Type-Based Termination with Sized Products

Gilles Barthe[1,⋆], Benjamin Grégoire[2], and Colin Riba[2]

[1] IMDEA Software, Madrid, Spain
gilles.barthe@imdea.org
[2] INRIA Sophia-Antipolis, France
{Benjamin.Gregoire,Colin.Riba}@sophia.inria.fr

Abstract. Type-based termination is a semantically intuitive method that ensures termination of recursive definitions by tracking the size of datatype elements, and by checking that recursive calls operate on smaller arguments. However, many systems using type-based termination rely on a semantical anomaly to guarantee strong normalization; namely, they impose that non-recursive elements of a datatype, e.g. the empty list, have size 1 instead of 0. This semantical anomaly also prevents functions such as quicksort to be given a precise typing.

The main contribution of this paper is a type system that remedies this anomaly, and still ensures termination. In addition, our type system features prenex stage polymorphism, a weakening of existential quantification over stages, and is precise enough to type quicksort as a non-size increasing function. Moreover, our system accomodate stage addition with all positive inductive types.

1 Introduction

Type-based termination is a method to guarantee termination of recursive definitions by a non-standard type system in which datatype elements are assigned a size, which is used by the typing rule for letrec to ensure that recursive calls are made on smaller elements, i.e. elements with a smaller size. The semantical intuition behind size-based termination is embedded in the (simplified) typing rule for recursive definitions, which states that the definition of a function on elements of size $\imath$ can only make recursive calls on elements of smaller size:

$$\frac{\Gamma, f : \mathsf{List}^{\imath}\tau \to \sigma \vdash e : \mathsf{List}^{\widehat{\imath}}\tau \to \sigma}{\Gamma \vdash \mathsf{letrec}\ f = e : \mathsf{List}^{\infty}\tau \to \sigma} \tag{1}$$

where $\imath$ is a size variable, $\mathsf{List}^{\imath}$ denotes the type of lists of size less or equal to $\imath$, and $\widehat{\ }$ is the successor function on stages, $\mathsf{List}^{\widehat{\imath}}$ denotes the type of lists of size less or equal to $\widehat{\imath}$ and List^{∞} denotes the usual type of lists.

One distinguishing feature of type-based termination is its expressiveness. Indeed, even the simplest systems of type-based termination are sufficiently expressive to allow to give precise typings for some structurally recursive functions:

$$\mathsf{map} : (X \to Y) \to \mathsf{List}^{\imath}X \to \mathsf{List}^{\imath}Y$$

⋆ Most of this work was performed while working at INRIA Sophia-Antipolis.

M. Kaminski and S. Martini (Eds.): CSL 2008, LNCS 5213, pp. 493–507, 2008.

and to type functions that are not structurally recursive such as the quicksort function:

$$\begin{array}{l}\mathsf{letrec}\ qs = \lambda l.\ \mathsf{case}\ l\ \mathsf{of}\\ \qquad\qquad \mid \mathsf{nil}\ \Rightarrow\ \mathsf{nil}\\ \qquad\qquad \mid \mathsf{cons}\ x\ xs\ \Rightarrow\ \mathsf{let}\ \langle z_1, z_2\rangle = (\mathsf{filter}\ x\ xs)\\ \qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \mathsf{app}\ (qs\ z_1)\ (\mathsf{cons}\ x\ (qs\ z_2))\end{array} \tag{2}$$

Many type-based termination systems [1, 2, 3, 7] allow the typing:

$$\mathsf{quicksort} : \mathsf{List}^{\infty} X \to \mathsf{List}^{\infty} X \tag{3}$$

but cannot yield the more precise typing:

$$\mathsf{quicksort} : \mathsf{List}^{\imath} X \to \mathsf{List}^{\imath} X \tag{4}$$

Achieving a precise typing for quicksort requires extending the type system so that it yields precise typings for app and filter: first, app must be given a precise typing by means of *stage addition*:

$$\mathsf{app} : \mathsf{List}^{\imath} X \to \mathsf{List}^{\jmath} X \to \mathsf{List}^{\imath+\jmath} X \tag{5}$$

Second, we have to express that filter divides a list of size $\imath$ into two lists whose respective sizes $\jmath_1$ and $\jmath_2$ sum up to $\imath$ which could be expressed using constrained existential types, as in [4, 10]:

$$\mathsf{filter} : X \to \mathsf{List}^{\imath} X \to \exists \jmath_1, \jmath_2\ (\jmath_1 + \jmath_2 = \imath).\ \mathsf{List}^{\jmath_1} X \times \mathsf{List}^{\jmath_2} X \tag{6}$$

Unfortunately, adding constrained existential quantification over stages may break subject reduction (see Section 2) and leads to complex type systems, where type checking requires solving constraints in Presburger arithmetic.

Furthermore, having nil of size at least $\widehat{0}$ (as in [1, 2, 3, 7]), we cannot type filter as in (6), and this prevents the typing quicksort as in (4). Thus, we must give the size 0 to nil. Alas, using the typing rule for fixpoints of [1, 2, 3, 7], and letting $\mathsf{nil} : \mathsf{List}^{0} X$, leads to typable non-terminating terms: using the typing rule for fixpoints of [1, 2, 3, 7], $(\mathsf{letrec}\ f\, x = f\, \mathsf{nil})\, \mathsf{nil}$ is typable (using the subtyping rule $\mathsf{List}^{0} X \leq \mathsf{List}^{\imath} X$) but not terminating.

Thus, defining a simple yet precise type system that enjoys good meta-theoretical properties is a challenge. The main contribution of this article is the definition of a type system $F_{\mathbb{X}}^{\widehat{}}$ that features a monoidal structure on stages (with zero and addition), that simulates existential quantification over stages, and still enjoys subject reduction and strong normalization for first-order and higher-order inductive types. Technically, we achieve subject reduction for existentials by attaching existential quantification to a container structure: this way, introduction and elimination of existential quantification is linked with introduction and elimination of the corresponding type constructor. This leads to a system which features subject reduction and where eliminations of existential quantification are easier to write for the user. The resulting system provides a well-behaved

intermediate step between basic type-based termination criterion [1, 2, 3, 7], and more powerful but less tractable constraint based approaches [4, 10]. For simplicity, in this paper we focus on a binary product, which we call *sized product*.

To achieve strong normalization, we resort to constraining the form of recursive definitions, requiring that the body of the function immediately performs a case analysis on its recursive argument; this syntactic restriction forces a style of definitions close to rewriting. However, in contrast with rewriting, the body of recursive definitions are still part of the terms. This allows for a more powerful intensional equality between functions than with rewriting. Another feature of $F^{\widehat{}}_{\times}$ is the associativity and commutativity of the addition on the stages of higher-order inductive datatypes. This is possible because, in the model construction for strong normalization, stage addition is interpreted by the natural addition on the ordinals which interpret the stages of higher-order inductive datatypes.

The paper is organized as follows. In Sect. 2 we discuss related works and presents informally the main characteristics of $F^{\widehat{}}_{\times}$. Sect. 3 is devoted to the formal definition of the system, while Sect. 4 and Sect. 5 outline respectively the proofs of subject reduction and strong normalization proofs.

2 Overview and Related Work

The purpose of this section is to present the main characteristics of $F^{\widehat{}}_{\times}$ and its relation with other works on type-based termination. We begin with a brief overview of the works that support precise typings for quicksort, and then explain the main specificities of our work.

There has been a lot of interest in using type systems to guarantee termination or to characterize the complexity of recursive functions, see e.g. [1] for an overview. Most systems share the semantical anomaly of $F^{\widehat{}}$ and we are only aware of three systems in which quicksort can be given its exact typing.

