Type-based termination of recursive definitions

G. BARTHE, M. J. FRADE, E. GIM&Eacute;NEZ, L. PINTO and T. UUSTALU

Mathematical Structures in Computer Science / Volume 14 / Issue 01 / February 2004, pp 97 - 141
DOI: 10.1017/S0960129503004122, Published online: 03 February 2004

Math. Struct. in Comp. Science (2004), *vol.* 14, *pp.* 97–141. © 2004 Cambridge University Press
DOI: 10.1017/S0960129503004122 Printed in the United Kingdom

Type-based termination of recursive definitions

G. BARTHE[†], M. J. FRADE[‡], E. GIMÉNEZ[§], L. PINTO[¶] and T. UUSTALU[‖]

[†]*INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France*

[‡]*Dep. de Informática, Universidade do Minho, Campus de Gualtar, P-4710-057 Braga, Portugal*

[§]*Trusted Logic, 5 rue du Bailliage, F-78000 Versailles, France and Instituto de Computación, Universidad de la República, Julio Herrera y Reissig 565, 11300 Montevideo, Uruguay*

[¶]*Dep. de Matemática, Universidade do Minho, Campus de Gualtar, P-4710-057 Braga, Portugal*

[‖]*Dep. de Informática, Universidade do Minho, Campus de Gualtar, P-4710-057 Braga, Portugal and Institute of Cybernetics, Akadeemia tee 21, EE-12618 Tallinn, Estonia*

Received 19 December 2000; revised 7 July 2002

This paper introduces $\widehat{\lambda}$, a simply typed lambda calculus supporting inductive types and recursive function definitions with termination ensured by types. The system is shown to enjoy subject reduction, strong normalisation of typable terms and to be stronger than a related system $\lambda_{\mathcal{G}}$ in which termination is ensured by a syntactic guard condition. The system can, at will, be extended to support coinductive types and corecursive function definitions also.

1. Introduction

Background Most functional programming languages (ML, Haskell, and so on) and proof development systems based on the proofs-as-programs paradigm of logic (Coq, HOL, PVS, and so on) rely on powerful type theories featuring inductive types such as natural numbers or lists. These languages come equipped with a mechanism for recursive definition of functions. However, there are significant differences between the mechanisms used in functional programming languages and in proof development systems.

The first difference concerns the termination of recursive functions. While in functional programming languages recursive functions are allowed to diverge, in proof development systems non-terminating functions must be banished from the language, as they almost always lead to logical paradoxes.

The second difference concerns the way in which recursive definitions are introduced. In functional programming languages, recursive functions are described in terms of a *pattern-matching operator* (`case`) and a general *fixpoint operator* (`let-rec`). For example, the addition of two natural numbers could be introduced as follows:

```
let rec plus n m =
  case m of
    0        -> n
 | (S p)     -> (S (plus n p))
 end
```

On the other hand, in the traditional presentations of type-based proof development systems (Coquand and Paulin 1990; Dybjer 1994; Luo 1994; Nordström *et al.* 1990), a recursive function $f : d \to \theta$ on an inductive type d is defined by means of the *elimination rule* of d, where both pattern matching and recursion are built into a single scheme that ensures termination. In this approach, the function `plus` can be encoded using the elimination rule of natural numbers $\mathrm{nat\_elim}\ \theta : \theta \to (\mathrm{Nat} \to \theta \to \theta) \to \mathrm{Nat} \to \theta$, which corresponds to the primitive recursion scheme:

```
let plus n m = (nat_elim nat n (fun p r -> (S r)) m)
```

This approach is theoretically sound. However, practice has shown that eliminators are rather cumbersome to use, whereas case-expressions and fixpoint expressions lead to more concise and readable definitions. In looking for a good compromise between termination and presentation issues, Coquand (1992) suggested that recursors should be replaced by case-expressions and a restricted form of fixpoint expressions, see also Giménez (1995). The restriction is imposed through a predicate $\mathscr{G}_f$ on untyped terms. This predicate enforces termination by constraining all recursive calls to be applied to terms smaller than the formal argument x of f – for instance, a pattern variable issued from a case expression on x. Hence the restricted typing rule for fixpoint expressions becomes:

$$\frac{f : \mathrm{Nat} \to \theta \vdash e : \mathrm{Nat} \to \theta}{\vdash (\mathsf{letrec}\ f = e) : \mathrm{Nat} \to \theta} \quad \text{if } \mathscr{G}_f(e) \qquad (*)$$

This alternative approach, called *guarded by destructors* recursion in Giménez (1995), has been implemented in the Coq system. Five years of experiments carried out with Coq have shown that it actually provides much more palatable representations of recursive functions.

However, the use of an external predicate $\mathscr{G}$ on untyped terms suffers from several weaknesses:

1 *The guard predicate is too syntax-sensitive and too weak.* The acceptance of a recursive definition becomes too sensitive to the syntactical shape of its body. Sometimes a small change in the definition could prevent it from satisfying the guardedness condition.

As an example, consider the following modification of the `plus` function, where the condition is no longer satisfied because of the introduction of a redex in the definition:

```
let comp f g x   = (f (g x))
let rec plus n m =
 case m of
    0         -> n
 | (S p)      -> (comp S (plus n) p)
 end
```

In addition, the guard predicate rejects many terminating recursive definitions such as the Euclidean division, Ackermann's function, or functions that swap arguments, such as subtyping algorithms for higher-order languages

```
let rec sub a a' =
 case a a' of
   (base b) (base b')         -> sub_base b b'
 | (fun b1 b2) (fun b'1 b'2)  -> (sub b'1 b1) && (sub b2 b'2)
 | ...                        -> ...
 end
```

2 *The guard predicate is hard to implement and hard to extend.* The guardedness condition is among the main sources of bugs in the implementation of the proof system. In order to improve the number of definitions accepted by the system, the guardedness condition has become more and more complicated, and thus prone to errors.

Besides, it is easier to extend the type system than to extend the guardedness condition: type conditions are expressed as local constraints associated to each construction of the language, whereas the guard predicate yields global constraints.

3 *The guard predicate is often defined on normal forms.* The guard predicate is often defined on normal forms only, which renders the typing rule (∗) useless in practice. Subsequently, the typing rule (∗) is usually replaced by the more liberal typing rule

$$\frac{f : \mathrm{Nat} \to \theta \ \vdash \ e : \mathrm{Nat} \to \theta}{\vdash \ (\mathsf{letrec}\, f = e) : \mathrm{Nat} \to \theta} \qquad \text{if } \mathcal{G}_f(\mathsf{nf}\ e)$$

where nf is the partial function associating the normal form to an expression. Now the modified rule introduces two further complications:

(a) *The new guard condition leads to inefficient type-checking.* Verifying the guardedness condition makes type-checking less efficient as the body of a recursive definition has to be reduced in order for it to be checked – expanding previously defined constants like the constant `comp` in the example above.

(b) *The new guard condition destroys strong normalisation.* For example, the normal form of the following definition satisfies the guardedness condition, but not the definition itself:

```
let K x y = x
let rec diverging_id n =
        case n of 0     -> K 0 (diverging_id n)
        |         (S p) -> S (diverging_id p)
        end
```

There is an infinite reduction sequence for the term `diverging_id 0`[†]:

$$\begin{aligned}\texttt{diverging\_id 0} &\to \texttt{(K 0 (diverging\_id 0))}\\ &\to \texttt{(K 0 (K 0 (diverging\_id 0)))}\\ &\to \dots\end{aligned}$$

One solution for getting around this problem (the solution has been considered for Coq) is to store recursive definitions with their bodies in normal forms, as enforced by the rule

$$\frac{f : \mathrm{Nat} \to \theta \ \vdash\ e : \mathrm{Nat} \to \theta}{\vdash\ (\mathsf{letrec}\, f = (\mathsf{nf}\ e)) : \mathrm{Nat} \to \theta} \qquad \text{if } \mathscr{G}_f(\mathsf{nf}\ e).$$

However, the rule has some severe drawbacks:

(1) proof terms become huge;

(2) the expressions being stored are not those constructed interactively by the user;

(3) the modified typing rule for fixpoint expressions is not syntax-directed, that is, one cannot guess the expression e appearing in the premise from the conclusion of the rule.

In order to circumvent those weaknesses, some authors have proposed semantically motivated type systems that ensure the termination of recursive definitions through typing (Giménez 1998; Amadio and Coupet-Grimal 1998; Barras 1999). The idea, which had already appeared in Mendler's work (Mendler 1991), consists of regarding an inductive type d as the least fixpoint of a monotonic operator $\widehat{-}^d$ on types, and to enforcing termination of recursive functions by requiring that the definition of $f : \widehat{\alpha}^d \to \theta$, where α may be thought of as a subtype of d, only relies on structurally smaller function calls, embodied by a function $f_{\mathrm{ih}} : \alpha \to \theta$. This approach to terminating recursion, which we call *type-based*, offers several advantages over the *guarded by destructors* approach. In particular, it addresses all the above-mentioned weaknesses.

This paper The purpose of this paper is to introduce $\lambda^{\widehat{}}$, a simply typed λ-calculus that supports type-based recursive definitions. Although greatly inspired by previous work by Giménez (Giménez 1998) and closely related to recent work by Amadio and Coupet (Amadio and Coupet-Grimal 1998), the technical machinery behind our system puts a slightly different emphasis on the interpretation of types. More precisely, we formalise the

[†] In fact, Coq 7.1 accepts this definition of `diverging_id`!

notion of type-based termination using a restricted form of type dependency (also known as indexed types), as popularised by Xi and Pfenning (1998; 1999). This leads to a simple and intuitive system that is robust under several extensions, such as mutually inductive datatypes and mutually recursive function definitions; however, such extensions are not treated in this paper.

The basic idea is to proceed as follows:

— First, 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. Here, we adopt a simple-minded approach and let stages range over the syntax

$$s := \imath \mid \widehat{s} \mid \infty$$

where $\imath$ ranges over stage variables, the hat operator $\widehat{-}$ is a function mapping a stage to its 'successor' and ∞ is the stage at which the iterative approximation process converges to the datatype itself.

— Second, a recursive definition of a function, say $f : d \to \theta$, should be given by a term e constructing a function $g' : d^{\widehat{\imath}} \to \theta$ from $g : d^{\imath} \to \theta$, where $\imath$ ranges over stages (in other words, e should be stage-polymorphic).

In order to illustrate the machinery involved, consider the inductive type Nat whose constructors are o : Nat and s : Nat$\to$Nat. The typing rules are

$$\frac{}{\vdash \mathsf{o} : \mathrm{Nat}^{\widehat{s}}} \qquad \frac{\vdash n : \mathrm{Nat}^{s}}{\vdash \mathsf{s}\, n : \mathrm{Nat}^{\widehat{s}}}$$

and, as an instance of the subsumption rule,

$$\frac{\vdash n : \mathrm{Nat}^{s}}{\vdash n : \mathrm{Nat}^{\widehat{s}}}$$

Finally, recursive functions from Nat to θ are constructed using the following typing rule:

$$\frac{f : \mathrm{Nat}^{\imath} \to \theta \vdash e : \mathrm{Nat}^{\widehat{\imath}} \to \theta}{\vdash (\mathsf{letrec}\, f = e) : \mathrm{Nat} \to \theta}$$

where $\imath$ is fresh with respect to θ. As we shall show later, such recursive functions are terminating and, despite its simplicity, this mechanism is powerful enough to capture course-of-value primitive recursion. Conformance with the scheme, and hence termination, is enforced through types.

Organisation of the paper The remainder of this paper is organised as follows. In Section 2 we present the system $\widehat{\lambda}$ formally. In Section 3 we show that $\widehat{\lambda}$ is well-behaved, and, in particular, enjoys subject reduction and strong normalisation. In Section 4 we introduce $\lambda_{\mathcal{G}}$ and prove that $\widehat{\lambda}$ strictly extends the system $\lambda_{\mathcal{G}}$. In Section 5 we consider an extension of $\widehat{\lambda}$ with coinductive types. We review related work in Section 6 and give conclusions in Section 7.

2. The System $\widehat{\lambda}$

In this section we will introduce $\widehat{\lambda}$, a simply typed lambda calculus featuring strongly positive, finitely iterated parametric inductive types (in the sense of, for example, Martin-Löf (1971)) and type-based termination of recursive definitions. The calculus is *à la* Curry: terms come without any type annotations.

