) *A substitution* $\rho : \mathcal{V}_\jmath \to \mathcal{S}_\jmath$ *satisfies a constraint* C*, notation* $\rho \models C$*, if and only if* $C \neq \bot$ *and* $s\rho \leq r\rho$ *is derivable using the rules of Def. 3.3 for all* $s \leq r \in C$.

(iii) *A subtyping assertion* $\overline{\tau} \sqsubseteq \overline{\sigma}$ *generates a constraint* $(\!|\overline{\tau} \sqsubseteq \overline{\sigma}|\!)$ *defined as follows*

$$
\begin{array}{rcll}
(\!|X \sqsubseteq X|\!) & =_{def} & \emptyset & \\
(\!|\overline{\tau}_2 \to \overline{\tau}_1 \sqsubseteq \overline{\sigma}_2 \to \overline{\sigma}_1|\!) & =_{def} & (\!|\overline{\sigma}_2 \sqsubseteq \overline{\tau}_2|\!) \cup_\bot (\!|\overline{\tau}_1 \sqsubseteq \overline{\sigma}_1|\!) & \\
(\!|\Pi X.\ \overline{\tau} \sqsubseteq \Pi X.\ \overline{\sigma}|\!) & =_{def} & (\!|\overline{\tau} \sqsubseteq \overline{\sigma}|\!) & \\
(\!|d^s\, \overline{\boldsymbol{\tau}} \sqsubseteq d^r\, \overline{\boldsymbol{\sigma}}|\!) & =_{def} & \{s \leq r\} \cup_\bot (\!|\overline{\boldsymbol{\tau}} \sqsubseteq \overline{\boldsymbol{\sigma}}|\!) & \\
(\!|\overline{\tau} \sqsubseteq \overline{\sigma}|\!) & =_{def} & \bot & \textit{in all other cases}
\end{array}
$$

where

$$
C_1 \cup_\bot C_2 \quad =_{def} \quad \begin{cases} \bot & \textit{if } C_1 = \bot \textit{ or } C_2 = \bot \\ C_1 \cup C_2 & \textit{otherwise} \end{cases}
$$

We write $s_1 \leq r_1, \ldots, s_n \leq r_n$ instead of $\{s_1 \leq r_1, \ldots, s_n \leq r_n\}$. Note that we have $\rho \models \emptyset$ for all $\rho : \mathcal{V}_\jmath \to \mathcal{S}_\jmath$, and that the empty substitution does not satisfies the false constraint $\bot$. The satisfaction of constraints generated by subtyping assertions corresponds exactly to the derivability of subtyping judgments.

Proposition 4.4. $\rho \models (\!|\overline{\tau} \sqsubseteq \overline{\sigma}|\!)$ *if and only if* $\overline{\tau}\rho \sqsubseteq \overline{\sigma}\rho$.

Example 4.5.

(i) The assertion $\mathsf{Nat}^\infty \sqsubseteq \mathsf{Bool}^\infty$ generates the constraint $\bot$. It follows that $\mathsf{Nat}^\infty \sqsubseteq \mathsf{Bool}^\infty$ is not derivable.

(ii) The assertion $\mathsf{Nat}^\imath \sqsubseteq \mathsf{Nat}^\infty$ generates the constraint $\{\imath \leq \infty\}$. The inequality $\imath \leq \infty$ is derivable, hence $\mathsf{Nat}^\imath \sqsubseteq \mathsf{Nat}^\infty$ is also derivable.

(iii) The assertion $\mathsf{Nat}^\imath \sqsubseteq \mathsf{Nat}^j$ generates the constraint $\{\imath \leq j\}$. The inequality $\imath \leq j$ is not derivable, hence $\mathsf{Nat}^\imath \sqsubseteq \mathsf{Nat}^j$ is not derivable. □

Inference rules. We have to adapt type inference to take into account the constraints generated by subtyping. We now consider judgments of the form

$$
C \,;\, \overline{\Gamma} \vdash e \uparrow \overline{\tau} \qquad \text{and} \qquad C \,;\, \overline{\Gamma} \vdash e \downarrow \overline{\tau}
$$

where C is a constraint. The constraints are generated by subtyping

$$
(\text{sub})\ \frac{C \,;\, \overline{\Gamma} \vdash e \uparrow \overline{\tau}}{C \cup_\bot (\!|\overline{\tau} \sqsubseteq \overline{\sigma}|\!) \,;\, \overline{\Gamma} \vdash e \downarrow \overline{\sigma}}
$$

and transmitted by the other rules. In all rules, constraints have to be read top-bottom. The system is presented in Fig. 7, and its correctness is stated the following Proposition. The rule (cons) and (case) are commented in the next paragraph.

Proposition 4.6 (Correctness).
(i) If $C \,;\, \overline{\Gamma} \vdash e \uparrow \overline{\tau}$ *and* $\rho \models C$ *then* $\overline{\Gamma}\rho \vdash e \;:\; \overline{\tau}\rho$
(ii) If $C \,;\, \overline{\Gamma} \vdash e \downarrow \overline{\tau}$ *and* $\rho \models C$ *then* $\overline{\Gamma}\rho \vdash e \;:\; \overline{\tau}\rho$

Let us now look at the derivation (5). We have

$$\frac{\emptyset \,;\, f : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\infty}, x : \mathsf{Nat}^{\alpha} \vdash x \uparrow \mathsf{Nat}^{\alpha}}{\alpha \leq \imath \,;\, f : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\infty}, x : \mathsf{Nat}^{\alpha} \vdash x \downarrow \mathsf{Nat}^{\imath}}$$

and it follows that

$$\frac{\alpha \leq \imath \,;\, f : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\infty}, x : \mathsf{Nat}^{\alpha} \vdash f\, x \uparrow \mathsf{Nat}^{\infty} \qquad (\mathsf{Nat}^{\alpha}, V) = \mathtt{Annot}(\mathsf{Nat}, \emptyset)}{\alpha \leq \imath \,;\, f : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\infty} \vdash \lambda x : \mathsf{Nat}.\;\; f\, x \uparrow \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\infty}}$$

By Prop. 4.6.(i), we get $f : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\infty} \vdash \lambda x : \mathsf{Nat}.\;\; f\, x \;:\; \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\infty}$.

Inductive datatypes. We now focus on the type inference rules (cons) and (case) that correspond respectively to the introduction and to the elimination of inductive datatypes. The only particular feature of the type inference rule for constructors is that it introduces a fresh inference stage variable:

$$\text{(cons)}\ \frac{c_k : \Pi \boldsymbol{X}.\, \overline{\boldsymbol{\theta}_{\boldsymbol{k}}} \to d^{\widehat{\imath}}\, \boldsymbol{X}}{\emptyset \,;\, \overline{\Gamma} \vdash c_k \uparrow \Pi \boldsymbol{X}.\, \overline{\boldsymbol{\theta}_{\boldsymbol{k}}}[\imath := \alpha] \to d^{\widehat{\alpha}}\, \boldsymbol{X}}\ \text{if } \alpha \notin \mathcal{V}_{\jmath}(\overline{\Gamma}) \text{ and } c_k \in \mathcal{C}(d) \text{ for some } d$$

Consider now the type inference rule (case) for case-analysis:

$$\frac{\begin{array}{lll} C \,;\, \overline{\Gamma} \vdash e \uparrow d^{s}\, \overline{\boldsymbol{\tau}} & \alpha \notin \mathcal{V}_{\jmath}(\overline{\Gamma}) & (\overline{\sigma}, V) = \mathtt{Annot}(\sigma, \mathcal{V}_{\jmath}(\overline{\Gamma}) \cup \{\alpha\}) \\ c_k : \Pi \boldsymbol{X}.\, \overline{\boldsymbol{\theta}_{\boldsymbol{k}}} \to d^{\widehat{\imath}}\, \boldsymbol{X} & C_k \,;\, \overline{\Gamma} \vdash e_k \downarrow \overline{\boldsymbol{\theta}_{\boldsymbol{k}}}[\boldsymbol{X} := \overline{\boldsymbol{\tau}}, \imath := \alpha] \to \overline{\sigma} & (1 \leq k \leq n) \end{array}}{\{s \leq \widehat{\alpha}\} \cup_{\perp} C \cup_{\perp} (\bigcup_{\perp} C_k) \;\,;\, \overline{\Gamma} \vdash \mathsf{case}_{\sigma}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \cdots \mid c_n \Rightarrow e_n\} \uparrow \overline{\sigma}}$$

where $\mathcal{C}(d) = \{c_1, \ldots, c_n\}$

For simplicity, we consider a simple case where the output type σ does not contain any datatype (hence $\overline{\sigma} = \sigma$) and where the datatype d subject of the case-analysis is the type of natural numbers. Assume that we want to infer the type of $\mathsf{case}_{\sigma}\ e\ \mathsf{of}\ \{0 \Rightarrow e_{\mathsf{o}} \mid \mathsf{s} \Rightarrow e_{\mathsf{s}}\}$ in the context $\overline{\Gamma}$. First, we infer the type of the subject e of the case-analysis, and we get a constraint C such that $C \,;\, \overline{\Gamma} \vdash e \uparrow \mathsf{Nat}^{s}$. Now, recall the typing rule case imposes that the subject can be typed with a stage of the form $\widehat{r}$, and that in our case, the branch e_{s} corresponding to the constructor s has type $\mathsf{Nat}^{r} \to \sigma$:

$$\frac{\overline{\Gamma} \vdash e \;:\; \mathsf{Nat}^{\widehat{r}} \qquad \overline{\Gamma} \vdash e_{\mathsf{o}} \;:\; \sigma \qquad \overline{\Gamma} \vdash e_{\mathsf{s}} \;:\; \mathsf{Nat}^{r} \to \sigma}{\overline{\Gamma} \vdash \mathsf{case}_{\sigma}\ e\ \mathsf{of}\ \{0 \Rightarrow e_{\mathsf{o}} \mid \mathsf{s} \Rightarrow e_{\mathsf{s}}\} \;:\; \sigma}$$