The first system is that of Chin and Khoo [5], which annotates every type with size annotations and infers a formula of Presburger arithmetic that guarantees termination. We believe that their system, while expressive, generates constraints which are too complex to be used in practice. The second system is that of Xi [10], which uses restricted dependent types to ensure termination. The third system is that of Blanqui and Riba [4]. Recursive functions are defined by rewrite rules, and as in $F^{\widehat{}}_{\times}$, having non-recursive constructors of size 0 is not problematic.

We now discuss the two main characteristics of $F^{\widehat{}}_{\times}$: the sized product and the typing of fixpoints.

2.1 The Sized Product

Advanced systems of type-based termination, such as Xi and Blanqui, feature constrained existential and universal stage quantification, respectively written $\exists \imath P.\tau$ and $\forall \imath P.\tau$, where P is a constraint and τ is a type. These systems deal with judgments of the form $K\,;\,\Gamma \vdash e : \tau$ where K is a conjunction of constraints, and their type checking algorithm generate constraints in Presburger arithmetic.

Apart from the inherent complexity of type checking, there are some known difficulties with existential types.

Fully explicit existential types, which are used by Xi [10], enjoy subject reduction but rely on a complex elaboration mechanism for type checking. In contrast, implicit existential types do not satisfy subject reduction. Blanqui and Riba [4] use the elimination rule:

$$(\exists\text{-elim})\ \frac{K\,;\,\Gamma \vdash e : \exists \imath P.\tau \qquad K, P\,;\,\Gamma, x{:}\tau \vdash e' : \sigma}{K\,;\,\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \sigma}\ \imath \notin K, \Gamma, \sigma$$

together with the let-reduction rule:

$$\mathsf{let}\ x = e\ \mathsf{in}\ e' \mapsto e'[x := e]$$

Subject reduction fails in this system (Tatsuta's example [9] is easily adapted). Because let-reduction is performed even if the typing of e does not end with an $\exists \imath P.\tau$-introduction, the sharing information given by the second premise of ($\exists$-elim), which is lost by let-reduction, is not regained by the witness information given by $\exists \imath P.\tau$-introductions. Note that the failure of subject reduction is actually not a so big problem in [4] since the constrained type system is used to analyze rewrite rules, while ensuring their termination in a standard type system which features subject reduction.

The above discussion illustrates the difficulties with existential quantification over stages and justifies our choice to focus on a simpler system that partially simulates, but does not have, existential quantification. Indeed, our type system $F^{\widehat{}}_{\times}$ achieves a similar effect using prenex stage quantification and using instead of explicitly existential quantification an embedding of existential quantification inside some specific type constructors. This way, introduction and elimination of existential quantification is linked with introduction and elimination of the corresponding type constructor. This leads to a system which features subject reduction and where eliminations of existential quantification are easier to write for the user. For simplicity, in this paper we focus on a binary product, which we call *sized product*. We explain it with an example. To express that $\mathsf{filter}\ x\ l$ computes a pair of lists whose sizes sum up to the size of l, we write

$$\mathsf{filter} : X \to \mathsf{List}^{\imath} X \to \mathsf{List}\, X \times^{\imath} \mathsf{List}\, X$$

The existential information on $\jmath_1$ and $\jmath_2$ in (6) is expressed using the rule (let), which is inspired by the usual elimination of existential quantification

$$(\mathrm{let})\ \frac{\begin{array}{l} K\,;\,\Gamma \vdash \mathsf{filter}\ x\ l : \mathsf{List}\ X \times^{\imath} \mathsf{List}\ X \\ K,\ \jmath_1 + \jmath_2 \leq \imath\ \ ;\ \ \Gamma,\ z_1{:}\mathsf{List}^{\jmath_1} X,\ z_2{:}\mathsf{List}^{\jmath_2} X \vdash e : \mathsf{List}^{\imath}\ X \end{array}}{K\,;\,\Gamma \vdash \mathsf{let}\ \langle z_1, z_2\rangle = \mathsf{filter}\, x\, l\ \mathsf{in}\ e : \mathsf{List}^{\imath}\ X}\ \jmath_1, \jmath_2 \notin K, \Gamma$$

The sized product allows to combine pair opening and let-reduction, which corresponds to the elimination of the existential information in the rule (let). This leads to the following rewrite rule, which is the key-point for subject reduction in $F^{\widehat{}}_{\times}$:

$$\mathsf{let}\ \langle x_1, x_2\rangle = \langle e_1, e_2\rangle\ \mathsf{in}\ e \quad \mapsto_{\theta} \quad e[x_1 := e_1, x_2 := e_2]$$

2.2 Typing of Fixpoints

In order to reject the non-terminating term in the introduction, we constrain the form of recursive definitions, requiring that the body of the function immediately performs a case analysis on its recursive argument, and distinguishing in the typing rule between recursive and non-recursive constructors. This approach is connected to definitions by rewriting [4], where fixpoints and case analysis are performed in a single definition. To avoid the non-normalizing term $(\mathsf{letrec}\ f\, x = f\, \mathsf{nil})\, \mathsf{nil}$ while having nil of size 0, we ensure that no evaluation strategy of a typable expression of the form $(\mathsf{letrec}\ f = e)\, \mathsf{nil}$ can make recursive calls to $\mathsf{letrec}\ f = e$. The simplest syntactic way to achieve this is to stick fixpoints definitions $\mathsf{letrec}\ f = e$ to case analysis, and to make a distinction between the *recursive* and the *non-recursive* constructors of a datatype d. Intuitively, d can not occur in the type of the arguments of a non-recursive constructors. Then, our fixpoints have the shape

$$\mathsf{letrec}\ f\ \mathsf{case}\ \{\boldsymbol{c^{nr}} \Rightarrow \boldsymbol{e^{nr}} \mid \boldsymbol{c^{r}} \Rightarrow \boldsymbol{e^{r}}\}$$

where $\boldsymbol{c^{nr}}$ are non-recursive constructors and $\boldsymbol{c^{r}}$ are recursive constructors, and where $\boldsymbol{e^{nr}}$ can not depend on f. Thus, the following term is strongly normalizing:

$$(\mathsf{letrec}\ f\ \mathsf{case}\ \{0 \Rightarrow 1 \mid \mathsf{s} \Rightarrow \lambda x.(f\ 0) + (f\ x)\})\ 0$$

It would be desirable to separate fixpoints from case analysis, as in $F^{\widehat{}}$, but we have been unable to find a strongly normalizing system for the usual syntax.

In contrast, Xi [10] can have $\mathsf{nil} : \mathsf{List}^0\tau$ without disturbing normalization. Instead of (1), fixpoints can be typed as follows:

$$\frac{K\,;\, \Gamma, f : \forall \imath < \jmath.\mathsf{List}^{\imath}\tau \to \sigma \vdash e : \mathsf{List}^{\jmath}\tau \to \sigma}{\Gamma \vdash \mathsf{letrec}\ f = e : \forall \imath.\mathsf{List}^{\imath}\tau \to \sigma}$$

It is possible for Xi to rely on a strict ordering on stages, because he relies on existential quantification to encode $\mathsf{List}^{\infty}\tau$ as $\exists \imath.\mathsf{List}^{\imath}\tau$. In our case, we cannot use such a strict ordering, because $\imath \leq \infty$ but *not* $\imath < \infty$.

3 System $F^{\widehat{}}_{\times}$

In this section, we present the syntax of system $F^{\widehat{}}_{\times}$. We begin by the stages, then define types, terms, reductions and present the typing rules of the system.

3.1 Stages

Stages expression are built from a set $\mathcal{V}_{\mathrm{S}} = \{\imath, \jmath, \kappa, \dots\}$ of stage variables. They use a binary stage addition $+$, a successor operation $\widehat{\cdot}$ and the constants 0, ∞, denoting respectively the least and the greatest stage.