2.1. *Datatypes and constructors*

Datatypes and constructors are named: we assume two denumerable sets $\mathscr{D}$ of *datatype identifiers* and $\mathscr{C}$ of *constructor identifiers*. We assume a stratification on datatypes that ensures that the dependency relation between datatypes is well-founded. Hence, each datatype d is assigned a stratum $\mathsf{str}(d) \in \mathbb{N}$. Datatypes and constructors may only accept a fixed number of arguments, so we stipulate that every datatype identifier d and constructor c have fixed *arities* $\mathsf{ar}(d) \in \mathbb{N}$ and $\mathsf{ar}(c) \in \mathbb{N}$ that indicate the number of parameters taken by d and c, respectively. Finally, we require that every datatype $d \in \mathscr{D}$ comes equipped with a vector $\mathsf{C}(d) \subseteq \mathscr{C}$ of constructors, and if $d \neq d'$, then $\mathsf{C}(d) \cap \mathsf{C}(d') = \emptyset$.

For the sake of clarity, we adopt the following naming conventions: $d, d', d_i, \ldots$ range over $\mathscr{D}$ and $c, c', c_i, \ldots$ range over $\mathscr{C}$.

2.2. *Terms and reduction*

Terms are built from variables, abstractions, applications, constructors, case-expressions and recursive definitions. Assume we have a denumerable set $\mathscr{V}_{\mathscr{E}}$ of *(object) variables*, and let $x, x', x_i, y, \ldots$ range over $\mathscr{V}_{\mathscr{E}}$.

Notation 2.1. For every set A, we let A^* denote the set of lists over A, and $\langle\rangle$ denote the empty list. $\vec{a}$ ranges over A^* if a ranges over A. $\#\vec{a}$ denotes the length of $\vec{a}$, and $\vec{a}[i]$ denotes, when it exists, the ith element of $\vec{a}$. For convenience, we will sometimes write lists in the form $\langle a_1, \ldots, a_n\rangle$ instead of $a_1 \ldots a_n$.

Definition 2.2 (Terms). The set $\mathscr{E}$ of *terms* is given by the abstract syntax

$$e, e' ::= x \mid \lambda x.\ e \mid e\ e' \mid c \mid \mathsf{case}\ e'\ \mathsf{of}\ \{\vec{c}\ \Rightarrow \vec{e}\} \mid (\mathsf{letrec}\ x = e)$$

where in the clause for case-expressions it is assumed that $\vec{c} = \mathsf{C}(d)$ for some $d \in \mathscr{D}$.

Free and bound variables, substitution, and so on, are defined as usual. We let $e[x := e']$ be the result of replacing all free occurrences of x in e by e'.

The reduction calculus is given by β-reduction for function application, ι-reduction for case analysis and μ-reduction for unfolding recursive definitions, which is only allowed in the context of application to a constructor application.

Definition 2.3 (Reduction calculus).

1 β-reduction $\to_\beta$ is defined as the compatible closure of the rule

$$(\lambda x.\ e)\ e' \ \to_\beta\ e[x := e'].$$

2 ι-reduction $\rightarrow_\iota$ is defined as the compatible closure of the rule

$$\mathsf{case}\ (c_i\ \vec{a})\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \ \rightarrow_\iota\ e_i\ \vec{a}$$

where $\#\vec{a} = \mathsf{ar}(c_i)$.

3 μ-reduction $\rightarrow_\mu$ is defined as the compatible closure of the rule

$$(\mathsf{letrec}\ f = e)\ (c\ \vec{a}) \ \rightarrow_\mu\ e[f := (\mathsf{letrec}\ f = e)]\ (c\ \vec{a})$$

where $\#\vec{a} = \mathsf{ar}(c)$.

4 $\beta\iota\mu$-reduction $\rightarrow_{\beta\iota\mu}$ is defined as $\rightarrow_\beta \cup \rightarrow_\iota \cup \rightarrow_\mu$.

Remark 2.4. Our formulation of the β- and μ-reduction rules relies on a variable convention: in the β-rule, the bound variables of e are assumed to be different from the free variables of e'; in the μ-rule, the bound and the free variables of e are assumed to be different.

The mechanics of the reduction calculus is illustrated by the following example.

Example 2.5. Consider the inductive type of natural numbers Nat with $\mathsf{C}(\mathrm{Nat}) = \{\mathsf{o}, \mathsf{s}\}$. Let plus $\equiv$ ($\mathsf{letrec}\ plus = \lambda x.\ \lambda y.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda x'.\ \mathsf{s}\ (plus\ x'\ y)\}$). The following is a reduction sequence that computes one plus two, where, as usual, $\twoheadrightarrow_\beta$ denotes the reflexive and transitive closure of $\rightarrow_\beta$.

$$\begin{aligned}
&\mathrm{plus}\ (\mathsf{s}\ \mathsf{o})\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o}))\\
&\qquad \rightarrow_\mu\ (\lambda x.\ \lambda y.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ y)\})\ (\mathsf{s}\ \mathsf{o})\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o}))\\
&\qquad \twoheadrightarrow_\beta\ \mathsf{case}\ \mathsf{s}\ \mathsf{o}\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{s}\ (\mathsf{s}\ \mathsf{o}) \mid \mathsf{s} \Rightarrow \lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o})))\}\\
&\qquad \rightarrow_\iota\ (\lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o}))))\ \mathsf{o}\\
&\qquad \rightarrow_\beta\ \mathsf{s}\ (\mathrm{plus}\ \mathsf{o}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o})))\\
&\qquad \rightarrow_\mu\ \mathsf{s}\ ((\lambda x.\ \lambda y.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ y)\})\ \mathsf{o}\ (\mathsf{s}(\mathsf{s}\ \mathsf{o})))\\
&\qquad \twoheadrightarrow_\beta\ \mathsf{s}\ (\mathsf{case}\ \mathsf{o}\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{s}\ (\mathsf{s}\ \mathsf{o}) \mid \mathsf{s} \Rightarrow \lambda x'.\ \mathsf{s}\ (\mathrm{plus}\ x'\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o})))\})\\
&\qquad \rightarrow_\iota\ \mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{o}))
\end{aligned}$$

2.3. *Types and typing system*

We now assume we are given two denumerable sets $\mathcal{V}_\mathcal{T}$ of *type variables* and $\mathcal{V}_\mathcal{S}$ of *stage variables*, and adopt the naming convention that $\alpha, \alpha', \alpha_i, \beta, \delta, \ldots$ range over $\mathcal{V}_\mathcal{T}$, and $\imath, \jmath, \ldots$ range over $\mathcal{V}_\mathcal{S}$. Proceeding from these, we define stage and type expressions. Stage expressions are built of stage variables, a symbol for the successor function on stages, and a symbol for the limit stage. A type expression is either a type variable, a function type expression or a datatype approximation expression.

Definition 2.6 (Stages and types).

1 The set $\mathcal{S}$ of *stage expressions* is given by the abstract syntax:

$$s, r ::= \imath \mid \infty \mid \widehat{s}.$$

2 The set $\mathscr{T}$ of *type expressions* is given by the abstract syntax:

$$\sigma, \tau ::= \alpha \mid \tau \to \sigma \mid d^s\ \vec{\tau}$$

where in the last clause, it is assumed that the length of $\vec{\tau}$ is exactly $\mathsf{ar}(d)$.

Notation 2.7. Very often we write $\vec{\tau} \to \sigma$ as an abbreviation for $\tau_1 \to \ldots \to \tau_n \to \sigma$, and $d\ \vec{\sigma}$ as an abbreviation for $d^\infty\ \vec{\sigma}$.

In order to present the typing rules for constructor and case expressions, we have to have a means for fixing the intended typings of the constructors. To this end, we introduce notions of constructor scheme and constructor scheme instantiation.

Definition 2.8 (Constructor scheme). A *constructor scheme* is a triple $(\delta, \vec{\alpha}, \vec{\sigma})$ where $\delta, \vec{\alpha} \in \mathscr{V}_{\mathscr{T}}$ and $\vec{\sigma} \in \mathscr{T}$ such that:

1 Each σ_i is positive with respect to δ, see Figure 5.
2 Each σ_i is positive with respect to each α_j, see Figure 5.
3 $\vec{\alpha}, \delta$ are pairwise distinct.
4 $\vec{\alpha}, \delta$ are the only type variables that can occur in $\vec{\sigma}$.
5 There are no occurrences of stage variables in $\vec{\sigma}$.

The set of constructor schemes is denoted by $\mathscr{CS}$.

Observe that type parameters can only appear positively in the argument types of the constructors. This makes it possible to parameterise the type of lists with respect to the type of elements, binary trees with respect to the type of node labels, and arbitrarily branching trees with respect to the type of node labels, but not with respect to the branching type.

In the rest of this paper, we assume a map $\mathsf{D} : \mathscr{C} \to \mathscr{CS}$ that respects arities: formally, for every datatype d and $c \in \mathsf{C}(d)$,

$$\mathsf{D}(c) = (\delta, \vec{\alpha}, \vec{\sigma}) \quad \text{with} \quad \#\vec{\alpha} = \mathsf{ar}(d) \quad \text{and} \quad \#\vec{\sigma} = \mathsf{ar}(c).$$

This mapping has to satisfy the following condition: if $c \in \mathsf{C}(d)$ and $\mathsf{D}(c) = (\delta, \vec{\alpha}, \vec{\sigma})$, any $d' \in \mathscr{D}$ appearing in $\vec{\sigma}$ satisfies $\mathsf{str}(d') < \mathsf{str}(d)$. This ensures that only finitely iterated inductive definitions are permitted (excluding mutual induction) and is made use of in the model construction (Definition 3.23) in the proof of strong normalisation.

Constructor schemes specify the possible typings for the arguments of each given constructor of every possible datatype: if δ is an approximation of the datatype, and $\vec{\alpha}$ are the parameters of the datatype, then $\vec{\sigma}$ is a possible typing for the arguments of the constructor.

Example 2.9. Consider $\mathrm{Bool}, \mathrm{Nat}, \mathrm{List}, \mathrm{Tree}, \mathrm{Ord} \in \mathscr{D}$. We have

$$\mathsf{C}(\mathrm{Bool}) = \{\mathsf{true}, \mathsf{false}\} \quad \begin{array}{lcl} \mathsf{D}(\mathsf{true}) & = & (\delta, \langle\rangle, \langle\rangle) \\ \mathsf{D}(\mathsf{false}) & = & (\delta, \langle\rangle, \langle\rangle) \end{array}$$

for the datatype of booleans;

$$\mathsf{C}(\mathrm{Nat}) = \{\mathsf{o}, \mathsf{s}\} \quad \begin{array}{lcl} \mathsf{D}(\mathsf{o}) & = & (\delta, \langle\rangle, \langle\rangle) \\ \mathsf{D}(\mathsf{s}) & = & (\delta, \langle\rangle, \langle\delta\rangle) \end{array}$$

$$\text{(refl)}\ \frac{}{s \preccurlyeq s} \qquad \text{(trans)}\ \frac{s \preccurlyeq r \quad r \preccurlyeq p}{s \preccurlyeq p} \qquad \text{(hat)}\ \frac{}{s \preccurlyeq \widehat{s}} \qquad \text{(infty)}\ \frac{}{s \preccurlyeq \infty}$$

Fig. 1. Stage comparison rules

$$\text{(refl)}\ \frac{}{\sigma \sqsubseteq \sigma} \qquad \text{(data)}\ \frac{s \preccurlyeq r \quad \tau_i \sqsubseteq \tau'_i \quad (1 \leqslant i \leqslant \mathsf{ar}(d))}{d^s \vec{\tau} \sqsubseteq d^r \vec{\tau'}} \qquad \text{(func)}\ \frac{\tau' \sqsubseteq \tau \quad \sigma \sqsubseteq \sigma'}{\tau \to \sigma \sqsubseteq \tau' \to \sigma'}$$

Fig. 2. Subtyping rules

for the datatype of natural numbers;

$$\mathsf{C}(\mathrm{List}) = \{\mathsf{nil}, \mathsf{cons}\} \quad \begin{array}{lcl} \mathsf{D}(\mathsf{nil}) & = & (\delta, \langle\alpha\rangle, \langle\rangle) \\ \mathsf{D}(\mathsf{cons}) & = & (\delta, \langle\alpha\rangle, \langle\alpha, \delta\rangle) \end{array}$$

for lists;

$$\mathsf{C}(\mathrm{Tree}) = \{\mathsf{branch}\} \quad \mathsf{D}(\mathsf{branch}) \ = \ (\delta, \langle\alpha\rangle, \langle\alpha, \mathrm{List}\ \delta\rangle)$$

for finitely branching trees; and

$$\mathsf{C}(\mathrm{Ord}) = \{\mathsf{zero}, \mathsf{succ}, \mathsf{lim}\} \quad \begin{array}{lcl} \mathsf{D}(\mathsf{zero}) & = & (\delta, \langle\rangle, \langle\rangle) \\ \mathsf{D}(\mathsf{succ}) & = & (\delta, \langle\rangle, \langle\delta\rangle) \\ \mathsf{D}(\mathsf{lim}) & = & (\delta, \langle\rangle, \langle \mathrm{Nat} \to \delta\rangle) \end{array}$$

for ordinals (or more exactly, for ordinal notations).