We express this by using a fresh stage variable α together with a constraint $s \leq \widehat{\alpha}$, and we typecheck the branch e_{s} against the type $\mathsf{Nat}^{\alpha} \to \sigma$. This gives:

$$\frac{C \,;\, \overline{\Gamma} \vdash e \uparrow \mathsf{Nat}^{s} \qquad C_{\mathsf{o}} \,;\, \overline{\Gamma} \vdash e_{\mathsf{o}} \downarrow \sigma \qquad C_{\mathsf{s}} \,;\, \overline{\Gamma} \vdash e_{\mathsf{s}} \downarrow \mathsf{Nat}^{\alpha} \to \sigma}{\{s \leq \widehat{\alpha}\} \cup_{\perp} C \cup_{\perp} C_{\mathsf{o}} \cup_{\perp} C_{\mathsf{s}} \,;\, \overline{\Gamma} \vdash \mathsf{case}_{\sigma}\ e\ \mathsf{of}\ \{0 \Rightarrow e_{\mathsf{o}} \mid \mathsf{s} \Rightarrow e_{\mathsf{s}}\} \uparrow \sigma}$$

where $\alpha \notin \mathcal{V}_{\jmath}(\overline{\Gamma})$

Example 4.7. Let us look at the following example, where $V \in \mathcal{V}_{\mathcal{T}}$:

$$? \ ; \ v : V,\ f : \mathsf{Nat}^{\alpha} \to V,\ y : \mathsf{Nat}^{\beta} \vdash \mathsf{case}_V\ y\ \mathsf{of}\ \{0 \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}.\ f\ z\} \uparrow ?$$

Since $\gamma \leq \alpha \ ; \ v : V,\ f : \mathsf{Nat}^{\alpha} \to V,\ y : \mathsf{Nat}^{\beta},\ z : \mathsf{Nat}^{\gamma} \vdash f\ z \uparrow V$, we deduce

$$\gamma \leq \alpha,\ \alpha \leq \gamma \ ; \ v : V,\ f : \mathsf{Nat}^{\alpha} \to V,\ y : \mathsf{Nat}^{\beta} \vdash \lambda z : \mathsf{Nat}.\ f\ z \downarrow \mathsf{Nat}^{\alpha} \to V$$

It follows that

$$\begin{array}{l} \gamma \leq \alpha,\ \alpha \leq \gamma,\ \beta \leq \widehat{\alpha}\ ; \\ v : V,\ f : \mathsf{Nat}^{\alpha} \to V,\ y : \mathsf{Nat}^{\beta} \vdash \mathsf{case}_V\ y\ \mathsf{of}\ \{0 \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}.\ f\ z\} \uparrow V \end{array}$$

There are two things to note about the constraint $\gamma \leq \alpha, \alpha \leq \gamma, \beta \leq \widehat{\alpha}$. First, it contains a variable γ that appears nowhere else in the sequent. It is in fact the inference stage variable that corresponds to the bound variable z. Hence, if the types recorded under abstractions were annotated by stages, we would have:

$$\begin{array}{l} \gamma \leq \alpha,\ \alpha \leq \gamma,\ \beta \leq \widehat{\alpha}\ ; \\ v : V,\ f : \mathsf{Nat}^{\alpha} \to V,\ y : \mathsf{Nat}^{\beta} \vdash \mathsf{case}_V\ y\ \mathsf{of}\ \{0 \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}^{\gamma}.\ f\ z\} \uparrow V \end{array}$$

Second, this constraint implies $\alpha = \gamma$, hence the type inference judgment above can be simplified to

$$\beta \leq \widehat{\alpha} \ ; \ v : V,\ f : \mathsf{Nat}^{\alpha} \to V,\ y : \mathsf{Nat}^{\beta} \vdash \mathsf{case}_V\ y\ \mathsf{of}\ \{0 \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}^{\alpha}.\ f\ z\} \uparrow V$$

Such notations are very convenient to write type inference judgments. □

4.3 Checking the correctness of recursive definitions

We now discuss the typing rule of fixpoints. For the sake of simplicity, we assume that d is an inductive datatype without parameters and that $\imath \notin \overline{\theta}$. In this case, the typing rule of fixpoints becomes

$$(\text{rec})\ \frac{\overline{\Gamma}, f\ :\ d^{\imath} \to \overline{\theta} \vdash e\ :\ d^{\widehat{\imath}} \to \overline{\theta}}{\overline{\Gamma} \vdash (\mathsf{letrec}_{d \to |\overline{\theta}|}\ f = e)\ :\ d^{s} \to \overline{\theta}}\ \text{ if } \imath \notin \overline{\Gamma}, \overline{\theta}$$

Consider now that rule from the point of view of type inference. If we start from

$$? \ ; \ \overline{\Gamma} \vdash (\mathsf{letrec}_{d \to \theta}\ f = e) \uparrow ?$$

then we have to compute a constraint C^{Rec} and a type $d^{\alpha} \to \overline{\theta}$ such that $|\overline{\theta}| = \theta$ and

$$C^{\mathsf{Rec}} \ ; \ \overline{\Gamma} \vdash (\mathsf{letrec}_{d \to \theta}\ f = e) \uparrow d^{\alpha} \to \overline{\theta}$$

The constraint C^{Rec} must be computed from the constraint C generated by typechecking the body e of the recursive definition. By analogy with (rec), the inference rule can be written

$$\frac{C \ ; \ \overline{\Gamma}, f\ :\ d^{\alpha} \to \overline{\theta} \vdash e \downarrow d^{\widehat{\alpha}} \to \overline{\theta}}{C^{\mathsf{Rec}} \ ; \ \overline{\Gamma} \vdash (\mathsf{letrec}_{d \to \theta}\ f = e) \uparrow d^{\alpha} \to \overline{\theta}}$$

(var) $$\frac{}{\emptyset \,;\, \overline{\Gamma}, x \,:\, \overline{\sigma} \vdash x \uparrow \overline{\sigma}}$$

(abs) $$\frac{(\overline{\tau}, V) := \mathtt{Annot}(\tau, \mathcal{V}_{\jmath}(\overline{\Gamma})) \qquad C \,;\, \overline{\Gamma}, x \,:\, \overline{\tau} \vdash e \uparrow \overline{\sigma}}{C \,;\, \overline{\Gamma} \vdash \lambda x : \tau.\ e \uparrow \overline{\tau} \to \overline{\sigma}}$$

(app) $$\frac{C_1 \,;\, \overline{\Gamma} \vdash e \uparrow \overline{\tau} \to \overline{\sigma} \qquad C_2 \,;\, \overline{\Gamma} \vdash e' \downarrow \overline{\tau}}{C_1 \cup_{\perp} C_2 \,;\, \overline{\Gamma} \vdash e\ e' \uparrow \overline{\sigma}}$$

(T-abs) $$\frac{C \,;\, \overline{\Gamma} \vdash e \uparrow \overline{\sigma}}{C \,;\, \overline{\Gamma} \vdash \Lambda X.\ e \uparrow \Pi X.\ \overline{\sigma}} \qquad \text{if } X \notin \overline{\Gamma}$$

(T-app) $$\frac{C \,;\, \overline{\Gamma} \vdash e \uparrow \Pi X.\ \overline{\sigma} \qquad (\overline{\tau}, V) = \mathtt{Annot}(\tau, \mathcal{V}_{\jmath}(\overline{\Gamma}))}{C \,;\, \overline{\Gamma} \vdash e\ \tau \uparrow \overline{\sigma}[X := \overline{\tau}]}$$

(cons) $$\frac{c_k : \Pi \boldsymbol{X}.\ \overline{\boldsymbol{\theta}_{\boldsymbol{k}}} \to d^{\widehat{\imath}}\ \boldsymbol{X}}{\emptyset \,;\, \overline{\Gamma} \vdash c_k \uparrow \Pi \boldsymbol{X}.\ \overline{\boldsymbol{\theta}_{\boldsymbol{k}}}[\imath := \alpha] \to d^{\widehat{\alpha}}\ \boldsymbol{X}} \text{ if } \alpha \notin \mathcal{V}_{\jmath}(\overline{\Gamma})$$
where $\mathcal{C}(d) = \{c_1, \ldots, c_n\}$ for some d