Definition 3.1 (Stages). *The set* $\mathcal{S}$ *of* stage expressions *is given by the abstract syntax:*

$$s, r \ ::= \ \mathcal{V}_{\mathcal{S}} \ \mid \ 0 \ \mid \ \infty \ \mid \ \widehat{s} \ \mid \ s + r$$

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

The system uses inequalities $s \leq r$ on stage expressions. They are derived by judgments of the form $K \vdash s \leq r$, where K is a conjunction of stages inequalities called *stage constraint*. These judgments are defined by the *substage relation*.

Definition 3.2. *A* stage constraint *is a finite set* $K \in \mathcal{K}$ *of* stage inequalities $s \leq r$ *with* $s, r \in \mathcal{S}$. *The* substage relation *is the smallest relation* $K \vdash s \leq r$, *where* $K \in \mathcal{K}$ *and* $s, r \in \mathcal{S}$, *such that*

$$(ax)\frac{}{K, s \leq r \vdash s \leq r} \qquad (refl)\frac{}{K \vdash s \leq s} \qquad (inf)\frac{}{K \vdash 0 \leq s} \qquad (sup)\frac{}{K \vdash s \leq \infty}$$

$$(trans)\frac{K \vdash s \leq r \quad K \vdash r \leq p}{K \vdash s \leq p} \qquad (mon)\frac{K \vdash s \leq r}{K \vdash s + p \leq r + p} \qquad (inj)\frac{K \vdash \widehat{s} \leq \widehat{r}}{K \vdash s \leq r}$$

$$(com)\frac{}{K \vdash s + r \leq r + s} \qquad (assoc)\frac{}{K \vdash s + (r + p) \leq (s + r) + p}$$

$$(succ)\frac{}{K \vdash s + \widehat{r} = \widehat{s + r}} \qquad (zero)\frac{}{K \vdash 0 + s = s}$$

where $\overline{K \vdash s = r}$ *abbreviates the conjunction of* $\overline{K \vdash s \leq r}$ *and* $\overline{K \vdash r \leq s}$.

Note that the rule *(sup)* implies $\widehat{\infty} \leq \infty$. Moreover, we can derive $K \vdash s \leq \widehat{s}$, $K \vdash (s + r) + p \leq s + (r + p)$, $K \vdash s \leq s + r$; and $K \vdash \widehat{s} \leq \widehat{r}$ from $K \vdash s \leq r$.

3.2 Types and Datatypes Declarations

The system $F_{\widehat{\times}}$ is an extension of Church's style System F. In addition to the function space and the second order type quantification $\Pi X.\tau$, sized types are built using the sized product $\_ \times^s \_$ and the *bounded* universal stage quantification $\forall \imath \leq s.\tau$, which binds $\imath$ in τ but *not* in s (so that, by Barendregt convention, we may always assume $\imath \notin s$). The bound s in universal stage quantification is essential for the typing of fixpoints (typing rule (rec)).

We consider three sets of types: erased types $|\tau| \in |\mathcal{T}|$, that do not carry size annotations, sized types $\tau \in \mathcal{T}$, in which stage variables are free, and constrained types $\underline{\tau} \in \underline{\mathcal{T}}$, which are built from sized types using *prenex* universal stage quantification. Erased types are needed because Church's typing imposes types to appear at the term level, while we do not want size annotations to appear in terms because it makes fail subject reduction (see Sect. 2.4 of [3]).

We assume given a set $\mathcal{V}_{\mathcal{T}} = \{X, Y, \dots\}$ of type variables and a set $\mathcal{D}$ of datatype identifiers. Each datatype identifier comes equipped with an arity $\mathsf{ar}(d)$.

Definition 3.3 (Types). *The sets $|\mathcal{T}|$, $\mathcal{T}$ and $\underline{\mathcal{T}}$ of erased types, sized types and constrained types are given by the following abstract syntaxes:*

$$\begin{array}{rcl} |\mathcal{T}| & ::= & \mathcal{V}_{\mathcal{T}} \mid |\mathcal{T}| \to |\mathcal{T}| \mid \Pi\mathcal{V}_{\mathcal{T}}.|\mathcal{T}| \mid \mathcal{D}\,|\boldsymbol{\mathcal{T}}| \mid |\mathcal{T}| \times |\mathcal{T}| \\ \mathcal{T} & ::= & \mathcal{V}_{\mathcal{T}} \mid \mathcal{T} \to \mathcal{T} \mid \Pi\mathcal{V}_{\mathcal{T}}.\mathcal{T} \mid \mathcal{D}^{\mathcal{S}}\,\boldsymbol{\mathcal{T}} \mid \mathcal{D}\,\boldsymbol{\mathcal{T}} \times^{\mathcal{S}} \mathcal{D}\,\boldsymbol{\mathcal{T}} \\ \underline{\mathcal{T}} & ::= & \mathcal{T} \mid \forall\imath \le s.\underline{\mathcal{T}} \end{array}$$

where $\imath \notin s$ and in the clause for datatypes, it is assumed that the length of the vectors $\boldsymbol{\mathcal{T}}$ and $|\boldsymbol{\mathcal{T}}|$ is exactly the arity of the datatype.

Let $|\tau|, |\theta|, |\sigma|, \ldots$ range over erased types, $\tau, \theta, \sigma, \ldots$ range over sized types and $\underline{\tau}, \underline{\theta}, \underline{\sigma}, \ldots$ range over constrained types. We write $\forall\imath.\underline{\tau}$ for $\forall\imath \le \infty.\underline{\tau}$ and $\forall \boldsymbol{\imath} \le \boldsymbol{s}.\underline{\tau}$ for $\forall\imath_1 \le s_1. \cdots \forall\imath_n \le s_n.\underline{\tau}$. Note that every $\underline{\tau} \in \underline{\mathcal{T}}$ can be written $\forall \boldsymbol{\imath} \le \boldsymbol{s}.\tau$ with $\tau \in \mathcal{T}$. Moreover, we denote by $|.|$ the obvious erasure map from $\underline{\mathcal{T}}$ to $|\mathcal{T}|$. A type is *closed* if it contains no free stage variables and no free type variables.

The subtyping relation is inherited from the substage relation. The rules are syntax-directed.

Definition 3.4 (Subtyping). *The* subtyping relation *is the smallest relation $K \vdash \underline{\tau} \sqsubseteq \underline{\sigma}$, where $K \in \mathcal{K}$ and $\underline{\tau}, \underline{\sigma} \in \underline{\mathcal{T}}$, such that*

$$(var)\frac{}{K \vdash X \sqsubseteq X} \qquad (cst)\frac{K \vdash r \le s \quad K, \imath \le r \vdash \underline{\tau} \sqsubseteq \underline{\sigma}}{K \vdash \forall\imath \le s.\underline{\tau} \sqsubseteq \forall\imath \le r.\underline{\sigma}}\ \imath \notin K$$

$$(func)\frac{K \vdash \tau' \sqsubseteq \tau \quad K \vdash \sigma \sqsubseteq \sigma'}{K \vdash \tau \to \sigma \sqsubseteq \tau' \to \sigma'} \qquad (prod)\frac{K \vdash \sigma \sqsubseteq \sigma'}{K \vdash \Pi X.\sigma \sqsubseteq \Pi X.\sigma'}$$

$$(data)\frac{K \vdash s \le r \quad K \vdash \boldsymbol{\tau} \sqsubseteq \boldsymbol{\tau'}}{K \vdash d^s\boldsymbol{\tau} \sqsubseteq d^r\boldsymbol{\tau'}} \qquad (pair)\frac{K \vdash s \le r \quad K \vdash \boldsymbol{\tau} \sqsubseteq \boldsymbol{\sigma} \quad K \vdash \boldsymbol{\tau'} \sqsubseteq \boldsymbol{\sigma'}}{K \vdash d\boldsymbol{\tau} \times^s d'\boldsymbol{\tau'} \sqsubseteq d\boldsymbol{\sigma} \times^r d'\boldsymbol{\sigma'}}$$

Note that the rule (cst) is contravariant wrt. stages inequalities.