Each particular legal typing for the arguments of a constructor is obtained by instantiating the associated constructor scheme. The concept of instance of a constructor scheme is formally defined as follows.

Definition 2.10 (Instance). Let $d \in \mathscr{D}$, $c \in \mathsf{C}(d)$, $s \in \mathscr{S}$ and $\vec{\tau} \in \mathscr{T}$ such that $\#\vec{\tau} = \mathsf{ar}(d)$. Assume $\mathsf{D}(c) = (\delta, \vec{\alpha}, \vec{\sigma})$. An *instance of c with respect to s and $\vec{\tau}$* is defined as follows

$$\mathsf{Inst}_c^s\ \vec{\tau} = \vec{\sigma}[\delta := d^s\vec{\tau}][\vec{\alpha} := \vec{\tau}].$$

We now turn to the typing system. We introduce a comparison relation on the stages. Importantly, the stage comparison rules state that all stages beyond the limiting stage are equivalent. On top of the stage comparison relation, another set of rules defines a subtyping relation on types. A crucial fact stated by these rules is that a given approximation of a datatype is always included in the next one.

Definition 2.11 (Stage comparison and subtyping). τ is a *subtype* of σ, written $\tau \sqsubseteq \sigma$, is defined by the rules of Figure 2, where $s \preccurlyeq r$ is defined by the rules of Figure 1.

Notation 2.12. We write $\vec{\sigma} \sqsubseteq \vec{\tau}$ if $\#\vec{\sigma} = \#\vec{\tau}$ and $\sigma[i] \sqsubseteq \tau[i]$ for $i = 1..\#\vec{\sigma}$.

Lemma 2.13. If $\sigma \sqsubseteq \tau$ and $\tau \sqsubseteq \theta$, then $\sigma \sqsubseteq \theta$.

Lemma 2.14. If $\widehat{r} \preccurlyeq \widehat{s}$, then $r \preccurlyeq s$.

Proof. By induction on the proof of $p \preccurlyeq \widehat{s}$, one can show that $p \preccurlyeq \widehat{s}$ implies $p \preccurlyeq s$ or $p = \widehat{s}$, from which the claim can be inferred by instantiating $p = \widehat{r}$. □

Lemma 2.15. If $r \preccurlyeq s$, then $\mathsf{Inst}_c^r\ \vec{\tau} \sqsubseteq \mathsf{Inst}_c^s\ \vec{\tau}$.

In order to define the typing relation between terms and type expressions, we need the concepts of context and judgment.

Definition 2.16 (Contexts and judgments).

1 A *context* is a finite sequence $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, where $x_1, \ldots, x_n$ are pairwise disjoint (object) variables and $\sigma_1, \ldots, \sigma_n$ are types.
2 A *typing judgment* is a triple of the form $\Gamma \vdash e : \sigma$, where Γ is a context, e is a term and σ is a type expression.

The definition of the typing relation itself depends on that of subtyping.

Definition 2.17 (Typing).

1 A typing judgment is *derivable* if it can be inferred from the rules of Figure 3 where the positivity condition $\imath\ \mathsf{pos}\ \sigma$ in the (rec) rule is defined in Figure 4.
2 A term $e \in \mathscr{E}$ is typable if $\Gamma \vdash e : \sigma$ is derivable for some context Γ and type σ.

The rules (var), (abs) and (app) come from the standard simply typed λ-calculus. The rule (sub) is present in any λ-calculus with subtyping and provides a linkage between the

$$\text{(var)} \qquad \frac{}{\Gamma \vdash x : \sigma} \qquad \text{if } (x : \sigma) \in \Gamma$$

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

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

$$\text{(cons)} \qquad \frac{}{\Gamma \vdash c : \mathsf{Inst}_c^s\ \vec{\tau} \to d^{\widehat{s}}\vec{\tau}} \qquad \text{if } c \in \mathsf{C}(d)$$

$$\text{(case)} \qquad \frac{\Gamma \vdash e' : d^{\widehat{s}}\vec{\tau} \qquad \Gamma \vdash e_i : \mathsf{Inst}_{c_i}^s\ \vec{\tau} \to \theta \quad (1 \leqslant i \leqslant n)}{\Gamma \vdash \mathsf{case}\ e'\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} : \theta} \qquad \text{if } \mathsf{C}(d) = \{c_1, \ldots, c_n\}$$

$$\text{(rec)} \qquad \frac{\Gamma, f : d^{\imath}\vec{\tau} \to \theta \vdash e : d^{\widehat{\imath}}\vec{\tau} \to \theta[\imath := \widehat{\imath}] \qquad \imath\ \mathsf{pos}\ \theta}{\Gamma \vdash (\mathsf{letrec}\ f = e) : d^{s}\vec{\tau} \to \theta[\imath := s]} \qquad \text{if } \imath \text{ not in } \Gamma, \vec{\tau}$$

$$\text{(sub)} \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma \sqsubseteq \sigma'}{\Gamma \vdash e : \sigma'}$$

Fig. 3. Typing rules for $\lambda^{\frown}$

$$\text{(sp1)}\quad \frac{}{\iota\ \mathsf{pos}\ \alpha} \qquad\qquad \text{(sn1)}\quad \frac{}{\iota\ \mathsf{neg}\ \alpha}$$

$$\text{(sp2)}\quad \frac{\iota\ \mathsf{neg}\ \tau \quad \iota\ \mathsf{pos}\ \sigma}{\iota\ \mathsf{pos}\ \tau \to \sigma} \qquad\qquad \text{(sn2)}\quad \frac{\iota\ \mathsf{pos}\ \tau \quad \iota\ \mathsf{neg}\ \sigma}{\iota\ \mathsf{neg}\ \tau \to \sigma}$$

$$\text{(sp3)}\quad \frac{\iota\ \mathsf{pos}\ \tau_i \quad (1 \leqslant i \leqslant \mathsf{ar}(d))}{\iota\ \mathsf{pos}\ d^s\vec{\tau}} \qquad\qquad \text{(sn3)}\quad \frac{\iota\ \mathsf{nocc}\ s \quad \iota\ \mathsf{neg}\ \tau_i \quad (1 \leqslant i \leqslant \mathsf{ar}(d))}{\iota\ \mathsf{neg}\ d^s\vec{\tau}}$$

Fig. 4. Positive and negative occurrences of a stage variable

$$\text{(pos1)}\quad \frac{}{\beta\ \mathsf{pos}\ \alpha} \qquad\qquad \text{(neg1)}\quad \frac{\beta \neq \alpha}{\beta\ \mathsf{neg}\ \alpha}$$

$$\text{(pos2)}\quad \frac{\beta\ \mathsf{neg}\ \tau \quad \beta\ \mathsf{pos}\ \sigma}{\beta\ \mathsf{pos}\ \tau \to \sigma} \qquad\qquad \text{(neg2)}\quad \frac{\beta\ \mathsf{pos}\ \tau \quad \beta\ \mathsf{neg}\ \sigma}{\beta\ \mathsf{neg}\ \tau \to \sigma}$$

$$\text{(pos3)}\quad \frac{\beta\ \mathsf{pos}\ \tau_i \quad (1 \leqslant i \leqslant \mathsf{ar}(d))}{\beta\ \mathsf{pos}\ d^s\vec{\tau}} \qquad\qquad \text{(neg3)}\quad \frac{\beta\ \mathsf{neg}\ \tau_i \quad (1 \leqslant i \leqslant \mathsf{ar}(d))}{\beta\ \mathsf{neg}\ d^s\vec{\tau}}$$

Fig. 5. Positive and negative occurrences of a type variable

subtyping and typing relations. The remaining rules – (cons), (case) and (rec) – deserve some brief comments.

The (cons) rule says that applying a constructor of a given datatype to values in an approximation of the datatype gives a value that is guaranteed to be an element in the next approximation. The (case) rule says that the converse is also true: any value in the approximation next to some given one is a result of applying one of the constructors of the datatype to values in the given approximation and can therefore be subjected to case analysis. Finally, the (letrec) rule says that any systematic way of extending a function defined on a given approximation of a datatype to work also on the next approximation induces a function defined on the whole datatype, the limit of the approximations. The premiss of this rule involves an implicit universal quantification over the set of all stages (freshness condition!).

2.4. *Some examples*

We now give a few examples of programming to illustrate the mechanics and expressive power of our calculus. We start from simple recursive definitions that are definable in all useful existing systems.

Example 2.18 (Standard examples).

— *The addition of two natural numbers.* The only recursive call in this program is made on a depth-one recursive component of the argument value.

$$\begin{array}{lll}
\mathsf{plus} \equiv (\mathsf{letrec} & \multicolumn{2}{l}{plus_{:\mathrm{Nat}^{\imath}\to\mathrm{Nat}\to\mathrm{Nat}} =} \\
& \lambda x_{:\mathrm{Nat}^{\widehat{\imath}}}.\ \lambda y_{:\mathrm{Nat}}.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o}\ \Rightarrow\ y & \\
& & \mid \mathsf{s}\ \Rightarrow\ \lambda x'_{:\mathrm{Nat}^{\imath}}.\ \mathsf{s}\ \underbrace{(plus\ x'\ y)}_{:\mathrm{Nat}} \\
& & \} \\
) : & \mathrm{Nat}^{s} \to \mathrm{Nat} \to \mathrm{Nat} &
\end{array}$$

— *The concatenation of two lists and the concatenation of a list of lists.*

$$\begin{array}{lll}
\mathsf{append} \equiv & \multicolumn{2}{l}{(\mathsf{letrec}\ append_{:\mathrm{List}^{\imath}\tau\to\mathrm{List}\,\tau\to\mathrm{List}\,\tau} = \lambda x_{:\mathrm{List}^{\widehat{\imath}}\tau}.\ \lambda y_{:\mathrm{List}\,\tau}.} \\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{nil}\ \Rightarrow\ y & \\
& & \mid \mathsf{cons}\ \Rightarrow\ \lambda z_{:\tau}.\ \lambda x'_{:\mathrm{List}^{\imath}\tau}.\ \mathsf{cons}\ z\ \underbrace{(append\ x'\ y)}_{:\mathrm{List}\,\tau} \\
& & \} \\
) : & \mathrm{List}^{s}\ \tau \to \mathrm{List}\ \tau \to \mathrm{List}\ \tau &
\end{array}$$

$$\begin{array}{lll}
\mathsf{conc} \equiv & \multicolumn{2}{l}{(\mathsf{letrec}\ conc_{:\mathrm{List}^{\imath}(\mathrm{List}\,\tau)\to\mathrm{List}\,\tau} = \lambda x_{:\mathrm{List}^{\widehat{\imath}}(\mathrm{List}\,\tau)}.} \\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{nil}\ \Rightarrow\ \mathsf{nil} & \\
& & \mid \mathsf{cons}\ \Rightarrow\ \lambda z_{:\mathrm{List}\,\tau}.\ \lambda x'_{:\mathrm{List}^{\imath}(\mathrm{List}\,\tau)}.\ \mathsf{append}\ z\ \underbrace{(conc\ x')}_{:\mathrm{List}\,\tau} \\
& & \} \\
) : & \mathrm{List}^{s}\,(\mathrm{List}\,\tau) \to \mathrm{List}\,\tau &
\end{array}$$

— *The addition of two ordinals.*

$$\begin{array}{lll}
\mathsf{add} \equiv & \multicolumn{2}{l}{(\mathsf{letrec}\ add_{:\mathrm{Ord}^{\imath}\to\mathrm{Ord}\to\mathrm{Ord}} = \lambda x_{:\mathrm{Ord}^{\widehat{\imath}}}.\ \lambda y_{:\mathrm{Ord}}.} \\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{zero}\ \Rightarrow\ y & \\
& & \mid \mathsf{succ}\ \Rightarrow\ \lambda x'_{:\mathrm{Ord}^{\imath}}.\ \mathsf{succ}\ \underbrace{(add\ x'\ y)}_{:\mathrm{Ord}} \\
& & \mid \mathsf{lim}\ \Rightarrow\ \lambda x'_{:\mathrm{Nat}\to\mathrm{Ord}^{\imath}}.\ \underbrace{\mathsf{lim}\ (\lambda z_{:\mathrm{Nat}}.\ \underbrace{add\ \underbrace{(x'\ z)}_{\mathrm{Ord}^{\imath}}\ y}_{:\mathrm{Ord}})}_{:\mathrm{Ord}} \\
& & \} \\
) : & \mathrm{Ord}^{s} \to \mathrm{Ord} \to \mathrm{Ord} &
\end{array}$$

The following example illustrates the use of subsumption.