(case) $$\frac{\begin{array}{l} C \,;\, \overline{\Gamma} \vdash e \uparrow d^{s}\ \overline{\boldsymbol{\tau}} \qquad \alpha \notin \mathcal{V}_{\jmath}(\overline{\Gamma}) \qquad (\overline{\sigma}, V) = \mathtt{Annot}(\sigma, \mathcal{V}_{\jmath}(\overline{\Gamma}) \cup \{\alpha\}) \\ c_k : \Pi \boldsymbol{X}.\ \overline{\boldsymbol{\theta}_{\boldsymbol{k}}} \to d^{\widehat{\imath}}\ \boldsymbol{X} \quad C_k \,;\, \overline{\Gamma} \vdash e_k \downarrow \overline{\boldsymbol{\theta}_{\boldsymbol{k}}}[\boldsymbol{X} := \overline{\boldsymbol{\tau}}, \imath := \alpha] \to \overline{\sigma} \quad (1 \le k \le n) \end{array}}{\{s \le \widehat{\alpha}\} \cup_{\perp} C \cup_{\perp} \left(\bigcup_{\perp} C_k\right) \,;\, \overline{\Gamma} \vdash \mathsf{case}_{\sigma}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \cdots \mid c_n \Rightarrow e_n\} \uparrow \overline{\sigma}}$$
where $\mathcal{C}(d) = \{c_1, \ldots, c_n\}$

(sub) $$\frac{C \,;\, \overline{\Gamma} \vdash e \uparrow \overline{\tau}}{C \cup_{\perp} (\!|\overline{\tau} \sqsubseteq \overline{\sigma}|\!) \,;\, \overline{\Gamma} \vdash e \downarrow \overline{\sigma}}$$

Fig. 7. Type inference with constraints

To guarantee correctness, as stated in Prop. 4.6, we want that $\rho \models C^{\mathsf{Rec}}$ entails

$$\overline{\Gamma}\rho \vdash (\mathsf{letrec}_{d\to\theta}\ f = e)\ :\ d^{\rho(\alpha)} \to \overline{\theta}\rho$$

For the above judgment to be derivable, it must be the case (applying inversion) that the premises of the rule (rec) must hold, and there must exist a substitution ρ' such that $\overline{\Gamma}\rho' = \overline{\Gamma}\rho'$, $\rho' \models C$, $\rho'(\alpha) = \imath$ and

$$\overline{\Gamma}\rho', f\ :\ d^{\imath} \to \overline{\theta}\rho' \vdash e\ :\ d^{\widehat{\imath}} \to \overline{\theta}\rho' \qquad \text{with} \quad \imath \notin \overline{\Gamma}\rho'$$

We explain how to define C^{Rec} by considering two examples of fixpoints definitions:

$$\begin{array}{lcl} \mathsf{fix}_1 & := & (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{o}) \\ \mathsf{fix}_2 & := & (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ (f\ \mathsf{o})) \end{array}$$

Note that $\mathsf{fix}_1\,\mathsf{o}$ terminates while $\mathsf{fix}_2\,\mathsf{o}$ does not. Hence, fix_1 must be typable while fix_2 must be rejected by the type inference algorithm. Let us now inspect the typechecking derivations of the bodies of these two functions.

– For the body of fix_1, we have

$$\frac{\emptyset\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta}, x : \mathsf{Nat}^{\gamma} \vdash \mathsf{o} \uparrow \mathsf{Nat}^{\widehat{\delta}}}{\emptyset\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta} \vdash \lambda x : \mathsf{Nat}.\ \mathsf{o} \uparrow \mathsf{Nat}^{\gamma} \to \mathsf{Nat}^{\widehat{\delta}}}$$

Since $(\!|\mathsf{Nat}^{\gamma} \to \mathsf{Nat}^{\widehat{\delta}} \sqsubseteq \mathsf{Nat}^{\widehat{\alpha}} \to \mathsf{Nat}^{\beta}|\!) = \widehat{\alpha} \leq \gamma,\ \widehat{\delta} \leq \beta$, it follows that

$$\widehat{\alpha} \leq \gamma,\ \widehat{\delta} \leq \beta\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta} \vdash \lambda x : \mathsf{Nat}.\ \mathsf{o} \downarrow \mathsf{Nat}^{\widehat{\alpha}} \to \mathsf{Nat}^{\beta}$$

– For the body of fix_2, we have

$$\frac{\emptyset\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta}, x : \mathsf{Nat}^{\gamma} \vdash \mathsf{o} \uparrow \mathsf{Nat}^{\widehat{\delta}}}{\widehat{\delta} \leq \alpha\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta}, x : \mathsf{Nat}^{\gamma} \vdash \mathsf{o} \downarrow \mathsf{Nat}^{\alpha}}$$

Hence

$$\frac{\widehat{\delta} \leq \alpha\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta}, x : \mathsf{Nat}^{\gamma} \vdash f\ \mathsf{o} \uparrow \mathsf{Nat}^{\beta}}{\widehat{\delta} \leq \alpha\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta} \vdash \lambda x : \mathsf{Nat}.\ (f\ \mathsf{o}) \uparrow \mathsf{Nat}^{\gamma} \to \mathsf{Nat}^{\beta}}$$

It follows that

$$\widehat{\delta} \leq \alpha,\ \widehat{\alpha} \leq \gamma\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta} \vdash \lambda x : \mathsf{Nat}.\ (f\ \mathsf{o}) \downarrow \mathsf{Nat}^{\widehat{\alpha}} \to \mathsf{Nat}^{\beta}$$

We arrive at these two derivations

$$\frac{\widehat{\alpha} \leq \gamma,\ \widehat{\delta} \leq \beta\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta} \vdash \lambda x : \mathsf{Nat}.\ \mathsf{o} \downarrow \mathsf{Nat}^{\widehat{\alpha}} \to \mathsf{Nat}^{\beta}}{C_1^{\mathsf{Rec}}\ ;\vdash (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{o}) \uparrow \mathsf{Nat}^{\widehat{\alpha}} \to \mathsf{Nat}^{\beta}} \qquad (\mathsf{fix}_1)$$

$$\frac{\widehat{\delta} \leq \alpha,\ \widehat{\alpha} \leq \gamma\ ;\ f : \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\beta} \vdash \lambda x : \mathsf{Nat}.\ (f\ \mathsf{o}) \downarrow \mathsf{Nat}^{\widehat{\alpha}} \to \mathsf{Nat}^{\beta}}{C_2^{\mathsf{Rec}}\ ;\vdash (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ (f\ \mathsf{o})) \uparrow \mathsf{Nat}^{\widehat{\alpha}} \to \mathsf{Nat}^{\beta}} \qquad (\mathsf{fix}_2)$$

where

$$\begin{array}{llll} C_1^{\mathsf{Rec}} & \text{is computed from} & C_1 & =_{\text{def}} \ \widehat{\alpha} \leq \gamma, \widehat{\delta} \leq \beta \\ C_2^{\mathsf{Rec}} & \text{is computed from} & C_2 & =_{\text{def}} \ \widehat{\delta} \leq \alpha, \widehat{\alpha} \leq \gamma \end{array}$$

The constraints C_1^{Rec} and C_2^{Rec} must satisfy different properties:

- We want fix_1 to be typable. Hence, C_1^{Rec} must be satisfiable, and moreover, for all substitution ρ such that $\rho \models C_1^{\mathsf{Rec}}$, there must be a substitution ρ' and a stage variable $\imath$ such that $\imath \notin \mathsf{codom}(\rho)$, $\rho'(\alpha) = \imath$, $\rho'(\beta) = \rho(\beta)$ and $\rho' \models \widehat{\alpha} \leq \gamma, \widehat{\delta} \leq \beta$.
- C_2^{Rec} must be unsatisfiable because fix_2 is not typable since it does not terminate. C_2^{Rec} will actually be the false constraint $\bot$.

These properties are provided by an algorithm called `RecCheck`, which we describe below. To ease the explanations, we introduce some terminology. Recall that stage expressions $s \in \mathcal{S}$ are either of the form $\widehat{\imath}^n$ with $\imath \in \mathcal{V}_{\mathcal{S}}$ or of the form $\widehat{\infty}^n$. Hence, any stage expression s has at most one occurrence of a unique stage variable $\imath \in \mathcal{V}_{\mathcal{S}}$. We call this stage variable the *base stage* of s, and write it $\lfloor s \rfloor$. Furthermore, consider a fixpoint $(\mathsf{letrec}_{d \to \theta}\ f = e)$ such that

$$C \ ; \overline{\Gamma}, f \ : \ d^{\alpha} \to \overline{\theta} \vdash e \downarrow d^{\widehat{\alpha}} \to \overline{\theta} \qquad \text{with} \quad \alpha \notin \overline{\Gamma}, \overline{\theta} \tag{6}$$

We say that α is the *fixpoint inference stage variable* of $(\mathsf{letrec}_{d \to \theta}\ f = e)$.

Example 4.8. The fixpoint stage variable of both fix_1 and fix_2 is α. □

Consider an instance of the judgment (6) for substitution ρ. In order to apply the rule (rec) with that instance as premise, the fixpoint inference stage variable α must be mapped to a fresh stage variable $\imath$. Moreover, we must have $\imath \notin \overline{\Gamma}\rho, \overline{\theta}\rho$. This imposes that no inference stage variable occurring in $\overline{\Gamma}, \overline{\theta}$ can be mapped to a stage expression with base stage $\imath$. Let $V^{\neq} =_{\text{def}} \mathcal{V}_{\mathcal{I}}(\overline{\Gamma}, \overline{\theta})$.

The algorithm works on a representation of C as a graph whose nodes are inference stages variables and ∞, and whose edges are integers. Each constraint in C is of the form $\infty \leq \widehat{\beta}^n$, or $\widehat{\alpha}_1^{n_1} \leq \widehat{\alpha}_2^{n_2}$. In the first case, one adds an edge from β to ∞ labeled with 0, in the second case one adds an edge from α_2 to α_1 labeled with $n_2 - n_1$. We do not represent edges for constraints of the form $\widehat{\beta}^n \leq \widehat{\beta}^n$.