Lemma 3.5. *The relation $K \vdash \_ \sqsubseteq \_$ is reflexive and transitive.*

We now turn to datatype declarations. We assume given a fixed set $\mathcal{C}$ of *constructors*, and a function $\mathsf{C} : \mathcal{D} \mapsto \wp(\mathcal{C})$ such that $\mathsf{C}(d) \cap \mathsf{C}(d') = \emptyset$ for every distinct $d, d' \in \mathcal{D}$. Each constructor $c \in \mathsf{C}(d)$ is given an erased type of the form $\Pi\boldsymbol{X}.|\boldsymbol{\theta}| \to d\,\boldsymbol{X}$, where $|\boldsymbol{\theta}|$ is an erased type in which d and $\boldsymbol{X}$ occur only positively [2]. Note that the arity condition on d imposes that $\boldsymbol{X}$ has the same length for all $c \in \mathsf{C}(d)$. We let $\mathsf{C} =_{\text{def}} \bigcup\{\mathsf{C}(d) \mid d \in \mathcal{D}\}$.

Moreover, we distinguish between recursive and non-recursive constructors. This is essential to annotate constructor types and to the reduction and typing rules of fixpoints. Formally, we assume that $\mathsf{C}(d) = \mathsf{C}_{nr}(d) \uplus \mathsf{C}_r(d)$. Then, $c \in \mathsf{C}(d)$ is *recursive* if $c \in \mathsf{C}_r(d)$ and *non-recursive* otherwise. Intuitively, c is non-recursive iff d does not occur in $|\boldsymbol{\theta}|$. For instance, the constructor $\mathsf{nil} : \Pi X.\mathsf{List}\,X$ is non-recursive, while $\mathsf{cons} : \Pi X.X \to \mathsf{List}\,X \to \mathsf{List}\,X$ is recursive. We write c^r (resp. c^{nr}) to denote a recursive (resp. non-recursive) constructor.

Constructor types are annotated as follows: each occurrence of $d' \neq d$ in $|\boldsymbol{\theta}|$ is annotated with ∞, and each occurrence of d in $|\boldsymbol{\theta}|$ is annotated with a stage

variable $\imath$. Then, the constrained type of c is of the form $\forall \imath.\Pi \boldsymbol{X}.\boldsymbol{\theta} \to d^{\widehat{\imath}}\boldsymbol{X}$ if c is recursive and of the form $\forall \imath.\Pi \boldsymbol{X}.\boldsymbol{\theta} \to d^{\imath}\boldsymbol{X}$ otherwise. In particular, we get $\mathsf{nil}\,|\tau| : \mathsf{List}^{\widehat{0}}\tau$ and $\mathsf{cons}\,|\tau|\,a\,l : \mathsf{List}^{\widehat{s}}\tau$ whenever $l : \mathsf{List}^{s}\tau$ and $a : \tau$.

Definition 3.6 (Inductive datatypes)

(i) A signature *is a map $\Sigma : \mathsf{C} \mapsto \underline{\mathcal{T}}$ such that for all $c \in \mathsf{C}(d)$, $\Sigma(c)$ is a closed type of the form $\forall \imath.\Pi \boldsymbol{X}.\boldsymbol{\theta} \to d^{\overline{\imath}}\boldsymbol{X}$ where*

– $\boldsymbol{\theta}$ are sized types on the abstract syntax T^+ (where $d' \neq d$):

$$\begin{array}{rcl} T^+ ::= & \mathcal{V}_{\mathcal{T}} \mid T^- \to T^+ \mid \Pi\mathcal{V}_{\mathcal{T}}.T^+ \mid d'^{\infty}\,\boldsymbol{T}^+ \mid d^{\imath}\,\boldsymbol{X} \\ T^- ::= & \mathcal{V}_{\mathcal{T}} \setminus \boldsymbol{X} \mid T^+ \to T^- \mid \Pi\mathcal{V}_{\mathcal{T}}.T^- \mid d'^{\infty}\,\boldsymbol{T}^- \end{array}$$

– if d occurs in $\boldsymbol{\theta}$ then $c \in \mathsf{C}_r(d)$ and $\overline{\imath} = \widehat{\imath}$; otherwise $c \in \mathsf{C}_{nr}(d)$ and $\overline{\imath} = \imath$. Moreover, let $\mathsf{Inst}(c, s, \boldsymbol{\tau}, \sigma) =_{def} \boldsymbol{\theta}[\boldsymbol{X} := \boldsymbol{\tau}, \imath := s] \to \sigma$.

(ii) Let $d \leq_\Sigma d'$ iff d occurs in $\Sigma(c)$ for some $c \in \mathsf{C}(d')$. Σ is well-formed *if $\leq_\Sigma$ is a partial order whose strict part $<_\Sigma$ is well-founded.*

Note that if $c \in \mathsf{C}(d)$ has type $\forall \imath.\Pi \boldsymbol{X}.\boldsymbol{\theta} \to d^{\overline{\imath}}\boldsymbol{X}$, then $\imath$ is the sole stage variable occurring in $\boldsymbol{\theta}$. Hence, if c is non-recursive, $\mathsf{Inst}(c, s, \boldsymbol{\tau}, \sigma)$ is of the form $\boldsymbol{\theta}[\boldsymbol{X} := \boldsymbol{\tau}] \to \sigma$, and we write it $\mathsf{Inst}(c, \_, \boldsymbol{\tau}, \sigma)$.

Note also that well-formed signatures rule out heterogeneous datatypes, and for simplicity, mutually inductive datatypes also. Besides, the positivity requirement for $d^{\imath}\boldsymbol{X}$ is standard 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. [8].

In the remaining of the paper we assume given a well-formed signature Σ.

3.3 Terms and Reductions

Terms are built from variables, abstractions, applications, constructors, case-expressions, pairs, let-expressions and recursive definitions letrec. Recursive definitions come with case analysis for pattern matching, and let-expressions bind *pairs* of variables. We assume given a set $\mathcal{V}_{\mathcal{E}} = \{f, x, y, z, \ldots\}$ of *term variables*.

Definition 3.7. *The set $\mathcal{E}$ of* terms *is given by the syntax (where $|\tau| \in |\mathcal{T}|$):*

$$\begin{array}{rcl} e, e' & ::= & \mathcal{V}_{\mathcal{E}} \mid \lambda x : |\tau|.e \mid e\,e' \mid \Lambda X.e \mid e\,|\tau| \mid c \\ & \mid & \mathsf{case}_{|\tau|}\ e\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{e}\} \\ & \mid & \langle e, e' \rangle \mid \mathsf{let}\ \langle x, x' \rangle = e\ \mathsf{in}\ e' \\ & \mid & \mathsf{letrec}_{|\tau|}\ f\ \mathsf{case}\ \{\boldsymbol{c^{nr}} \Rightarrow \boldsymbol{e^{nr}} \mid \boldsymbol{c^{r}} \Rightarrow \boldsymbol{e^{r}}\} \end{array}$$

Free and bound variables are defined as usual with the following proviso: in letrec expressions, the (fixpoint) variable f is bound in the branches $\boldsymbol{e^r}$ for the recursive constructors, but *not* in the branches $\boldsymbol{e^{nr}}$ for the non-recursive ones. Hence, by Barendregt convention, we may assume that $f \notin \boldsymbol{e^{nr}}$. This is important for the typing and reduction rules of letrec. Note that no stage variable occurs in a term $e \in \mathcal{E}$.

Substitutions are maps $\rho : (\mathcal{V}_{\mathcal{E}} \mapsto \mathcal{E}) \uplus (\mathcal{V}_{\mathcal{T}} \mapsto |\mathcal{T}|)$ of finite domain. The capture-avoiding application of ρ to e is denoted $e\rho$, but we may also write $e[\boldsymbol{x} := \rho(\boldsymbol{x}), \boldsymbol{X} := \rho(\boldsymbol{X})]$ when $\mathsf{dom}(\rho) = \boldsymbol{x} \uplus \boldsymbol{X}$. The reductions are as follows.