Example 2.19 (Example using subsumption). The predicate that decides whether a natural number is even or not may be defined as follows. This program involves a recursive call on a deep recursive component of the argument value. To type it, therefore, the subsumption rule has to be used.

$$
\begin{array}{ll}
(\mathsf{letrec} & \mathit{even}_{:\mathrm{Nat}^{\imath}\to\mathrm{Bool}} = \lambda x_{:\mathrm{Nat}^{\widehat{\imath}}}.\\
& \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{true}\\
& \qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x'_{:\mathrm{Nat}^{\imath}\sqsubseteq\mathrm{Nat}^{\widehat{\imath}}}.\ \mathsf{case}\ x'\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{false}\\
& \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x''_{:\mathrm{Nat}^{\imath}}.\ \underbrace{\mathit{even}\ x''}_{:\mathrm{Bool}}\ \}\\
& \qquad\qquad \}\\
):& \mathrm{Nat}^{s}\to\mathrm{Bool}
\end{array}
$$

The following examples demonstrate the specific, novel features of $\widehat{\lambda}$. First of all, stages provide a limited means of controlling the effect of a recursively defined function in terms of the relation between the depths of argument and result values.

Example 2.20 (Examples of 'exact' typings).

— *The length of a list*. This standard program for calculating the length of a list admits an unusually 'exact' (that is, tight and therefore informative) type in $\widehat{\lambda}$.

$$
\begin{array}{ll}
\mathsf{length} \equiv & (\mathsf{letrec}\ \mathit{length}_{:\mathrm{List}^{\imath}\tau\to\mathrm{Nat}^{\imath}} =\\
& \lambda x_{:\mathrm{List}^{\widehat{\imath}}\tau}.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{o}\\
& \qquad\qquad\qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z_{:\tau}.\ \lambda x'_{:\mathrm{List}^{\imath}\tau}.\ \underbrace{\mathsf{s}\ \underbrace{(\mathit{length}\ x')}_{:\mathrm{Nat}^{\imath}}}_{:\mathrm{Nat}^{\widehat{\imath}}}\\
& \qquad\qquad\qquad\qquad \}\\
):& \mathrm{List}^{s}\tau\to\mathrm{Nat}^{s}
\end{array}
$$

— *The map of a function on a list*. This program is very similar to that for the length function and also admits an 'exact' typing, but becomes crucial in an example below.

$$
\begin{array}{l}
\mathsf{map} \equiv \lambda f_{:\tau\to\sigma}.\ (\mathsf{letrec}\ \mathit{map}_{:\mathrm{List}^{\imath}\tau\to\mathrm{List}^{\imath}\sigma} = \lambda x_{:\mathrm{List}^{\widehat{\imath}}\tau}.\\
\qquad\qquad \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}\\
\qquad\qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda z_{:\tau}.\ \lambda x'_{:\mathrm{List}^{\imath}\tau}.\ \underbrace{\mathsf{cons}\ \underbrace{(f\ z)}_{:\sigma}\ \underbrace{(\mathit{map}\ x')}_{:\mathrm{List}^{\imath}\sigma}}_{:\mathrm{List}^{\widehat{\imath}}\sigma}\\
\qquad\qquad\qquad \}\\
\qquad)_{:\mathrm{List}^{s}\tau\to\mathrm{List}^{s}\sigma}
\end{array}
$$

Further, recursive calls are allowed on structurally smaller arguments that cannot be verified to be structurally smaller using a viable syntactic criterion.

Example 2.21 (Examples not handled by the guard condition).

— *The Euclidean division* $\lceil\frac{x}{y+1}\rceil$. This program for the Euclidean division depends on a program for subtraction. It is not typable in systems with a syntactic guard predicate, as, syntactically, (minus x' y) is not properly structurally smaller than x in the program below. In $\widehat{\lambda}$, it is typable because of the 'exact' type assignable to minus.

$$\begin{array}{ll}
\text{minus} \equiv & (\mathsf{letrec}\ \mathit{minus}_{:\mathrm{Nat}^{\imath}\to\mathrm{Nat}\to\mathrm{Nat}^{\imath}} = \lambda x_{:\mathrm{Nat}^{\hat{\imath}}}.\ \lambda y_{:\mathrm{Nat}}. \\
& \quad \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow x \\
& \qquad\qquad\quad \mid \mathsf{s} \Rightarrow \lambda x'_{:\mathrm{Nat}^{\imath}}.\ \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow x \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda y'_{:\mathrm{Nat}}.\ \underbrace{\mathit{minus}\ x'\ y'}_{:\mathrm{Nat}^{\imath}}\ \} \\
& \qquad\qquad\quad \} \\
)\ : & \mathrm{Nat}^{s} \to \mathrm{Nat} \to \mathrm{Nat}^{s}
\end{array}$$

$$\begin{array}{ll}
(\mathsf{letrec} & \mathit{div}_{:\mathrm{Nat}^{\imath}\to\mathrm{Nat}\to\mathrm{Nat}^{\imath}} = \\
& \lambda x_{:\mathrm{Nat}^{\hat{\imath}}}.\ \lambda y_{:\mathrm{Nat}}.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{o} \\
& \qquad\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x'_{:\mathrm{Nat}^{\imath}}.\ \mathsf{s}\ \underbrace{(\mathit{div}\ \underbrace{(\mathrm{minus}\ x'\ y)}_{:\mathrm{Nat}^{\imath}}\ y)}_{:\mathrm{Nat}^{\imath}} \\
& \qquad\qquad\qquad\qquad\qquad \} \\
)\ : & \mathrm{Nat}^{s} \to \mathrm{Nat} \to \mathrm{Nat}^{s}
\end{array}$$

— *Flattening of finitely branching trees.* This program depends on map and conc. As with div, it is not typable in systems with a syntactic guard predicate.

$$\begin{array}{ll}
(\mathsf{letrec} & \mathit{flatten}_{:\mathrm{Tree}^{\imath}\tau\to\mathrm{List}\ \tau} = \\
& \lambda x_{:\mathrm{Tree}^{\hat{\imath}}\tau}.\ \mathsf{case}\ x\ \mathsf{of}\ \{ \\
& \quad \mathsf{branch} \Rightarrow \lambda z_{:\tau}.\ \lambda x'_{:\mathrm{List}\ (\mathrm{Tree}^{\imath}\tau)}.\ \mathsf{cons}\ z\ \underbrace{(\mathrm{conc}\ \underbrace{(\mathrm{map}\ \mathit{flatten}\ x')}_{:\mathrm{List}\ (\mathrm{List}\ \tau)})}_{:\mathrm{List}\ \tau} \\
& \qquad\qquad \} \\
)\ : & \mathrm{Tree}^{s}\tau \to \mathrm{List}\ \tau
\end{array}$$

Finally, we give an example demonstrating the usefulness of having the capability of naming stages explicitly: if one recursion is nested in another, we need two distinct free stage variables.

Example 2.22 (Examples involving several stage variables). *The Ackermann function.* The natural definition of the Ackermann function is not typable in our system. A definition with two recursions, one nesting the other, however, is easy to type.

$$\begin{array}{ll}
\text{ack} \equiv & (\mathsf{letrec}\ \mathit{ack}_{:\mathrm{Nat}^{\imath}\to\mathrm{Nat}\to\mathrm{Nat}} = \lambda x_{:\mathrm{Nat}^{\hat{\imath}}}. \\
& \quad \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \underbrace{\lambda z.(\mathsf{s}\ z)}_{:\mathrm{Nat}\to\mathrm{Nat}} \\
& \qquad\qquad \mid \mathsf{s} \Rightarrow \lambda x'_{:\mathrm{Nat}^{\imath}}.\ (\mathsf{letrec}\ \mathit{ack\_x}_{:\mathrm{Nat}^{\jmath}\to\mathrm{Nat}} = \lambda y_{:\mathrm{Nat}^{\hat{\jmath}}}. \\
& \qquad\qquad\qquad\qquad \mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \underbrace{\mathit{ack}\ x'\ (\mathsf{s}\ \mathsf{o})}_{:\mathrm{Nat}} \\
& \qquad\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda y'_{:\mathrm{Nat}^{\jmath}}.\ \underbrace{\mathit{ack}\ x'\ \underbrace{(\mathit{ack\_x}\ y')}_{:\mathrm{Nat}}}_{:\mathrm{Nat}}\ \} \\
& \qquad\qquad\qquad\qquad)_{:\mathrm{Nat}\to\mathrm{Nat}} \\
& \qquad\qquad \} \\
)\ : & \mathrm{Nat}^{s} \to \mathrm{Nat} \to \mathrm{Nat}
\end{array}$$

3. Meta-theoretical results

In this section, we prove two important meta-theoretic properties of $\widehat{\lambda}$ – subject reduction and strong normalisability of typable terms.

3.1. *Subject reduction*

The proof of subject reduction for $\widehat{\lambda}$ is quite standard. Before going into this proof, some lemmas involving monotonicity and substitution properties for stages, as well as generation and substitution properties for typing, are considered.

Lemma 3.1 (Generation lemma for subtyping).

1 $\sigma \sqsubseteq \tau_1 \to \tau_2 \quad \Rightarrow \quad \sigma \equiv \tau'_1 \to \tau'_2 \;\wedge\; \tau_1 \sqsubseteq \tau'_1 \;\wedge\; \tau'_2 \sqsubseteq \tau_2$.
2 $\tau_1 \to \tau_2 \sqsubseteq \sigma \quad \Rightarrow \quad \sigma \equiv \tau'_1 \to \tau'_2 \;\wedge\; \tau'_1 \sqsubseteq \tau_1 \;\wedge\; \tau_2 \sqsubseteq \tau'_2$.
3 $\theta \sqsubseteq d^s\vec{\tau} \quad \Rightarrow \quad \theta \equiv d^r\vec{\sigma} \;\wedge\; r \preccurlyeq s \;\wedge\; \vec{\sigma} \sqsubseteq \vec{\tau}$.
4 $d^s\vec{\tau} \sqsubseteq \theta \quad \Rightarrow \quad \theta \equiv d^r\vec{\sigma} \;\wedge\; s \preccurlyeq r \;\wedge\; \vec{\tau} \sqsubseteq \vec{\sigma}$.
5 $\alpha \sqsubseteq \sigma \quad \Rightarrow \quad \sigma \equiv \alpha$.
6 $\sigma \sqsubseteq \alpha \quad \Rightarrow \quad \sigma \equiv \alpha$.

Proof. The proof is immediate by analysis of the subtyping rules. □

Lemma 3.2 (Generation lemma for typing).