Example 4.9. The graphs representing C_1 and C_2 are depicted in Fig. 8. □

Before explaining the algorithm, let us make an important remark on the graph of constraints. It may happen that the graph contains a negative cycle, i.e. a cycle where the sum of the edges is strictly negative. Such cycles imply $\widehat{\beta}^{k+1} \leq \beta$, or equivalently $\infty \leq \beta$, for the variable β in the cycle. Hence, every variable in a negative cycle *must* be mapped to ∞. Therefore, at some stage in the algorithm, it is necessary to compute negative cycles. This can be done using Bellman's algorithm, which runs n^2, where n is number of edges of the graph, hence the number of inference stage variables in C.

The algorithm runs in two phases. The first phase ensures that the fixpoint inference stage variable α can be mapped to $\imath$, and the second phase ensures that no variable in $V^{\neq}$ must be mapped to a stage expression with base stage $\imath$.

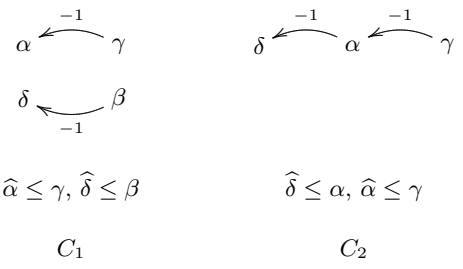


Fig. 8. The graphs of the constraints C_1 and C_2.

First phase: variables that must be mapped to $\imath$. First, note that the constraint $\widehat{\delta} \leq \alpha$, $\widehat{\alpha} \leq \gamma$ of fix_2 cannot be satisfied by a substitution which sends α to the stage variable $\imath$. Indeed, the constraint $\widehat{\delta} \leq \alpha$ would lead to a stage inequality of the form $\widehat{s} \leq \imath$, which is derivable for no stage expression s.

We briefly indicate how to detect this kind of situation. Observe that constraints of the form $\widehat{\beta}^n \leq \widehat{\alpha}^m$ force β to be mapped to a stage expression with base stage $\imath$. This is the case of the variable δ in the constraint C_2. So, we define $S^\imath$, the set of inference stage variable that must be mapped to a stage expression with base stage $\imath$, as the downward closure of $\{\alpha\}$:

$$\alpha \in S^\imath \qquad \text{and} \qquad \beta \in S^\imath \quad \text{if} \quad \widehat{\beta}^n \leq \widehat{\gamma}^m \in C \quad \text{with} \quad \gamma \in S^\imath$$

Example 4.10. For fix_1 we have $S^\imath = \{\alpha\}$ and for fix_2 we have $S^\imath = \{\alpha, \delta\}$. □

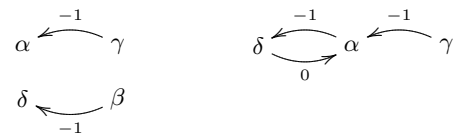
Note that if $\beta \in S^\imath$ is mapped to a s depending on $\imath$, then we must have $\imath \leq s$. We represent this by computing a new set of constraints

$$C^1 \;=_{\text{def}}\; C \cup \{\alpha \leq \beta \mid \beta \in S^\imath\}$$

Example 4.11. These constraints for fix_1 and fix_2, denoted respectively C_1^1 and C_2^1, are depicted in Fig. 9. We have $C_1^1 = C_1$. On the other hand, the graph of C_2^1 contains a negative cycle which forces α to be mapped to ∞. Hence α cannot be mapped to $\imath$. □

Therefore, we now have to check for negative cycles in the graph of C^1. For each such cycle starting from β, we compute the set $V_{\beta\leq}$ of variables greater or equal to β, remove all inequalities about variables in $V_{\beta\leq}$ and add the constraints $\infty \leq \gamma$ for $\gamma \in V_{\beta\leq}$. Hence, we get a new set of constraints C^2 that does not contain cycles.

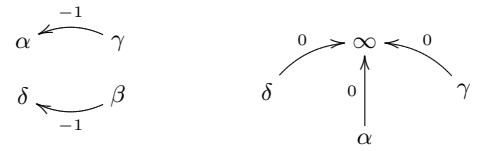
Example 4.12. The sets C_1^2 and C_2^2 for fix_1 and fix_2 are depicted on Fig. 10. The negative cycle involving α in C_2^1 implies that C_2^2 forces α to be mapped to ∞, which makes impossible to map α to $\imath$. Therefore, we can already discard fix_2 at this point, and put $C_2^{\mathsf{Rec}} =_{\text{def}} \bot$. On the other hand, the constraint C_1^2 poses no problem. □



$\widehat{\alpha} \leq \gamma,\ \widehat{\delta} \leq \beta$ $\qquad$ $\widehat{\delta} \leq \alpha,\ \widehat{\alpha} \leq \gamma,\ \alpha \leq \delta$

C_1^1 $\qquad$ C_2^1

Fig. 9. The graphs of the constraints C_1^1 and C_2^1.



$\widehat{\alpha} \leq \gamma,\ \widehat{\delta} \leq \beta$ $\qquad$ $\infty \leq \delta,\ \infty \leq \alpha,\ \infty \leq \gamma$

C_1^2 $\qquad$ C_2^2

Fig. 10. The graphs of the constraints C_1^2 and C_2^2.

Second phase: ensuring that $\imath \notin \overline{\Gamma}\rho, \overline{\theta}\rho$. Since we must have $\imath \notin \overline{\Gamma}\rho, \overline{\theta}\rho$, no inference stage variables occurring in $\overline{\Gamma}, \overline{\theta}$ can be mapped by ρ to a stage expression with base stage $\imath$. Moreover, if β is such a variable and $\widehat{\beta}^n \leq \widehat{\delta}^m$ is derivable from C^2 (i.e. if there is a path from δ to β in the graph of C^2), then δ cannot be mapped to a stage expression with base stage $\imath$. Indeed, $s \leq \widehat{\imath}^p$ implies that s is of the form $\widehat{\imath}^q$. Hence, mapping δ to $\widehat{\imath}^p$ for some p forces to map β to $\widehat{\imath}^q$ some q. Therefore, we let $S^{\neg\imath}$, the set of variables that cannot be mapped to a stage expression with base stage $\imath$, be the upward closure of $V^{\neq}$:

$$V^{\neq} \subseteq S^{\neg\imath} \qquad \text{and} \qquad \delta \in S^{\neg\imath} \quad \text{if} \quad \widehat{\beta}^n \leq \widehat{\delta}^m \in C^2 \quad \text{with} \quad \beta \in S^{\neg\imath}$$

(recall that $V^{\neq} = \mathcal{V}_{\mathfrak{I}}(\overline{\Gamma}, \overline{\theta})$).

Example 4.13. For fix_1 and fix_2 we have $S_1^{\neg\imath} = S_2^{\neg\imath} = \{\beta\} = V_1^{\neq} = V_2^{\neq}$. □

Now, assume that there is in C^2 a path from a variable $\delta \in S^{\neg\imath}$ to a variable $\beta \in S^{\imath}$ (hence that $\widehat{\beta}^n \leq \widehat{\delta}^m$ is derivable from C^2). Since β must be mapped to an expression with base stage $\imath$, there are two possibilities for δ: it must be mapped either to a stage expression with base stage $\imath$ or to ∞. But δ cannot be mapped to a stage expression with base stage $\imath$. It must therefore be mapped to ∞, which is expressed by the constraint $\infty \leq \delta$. Now, it may happen that such new constraints force a variable $\gamma \in S^{\imath}$ to be mapped to ∞ (see for instance Ex. 4.14 below). In this case the algorithm fails, otherwise it succeeds.

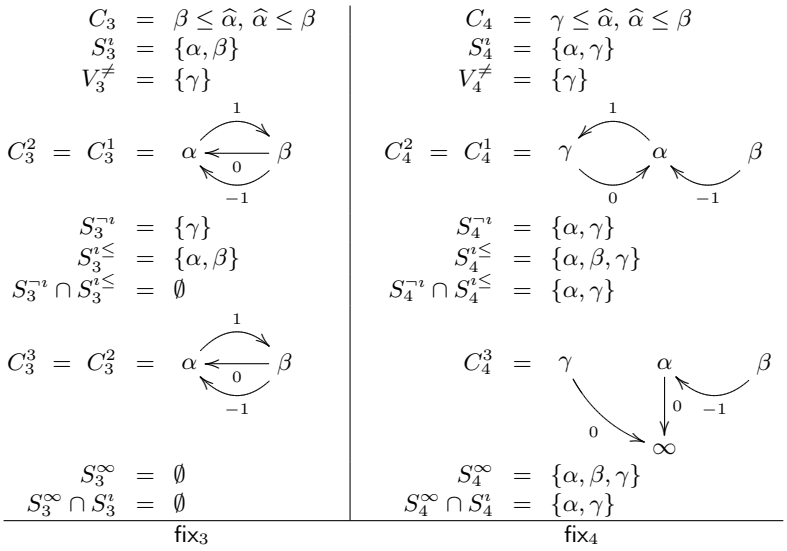


Fig. 11. Runs of `RecCheck` on fix_3 and fix_4 of Ex. 4.14.

Formally, we proceed as follows.