Definition 3.8 (Reductions). *The relation of $\beta\iota\mu\theta$-reduction $\rightarrow$ is the smallest rewrite relation containing $\mapsto_{\beta\iota\mu\theta}$, where*

$$(\lambda x : |\tau|.e)\ e' \ \mapsto_\beta\ e[x := e'] \qquad (\Lambda X.e)\ |\tau| \ \mapsto_\beta\ e[X := |\tau|]$$

$$\begin{array}{rl} \mathsf{case}_{|\tau|}\ (c_i\ |\boldsymbol{\sigma}|\ \boldsymbol{a})\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{e}\} & \mapsto_\iota\ e_i\ \boldsymbol{a} \\ \quad \mathsf{let}\ \langle x_1, x_2\rangle = \langle e_1, e_2\rangle\ \mathsf{in}\ e & \mapsto_\theta\ e[x_1 := e_1, x_2 := e_2] \end{array}$$

$$\begin{array}{rl} \mathsf{letrec}_{|\tau|}\ f\ \mathsf{case}\ \{\boldsymbol{c^{nr}} \Rightarrow \boldsymbol{e^{nr}} \mid \boldsymbol{c^r} \Rightarrow \boldsymbol{e^r}\}\ (c_i^{nr}\ |\boldsymbol{\sigma}|\ \boldsymbol{a}) & \mapsto_\mu\ e_i^{nr}\ \boldsymbol{a} \\ \mathsf{letrec}_{|\tau|}\ f\ \mathsf{case}\ \{\boldsymbol{c^{nr}} \Rightarrow \boldsymbol{e^{nr}} \mid \boldsymbol{c^r} \Rightarrow \boldsymbol{e^r}\}\ (c_i^{r}\ |\boldsymbol{\sigma}|\ \boldsymbol{a}) & \mapsto_\mu\ e_i^{r}[f := e_f]\ \boldsymbol{a} \end{array}$$

with $e_f = \mathsf{letrec}_{|\tau|}\ f\ \mathsf{case}\ \{\boldsymbol{c^{nr}} \Rightarrow \boldsymbol{e^{nr}} \mid \boldsymbol{c^r} \Rightarrow \boldsymbol{e^r}\}$.

The rewrite system $\mapsto_{\beta\iota\mu\theta}$ is orthogonal and thus confluent.

3.4 Typing Rules

The typing system is an extension of [3] with sized products and prenex bounded universal stage quantification. Recall that $\underline{\tau} \in \underline{\mathcal{T}}$ denotes a constrained type, ie. a type of the form $\forall \boldsymbol{\imath} \leq \boldsymbol{s}.\tau$ where $\tau \in \mathcal{T}$ is a sized type. The capture-avoiding substitution of τ for X in σ is written $\sigma[X := \tau]$.

Definition 3.9 (Typing). *A* context *is a map $\Gamma : \mathcal{V}_\mathcal{E} \mapsto \underline{\mathcal{T}}$ of finite domain. The* typing relation *is the smallest relation $K\,;\,\Gamma \vdash e : \underline{\tau}$ which is closed by the rules of Fig. 1.*

The positivity condition $\imath\ \mathsf{pos}\ \sigma$ in the rule (rec) is defined in the usual way [2]. Note that the expression $\lambda x : \mathrm{Nat}.x$ has type $\forall \imath.\mathrm{Nat}^\imath \rightarrow \mathrm{Nat}^\imath$.

The rule (rec) combines the usual rule of fixpoints (1) with the rule (case). The first premise line is the typing of the branches corresponding to non-recursive constructors. They can not depend on the fixpoint variable f. The others premises are the branches for the recursive constructors, which can depend on the fixpoint variable f. The intuition of the termination argument is the following. Assume that we typecheck the approximation of f at type $d^{\widehat{\jmath}}\boldsymbol{\tau} \rightarrow \theta[\imath := \widehat{\jmath}]$ and let $c_k^r : \forall \imath.\Pi \boldsymbol{X}.\boldsymbol{\theta} \rightarrow d^{\widehat{\imath}}\boldsymbol{X}$. The branch e_k^r corresponding to c_k^r must be of type $\boldsymbol{\theta}[\boldsymbol{X} := \boldsymbol{\tau}, \imath := \jmath] \rightarrow \theta[\imath := \widehat{\jmath}]$, provided that f is used with type $\forall \imath \leq \jmath.d^\imath\boldsymbol{\tau} \rightarrow \theta$. That is, only strictly less defined approximations of f can be used to type e_k^r.

The μ-reduction of fixpoints takes into account the difference between recursive and non-recursive constructors: the fixpoint variable is only substituted in the recursive branches.

Finally, a crucial point with constrained-based approaches is that the satisfiability of the constraints K in judgments $K\,;\,\Gamma \vdash e : \tau$ must be preserved by typing rules read bottom up. With general constraints systems like [4, 10], satisfiability tests of K during type-checking generate existential constraints. This is manageable when stages are interpreted by natural numbers, but this may not be the case when constraints have to be interpreted by countable ordinals. By restricting to *bounded universal quantifications* $\forall \imath \leq s.\tau$, type checking generates constraints of the form $\imath \leq s$, which are always satisfiable by $[\imath := s]$. As a

(var) $$\frac{}{K\,;\,\Gamma, x{:}\underline{\sigma} \vdash x : \underline{\sigma}}$$

(abs) $$\frac{K\,;\,\Gamma, x{:}\tau \vdash e : \sigma}{K\,;\,\Gamma \vdash \lambda x : |\tau|.e : \tau \to \sigma}$$

(app) $$\frac{K\,;\,\Gamma \vdash e : \tau \to \sigma \qquad K\,;\,\Gamma \vdash e' : \tau}{K\,;\,\Gamma \vdash e\ e' : \sigma}$$

(T-abs) $$\frac{K\,;\,\Gamma \vdash e : \sigma}{K\,;\,\Gamma \vdash \Lambda X.e : \Pi X.\sigma} \quad \text{if } X \notin \Gamma$$

(T-app) $$\frac{K\,;\,\Gamma \vdash e : \Pi X.\sigma}{K\,;\,\Gamma \vdash e\ |\tau| : \sigma[X := \tau]}$$

(S-gen) $$\frac{K, \imath \le s\,;\,\Gamma \vdash e : \underline{\tau}}{K\,;\,\Gamma \vdash e : \forall \imath \le s.\underline{\tau}} \quad \text{if } \imath \notin \Gamma, K, s$$

(S-inst) $$\frac{K\,;\,\Gamma \vdash e : \forall \imath \le s.\underline{\tau} \qquad K \vdash r \le s}{K\,;\,\Gamma \vdash e : \underline{\tau}[\imath := r]}$$

(cons) $$\frac{c \in \mathsf{C}(d) \text{ for some } d}{K\,;\,\Gamma \vdash c : \Sigma(c)}$$

(pair) $$\frac{K\,;\,\Gamma \vdash e_1 : d_1^{s_1}\ \boldsymbol{\tau_1} \qquad K\,;\,\Gamma \vdash e_2 : d_2^{s_2}\ \boldsymbol{\tau_2}}{K\,;\,\Gamma \vdash \langle e_1, e_2\rangle : d_1\ \boldsymbol{\tau_1} \times^{s_1+s_2} d_2\ \boldsymbol{\tau_2}}$$