1 $\Gamma \vdash x : \sigma \quad \Rightarrow \quad (x : \tau) \in \Gamma \;\wedge\; \tau \sqsubseteq \sigma$.
2 $\Gamma \vdash a\, b : \sigma \quad \Rightarrow \quad \Gamma \vdash a : \tau \to \sigma' \;\wedge\; \Gamma \vdash b : \tau \;\wedge\; \sigma' \sqsubseteq \sigma$.
3 $\Gamma \vdash \lambda x.e : \sigma \quad \Rightarrow \quad \sigma \equiv \tau_1 \to \tau_2 \;\wedge\; \Gamma, x : \tau'_1 \vdash e : \tau'_2 \;\wedge\; \tau_1 \sqsubseteq \tau'_1 \;\wedge\; \tau'_2 \sqsubseteq \tau_2$.
4 $\Gamma \vdash c : \sigma \quad \Rightarrow \quad \sigma \equiv \vec{\gamma} \to \theta \;\wedge\; \vec{\gamma} \sqsubseteq \mathsf{Inst}^s_c\ \vec{\tau} \;\wedge\; d^{\widehat{s}}\vec{\tau} \sqsubseteq \theta \;\wedge\; c \in \mathsf{C}(d)$.
5 $\Gamma \vdash \mathsf{case}\ a\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} : \sigma \quad \Rightarrow \quad \Gamma \vdash a : d^{\widehat{s}}\vec{\tau} \;\wedge\; \Gamma \vdash b_i : \mathsf{Inst}^s_{c_i}\ \vec{\tau} \to \theta \;\wedge\; \theta \sqsubseteq \sigma$.
6 $\Gamma \vdash \mathsf{letrec}\ f = e : \sigma \quad \Rightarrow \quad \Gamma, f : d^\imath\vec{\tau} \to \theta \vdash e : (d^\imath\vec{\tau} \to \theta)[\imath := \widehat{\imath}] \;\wedge\; (d^\imath\vec{\tau} \to \theta)[\imath := s] \sqsubseteq \sigma$ with $\imath \in \mathcal{V}_{\mathcal{S}}$, $\imath$ pos θ and $\imath$ fresh in $\Gamma, \vec{\tau}$.

Proof. The proof is by inspection on the derivation of the antecedent judgments. □

Lemma 3.3.

1 If $\imath$ pos θ and $r \preccurlyeq s$, then $\theta[\imath := r] \sqsubseteq \theta[\imath := s]$.
2 If $\imath$ neg θ and $r \preccurlyeq s$, then $\theta[\imath := s] \sqsubseteq \theta[\imath := r]$.
3 If $\tau \sqsubseteq \sigma$ and α pos θ, then $\theta[\alpha := \tau] \sqsubseteq \theta[\alpha := \sigma]$.
4 If $\tau \sqsubseteq \sigma$ and α neg θ, then $\theta[\alpha := \sigma] \sqsubseteq \theta[\alpha := \tau]$.

Proof. Properties 1 and 2 are proved by simultaneous induction on the structure of θ. Properties 3 and 4 are proved in the same way. □

Lemma 3.4.

1 If $\sigma \sqsubseteq \sigma'$, then $\sigma[\imath := s] \sqsubseteq \sigma'[\imath := s]$.
2 If $\sigma \sqsubseteq \sigma'$, then $\sigma[\alpha := \tau] \sqsubseteq \sigma'[\alpha := \tau]$.

Proof. The proof is by induction on the structure of σ. □

Lemma 3.5. If $r \preccurlyeq s$ and $\vec{\tau} \sqsubseteq \vec{\sigma}$, then $\mathsf{Inst}^r_c\ \vec{\tau} \sqsubseteq \mathsf{Inst}^s_c\ \vec{\sigma}$.

Proof. This follows from the previous lemmas. □

Lemma 3.6. $\Gamma_1, x : \tau, \Gamma_2, \Gamma_3 \vdash a : \sigma \Rightarrow \Gamma_1, \Gamma_2, x : \tau, \Gamma_3 \vdash a : \sigma$.

Proof. The proof is by induction on the derivation of $\Gamma_1, x : \tau, \Gamma_2, \Gamma_3 \vdash a : \sigma$. □

Lemma 3.7 (Substitution lemma).
If $\Gamma, x : \tau \vdash a : \sigma$ and $\Gamma \vdash b : \tau$, then $\Gamma \vdash a[x := b] : \sigma$.

Proof. The proof is by induction on the derivation of $\Gamma, x : \tau \vdash a : \sigma$. □

The following lemma shows the polymorphic nature of stage variables. In fact, in a derivable judgment a stage variable can be replaced throughout by a stage without affecting derivability.

Lemma 3.8. If $\Gamma \vdash a : \sigma$, then $\Gamma[\imath := s] \vdash a : \sigma[\imath := s]$.

Proof. Without loss of generality, one can assume $\imath$ nocc s, otherwise one could first apply this weaker version of the lemma with $\imath$ being replaced by a new stage variable κ (for the set of stage variables is infinite) and then use again the weaker version of the lemma with κ replaced by s.

We then use induction on the derivation of $\Gamma \vdash a : \sigma$. The only interesting case is when the last rule applied is (rec). (The other cases can be easily proved using the induction hypothesis.) Assume the last step is

$$\frac{\Gamma, f : d^{\jmath}\vec{\tau} \to \theta \vdash e : d^{\widehat{\jmath}}\vec{\tau} \to \theta[\jmath := \widehat{\jmath}] \qquad \jmath \ \mathsf{pos}\ \theta}{\Gamma \vdash (\mathsf{letrec}\, f = e) : d^{r}\vec{\tau} \to \theta[\jmath := r]} \quad \jmath \text{ fresh in } \Gamma, \vec{\tau}$$

A stage variable κ can be chosen such that κ is fresh in $\Gamma, \vec{\tau}, \theta$, $\kappa \neq \imath$ and κ nocc s. Then, from the induction hypothesis, $\Gamma, f : d^{\kappa}\vec{\tau} \to \theta[\jmath := \kappa] \vdash e : d^{\widehat{\kappa}}\vec{\tau} \to \theta[\jmath := \widehat{\jmath}][\jmath := \kappa]$. Therefore, $\Gamma, f : d^{\kappa}\vec{\tau} \to \theta[\jmath := \kappa] \vdash e : d^{\widehat{\kappa}}\vec{\tau} \to \theta[\jmath := \kappa][\kappa := \widehat{\kappa}]$ and therefore, using the induction hypothesis again,

$$\begin{aligned}\Gamma[\imath := s],\ f : d^{\kappa}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\imath := s] &\vdash \\ e \ :\ d^{\widehat{\kappa}}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\kappa := \widehat{\kappa}][\imath := s]&.\end{aligned}$$

Since κ nocc s, the substitutions $[\kappa := \widehat{\kappa}]$ and $[\imath := s]$ can be exchanged to give

$$\begin{aligned}\Gamma[\imath := s],\ f : d^{\kappa}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\imath := s] &\vdash \\ e \ :\ d^{\widehat{\kappa}}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\imath := s][\kappa := \widehat{\kappa}]&, \end{aligned} \tag{1}$$

and, as κ is fresh in $\Gamma[\imath := s]$ and in $\vec{\tau}[\imath := s]$, one can apply the rule (rec).

Let $u \equiv r[\imath := s]$. From (1) by (rec),

$$\Gamma[\imath := s] \vdash (\mathsf{letrec}\, f = e) : d^{u}\vec{\tau}[\imath := s] \to \theta[\jmath := \kappa][\imath := s][\kappa := u].$$

So, as κ nocc s, $\imath$ nocc s and κ is θ-fresh, $\Gamma[\imath := s] \vdash (\mathsf{letrec}\, f = e) : (d^{r}\vec{\tau})[\imath := s] \to \theta[\jmath := u][\imath := s]$. Hence, $\Gamma[\imath := s] \vdash (\mathsf{letrec}\, f = e) : (d^{r}\vec{\tau} \to \theta[\jmath := r])[\imath := s]$. □

We are now ready to prove that $\widehat{\lambda}$ enjoys the property of subject reduction.

Proposition 3.1 (Subject reduction).

$$\Gamma \vdash e_1 : \sigma \quad \wedge \quad e_1 \to_{\beta\iota\mu} e_2 \quad \Rightarrow \quad \Gamma \vdash e_2 : \sigma.$$

Proof. The proof is by induction on the derivation of $\Gamma \vdash e_1 : \sigma$. The interesting cases are when the last rule applied is (app) or (case):

(app) Assume $e_1 \equiv a\,b$ and the last step is

$$\frac{\Gamma \vdash a : \tau_1 \to \tau_2 \qquad \Gamma \vdash b : \tau_1}{\Gamma \vdash a\,b : \tau_2}.$$

Then one may have the following cases:

$e_2 \equiv e[x := b]$, with $a \equiv \lambda x.e$

From the typing derivation for a, using Lemma 3.2, it follows that $\Gamma, x : \tau_1' \vdash e : \tau_2'$, $\tau_1 \sqsubseteq \tau_1'$ and $\tau_2' \sqsubseteq \tau_2$, and from the typing derivation for b, by (sub), we derive $\Gamma \vdash b : \tau_1'$. Thus, by Lemma 3.7, $\Gamma \vdash e[x := b] : \tau_2'$, and, finally, by the rule (sub), $\Gamma \vdash e[x := b] : \tau_2$.

$e_2 \equiv (e[f := (\mathsf{letrec}\, f = e)])\,(c\,\vec{a})$, with $a \equiv (\mathsf{letrec}\, f = e)$ and $b \equiv (c\,\vec{a})$

Applying Lemma 3.2 to the typing derivation for a,

$$\Gamma, f : d^{\imath}\vec{\tau} \to \theta \vdash e : (d^{\imath}\vec{\tau} \to \theta)[\imath := \hat{\imath}], \tag{2}$$

$$(d^{\imath}\vec{\tau} \to \theta)[\imath := s] \sqsubseteq \tau_1 \to \tau_2, \tag{3}$$

$$\imath \in \mathscr{V}_{\mathscr{S}} \;\wedge\; \imath \;\mathsf{pos}\; \theta \;\wedge\; \imath \;\mathsf{fresh\ in}\; \Gamma, \vec{\tau}. \tag{4}$$

From (3) by Lemma 3.1, $\tau_1 \sqsubseteq d^{s}\vec{\tau} \;\wedge\; \theta[\imath := s] \sqsubseteq \tau_2$ and thus, using Lemma 3.1 again,

$$\tau_1 \equiv d^{p}\vec{\tau}' \quad \wedge \quad p \preccurlyeq s \quad \wedge \quad \vec{\tau}' \sqsubseteq \vec{\tau}.$$

From (2) and (4), by (rec), $\Gamma \vdash (\mathsf{letrec}\ f = e) : (d^{\imath}\vec{\tau} \to \theta)[\imath := q]$ holds for an arbitary stage q. Therefore, choosing $q \equiv \imath$ and taking (2) into account, by Lemma 3.7, we can derive

$$\Gamma \vdash e[f := (\mathsf{letrec}\, f = e)] : (d^{\imath}\vec{\tau} \to \theta)[\imath := \hat{\imath}]. \tag{5}$$

Thus we have $\Gamma \vdash (c\,\vec{a}) : d^{p}\vec{\tau}'$, so, by Lemmas 3.1 and 3.2, one of two possibilities for p must arise: $p \equiv \jmath^{n}$ with $n \geqslant 1$ or $p \equiv \infty^{m}$ with $m \geqslant 0$, where for a stage s and for $k \in \mathbb{N}$, s^{k} means s hatted k times.

— Case $p \equiv \jmath^{n}$ with $n \geqslant 1$. Using Lemma 3.8 on (5) with substitution $[\imath := \jmath^{(n-1)}]$, and since $\imath$ is fresh with respect to Γ and $\vec{\tau}$,

$$\Gamma \vdash e[f := (\mathsf{letrec}\, f = e)] : d^{\jmath^{n}}\vec{\tau} \to \theta[\imath := \jmath^{n}].$$

Thus, since $\Gamma \vdash (c\,\vec{a}) : \tau_1$ and $\tau_1 \equiv d^{p}\vec{\tau}'$, by (sub) and (app), we have $\Gamma \vdash e[f := (\mathsf{letrec}\, f = e)]\,(c\,\vec{a}) : \theta[\imath := \jmath^{n}]$. We have $\jmath^{n} \preccurlyeq s$ and $\imath$ pos θ, so, by Lemma 3.3, $\theta[\imath := \jmath^{n}] \sqsubseteq \theta[\imath := s]$ and the proof of this case is concluded using the rule (sub).