1. We compute the upward closed set $S^{\imath\le}$ of stage variables that must be mapped to ∞ or to a stage expression with base stage $\imath$:
$$S^\imath \subseteq S^{\imath\le} \qquad \text{and} \qquad \delta \in S^{\imath\le} \quad \text{if} \quad \widehat{\beta}^n \le \widehat{\delta}^m \in C^2 \quad \text{with} \quad \beta \in S^{\imath\le}$$
For fix_1, we have $S_1^{\imath\le} = \{\alpha, \gamma\}$.
2. Proceeding as for computing C^2, we set all variables $\beta \in S^{\neg\imath} \cap S^{\imath\le}$ to ∞. This generates a new set of constraints C^3. For fix_1, we have $S_1^{\neg\imath} \cap S_1^{\imath\le} = \emptyset$, hence $C_1^3 = C_1^2$.
3. We compute the upward closed set S^∞ of stage variables that must be mapped to ∞:
$$\beta \in S^\infty \qquad \text{if} \qquad \infty \le \widehat{\beta}^k \in C^3 \quad \text{or} \quad \left(\widehat{\delta}^n \le \widehat{\beta}^m \in C^3 \quad \text{with} \quad \delta \in S^\infty\right)$$

Now, if $S^\imath \cap S^\infty = \emptyset$, then the algorithm succeeds and we put $C^{\mathsf{Rec}} := C^3$, otherwise it fails and we put $C^{\mathsf{Rec}} := \bot$.

This second phase succeeds for fix_1. On the other hand, the failure of the algorithm for fix_2 was already known at the end of the first phase. In Ex. 4.14 below, we give an expression for which the algorithm succeeds on the first phase and fails on the second.

In the case of fix_1, the algorithm succeeds and we have $C_1^{\mathsf{Rec}} = C_1^2 = C_1^1 = C_1$. We arrive at the following inference derivation:

$$\frac{\widehat{\alpha} \le \gamma, \widehat{\delta} \le \beta\ ;\ f : \mathsf{Nat}^\alpha \to \mathsf{Nat}^\beta \vdash \lambda x : \mathsf{Nat}.\ \mathsf{o} \downarrow \mathsf{Nat}^{\widehat{\alpha}} \to \mathsf{Nat}^\beta}{\widehat{\alpha} \le \gamma, \widehat{\delta} \le \beta\ ;\ \vdash (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{o}) \uparrow \mathsf{Nat}^\alpha \to \mathsf{Nat}^\beta}$$

We check that the constraint C_1^{Rec} allows applying the rule (rec):

$$(\text{rec})\ \frac{f : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{r} \vdash \lambda x : \mathsf{Nat}.\ \mathsf{o}\ :\ \mathsf{Nat}^{\widehat{\imath}} \to \mathsf{Nat}^{r}}{\vdash (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{o})\ :\ \mathsf{Nat}^{s} \to \mathsf{Nat}^{r}}\ \text{if } \imath \notin r$$

Let ρ such that $\widehat{\rho(\alpha)} \leq \rho(\gamma)$ and $\widehat{\rho(\delta)} \leq \rho(\beta)$. Moreover, let $s =_{\text{def}} \rho(\alpha)$ and $r =_{\text{def}} \rho(\beta)$. Let $\imath \notin \rho(\beta), \rho(\delta)$ and define ρ' such that

$$\rho'(\alpha) = \imath \qquad \rho'(\gamma) = \widehat{\imath} \qquad \rho'(\beta) = \rho(\beta) \qquad \rho'(\delta) = \rho(\delta)$$

We thus have $\widehat{\rho'(\alpha)} \leq \rho'(\gamma)$ and $\widehat{\rho'(\delta)} \leq \rho'(\beta)$. It follows that

$$f : \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{r} \vdash \lambda x : \mathsf{Nat}.\ \mathsf{o}\ :\ \mathsf{Nat}^{\widehat{\imath}} \to \mathsf{Nat}^{r}$$

Since $\imath \notin r$, by (rec) deduce that

$$\vdash (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{o})\ :\ \mathsf{Nat}^{s} \to \mathsf{Nat}^{r}$$

hence $\vdash (\mathsf{letrec}_{\mathsf{Nat}\to\mathsf{Nat}}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{o})\ :\ \mathsf{Nat}^{\rho(\alpha)} \to \mathsf{Nat}^{\rho(\beta)}$.

Example 4.14 (The second phase of `RecCheck`*).* Consider the expressions:

$$\begin{array}{rcl} \mathsf{fix}_3 & := & \lambda v : V.\ \ \lambda y : \mathsf{Nat}.\ \ (\mathsf{letrec}\ f = \lambda x : \mathsf{Nat}. \\ & & \mathsf{case}_V\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}.\ \ f\, z\}) \end{array}$$

$$\begin{array}{rcl} \mathsf{fix}_4 & := & \lambda v : V.\ \ \lambda y : \mathsf{Nat}.\ \ (\mathsf{letrec}\ f = \lambda x : \mathsf{Nat}. \\ & & \mathsf{case}_V\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}.\ \ f\, z\}) \end{array}$$

Forgetting stages, fix_3 and fix_4 would have type $V \to \mathsf{Nat} \to \mathsf{Nat} \to V$, hence

$$v : V \vdash \mathsf{fix}_3\ v\ (\mathsf{s}\ \mathsf{o})\ \mathsf{o}\ :\ V \qquad \text{and} \qquad v : V \vdash \mathsf{fix}_4\ v\ (\mathsf{s}\ \mathsf{o})\ \mathsf{o}\ :\ V$$

The only difference between fix_3 and fix_4 is the variable subject to the case-analysis: in fix_3, this is the variable x, bound inside the fixpoint, while in fix_4, this is the variable y, bound outside the fixpoint. This leads to very different behaviors: $\mathsf{fix}_3\ v\ (\mathsf{s}\,\mathsf{o})\ \mathsf{o}$ is strongly normalizing and reduces to v, whereas $\mathsf{fix}_4\ v\ (\mathsf{s}\,\mathsf{o})\ \mathsf{o}$ has no normal form. Indeed, we have $\mathsf{fix}_4\ v\ (\mathsf{s}\,\mathsf{o})\ \mathsf{o} \to^* \mathsf{fix}'_4\mathsf{o}$ and

$$\mathsf{fix}'_4\ \mathsf{o} \to^* \mathsf{case}_V\ (\mathsf{s}\,\mathsf{o})\ \mathsf{of}\ \{\mathsf{o} \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}.\ \mathsf{fix}'_4\, z\} \to \mathsf{fix}'_4\ \mathsf{o} \to \ \ldots$$

where $\mathsf{fix}'_4 := (\mathsf{letrec}\ f = \lambda x : \mathsf{Nat}.\ \mathsf{case}_V\ (\mathsf{s}\,\mathsf{o})\ \mathsf{of}\ \{\mathsf{o} \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}.\ \ f\, z\})$. Let us have a look at the constraints generated during the typechecking of fix_3 and fix_4. Reasoning as in Ex. 4.7 (and doing the same simplifications), we obtain the following judgment for fix_3:

$$\beta \leq \widehat{\alpha}\ ;\ v : V,\ y : \mathsf{Nat}^{\gamma},\ f : \mathsf{Nat}^{\alpha} \to V,\ x : \mathsf{Nat}^{\beta} \vdash \mathsf{case}_V\ x\ \mathsf{of}\ \{0 \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}^{\alpha}.\ f\, z\} \uparrow V$$

and for fix_4:

$$\gamma \leq \widehat{\alpha}\ ;\ v : V,\ y : \mathsf{Nat}^{\gamma},\ f : \mathsf{Nat}^{\alpha} \to V,\ x : \mathsf{Nat}^{\beta} \vdash \\ \mathsf{case}_V\ y\ \mathsf{of}\ \{0 \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}^{\alpha}.\ f\ z\} \uparrow V$$

This leads to the following premises for the inference rule (rec): for fix_3, we have

$$\beta \leq \widehat{\alpha},\ \widehat{\alpha} \leq \beta\ ;\ v : V,\ y : \mathsf{Nat}^{\gamma},\ f : \mathsf{Nat}^{\alpha} \to V \vdash \\ \lambda x : \mathsf{Nat}^{\beta}.\ \ \mathsf{case}_V\ x\ \mathsf{of}\ \{0 \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}^{\alpha}.\ f\ z\} \downarrow \mathsf{Nat}^{\widehat{\alpha}} \to V$$

and for fix_4, we have

$$\gamma \leq \widehat{\alpha},\ \widehat{\alpha} \leq \beta\ ;\ v : V,\ y : \mathsf{Nat}^{\gamma},\ f : \mathsf{Nat}^{\alpha} \to V \vdash \\ \lambda x : \mathsf{Nat}^{\beta}.\ \ \mathsf{case}_V\ y\ \mathsf{of}\ \{0 \Rightarrow v \mid \mathsf{s} \Rightarrow \lambda z : \mathsf{Nat}^{\alpha}.\ f\ z\} \downarrow \mathsf{Nat}^{\widehat{\alpha}} \to V$$

Let $C_3 \ =_{\text{def}}\ \beta \leq \widehat{\alpha}, \widehat{\alpha} \leq \beta$ and $C_4 \ =_{\text{def}}\ \gamma \leq \widehat{\alpha}, \widehat{\alpha} \leq \beta$. In both cases, when checking the body of the recursive definition, α is the fixpoint stage variable (i.e. it must be mapped to $\imath$), while γ cannot be mapped to a stage expression with base stage $\imath$. The variable β must normally not appear as an annotation of Nat under the λ-abstraction; we write it just for convenience. We have $V_3^{\neq} = V_4^{\neq} = \{\gamma\}$, hence there is *a priori* no restriction on β. The constraint C_4 impose γ to be mapped to $\imath$, and the algorithm will fail on fix_4 for this reason. We display on Fig.11 the runs of the algorithm on fix_3 and fix_4. The algorithm succeeds on fix_3. On fix_4, the first phase succeeds but the second fails. □

4.4 Type inference in System $F^{\frown}$

We now turn to the formal and precise description of the type inference algorithm for $F^{\frown}$. Our presentation follows [6], where more details can be found.