(let) $$\frac{\begin{array}{l}K\,;\,\Gamma \vdash e : d_1\ \boldsymbol{\tau_1} \times^{s} d_2\ \boldsymbol{\tau_2} \\ K, \imath_1 + \imath_2 \le s\,;\,\Gamma, x_1{:}d_1^{\imath_1}\ \boldsymbol{\tau_1}, x_2{:}d_2^{\imath_2}\ \boldsymbol{\tau_2} \vdash e' : \sigma\end{array}}{K\,;\,\Gamma \vdash \mathsf{let}\ \langle x_1, x_2\rangle = e\ \mathsf{in}\ e' : \sigma}$$
where $\imath_1, \imath_2 \notin \Gamma, K, \sigma, s, \boldsymbol{\tau_1}, \boldsymbol{\tau_2}$

(case) $$\frac{\begin{array}{c}K\,;\,\Gamma \vdash e : \widehat{d^{s}}\boldsymbol{\tau} \\ \mathsf{C}(d) = \{c_1, \ldots, c_n\} \quad K\,;\,\Gamma \vdash e_k : \mathsf{Inst}(c_k, s, \boldsymbol{\tau}, \theta) \quad (1 \le k \le n)\end{array}}{K\,;\,\Gamma \vdash \mathsf{case}_{|\theta|}\ e\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{e}\} : \theta}$$

(rec) $$\frac{\begin{array}{l}\mathsf{C}_{nr}(d) = \{c_1^{nr}, \ldots, c_n^{nr}\} \qquad K\,;\,\Gamma \vdash e_k^{nr} : \mathsf{Inst}(c_k^{nr}, \_, \boldsymbol{\tau}, \theta) \quad (1 \le k \le n) \\ \mathsf{C}_r(d) = \{c_1^r, \ldots, c_m^r\} \\ K\,;\,\Gamma, f{:}\forall \imath \le \jmath.d^{\imath}\boldsymbol{\tau} \to \theta \vdash e_k^r : \mathsf{Inst}(c_k^r, \jmath, \boldsymbol{\tau}, \theta[\imath := \widehat{\jmath}]) \quad (1 \le k \le m)\end{array}}{K\,;\,\Gamma \vdash \mathsf{letrec}_{|\tau|}\ f\ \mathsf{case}\ \{\boldsymbol{c^{nr}} \Rightarrow \boldsymbol{e^{nr}} \mid \boldsymbol{c^{r}} \Rightarrow \boldsymbol{e^{r}}\} : \forall \imath.d^{\imath}\boldsymbol{\tau} \to \theta}$$
where $\imath \notin K, \Gamma, \boldsymbol{\tau} \qquad \imath\ \mathsf{pos}\ \theta \qquad \jmath \notin \imath, K, \Gamma, \boldsymbol{\tau}, \theta \qquad |\tau| = d|\boldsymbol{\tau}| \to |\theta|$

(sub) $$\frac{K\,;\,\Gamma \vdash e : \underline{\sigma} \qquad K \vdash \underline{\sigma} \sqsubseteq \underline{\tau}}{K\,;\,\Gamma \vdash e : \underline{\tau}}$$

Fig. 1. Typing rules for $F_{\times}^{\widehat{}}$

consequence, the satisfiability of K in $K\,;\,\Gamma \vdash e : \tau$ is preserved by typing rules, and there is no need of satisfiability tests during type checking. That is why we have a tractable system with stage addition and all positive inductive types.

4 Subject Reduction

The proof of the subject reduction property relies on the usual intermediate properties, namely inversion and substitution.

For inversion of typing, it is convenient to work in an equivalent type system, in which stage quantification is independent from the introduction and elimination rules of term constructs.

Definition 4.1. *Let $F_\times^{\curlywedge\prime}$ be the type system identical to $F_\times^{\curlywedge}$, except for the rules (cons) and (rec) which are replaced with*

$$\textit{(cons')} \qquad \frac{c \in \mathsf{C}(d) \text{ for some } d \qquad \Sigma(c) = \forall \imath.\sigma}{K\,;\,\Gamma \vdash c : \sigma[\imath := s]}$$

$$\textit{(rec')} \qquad \frac{\begin{array}{l}\mathsf{C}_{nr}(d) = \{c_1^{nr}, \ldots, c_n^{nr}\} \qquad K\,;\,\Gamma \vdash e_k^{nr} : \mathsf{Inst}(c_k^{nr}, \_, \boldsymbol{\tau}, \theta) \quad (1 \le k \le n)\\ \mathsf{C}_r(d) = \{c_1^r, \ldots, c_m^r\}\\ K\,;\,\Gamma, f : \forall \imath \le \jmath.d^s\boldsymbol{\tau} \to \theta \vdash e_k^r : \mathsf{Inst}(c_k^r, \jmath, \boldsymbol{\tau}, \theta[\imath := \widehat{\jmath}]) \quad (1 \le k \le m)\end{array}}{\begin{array}{c}K\,;\,\Gamma \vdash \mathsf{letrec}_{|\tau|}\ f\ \mathsf{case}\ \{\boldsymbol{c^{nr}} \Rightarrow \boldsymbol{e^{nr}} \mid \boldsymbol{c^r} \Rightarrow \boldsymbol{e^r}\} : d^s\boldsymbol{\tau} \to \theta[\imath := s]\\ \textit{where} \quad \imath \notin K, \Gamma, \boldsymbol{\tau} \qquad \imath\ \mathsf{pos}\ \theta \qquad \jmath \notin \imath, K, \Gamma, \boldsymbol{\tau}, \theta \qquad |\tau| = d|\boldsymbol{\tau}| \to |\theta|\end{array}}$$

Lemma 4.2. *$K\,;\,\Gamma \vdash e : \underline{\tau}$ is derivable in $F_\times^{\curlywedge}$ iff it is derivable in $F_\times^{\curlywedge\prime}$.*

Proposition 4.3 (Inversion of stage quantification). *Let $e \notin \mathcal{V}_\mathcal{X}$. In $F_\times^{\curlywedge\prime}$, if $K\,;\,\Gamma \vdash e : \forall \boldsymbol{\imath} \le \boldsymbol{p}.\tau$ then there is a sized type σ such that $K, \boldsymbol{\imath} \le \boldsymbol{p} \vdash \sigma \sqsubseteq \tau$ and $K, \boldsymbol{\imath} \le \boldsymbol{p}\,;\,\Gamma \vdash e : \sigma$ is derivable in a derivation whose last rule is neither (S-gen), (S-inst) nor (sub).*

The main point in using $F_\times^{\curlywedge\prime}$ instead of $F_\times^{\curlywedge}$ in Prop. 4.3 is to ensure that the last rule of the derivation is the rule corresponding to the top symbol of e, when e is not a variable. This leads to the usual inversion properties for typing in $F_\times^{\curlywedge\prime}$, and thus in $F_\times^{\curlywedge}$ using Lem. 4.2.

Theorem 4.4 (Subject reduction). *In $F_\times^{\curlywedge}$,*

$$(K\,;\,\Gamma \vdash e_1 : \underline{\tau} \quad \wedge \quad e_1 \to e_2) \quad \Longrightarrow \quad K\,;\,\Gamma \vdash e_2 : \underline{\tau}$$

5 Strong Normalization

We outline a realizability proof that typable terms are strongly normalizing. We begin by the interpretation of stages, and then turn to the strong normalization proof itself, which relies on Tait's saturated sets.

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. Existing systems with stage addition [4, 10] are restricted to first-order inductive types, and stages constraints are formulas of Presburger arithmetic.

We go one step further by allowing at the same time all positive inductive types and stage addition. Stage addition is interpreted by the *natural addition* on countable ordinals. This operation is associative and commutative, in contrast with the usual ordinal addition which is in general *not* commutative.

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

5.1 The Stage Model

Stages are interpreted by ordinals below the first uncountable cardinal.