— Case $p \equiv \infty^{m}$ with $m \geqslant 0$. Observing that, by (sub), $\Gamma \vdash (c\,\vec{a}) : d^{\infty^{(m+1)}}\vec{\tau}'$, and the proof can now be completed arguing as in the previous case.

The remaining cases, where $e_2 \equiv a'\,b$ with $a \to_{\beta\iota\mu} a'$, or $e_2 \equiv a\,b'$ with $b \to_{\beta\iota\mu} b'$, follow by routine induction.

(case) Assume $e_1 \equiv \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$ and the last step is

$$\frac{\Gamma \vdash a : \widehat{d^s}\vec{\tau} \qquad \Gamma \vdash b_i : \mathsf{Inst}^s_{c_i}\ \vec{\tau} \to \theta \qquad (1 \leqslant i \leqslant n)}{\Gamma \vdash \mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \theta}.$$

Then we may have:

$e_2 \equiv b_i\,a_1 \ldots a_{\mathsf{ar}(c_i)}$, with $a \equiv c_i\,a_1 \ldots a_{\mathsf{ar}(c_i)}$

From $\Gamma \vdash c_i\,a_1 \ldots a_{\mathsf{ar}(c_i)} : \widehat{d^s}\vec{\tau}$, by Lemma 3.2, it follows that

$$\Gamma \vdash c_i : \vec{\gamma} \to \sigma \quad \wedge \quad \sigma \sqsubseteq \widehat{d^s}\vec{\tau}$$

and, also, for $1 \leqslant j \leqslant \mathsf{ar}(c_i)$

$$\Gamma \vdash a_j : \gamma_j \quad \wedge \quad \gamma_j \sqsubseteq \mathsf{Inst}^r_{c_i}\ \vec{\psi}[j] \quad \wedge \quad \widehat{d^r}\vec{\psi} \sqsubseteq \sigma.$$

So, $\widehat{d^r}\vec{\psi} \sqsubseteq \widehat{d^s}\vec{\tau}$, and therefore, by Lemmas 3.1 and 2.14, $r \preccurlyeq s$ and $\vec{\psi} \sqsubseteq \vec{\tau}$. Using Lemma 3.5 and the (sub) rule, we have $\Gamma \vdash a_j : \mathsf{Inst}^s_{c_i}\ \vec{\tau}[j]$ for $1 \leqslant j \leqslant \mathsf{ar}(c_i)$, which can be combined with the typing derivation of b_i, by means of the rule (app), to conclude the proof of this case.

The remaining cases,

$$e_2 \equiv \mathsf{case}\ a'\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}\ \text{with}\ a \to_{\beta\iota\mu} a',$$

and

$$e_2 \equiv \mathsf{case}\ a'\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_i \Rightarrow b'_i \mid \ldots \mid c_n \Rightarrow b_n\}\ \text{with}\ b_i \to_{\beta\iota\mu} b'_i,$$

follow by routine induction. □

3.2. *Strong normalisation*

As usual, we say that a term e is *strongly normalising* with respect to $\to_{\beta\iota\mu}$ if all $\beta\iota\mu$-reduction sequences starting with e terminate. Let SN denote the set of terms that are strongly normalising with respect to $\to_{\beta\iota\mu}$.

To prove that every typable term is in SN, we use the method of saturated sets. This is a standard technique, see, for example, Luo (1994). The idea is to provide the system with a semantics in which terms are interpreted as terms and type expressions as sets of terms known by construction only to contain strongly normalising terms and always to be non-empty (saturated sets). The strong normalisability of all typable terms then follows immediately as soon it is proved that the system is sound with respect to it.

3.2.1. *Saturated sets and interpretation domains* We start by defining the notions of base terms and key reduction (which is also known as weak head reduction).

Definition 3.10 (Base terms). The set Base of *base terms* is defined inductively as follows:

— $\mathscr{V}_{\mathscr{E}} \subseteq \mathsf{Base}$.
— If $b \in \mathsf{Base}$ and $e \in \mathsf{SN}$, then $b\,e \in \mathsf{Base}$.

— If $b \in \mathsf{Base}$ and $e_1, \ldots, e_n \in \mathsf{SN}$, then $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \in \mathsf{Base}$.
— If $b \in \mathsf{Base}$ and $e \in \mathsf{SN}$, then $(\mathsf{letrec}\ f = e)\, b \in \mathsf{Base}$.

Every base term is strongly normalising.

Lemma 3.11. $\mathsf{Base} \subseteq \mathsf{SN}$.

Definition 3.12 (Key reduction). The relation of *key reduction* between terms is defined inductively as follows:

— If e is a $\beta\iota\mu$-redex and e' is the contractum, then $e \rightarrow_k e'$.
— If $a \rightarrow_k a'$, then $a\, e \rightarrow_k a'\, e$.
— If $a \rightarrow_k a'$, then $\mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \rightarrow_k \mathsf{case}\ a'\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}$.
— If $a \rightarrow_k a'$, then $(\mathsf{letrec}\ f = e)\, a \rightarrow_k (\mathsf{letrec}\ f = e)\, a'$.

Key reduction commutes with reduction in the following sense.

Lemma 3.13. If $a \rightarrow_k b$ and $a \rightarrow a' \neq b$, then $a' \rightarrow_k b'$ and $b \twoheadrightarrow b'$ for some b'.

The following two lemmas provide sufficient conditions for an expression to be strongly normalising.

Lemma 3.14.

1 If $a \in \mathsf{SN}$, $a \rightarrow_k b$ and $b\ e \in \mathsf{SN}$, then $a\ e \in \mathsf{SN}$.
2 If $a \in \mathsf{SN}$, $a \rightarrow_k b$ and $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \in \mathsf{SN}$, then $\mathsf{case}\ a\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \in \mathsf{SN}$.
3 If $a \in \mathsf{SN}$, $a \rightarrow_k b$ and $(\mathsf{letrec}\ f = e)\ b \in \mathsf{SN}$, then $(\mathsf{letrec}\ f = e)\ a \in \mathsf{SN}$.

Proof. We will just prove (1). Suppose $a \in \mathsf{SN}$, $a \rightarrow_k b$ and $b\ e \in \mathsf{SN}$. First note that $e \in \mathsf{SN}$ as $b\ e \in \mathsf{SN}$. The proof of $a\ e \in \mathsf{SN}$ is by simultaneous induction on '$a \in \mathsf{SN}$' and '$e \in \mathsf{SN}$'. We have to prove that $c \in \mathsf{SN}$ for any one c such that $a\ e \rightarrow c$. As a can be neither a lambda-abstraction nor a letrec, there are two cases: either $c = a'\ e$ and $a \rightarrow a'$ or $c = a\ e'$ and $e \rightarrow e'$.

— Suppose $c = a'\ e$ and $a \rightarrow a'$. If $a' = b$, then $c = a'\ e \in \mathsf{SN}$, since $b\ e \in \mathsf{SN}$. Otherwise, by Lemma 3.13, there is b' such that $a' \rightarrow_k b'$ and $b \twoheadrightarrow b'$. We have $a' \in \mathsf{SN}$ (as $a \in \mathsf{SN}$), $a' \rightarrow_k b'$ and $b'\ e \in \mathsf{SN}$ (as $b\ e \in \mathsf{SN}$). By the induction hypothesis, $c = a'\ e \in \mathsf{SN}$.
— Suppose $c = a\ e'$ and $e \rightarrow e'$. We have $a \in \mathsf{SN}$, $a \rightarrow_k b$, $b\ e' \in \mathsf{SN}$ (as $b\ e \in \mathsf{SN}$). By the induction hypothesis, $c = a\ e' \in \mathsf{SN}$. □

Lemma 3.15.

1 If $a, e, a[x := e] \in \mathsf{SN}$, then $(\lambda x.\ a)\ e \in \mathsf{SN}$.
2 If $\vec{a}, e_1, \ldots, e_n, e_i\ \vec{a} \in \mathsf{SN}$, then $\mathsf{case}\ (c_i\ \vec{a})\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} \in \mathsf{SN}$.
3 If $\vec{a}, e, e[f := (\mathsf{letrec}\ f = e)]\ (c\ \vec{a}) \in \mathsf{SN}$, then $(\mathsf{letrec}\ f = e)\ (c\ \vec{a}) \in \mathsf{SN}$.

Next we define saturated sets and state some of their closure properties. Saturated sets are sets of strongly normalising terms containing the base terms and closed with respect to key expansion.

Definition 3.16 (Saturated sets).

1 A set $X \subseteq \mathscr{E}$ is said to be a *saturated set*, if

— $X \subseteq \mathsf{SN}$,

— $\mathsf{Base} \subseteq X$,

— if $a \in \mathsf{SN}$ and $a \to_k a'$ for some $a' \in X$, then $a \in X$.

The set of all saturated sets is denoted by SAT.

2 For any $X \subseteq \mathscr{E}$, let $\ulcorner X \urcorner = \{a \in \mathsf{SN} \mid \exists b \in \mathsf{Base} \cup X . a \twoheadrightarrow_k b\}$.

The following lemma establishes some basic properties of the closure operator $\ulcorner \cdot \urcorner$.

Lemma 3.17.

1 If $X \subseteq \mathsf{SN}$, then $\ulcorner X \urcorner$ is a saturated set, in fact, the smallest saturated set containing X.
2 $\ulcorner X_1 \cup \ldots \cup X_n \urcorner = \ulcorner X_1 \urcorner \cup \ldots \cup \ulcorner X_n \urcorner$.
3 If X_i is a saturated set for any $i \in I$, then $\bigcup i \in I . X_i$ is a saturated set. (We say that $\bigcup i \in \varnothing . X_i = \ulcorner \varnothing \urcorner$.)

On saturated sets, we can define a function-space forming operation. This is needed for the interpretation of function-space types.

Definition 3.18. For any $X, Y \subseteq \mathscr{E}$, let $X \to Y = \{a \in \mathscr{E} \mid \forall e \in X . a\, e \in Y\}$.

Lemma 3.19. If X and Y are saturated sets, then so is $X \to Y$.

Proof. Suppose X and Y are saturated. Clearly any $a \in X \to Y$ is strongly normalising: as X is non-empty, we can pick some $e \in X$, and then $a \in \mathsf{SN}$ because $a\, e \in Y \subseteq \mathsf{SN}$. Let us check that $X \to Y$ satisfies the conditions of saturatedness.

— Suppose $b \in \mathsf{Base}$ and consider any $e \in X$. As $e \in \mathsf{SN}$, we have that $b\ e \in \mathsf{Base} \subseteq Y$. Hence $b \in X \to Y$.
— Suppose $a \in \mathsf{SN}$, $a \to_k a'$ and $a' \in X \to Y$. We have to show that $a \in X \to Y$, that is, that, for any $e \in X$, $a\, e \in Y$. Consider any $e \in X$. We have $a\ e \to_k a'\ e$ and $a'\ e \in Y \subseteq \mathsf{SN}$, hence Lemma 3.14 applies and $a\ e \in \mathsf{SN}$. Since Y is saturated, we get $a\ e \in Y$. □

3.2.2. *Type and term interpretation* In the following we define a semantics of the language of stages, types and terms, and show that the rules of stage comparison, subtyping and typing are sound with respect to that semantics. Types will be interpreted as saturated sets of terms, terms will be interpreted as terms.

We start with the definitions of valuations and interpretation for stages and types. Stages will be interpreted as ordinals below Ω, the first uncountable ordinal, types as saturated sets of terms. Inductive types are interpreted as limits of a monotone approximation process from below. As the universe, SN, is countable, the approximation process is guaranteed to converge before Ω.

Definition 3.20 (Stage valuation).