System $F^{\frown\star}$. When we discussed the typing rule of fixpoint in the last section (Sect. 4.3), we silenced one issue concerning the types appearing as tags in terms. Consider the typing rule (rec)

$$\text{(rec)}\ \frac{\overline{\Gamma}, f\ :\ d^{\imath}\overline{\boldsymbol{\tau}} \to \overline{\theta} \vdash e\ :\ d^{\widehat{\imath}}\overline{\boldsymbol{\tau}} \to \overline{\theta}[\imath := \widehat{\imath}] \qquad \imath\ \mathsf{pos}\ \overline{\theta}}{\overline{\Gamma} \vdash (\mathsf{letrec}_{d|\overline{\boldsymbol{\tau}}|\to|\overline{\theta}|}\ f = e)\ :\ d^{s}\overline{\boldsymbol{\tau}} \to \overline{\theta}[\imath := s]}\ \text{if } \imath \notin \overline{\Gamma}, \overline{\boldsymbol{\tau}}$$

In the general case, the stage variable may appear in $\overline{\theta}$. But when inferring the type of a fixpoint $? \ ; \overline{\Gamma} \vdash (\mathsf{letrec}_{d\,\overline{\boldsymbol{\tau}}\to\theta}\ f = e) \uparrow ?$ we have *a priori* no indication on how the θ should be decorated. Therefore, type inference is performed in a system $F^{\frown\star}$ such that the type appearing in fixpoints convey an indication, symbolized by the tag $\star$, of the positions at which the fixpoint stage variable must appear. Such types are called *position types* and are given by the abstract syntax:

$$\mathcal{T}^{\star} \ ::= \ \mathcal{V}_{\mathcal{T}} \mid \mathcal{T}^{\star} \to \mathcal{T}^{\star} \mid \Pi\mathcal{V}_{\mathcal{T}}.\ \mathcal{T}^{\star} \mid \mathcal{D}^{\star}\ \boldsymbol{\mathcal{T}}^{\star} \mid \mathcal{D}\ \boldsymbol{\mathcal{T}}^{\star}$$

where in the clause for datatypes, it is assumed that the length of the vector $\mathfrak{T}^\star$ is exactly the arity of the datatype. We let $|.| : \mathfrak{T}^\star \to \mathfrak{T}$ be the obvious erasure map from $\mathfrak{T}^\star$ to $\mathfrak{T}$.

The terms of $F^{\widehat{\star}}$ are those of $F^{\widehat{}}$, excepted that fixpoints are now of the form $(\mathsf{letrec}_{\tau^\star}\ f = e)$ where $\tau^\star \in \mathfrak{T}^\star$ is a position type. The typing rules of $F^{\widehat{\star}}$ are those of $F^{\widehat{}}$, excepted that the typing rule (rec) of fixpoints is now

$$\frac{\overline{\Gamma}, f\ :\ d^\imath\overline{\boldsymbol{\tau}} \to \overline{\theta} \vdash e\ :\ d^{\widehat{\imath}}\overline{\boldsymbol{\tau}} \to \overline{\theta}[\imath := \widehat{\imath}]}{\overline{\Gamma} \vdash (\mathsf{letrec}_{\tau^\star}\ f = e)\ :\ d^s\overline{\boldsymbol{\tau}} \to \overline{\theta}[\imath := s]}\ \text{if}\ \begin{cases} \imath \notin \overline{\Gamma}, \overline{\boldsymbol{\tau}} \text{ and } \imath\ \mathsf{pos}\ \overline{\theta} \\ \tau^\star \text{ is } \imath\text{-compatible with } d^\imath\overline{\boldsymbol{\tau}} \to \overline{\theta} \end{cases}$$

where we say that a position type $\sigma^\star$ is *$\imath$-compatible* with a sized type $\overline{\sigma}$ if $\sigma^\star$ can be obtained from $\overline{\sigma}$ by replacing all stage annotations containing $\imath$ by $\star$ and by erasing all other size annotations. For instance, $\mathsf{Nat}^\imath \to \mathsf{Nat}^j \to \mathsf{Nat}^{\widehat{\imath}}$ is $\imath$-compatible with $\mathsf{Nat}^\star \to \mathsf{Nat} \to \mathsf{Nat}^\star$, while $\mathsf{Nat}^\imath \to \mathsf{Nat}^\imath \to \mathsf{Nat}^{\widehat{\imath}}$ is not.

Remark 4.15 (The shape of $\tau^\star$). Note that the conjunction of the conditions $\imath \notin \overline{\boldsymbol{\tau}}$ and $\tau^\star$ $\imath$-compatible with $d^\imath\overline{\boldsymbol{\tau}} \to \overline{\theta}$ implies that $\tau^\star$ is of the form $d^\star\boldsymbol{\tau} \to \theta^\star$, where $\boldsymbol{\tau}$ are not position types (i.e. they convey no tag $\star$).

The annotation of a position type $\tau^\star$ is performed by a function $\mathtt{Annot}^\star$, which is similar to $\mathtt{Annot}$ but takes as input a position type instead of a bare type. Intuitively, if V is a set of inference stage variables and $\sigma^\star \in \mathfrak{T}^\star$ is a position type, then $\mathtt{Annot}^\star(\sigma^\star, V)$ returns a tuple $(\overline{\sigma}, V', V^\star)$ such that $|\sigma^\star| = |\overline{\sigma}|$ and:

- as with $\mathtt{Annot}$, each occurrence of an inductive datatype (tagged or not) in $\sigma^\star$ is annotated with a distinct inference stage variable $\alpha \notin V$ (so that α occurs at most once in $\sigma^\star$), and $V' = V \cup \mathcal{V}_\mathcal{J}(\overline{\sigma})$,
- $V^\star$ is the set of inference stages variables occurring in $\overline{\sigma}$ at positions that where tagged in σ.

E.g., $\mathtt{Annot}^\star(\mathsf{Nat}^\star \to \mathsf{Nat}, \{\alpha_1, \alpha_2\}) = (\mathsf{Nat}^{\alpha_3} \to \mathsf{Nat}^{\alpha_4}, \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}, \{\alpha_3\})$. Therefore, the inference rule for fixpoints will have the following shape:

$$\frac{C\ ;\ \overline{\Gamma}, f\ :\ d^\alpha\overline{\boldsymbol{\tau}} \to \overline{\theta} \vdash e \downarrow d^{\widehat{\alpha}}\overline{\boldsymbol{\tau}} \to \overline{\theta} \qquad (d^\alpha\overline{\boldsymbol{\tau}} \to \overline{\theta}, V', V^\star) = \mathtt{Annot}^\star(d^\star\boldsymbol{\tau} \to \theta^\star, V)}{C^{\mathsf{Rec}}\ ;\ \overline{\Gamma} \vdash (\mathsf{letrec}_{d^\star\boldsymbol{\tau} \to \theta^\star}\ f = e) \uparrow d^\alpha\overline{\boldsymbol{\tau}} \to \overline{\theta}}$$

where C^{Rec} is computed by $\mathtt{RecCheck}$ and $V = \mathcal{V}_\mathcal{J}(\overline{\Gamma})$. Before giving the formal definition of $\mathtt{RecCheck}$, let us stress some important points. Recall that there must be a fresh stage variable $\imath$ such that the variables $\beta \in V^\star$ must all be mapped to a stage expression of base stage $\imath$, and moreover the fixpoint variable α, which belongs to $V^\star$, must be mapped to $\imath$. Note that $V^\star \setminus \{\alpha\}$ contains exactly the set of inference variables occurring in $\overline{\theta}$ that must be mapped to a stage expression with base stage $\imath$. Therefore, the variables $\beta \in V \setminus V^\star$ are annotations of untagged datatypes occurrences in $\tau^\star$, and cannot be mapped to stage expressions depending on $\imath$. Moreover the typing rule of fixpoint imposes that for all substitution ρ, all occurrences of $\imath$ in $\overline{\theta}\rho$ must be positive. It follows that all occurrences of $\beta \in V^\star \setminus \{\alpha\}$ in $\overline{\theta}$ must be positive. The type inference algorithm checks this by encoding positivity tests in constraints.

Lemma 4.16. $\imath$ pos $\overline{\tau}$ *if and only if* $\overline{\tau} \sqsubseteq \overline{\tau}[\imath := \widehat{\imath}]$.