Definition 5.1 (Countable ordinals). *We denote by $(\Omega, \leq_\Omega)$ the well-ordered set of countable ordinals and by $+_\Omega$ the usual ordinal addition on Ω.*

Recall that $\mathbb{N}$ can be seen as a proper subset of Ω and that $+_\Omega$ coincide with the usual addition on $\mathbb{N}$. We want an associative and commutative addition on Ω, but $+_\Omega$ is in general not commutative on Ω. Instead, we use the *natural addition* on ordinals. To define it, we use the well-known fact that every $\alpha \in \Omega$ can be written in *Cantor normal form*,

$$\alpha \quad = \quad c_n.\omega^{\alpha_n} +_\Omega \cdots +_\Omega c_1.\omega^{\alpha_1}$$

where $\alpha_1 <_\Omega \ldots <_\Omega \alpha_n \in \Omega$ and $c_1, \ldots, c_n \in \mathbb{N}$. The *natural addition* $\oplus$ on Ω is then defined as

$$\begin{array}{ll} & (c_n.\omega^{\alpha_n} +_\Omega \cdots +_\Omega c_1.\omega^{\alpha_1}) \ \oplus \ (d_n.\omega^{\alpha_n} +_\Omega \cdots +_\Omega d_1.\omega^{\alpha_1}) \\ =_{\text{def}} & (c_n +_\Omega d_n).\omega^{\alpha_n} +_\Omega \cdots +_\Omega (c_1 +_\Omega d_1).\omega^{\alpha_1} \end{array}$$

Proposition 5.2. *The* natural addition *is associative, commutative and with neutral element* 0*. Moreover, for all $\alpha \in \Omega$, the successor ordinal of α is $\alpha \oplus 1$.*

We are now ready to define our stage model. Each inductive type can be interpreted using an induction up to a countable ordinal (see Prop. 5.9). We can thus interpret ∞ by Ω. This motivates the following definition.

Definition 5.3 (Stage model). *Let $\widehat{\Omega} =_{def} \Omega \cup \{\Omega\}$. For all $\alpha, \beta \in \widehat{\Omega}$, let*

$$\alpha < \beta \ \ \textit{iff} \ \ (\beta = \Omega \ \vee \ \alpha <_\Omega \beta) \qquad \textit{and} \qquad \alpha + \beta =_{def} \begin{cases} \alpha \oplus \beta & \textit{if } \alpha, \beta \in \Omega \\ \Omega & \textit{otherwise} \end{cases}$$

So we have an addition $+$ on stages which is monotone, associative and commutative. Moreover we have $\alpha + \beta < \Omega$ for all $\alpha, \beta < \Omega$.

Definition 5.4 (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 *$(\!|.|\!)_\pi : \mathcal{S} \mapsto \widehat{\Omega}$ as follows:*

$$(\!|0|\!)_\pi =_{def} 0 \qquad (\!|\infty|\!)_\pi =_{def} \Omega \qquad (\!|\widehat{s}|\!)_\pi =_{def} (\!|s|\!)_\pi + 1 \qquad (\!|s + r|\!)_\pi =_{def} (\!|s|\!)_\pi + (\!|r|\!)_\pi$$

We let $\pi \models K$ if $(\!|s|\!)_\pi \leq (\!|p|\!)_\pi$ for all $s \leq p \in K$, and let $K \models s \leq r$ if $\pi \models s \leq r$ for all π such that $\pi \models K$. K is satisfiable *if there is π such that $\pi \models K$.*

5.2 Type Interpretation

Let SN be the set of strongly normalizing terms. We interpret types by saturated sets. It is convenient to define them by means of *elimination contexts*:

$$\begin{array}{rcl} E[\,] & ::= & [\,] \mid E[\,]\, e \mid E[\,]\,|\tau| \mid \mathsf{case}_{|\tau|}\ E[\,]\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{e}\} \\ & \mid & \mathsf{let}\ \langle x, x'\rangle = E[\,]\ \mathsf{in}\ e' \mid \mathsf{letrec}_{|\tau|}\ f\ \mathsf{case}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{e}\}\ E[\,] \end{array}$$

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 $[\,]$.

Let $E[e] \to_{\mathrm{wh}} E[e']$ if $e \mapsto_{\beta\iota\mu\theta} e'$.

Definition 5.5 (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 $x \in \mathcal{V}_{\mathcal{X}}$,*
(SAT2) *if $e \in \mathsf{SN}$ and $e \to_{wh} e'$ for some $e' \in S$ then $e \in S$.*

It is well-known that $\mathsf{SN} \in \mathsf{SAT}$ and that $\bigcap \mathcal{Y}, \bigcup \mathcal{Y} \in \mathsf{SAT}$ for all non-empty $\mathcal{Y} \subseteq \mathsf{SAT}$. Hence, for each $X \subseteq \mathsf{SN}$ there is a smallest saturated set containing X, written $\overline{X}$.

As usual, the *function space* on SAT is given for $X, Y \in \mathsf{SAT}$ by:

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

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

$$\begin{array}{rcl} [\![\forall \iota \le s.\underline{\tau}]\!]^I_{\pi,\xi} & = & \bigcap\{[\![\underline{\tau}]\!]^I_{\pi(\iota:=\alpha),\xi} \mid \alpha \le (\!|s|\!)_\pi\} \\ [\![X]\!]^I_{\pi,\xi} & = & \xi(X) \\ [\![\tau \to \sigma]\!]^I_{\pi,\xi} & = & [\![\tau]\!]^I_{\pi,\xi} \to [\![\sigma]\!]^I_{\pi,\xi} \\ [\![\Pi X.\tau]\!]^I_{\pi,\xi} & = & \{e \mid \forall |\sigma| \in |\mathcal{T}|,\ \forall S \in \mathsf{SAT},\quad e\ |\sigma| \in [\![\tau]\!]^I_{\pi,\xi(X:=S)}\} \\ [\![d\boldsymbol{\tau} \times^s d'\boldsymbol{\tau'}]\!]^I_{\pi,\xi} & = & \bigcup\{\langle I_d([\![\boldsymbol{\tau}]\!]^I_{\pi,\xi}, \alpha), I_{d'}([\![\boldsymbol{\tau'}]\!]^I_{\pi,\xi}, \alpha')\rangle \mid \alpha + \alpha' \le (\!|s|\!)_\pi\} \\ [\![d^s\boldsymbol{\tau}]\!]^I_{\pi,\xi} & = & I_d([\![\boldsymbol{\tau}]\!]^I_{\pi,\xi}, (\!|s|\!)_\pi) \end{array}$$

where $\langle S_1, S_2\rangle =_{def} \overline{\{\langle e_1, e_2\rangle \mid e_1 \in S_1 \ \wedge\ e_2 \in S_2\}}$ for all $S_1, S_2 \in \mathsf{SAT}$.

Note that unions and intersections are always taken over non-empty sets of saturated sets.

We now define the interpretation of inductive datatypes. Recall that the relation $<_\Sigma$ is assumed to be a well-founded strict partial order (see Def. 3.6). The interpretation $(I_d)_{d\in\mathcal{D}}$ is defined by induction on $<_\Sigma$, and for each $d \in \mathcal{D}$, the map $I_d : \mathsf{SAT}^{\mathsf{ar}(d)} \times \widehat{\Omega} \mapsto \mathsf{SAT}$ is defined by induction on $\widehat{\Omega}$.