1 A *stage valuation* is a map $\pi : \mathscr{V}_{\mathscr{S}} \to \Omega + 1$.

2 For every stage valuation π, $\imath \in \mathscr{V}_{\mathscr{S}}$, and $x \in \Omega + 1$, the stage valuation $\pi(\imath := x)$ is defined as follows:

$$\pi(\imath := x)(\imath') = \begin{cases} x & \text{if } \imath' \equiv \imath \\ \pi(\imath') & \text{if } \imath' \not\equiv \imath. \end{cases}$$

Definition 3.21 (Interpretation of stages). Let π be a type valuation. The corresponding stage interpretation function $[\![.]\!]_\pi : \mathscr{S} \rightarrow \Omega + 1$ is defined as follows:

$$\begin{aligned} [\![\imath]\!]_\pi &= \pi(\imath) \text{ if } \imath \in \mathscr{V}_{\mathscr{S}} \\ [\![\infty]\!]_\pi &= \Omega \\ [\![\hat{s}]\!]_\pi &= \begin{cases} [\![s]\!]_\pi + 1 & \text{if } [\![s]\!]_\pi < \Omega \\ [\![s]\!]_\pi & \text{if } [\![s]\!]_\pi = \Omega. \end{cases} \end{aligned}$$

Definition 3.22 (Type valuation).

1 A *type valuation* is a map $\xi : \mathscr{V}_{\mathscr{T}} \rightarrow \mathsf{SAT}$.
2 For every type valuation ξ, $\alpha \in \mathscr{V}_{\mathscr{T}}$ and $X \in \mathsf{SAT}$, the type valuation $\xi(\alpha := X)$ is defined as follows:

$$\xi(\alpha := X)(\alpha') = \begin{cases} X & \text{if } \alpha' \equiv \alpha \\ \xi(\alpha') & \text{if } \alpha' \not\equiv \alpha. \end{cases}$$

Definition 3.23 (Interpretation of types). Let π be a stage valuation and ξ a type valuation. The corresponding type interpretation function $[\![.]\!]_{\pi,\xi} : \mathscr{T} \rightarrow \mathsf{SAT}$ is defined by induction on heights (because of the stratification on datatype identifiers, every type has finite height):

$$\begin{aligned} [\![\alpha]\!]_{\pi,\xi} &= \xi(\alpha) \text{ if } \alpha \in \mathscr{V}_{\mathscr{T}} \\ [\![\tau \rightarrow \sigma]\!]_{\pi,\xi} &= [\![\tau]\!]_{\pi,\xi} \rightarrow [\![\sigma]\!]_{\pi,\xi} \\ [\![d^s \vec{\tau}]\!]_{\pi,\xi} &= D([\![\vec{\tau}]\!]_{\pi,\xi}, [\![s]\!]_\pi) \end{aligned}$$

where $D(\vec{X}, x)$ is defined by induction on x by

$$\begin{aligned} D(\vec{X}, 0) &= \ulcorner \varnothing \urcorner \\ D(\vec{X}, y+1) &= \ulcorner c_1\, [\![\vec{\sigma_1}]\!]_{\pi,\xi(\delta := D(\vec{X},y), \vec{\alpha} := \vec{X})} \cup \ldots \cup c_n\, [\![\vec{\sigma_n}]\!]_{\pi,\xi(\delta := D(\vec{X},y), \vec{\alpha} := \vec{X})} \urcorner \\ & \qquad \text{where } \mathsf{D}(c_i) = (\delta, \vec{\alpha}, \vec{\sigma_i}) \\ D(\vec{X}, x) &= \bigcup y < x.\, D(\vec{X}, y) \text{ if } x \text{ is a limit ordinal.} \end{aligned}$$

Lemma 3.24 (Substitution lemma for the interpretation of types).

1 $[\![\sigma[\imath := s]]\!]_{\pi,\xi} = [\![\sigma]\!]_{\pi(\imath := [\![s]\!]_\pi),\xi}$
2 $[\![\sigma[\alpha := \tau]]\!]_{\pi,\xi} = [\![\sigma]\!]_{\pi,\xi(\alpha := [\![\tau]\!]_{\pi,\xi})}$.

The following lemma states that the sequence of approximates of any datatype is non-decreasing with respect to set inclusion and converges before Ω.

Lemma 3.25.

1 If $X \subseteq X'$ and α pos σ, then $[\![\sigma]\!]_{\pi,\xi(\alpha := X)} \subseteq [\![\sigma]\!]_{\pi,\xi(\alpha := X')}$.
If $X \subseteq X'$ and α neg σ, then $[\![\sigma]\!]_{\pi,\xi(\alpha := X')} \subseteq [\![\sigma]\!]_{\pi,\xi(\alpha := X)}$.

2 If $x \leqslant x'$, then $D(\vec{X}, x) \subseteq D(\vec{X}, x')$.
3 $D(\vec{X}, \Omega + 1) = D(\vec{X}, \Omega)$.

Proof.

1 The proof is by mutual induction on the height of σ.
2 The proof is from (1), by induction on x.
3 The proof is from (2), using the fact that $\mathscr{E}$ is countable. The iteration process has to converge before Ω: the opposite would imply that $\mathscr{E}$ is uncountable, as Ω is uncountable. □

Lemma 3.26.

1 $[\![\widehat{d^s}\vec{\tau}]\!]_{\pi,\xi} = \ulcorner c_1\ [\![\mathsf{Inst}^s_{c_1}\ \vec{\tau}]\!]_{\pi,\xi} \cup \ldots \cup c_n\ [\![\mathsf{Inst}^s_{c_n}\ \vec{\tau}]\!]_{\pi,\xi}\urcorner$
2 $[\![d^\infty\vec{\tau}]\!]_{\pi,\xi} = \ulcorner c_1\ [\![\mathsf{Inst}^\infty_{c_1}\ \vec{\tau}]\!]_{\pi,\xi} \cup \ldots \cup c_n\ [\![\mathsf{Inst}^\infty_{c_n}\ \vec{\tau}]\!]_{\pi,\xi}\urcorner$.

Next we define valuations and interpretation for terms.

Definition 3.27 (Term valuation).

1 A *term valuation* is a map $\rho : \mathscr{V}_\mathscr{E} \to \mathscr{E}$.
2 For every term valuation ρ, $e \in \mathscr{E}$ and $x \in \mathscr{V}_\mathscr{E}$, the term valuation $\rho(x := e)$ is defined as follows:

$$\rho(x := e)(z) = \begin{cases} e & \text{if } z \equiv x \\ \rho(z) & \text{if } z \not\equiv x. \end{cases}$$

Definition 3.28 (Interpretation of terms). For any term valuation ρ, the map $([.])_\rho : \mathscr{E} \to \mathscr{E}$ is defined inductively as follows:

$$\begin{aligned} ([x])_\rho &= \rho(x) \\ ([\lambda x.\, e])_\rho &= \lambda x.([e])_{\rho(x:=x)} \\ ([e\, e'])_\rho &= ([e])_\rho\, ([e'])_\rho \\ ([c_k])_\rho &= c_k \\ ([\mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}])_\rho &= \mathsf{case}\ ([e])_\rho\ \mathsf{of}\ \{c_1 \Rightarrow ([e_1])_\rho \mid \ldots \mid c_n \Rightarrow ([e_n])_\rho\} \\ ([\mathsf{letrec}\ f = e])_\rho &= \mathsf{letrec}\ f = ([e])_{\rho(f:=f)}. \end{aligned}$$

Remark 3.29. The clauses for lambda-abstraction and letrec rely on a form of variable convention: namely, x and f, respectively, are assumed not to appear as a free variable in any of the terms $\rho(y)$ where y is free in e. Alternatively, without the convention, some variable renaming may be necessary: in the case of lambda-abstraction, one would set

$$([\lambda x.\, e])_\rho = \lambda x'.([e])_{\rho(x:=x')}$$

where x' is some variable free in no $\rho(y)$ where y is free in e.

Lemma 3.30 (Substitution lemma for the interpretation of terms).

— $([e[x := e']])_\rho = ([e])_{\rho(x:=([e'])_\rho)}$.
— $([e])_\rho = e[\vec{y} := \rho(\vec{y})]$ where $\vec{y}$ are the free variables of e.

The notion of satisfaction and validity are defined as usual: satisfaction of subtyping is set inclusion, and satisfaction of typing is set membership.

Definition 3.31 (Satisfaction, validity).

1 A stage valuation π *satisfies* a stage comparison judgment $s \preccurlyeq s'$ if $[\![s]\!]_\pi \leqslant [\![s']\!]_\pi$. A stage comparison judgment $s \preccurlyeq s'$ is valid, if every stage valuation satisfies it.
2 A stage valuation π and a type valuation ξ *satisfy* a subtyping judgment $\sigma \sqsubseteq \sigma'$ if $[\![\sigma]\!]_{\pi,\xi} \subseteq [\![\sigma']\!]_{\pi,\xi}$. A subtyping judgment $\sigma \sqsubseteq \sigma'$ is *valid* if every pair of stage and type valuations satisfies it.
3 A *valuation* is a triple (π, ξ, ρ), where π is a stage valuation, ξ is a type valuation and ρ is a term valuation.
4 Let (π, ξ, ρ) be a valuation.
   (a) (π, ξ, ρ) *satisfies a context* Γ, written $(\pi, \xi, \rho) \models \Gamma$, if $\rho(x) \in [\![\tau]\!]_{\pi,\xi}$ for each $(x : \tau) \in \Gamma$.
   (b) (π, ξ, ρ) *satisfies a typing judgment* $\Gamma \vdash e : \sigma$ if
$$(\pi, \xi, \rho) \models \Gamma \quad \Rightarrow \quad (\![e]\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}.$$
5 A typing judgment $\Gamma \vdash e : \sigma$ is *valid*, written $\Gamma \models e : \sigma$ if every valuation satisfies it.

3.2.3. *Soundness with respect to the semantics* Next we prove that the rules of $\widehat{\lambda}$ for stage comparison, subtyping and typing are sound with respect to the semantics just defined. The strong normalisation theorem follows as a corollary from the typing soundness.

Proposition 3.32 (Stage comparison soundness).

$$s \preccurlyeq s' \text{ derivable} \quad \Rightarrow \quad s \preccurlyeq s' \text{ valid.}$$

Proof. The proof is by induction on the derivation of $s \preccurlyeq s'$. □

Proposition 3.33 (Subtyping soundness).

$$\sigma \sqsubseteq \sigma' \text{ derivable} \quad \Rightarrow \quad \sigma \sqsubseteq \sigma' \text{ valid.}$$

Proof. The proof is by induction on the derivation of $\sigma \sqsubseteq \sigma'$. □

Lemma 3.34. Let ξ be a type valuation. Then:

1 If $\imath$ pos σ and $x \leqslant x'$, then $[\![\sigma]\!]_{\pi(\imath:=x),\xi} \subseteq [\![\sigma]\!]_{\pi(\imath:=x'),\xi}$.
2 If $\imath$ neg σ and $x \leqslant x'$, then $[\![\sigma]\!]_{\pi(\imath:=x),\xi} \supseteq [\![\sigma]\!]_{\pi(\imath:=x'),\xi}$.

Proof. The proof is by simultaneous induction on the structure of σ. □

Proposition 3.35 (Typing soundness).

$$\Gamma \vdash e : \sigma \text{ derivable} \quad \Rightarrow \quad \Gamma \models e : \sigma.$$

Proof. The proof is by induction on the derivation of $\Gamma \vdash e : \sigma$.

(var) Assume the last (and only) step is

$$\frac{}{\Gamma \vdash x : \tau} \quad \text{and} \quad (x : \tau) \in \Gamma.$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![x]\!)_\rho \in [\![\tau]\!]_{\pi,\xi}$. This is true, as $(x : \tau) \in \Gamma$.