Note that by definition of $\texttt{Annot}^\star$, the fixpoint variable α does not occur in $\overline{\theta}$. Hence the positivity test for the occurrences of variables $\beta \in V^\star$ in $\overline{\theta}$ can be coded by the constraint $(\!|\overline{\theta} \sqsubseteq \widehat{\theta}|\!)$ where $\widehat{\theta}$ is defined as $\widehat{\theta} =_{\text{def}} \overline{\theta}[\beta := \widehat{\beta}]_{\beta \in V^\star}$. Therefore, in addition to the constraint C^{Rec} computed by $\texttt{RecCheck}$, the type inference rule of fixpoints produces a constraint $(\!|\overline{\theta} \sqsubseteq \widehat{\theta}|\!)$:

$$\frac{C \,;\, \overline{\Gamma}, f \,:\, d^\alpha \overline{\boldsymbol{\tau}} \to \overline{\theta} \vdash e \downarrow d^{\widehat{\alpha}} \overline{\boldsymbol{\tau}} \to \overline{\theta} \qquad (d^\alpha \overline{\boldsymbol{\tau}} \to \overline{\theta}, V', V^\star) = \texttt{Annot}^\star(d^\star \boldsymbol{\tau} \to \theta^\star, V)}{C^{\mathsf{Rec}} \cup_\perp (\!|\overline{\theta} \sqsubseteq \widehat{\theta}|\!) \,;\, \overline{\Gamma} \vdash (\mathsf{letrec}_{d^\star \boldsymbol{\tau} \to \theta^\star}\ f = e) \uparrow d^\alpha \overline{\boldsymbol{\tau}} \to \overline{\theta}}$$

where $V = \mathcal{V}_{\mathcal{J}}(\overline{\Gamma})$.

The `RecCheck` algorithm. We now turn to the computation of `RecCheck`. This algorithm is at the core of guaranteeing termination. As we have in Sect.4.3, it takes as input the set of constraints that has been inferred for the body of the recursive definition, and either returns an error if the definition is unsound w.r.t. the type system, or the set of constraints for the recursive definition, if the definition is sound w.r.t. the type system. The formal definition of the function is given below. Let us look at its signature. Starting from $(d^\alpha \overline{\boldsymbol{\tau}} \to \overline{\theta}, V, V^\star) := \texttt{Annot}^\star(d^\star \boldsymbol{\tau} \to \theta^\star, \mathcal{V}_{\mathcal{J}}(\overline{\Gamma}))$, we have

- an inference fixpoint variable α which must be mapped to a fresh stage variable $\imath$,
- a set of inference stage variables $V^\star$, containing α, which must be mapped to a stage expression with base stage $\imath$,
- and a set $V \setminus V^\star$ which cannot be mapped to stage expressions with base stage $\imath$.

This corresponds to the first three arguments of `RecCheck`. The last one is the constraint C generated by typechecking the body of the recursive definition. Formally, the function `RecCheck` takes as input a tuple $(\alpha, V^\star, V^{\neq}, C)$, where

- α is the fixpoint inference stage variable of the recursive definition ; it must be mapped to a fresh base stage $\imath$;
- $V^\star$ is a set of inference stage variables that must be mapped to a stage expression with the same base stage as α. The set $V^\star$ is determined by the position types in the tag of the recursive definition. In particular, we have $\alpha \in V^\star$;
- $V^{\neq}$ is a set of inference stage variables that must be mapped to a stage expression with a base stage different from $\imath$;
- C is the constraint inferred by typechecking the body of the recursive definition.

The algorithm $\texttt{RecCheck}(\alpha, V^\star, V^{\neq}, C)$ returns $\perp$ or a set of constraints subject to conditions that will be presented later.

We now turn to the computation of $\texttt{RecCheck}(\alpha, V^\star, V^{\neq}, C)$. Assuming that we intend to map α to a fresh stage variable $\imath$, the computation proceeds as follows:

1. it computes the downwards closed set $S^\imath$ of stage variables that must be mapped to a stage expression with base stage $\imath$. The rules are $V^\star \subseteq S^\imath$, and if $\alpha_1 \in S^\imath$ and $\widehat{\alpha}_2^{n_2} \leq \widehat{\alpha}_1^{n_1} \in C$ then $\alpha_2 \in S^\imath$;
2. the algorithm must enforce that α is the smallest variable in $S^\imath$. It does so by adding to C the constraints $\alpha \leq S^\imath$. Let $C_1 = C \cup \alpha \leq S^\imath$;
3. the algorithm checks for negative cycles in the graph representation of C_1. Each time it finds such a cycle starting from β, the algorithm computes the set $V_{\sqsupseteq\beta}$ of variables greater or equal to β, removes all inequalities about variables in $V_{\sqsupseteq\beta}$ and adds the constraints $\infty \leq V_{\sqsupseteq\beta}$. At the end of this step there are no more negative cycles in the graph, and we get a new set of constraints C_2;
4. the algorithm computes the upwards closed set $S^{\imath\leq}$ of stage variables that must be mapped to ∞ or to a stage expression with base stage $\imath$. The rules are $S^\imath \subseteq S^{\imath\leq}$ and if $\alpha_1 \in S^{\imath\leq}$ and $\widehat{\alpha}_1^{n_1} \leq \widehat{\alpha}_2^{n_2} \in C_2$ then $\alpha_2 \in S^{\imath\leq}$;
5. the algorithm computes the upwards closed set $S^{\neg\imath}$ of stage variables that cannot be mapped to a stage expression with base stage $\imath$. The rules are $V^{\neq} \subseteq S^{\neg\imath}$ and if $\alpha_1 \in S^{\neg\imath}$ and $\widehat{\alpha}_1^{n_1} \leq \widehat{\alpha}_2^{n_2} \in C_2$ then α_2 is in $S^{\neg\imath}$;
6. the algorithm sets all variables $\beta \in S^{\neg\imath} \cap S^{\imath\leq}$ to ∞ (as in Step 3). At the end of this step we get a new set of constraints C_3;
7. the algorithm computes the upwards closed set S^∞ of stage variables that must be mapped to ∞. If $\infty \leq \widehat{\beta}^k \in C_3$ then β is in S^∞, and if $\alpha_1 \in S^\infty$ and $\widehat{\alpha}_1^{n_1} \leq \widehat{\alpha}_2^{n_2} \in C_3$ then α_2 is in S^∞.
8. if $S^\infty \cap S^\imath = \emptyset$ the algorithm returns the new set of constraints, else it fails.

We have already explained this algorithm in Sect. 4.3. Note that at the end of step 3, the algorithm can already stop and fail if $C_2 \vdash \infty \leq \alpha$.

The inference algorithm. We now turn to the formal definition of the inference algorithm. Contrary to what we have done up to now, it is not presented by inference rules. The cause is that we want to have a very precise control on the fresh variables introduced during type inference. The defect of the presentation by inference rules can be seen on the inference rule (app) for the application:

$$\text{(app)}\ \frac{C_1\ ;\ \overline{\Gamma} \vdash e \uparrow \overline{\tau} \to \overline{\sigma} \qquad C_2\ ;\ \overline{\Gamma} \vdash e' \downarrow \overline{\tau}}{C_1 \cup_\perp C_2\ ;\ \overline{\Gamma} \vdash e\ e' \uparrow \overline{\sigma}}$$

Assume that the derivation of $C_1\ ;\ \overline{\Gamma} \vdash e \uparrow \overline{\tau} \to \overline{\sigma}$ has generated fresh inference variables $\boldsymbol{\alpha}$ occurring in C_1 but not in $\overline{\Gamma}$. If we need fresh inference variables in the derivation of $C_2\ ;\ \overline{\Gamma} \vdash e' \downarrow \overline{\tau}$, then we must ensure that these variables are not taken among $\boldsymbol{\alpha}$. Hence, we have to transmit the current set of non-fresh inference variable along the derivations of type inference. The easiest to do this is to use two functions `Infer` and `Check` such that

– `Infer` takes as input a tuple $(V, \overline{\Gamma}, e)$, where V is a set of already used inference variables such that $\mathcal{V}_\jmath(\overline{\Gamma}) \subseteq V$. It returns a tuple $(V', C, \overline{\tau})$ such that $C\ ;\ \overline{\Gamma} \vdash e \uparrow \overline{\tau}$ and $\mathcal{V}_\jmath(C, \overline{\tau}) \cup V \subseteq V'$.