Substitution	$(\!\|p[\imath := s]\|\!)_\pi$	$=$	$(\!\|p\|\!)_{\pi(\imath:=(\!\|s\|\!)_\pi)}$
	$[\![\underline{\tau}[\imath := s]]\!]_{\pi,\xi}$	$=$	$[\![\underline{\tau}]\!]_{\pi(\imath:=(\!\|s\|\!)_\pi),\xi}$
	$[\![\underline{\tau}[X := \sigma]]\!]_{\pi,\xi}$	$=$	$[\![\underline{\tau}]\!]_{\pi,\xi(X:=[\![\sigma]\!]_{\pi,\xi})}$
Stage monotony	$\alpha \leq \beta$	$\Rightarrow$	$I_d(\boldsymbol{S},\alpha) \subseteq I_d(\boldsymbol{S},\beta)$
	$\alpha \leq \beta \ \wedge\ \imath \ \mathsf{pos}\ \theta$	$\Rightarrow$	$[\![\theta]\!]_{\pi(\imath:=\alpha),\xi} \subseteq [\![\theta]\!]_{\pi(\imath:=\beta),\xi}$
	$\alpha \leq \beta \ \wedge\ \imath \ \mathsf{neg}\ \theta$	$\Rightarrow$	$[\![\theta]\!]_{\pi(\imath:=\beta),\xi} \subseteq [\![\theta]\!]_{\pi(\imath:=\alpha),\xi}$
Substage soundness	$K \vdash s \leq p$	$\Rightarrow$	$K \models s \leq p$
Subtyping soundness	$K \vdash \underline{\tau} \sqsubseteq \underline{\sigma} \ \wedge\ \pi \models K$	$\Rightarrow$	$[\![\underline{\tau}]\!]_{\pi,\xi} \subseteq [\![\underline{\sigma}]\!]_{\pi,\xi}$

Fig. 2. Properties of the type interpretation

Definition 5.7. *For all $d \in \mathcal{D}$, all $\boldsymbol{S} \in \mathsf{SAT}^{\mathsf{ar}(d)}$ and all $\alpha \in \widehat{\Omega}$, we define $I_d(\boldsymbol{S},\alpha)$, by induction on pairs (d,α) ordered by $(<_\Sigma, <)_{lex}$, as follows:*

$$I_d(\boldsymbol{S},0) = \bigcup\{c^{nr}\,[\![\boldsymbol{\theta}]\!]^I_{\emptyset,\boldsymbol{X}:=\boldsymbol{S}} \mid c^{nr} \in \mathsf{C}_{nr}(d) \wedge \Sigma(c^{nr}) = \forall\imath.\Pi\boldsymbol{X}.\boldsymbol{\theta} \to d^\imath\boldsymbol{X}\}$$

$$I_d(\boldsymbol{S},\alpha\oplus 1) = \bigcup\{c\,[\![\boldsymbol{\theta}]\!]^I_{\imath:=\alpha,\boldsymbol{X}:=\boldsymbol{S}} \mid c \in \mathsf{C}(d) \wedge \Sigma(c) = \forall\imath.\Pi\boldsymbol{X}.\boldsymbol{\theta} \to d^{\overline{\imath}}\boldsymbol{X}\}$$

$$I_d(\boldsymbol{S},\lambda) = \bigcup\{I_d(\boldsymbol{S},\alpha) \mid \alpha < \lambda\} \qquad \text{if } \lambda \text{ is a limit ordinal}$$

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 $I_d(\boldsymbol{S},\alpha\oplus 1)$ only uses $c[\![\boldsymbol{\theta}]\!]^I_{\imath:=\alpha,\boldsymbol{X}:=\boldsymbol{S}}$ with $c \in \mathsf{C}(d)$, which in turn only uses $I_{d'}(\boldsymbol{U},\beta)$ with $(d',\beta)\,(<_\Sigma,<)_{\mathrm{lex}}\,(d,\alpha\oplus 1)$.

Definition 5.8. *Let* $[\![.]\!]_{\pi,\xi} =_{def} [\![.]\!]^I_{\pi,\xi}$.

Fig. 2 collects some essential 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. 5.10. The key-point is that for every *countable* $S \subseteq \Omega$, there is $\beta \in \Omega$ such that $\alpha < \beta$ for all $\alpha \in S$ [6].

Proposition 5.9. *For all $d \in \mathcal{D}$ and all $\boldsymbol{S} \in \mathsf{SAT}^{\mathsf{ar}(d)}$, there is an ordinal $\alpha < \Omega$ such that $I_d(\boldsymbol{S},\alpha) = I_d(\boldsymbol{S},\beta)$ for all β such that $\alpha \leq \beta \leq \Omega$.*

As usual, soundness is shown by induction on typing derivations. Note that if K is satisfied by π, then every K' occurring in the derivation of $K\,;\,\Gamma \vdash e : \underline{\tau}$ is satisfied by an extension of π. Thus, in contrast to [4], there is no need of tests of the form $K \vdash \exists\imath.P$ in typing derivations.

Theorem 5.10 (Typing soundness). *Given $\pi : \mathcal{V}_\mathcal{S} \mapsto \widehat{\Omega}$, $\xi : \mathcal{V}_\mathcal{T} \mapsto \mathsf{SAT}$ and $\rho : (\mathcal{V}_\mathcal{E} \mapsto \mathcal{E}) \uplus (\mathcal{V}_\mathcal{T} \mapsto |\mathcal{T}|)$, we let $(\pi,\xi,\rho) \models K;\Gamma$ if and only if $\pi \models K$ and $\rho(x) \in [\![\Gamma(x)]\!]_{\pi,\xi}$ for all $x \in \mathsf{dom}(\Gamma)$.*

If $K\,;\,\Gamma \vdash e : \underline{\tau}$, then $e\rho \in [\![\underline{\tau}]\!]_{\pi,\xi}$ for all π,ξ,ρ such that $(\pi,\xi,\rho) \models K;\Gamma$.

We deduce the strong normalization of terms typable with satisfiable K.

Corollary 5.11. *If $K\,;\,\Gamma \vdash e : \underline{\tau}$ with K satisfiable, then $e \in \mathsf{SN}$.*

6 Conclusion

$F^{\widehat{}}_{\times}$ is a variant of $F^{\widehat{}}$ that supports simple yet precise typing by using sized products instead of existential quantification, for which subject reduction is problematic. We have proved strong normalization and subject reduction of $F^{\widehat{}}_{\times}$, and conjecture that type-checking is tractable. On the other hand, size inference seems more difficult than in [3], in particular for precise annotations with addition such as for the function append on lists.

Our main next objective is to extend our results to the Calculus of Inductive Constructions, and to implement type-based termination in Coq. It would also be interesting to study the expressivity of more general forms of sized products, both with general container types instead of cartesian products, and arbitrary binary operators instead of $+$. Moreover, it would be interesting to study the tractability of more liberal subtyping relations for universal stage quantifications. Finally, an outstanding issue is the design of a strongly normalizing type system in which non-recursive constructors can be given the size zero while keeping fixpoint definitions separated from case analysis.

References

1. Abel, A.: Type-Base Termination. A Polymorphic Lambda-Calculus with Sized Higher-Order Types. PhD thesis, LMU University, Munich (2006)
2. Barthe, G., Frade, M.J., Giménez, E., Pinto, L., Uustalu, T.: Type-Based Termination of Recursive Definitions. Mathematical Structures in Computer Science 14(1), 97–141 (2004)
3. Barthe, G., Grégoire, B., Pastawski, F.: Practical Inference for Type-Based Termination in a Polymorphic Setting. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 71–85. Springer, Heidelberg (2005)
4. Blanqui, F., Riba, C.: Combining Typing and Size Constraints for Checking the Termination of Higher-Order Conditional Rewrite Systems. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 105–119. Springer, Heidelberg (2006)
5. Chin, W.-N., Khoo, S.-C.: Calculating Sized Types. Higher-Order and Symbolic Computation 14(2–3), 261–300 (2001)
6. Gallier, J.H.: 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)
7. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: Proceedings of POPL 1996, pp. 410–423. ACM, New York (1996)
8. Steffen, M.: Polarized Higher-order Subtyping. PhD thesis, Department of Computer Science, University of Erlangen (1997)
9. Tatsuta, M.: Simple Saturated Sets for Disjunction and Second-Order Existential Quantification. In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 366–380. Springer, Heidelberg (2007)
10. Xi, H.: Dependent Types for Program Termination Verification. Higher-Order and Symbolic Computation 15(1), 91–131 (2002)