$$
\begin{array}{lcl}
\texttt{Check}(V, \overline{\Gamma}, e, \overline{\tau}) & = & (V_e, (C_e \cup_\perp (\!|\overline{\tau}_e \sqsubseteq \overline{\tau}|\!))) \\
 & \text{where} & (V_e, C_e, \overline{\tau}_e) := \texttt{Infer}(\overline{\Gamma}, e) \\
\texttt{Infer}(V, \overline{\Gamma}, x) & = & \big(V, \emptyset, \overline{\Gamma}(x)\big) \\
\texttt{Infer}(V, \overline{\Gamma}, \lambda x : \tau_1.\ e) & = & (V_e, C_e, \overline{\tau}_1 \to \overline{\tau}_2) \\
 & \text{where} & (V_1, \overline{\tau}_1) := \texttt{Annot}(V, \tau_1) \\
 & & (V_e, C_e, \overline{\tau}_2) := \texttt{Infer}(V_1, \overline{\Gamma}, x\ :\ \overline{\tau}_1, e) \\
\texttt{Infer}(V, \overline{\Gamma}, \Lambda A.\ e) & = & (V_e, C_e, \Pi A.\ \overline{\tau}) \\
 & \text{where} & (V_e, C_e, \overline{\tau}) := \texttt{Infer}(V, \overline{\Gamma}, e) \\
 & \text{if} & A \text{ does not occur in } \overline{\Gamma} \\
\texttt{Infer}(V, \overline{\Gamma}, e_1\ e_2) & = & (V_2, (C_1 \cup_\perp C_2), \overline{\tau}) \\
 & \text{where} & (V_1, C_1, \overline{\tau}_2 \to \overline{\tau}) := \texttt{Infer}(V, \overline{\Gamma}, e_1) \\
 & & (V_2, C_2) := \texttt{Check}(V_1, \overline{\Gamma}, e_2, \overline{\tau}_2) \\
\texttt{Infer}(V, \overline{\Gamma}, e\ \tau) & = & (V_e, C_e, \overline{\tau}_e[A := \overline{\tau}]) \\
 & \text{where} & (V_1, \overline{\tau}) := \texttt{Annot}(V, \tau) \\
 & & (V_e, C_e, \Pi A.\ \overline{\tau}_e) := \texttt{Infer}(V_1, \overline{\Gamma}, e) \\
\texttt{Infer}(V, \overline{\Gamma}, c) & = & ((V \cup \{\alpha\}), \emptyset, \mathsf{Type}(c, \alpha)) \\
 & \text{with} & \alpha \notin V \\
\texttt{Infer}(V, \overline{\Gamma}, \mathsf{case}_\theta\ e\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{e}\}) & = & \big(V_n, (\{s \le \widehat{\alpha}\} \cup_\perp C_e \cup_\perp \bigcup_{i=1}^n C_i), \overline{\theta}\big) \\
 & \text{where} & \alpha \notin V \\
 & & (V_\theta, \overline{\theta}) := \texttt{Annot}(V \cup \{\alpha\}, \theta) \\
 & & (V_0, C_e, d^s \overline{\boldsymbol{\tau}}) := \texttt{Infer}(V_\theta, \overline{\Gamma}, e) \\
 & & (V_i, C_i) := \texttt{Check}(V_{i-1}, \overline{\Gamma}, e_i, \mathsf{Inst}(c_i, \alpha, \overline{\boldsymbol{\tau}}, \overline{\theta})) \\
 & \text{if} & \mathcal{C}(d) = \{c_1, \ldots, c_n\} \\
\texttt{Infer}(V, \overline{\Gamma}, (\mathsf{letrec}_{d^s \boldsymbol{\tau} \to \theta}\ f = e)) & = & \big(V_e, C_f, d^\alpha \overline{\boldsymbol{\tau}} \to \overline{\theta}\big) \\
 & \text{where} & (V_1, V^\star, d^\alpha \overline{\boldsymbol{\tau}} \to \overline{\theta}) := \texttt{Annot}^\star(V, d^\star \boldsymbol{\tau} \to \theta) \\
 & & \widehat{\theta} := \overline{\theta}[\alpha_i := \widehat{\alpha_i}]_{\alpha_i \in V^\star} \\
 & & (V_e, C_e) := \texttt{Check}(V_1, \overline{\Gamma}, f\ :\ d^\alpha \overline{\boldsymbol{\tau}} \to \overline{\theta}, e, d^{\widehat{\alpha}} \overline{\boldsymbol{\tau}} \to \widehat{\theta}) \\
 & & C_f := \texttt{RecCheck}(\alpha, V^\star, V_1 \backslash V^\star, C_e \cup_\perp (\!|\overline{\theta} \sqsubseteq \widehat{\theta}|\!))
\end{array}
$$

Fig. 12. Inference algorithm

– `Check` takes as input a tuple $(V, \overline{\Gamma}, e)$, where V is a set of already used inference variables such that $\mathcal{V}_\jmath(\overline{\Gamma}, \overline{\tau}) \subseteq V$. It returns a tuple (V', C) such that $C\ ;\ \overline{\Gamma} \vdash e \downarrow \overline{\tau}$ and $\mathcal{V}_\jmath(C, \overline{\tau}) \cup V \subseteq V'$.

For instance, the inference rule for the application now becomes:

$$
\begin{array}{lcl}
\texttt{Infer}(V, \overline{\Gamma}, e_1\ e_2) & = & (V_2, C_1 \cup C_2, \overline{\tau}) \\
 & \text{where} & (V_1, C_1, \overline{\tau}_2 \to \overline{\tau}) := \texttt{Infer}(V, \overline{\Gamma}, e_1) \\
 & & (V_2, C_2) := \texttt{Check}(V_1, \overline{\Gamma}, e_2, \overline{\tau}_2)
\end{array}
$$

Hence, if the derivation of $C_1\ ;\ \overline{\Gamma} \vdash e \uparrow \overline{\tau} \to \overline{\sigma}$ uses stage variables in V_1, then no fresh variable used in the derivation of $C_2\ ;\ \overline{\Gamma} \vdash e' \downarrow \overline{\tau}$ belongs to V_1. The whole inference algorithm is presented in Fig. 12.

We end the description of the type inference algorithm by the following observation about constraints.

Remark 4.17. The principal type returned by the inference algorithm may not be represented in its most compact form: for example, the inference algorithm will infer for the usual definition of addition the unconstrained type $\mathsf{Nat}^i \to \mathsf{Nat}^j \to \mathsf{Nat}^\infty$ whereas it would be more readable to use the equivalent type $\mathsf{Nat}^\infty \to \mathsf{Nat}^\infty \to \mathsf{Nat}^\infty$. Formally, we can define a notion of equivalence between pairs of types and constraint: we say $(C, \overline{\tau}) \preceq (C', \overline{\tau'})$ iff for every ρ s.t. $\rho \models C$ there exists ρ' s.t. $\rho' \models C'$ and $\rho'\overline{\tau'} \sqsubseteq \rho\overline{\tau}$. Then, we define $(C, \overline{\tau}) \simeq (C', \overline{\tau'})$ iff $(C, \overline{\tau}) \preceq (C', \overline{\tau'})$ and $(C', \overline{\tau'}) \preceq (C, \overline{\tau})$. Now, we can define a set of heuristics that transform a pair $(C, \overline{\tau})$ into simpler ones. For example, one can replace by ∞ all size variables that only occur in negative positions in the type component of such pairs. One can also perform simplifications in the constraint, in the spirit of the rules that are used in the algorithm that checks the correctness of recursive definitions. We do not formalize the notion of simplification, but it is possible to define a notion of canonical form and provide a set of rules that transform every such pair to an equivalent, canonical one.

The soundness and completeness of the algorithm, which is stated in Proposition 4.1, is proved by induction on derivations, and relies on a proof that the algorithm `RecCheck` is itself sound and complete.

5 Further reading

The material presented in this paper is based on the research papers [5,6]; the first paper introduces $\lambda^\frown$, a simply typed fragment of the $F^\frown$, and shows that it enjoys strong normalization, whereas the second paper introduces $F^\frown$ and presents the inference algorithm.

Type-based termination has its origins in Mendler's formulation of recursion in the style of fixpoints [14]. Mendler's ideas were further developed in a series of works, including [7,19], and adapted to reactive programming by Pareto, Hughes, and Sabry [13], and to type theory by Giménez [10]. Abel [2] and Barthe *et al* provide a more detailed account of related work in the area of type-based termination. Coquand and Dybjer [8] provide an historical account of inductive definitions in type theory, whereas Aczel [4] provides an introduction to inductive definitions in a set-theoretical setting.

References

1. A. Abel. Termination Checking with Types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue (FICS'03).
2. A. Abel. *Type-Based Termination. A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* PhD thesis, LMU University, Munich, 2006.
3. A. Abel. Semi-Continuous Sized Types and Termination. *LMCS*, 4(2:3), 2008.

4. P. Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of mathematical logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. North-Holland, 1977.
5. 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):97–141, 2004.
6. G. Barthe, B. Grégoire, and F. Pastawski. Practical Inference for Type-Based Termination in a Polymorphic Setting. In *Proceedings of TLCA'05*, pages 71–85, 2005.
7. W.-N. Chin and S.-C. Khoo. Calculating Sized Types. *Higher-Order and Symbolic Computation*, 14(2–3):261–300, 2001.
8. T. Coquand and P. Dybjer. Inductive Definitions and Type Theory: an Introduction (Preliminary Version). In P.S. Thiagarajan, editor, *Proceedings of FSTTCS'94*, volume 880 of *LNCS*, pages 60–76. Springer, 1994.
9. J.H. Gallier. What's So Special About Kruskal's Theorem and the Ordinal Γ_0? A Survey of Some Results in Proof Theory. *Annals of Pure and Applied Logic*, 53(3):199–260, 1991.
10. E Giménez. Structural Recursive Definitions in Type Theory. In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 397–408. Springer, 1998.
11. J.-Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université Paris 7, 1972.
12. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
13. J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of POPL'96*, pages 410–423. ACM, 1996.
14. N. P. Mendler. Recursive Types and Type Constraints in Second Order Lambda-Calculus. In *Proceedings of LiCS'87*, pages 30–36. IEEE Computer Society, 1987.
15. M. Parigot. On the Representation of Data in Lambda-Calculus. In *Proceedings of CSL'89*, volume 440 of *LNCS*, pages 309–321, 1989.
16. Z. Spławski and P. Urzyczyn. Type Fixpoints: Iteration vs. Recursion. In *Proceedings of ICFP'99*, pages 102–113. ACM, 1999.
17. M. Steffen. *Polarized Higher-order Subtyping*. PhD thesis, Department of Computer Science, University of Erlangen, 1997.
18. W. W. Tait. A Realizability Interpretation of the Theory of Species. In R. Parikh, editor, *Logic Colloquium*, volume 453 of *LNCS*, pages 240–251, 1975.
19. H. Xi. Dependent Types for Program Termination Verification. *Higher-Order and Symbolic Computation*, 15(1):91–131, 2002.

Table of Contents