(abs) Assume the last step is

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.\, e : \tau \to \sigma}.$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![\lambda x.\, e]\!)_\rho \in [\![\tau \to \sigma]\!]_{\pi,\xi}$. Since $[\![\tau \to \sigma]\!]_{\pi,\xi} = [\![\tau]\!]_{\pi,\xi} \to [\![\sigma]\!]_{\pi,\xi}$ and $(\![\lambda x.\, e]\!)_\rho = \lambda x.(\![e]\!)_{\rho_0}$, where $\rho_0 = \rho(x := x)$, this amounts to showing that $(\lambda x.\, (\![e]\!)_{\rho_0})\, a \in [\![\sigma]\!]_{\pi,\xi}$ for any $a \in [\![\tau]\!]_{\pi,\xi}$.
Observe first that, since $(\pi, \xi, \rho_0) \models \Gamma$ and $\rho_0(x) = x \in \mathscr{V}_{\mathscr{E}} \subseteq [\![\tau]\!]_{\pi,\xi}$, the induction hypothesis tells us that $(\![e]\!)_{\rho_0} \in [\![\sigma]\!]_{\pi,\xi} \subseteq \mathsf{SN}$.
Now suppose $a \in [\![\tau]\!]_{\pi,\xi} \subseteq \mathsf{SN}$, and let $\rho' = \rho(x := a)$. Since $(\pi, \xi, \rho') \models \Gamma$ and $\rho'(x) = a \in [\![\tau]\!]_{\pi,\xi}$, by the induction hypothesis, we get that $(\![e]\!)_{\rho'} \in [\![\sigma]\!]_{\pi,\xi} \subseteq \mathsf{SN}$. Write $\vec{y}$ for the free variables of e, then $(\lambda x.\, (\![e]\!)_{\rho_0})\, a \to_k (\![e]\!)_{\rho_0}[x := a] = e[\vec{y} := \rho_0(\vec{y})][x := a] = e[\vec{y} := \rho'(\vec{y})] = (\![e]\!)_{\rho'}$ (by the variable convention, Remark 3.29, x does not occur free in $\rho_0(\vec{y})$). By Lemma 3.15, $(\lambda x.\, (\![e]\!)_{\rho_0})\, a \in \mathsf{SN}$. As $[\![\sigma]\!]_{\pi,\xi}$ is a saturated set, we get that $(\lambda x.\, (\![e]\!)_{\rho_0})\, a \in [\![\sigma]\!]_{\pi,\xi}$.

(app) Assume the last step is

$$\frac{\Gamma \vdash e : \tau \to \sigma \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \sigma}.$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![e\, e']\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$. As $(\![e\, e']\!)_\rho = (\![e]\!)_\rho\, (\![e']\!)_\rho$, which amounts to showing that $(\![e]\!)_\rho\, (\![e']\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$.
As $(\pi, \xi, \rho) \models \Gamma$, the induction hypothesis gives $(\![e]\!)_\rho \in [\![\tau \to \sigma]\!]_{\pi,\xi} = [\![\tau]\!]_{\pi,\xi} \to [\![\sigma]\!]_{\pi,\xi}$ and $(\![e']\!)_\rho \in [\![\tau]\!]_{\pi,\xi}$. Thus $(\![e]\!)_\rho\, (\![e']\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$.

(cons) Assume the last (and the only) step is

$$\frac{}{\Gamma \vdash c_k : \mathsf{Inst}^s_{c_k}\ \vec{\tau} \to \widehat{d^s}\vec{\tau}} \quad \text{and} \quad k \in 1..n.$$

Suppose that $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![c_k]\!)_\rho \in [\![\mathsf{Inst}^s_{c_k}\ \vec{\tau} \to \widehat{d^s}\vec{\tau}]\!]_{\pi,\xi} = [\![\mathsf{Inst}^s_{c_k}\ \vec{\tau}]\!]_{\pi,\xi} \to [\![\widehat{d^s}\vec{\tau}]\!]_{\pi,\xi}$. As $(\![c_k]\!)_\rho = c_k$, this amounts to showing that $c_k\vec{a} \in [\![\widehat{d^s}\vec{\tau}]\!]_{\pi,\xi}$ for any $\vec{a} \in [\![\mathsf{Inst}^s_{c_k}\ \vec{\tau}]\!]_{\pi,\xi}$. But this holds trivially, since $[\![\widehat{d^s}\vec{\tau}]\!]_{\pi,\xi} = \ulcorner c_1\, [\![\mathsf{Inst}^s_{c_1}\ \vec{\tau}]\!]_{\pi,\xi} \cup \ldots \cup c_n\, [\![\mathsf{Inst}^s_{c_n}\ \vec{\tau}]\!]_{\pi,\xi} \urcorner$.

(case) Assume the last step is

$$\frac{\Gamma \vdash e : \widehat{d^s}\vec{\tau} \quad \Gamma \vdash e_1 : \mathsf{Inst}^s_{c_1}\ \vec{\tau} \to \theta \quad \ldots \quad \Gamma \vdash e_n : \mathsf{Inst}^s_{c_n}\ \vec{\tau} \to \theta}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} : \theta}.$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![\mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}]\!)_\rho \in [\![\theta]\!]_{\pi,\xi}$. As $(\![\mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}]\!)_\rho = \mathsf{case}\ (\![e]\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\}$, this amounts to showing that $\mathsf{case}\ (\![e]\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \in [\![\theta]\!]_{\pi,\xi}$.
As $(\pi, \xi, \rho) \models \Gamma$, from the induction hypothesis we get that $(\![e]\!)_\rho \in [\![\widehat{d^s}\vec{\tau}]\!]_{\pi,\xi} \subseteq \mathsf{SN}$ and $(\![e_k]\!)_\rho \in [\![\mathsf{Inst}^s_{c_k}\ \vec{\tau} \to \theta]\!]_{\pi,\xi} = [\![\mathsf{Inst}^s_{c_k}\ \vec{\tau}]\!]_{\pi,\xi} \to [\![\theta]\!]_{\pi,\xi} \subseteq \mathsf{SN}$ for each $k \in 1..n$.

Since $[\![d^{\widehat{s}}\vec{\tau}]\!]_{\pi,\xi} = \ulcorner c_1\ [\![\mathsf{Inst}^s_{c_1}\ \vec{\tau}]\!]_{\pi,\xi} \cup \ldots \cup c_n\ [\![\mathsf{Inst}^s_{c_n}\ \vec{\tau}]\!]_{\pi,\xi}\urcorner$, it must be the case that $(\![e]\!)_\rho \twoheadrightarrow_k b$ for some $b \in \mathsf{Base} \cup c_1\ [\![\mathsf{Inst}^s_{c_1}\ \vec{\tau}]\!]_{\pi,\xi} \cup \ldots \cup c_n\ [\![\mathsf{Inst}^s_{c_n}\ \vec{\tau}]\!]_{\pi,\xi}$.
From $b \in \mathsf{Base} \cup c_1\ [\![\mathsf{Inst}^s_{c_1}\ \vec{\tau}]\!]_{\pi,\xi} \cup \ldots \cup c_n\ [\![\mathsf{Inst}^s_{c_n}\ \vec{\tau}]\!]_{\pi,\xi}$, it follows that $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \in [\![\theta]\!]_{\pi,\xi} \subseteq \mathsf{SN}$. Indeed, if $b \in \mathsf{Base}$, then $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \in \mathsf{Base} \subseteq [\![\theta]\!]_{\pi,\xi}$, as $(\![e_k]\!)_\rho \in \mathsf{SN}$ for each $k \in 1..n$; if $b \in c_k\,[\![\mathsf{Inst}^s_{c_k}\ \vec{\tau}]\!]_{\pi,\xi}$ for some $k \in 1..n$, then $b = c_k\vec{a}$ for some $\vec{a} \in [\![\mathsf{Inst}^s_{c_k}\ \vec{\tau}]\!]_{\pi,\xi}$ and therefore $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \rightarrow_k (\![e_k]\!)_\rho\vec{a} \in [\![\theta]\!]_{\pi,\xi}$ and, by Lemma 3.15, $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \in \mathsf{SN}$, and hence $\mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \in [\![\theta]\!]_{\pi,\xi}$ as $[\![\theta]\!]_{\pi,\xi}$ is saturated.
From $(\![e]\!)_\rho \twoheadrightarrow_k b$ it follows that $\mathsf{case}\ (\![e]\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \twoheadrightarrow_k \mathsf{case}\ b\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\}$; furthermore, by Lemma 3.14, we get $\mathsf{case}\ (\![e]\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \in \mathsf{SN}$. Since $[\![\theta]\!]_{\pi,\xi}$ is saturated, we get that $\mathsf{case}\ (\![e]\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\![e_1]\!)_\rho \mid \ldots \mid c_n \Rightarrow (\![e_n]\!)_\rho\} \in [\![\theta]\!]_{\pi,\xi}$.

(rec) Assume the last step is

$$\frac{\Gamma, f : d^\imath\vec{\tau} \to \theta \ \vdash\ e : d^{\widehat{\imath}}\vec{\tau} \to \theta[\imath := \widehat{\imath}] \quad \imath\ \mathsf{pos}\ \theta}{\Gamma \ \vdash\ (\mathsf{letrec}\ f = e) : d^s\vec{\tau} \to \theta[\imath := s]} \quad \text{and } \imath \text{ fresh in } \Gamma,\ \vec{\tau}.$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![(\mathsf{letrec}\ f = e)]\!)_\rho \in [\![d^s\vec{\tau} \to \theta[\imath := s]]\!]_{\pi,\xi}$.
As $[\![d^s\vec{\tau} \to \theta[\imath := s]]\!]_{\pi,\xi} = [\![d^\imath\vec{\tau} \to \theta]\!]_{\pi_0,\xi} = [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi} \to [\![\theta]\!]_{\pi_0,\xi}$ and $(\![(\mathsf{letrec}\ f = e)]\!)_\rho = (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})$ where $\pi_0 = \pi(\imath := [\![s]\!]_\pi)$ and $\rho_0 = \rho(f := f)$, this amounts to showing that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,a \in [\![\theta]\!]_{\pi_0,\xi}$ for any $a \in [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi}$.
As $(\pi_0, \xi, \rho_0) \models \Gamma$ and $\rho_0(f) = f \in \mathscr{V}_{\mathscr{E}} \subseteq [\![d^\imath\vec{\tau} \to \theta]\!]_{\pi_0,\xi}$, by the induction hypothesis we get $(\![e]\!)_{\rho_0} \in [\![d^{\widehat{\imath}}\vec{\tau} \to \theta[\imath := \widehat{\imath}]]\!]_{\pi_0,\xi} \subseteq \mathsf{SN}$.
We prove our goal by induction on $\pi_0(\imath)$.

$(\pi_0(\imath) = 0)$ Suppose $a \in [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi} = \ulcorner\varnothing\urcorner \subseteq \mathsf{SN}$. Then $a \twoheadrightarrow_k b$ for some $b \in \mathsf{Base}$.
Since $(\![e]\!)_{\rho_0} \in \mathsf{SN}$, from $b \in \mathsf{Base}$ it follows that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,b \in \mathsf{Base} \subseteq [\![\theta]\!]_{\pi_0,\xi} \subseteq \mathsf{SN}$.
From $a \twoheadrightarrow_k b$ it follows that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,a \twoheadrightarrow_k (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,b$ and, by Lemma 3.14, $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,a \in \mathsf{SN}$.
Since $[\![\theta]\!]_{\pi_0,\xi}$ is a saturated set, we can conclude that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,a \in [\![\theta]\!]_{\pi_0,\xi}$.

$(\pi_0(\imath) = y + 1)$ Let $\pi' = \pi(\imath := y)$ and $\rho' = \rho(f := (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0}))$. As $(\pi', \xi, \rho') \models \Gamma$, and, as by the inner induction hypothesis $\rho'(f) = (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0}) \in [\![d^\imath\vec{\tau}]\!]_{\pi',\xi} \to [\![\theta]\!]_{\pi',\xi} = [\![d^\imath\vec{\tau} \to \theta]\!]_{\pi',\xi}$, we get by the outer induction hypothesis that $(\![e]\!)_{\rho'} \in [\![d^{\widehat{\imath}}\vec{\tau} \to \theta[\imath := \widehat{\imath}]]\!]_{\pi',\xi} = [\![d^\imath\vec{\tau} \to \theta]\!]_{\pi_0,\xi}$.
Suppose $a \in [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi} = \ulcorner c_1\ [\![\mathsf{Inst}^\imath_{c_1}\ \vec{\tau}]\!]_{\pi',\xi} \cup \ldots \cup c_n\ [\![\mathsf{Inst}^\imath_{c_n}\ \vec{\tau}]\!]_{\pi',\xi}\urcorner \subseteq \mathsf{SN}$. Then $a \twoheadrightarrow_k b$ for some $b \in \mathsf{Base} \cup c_1\ [\![\mathsf{Inst}^\imath_{c_1}\ \vec{\tau}]\!]_{\pi',\xi} \cup \ldots \cup c_n\ [\![\mathsf{Inst}^\imath_{c_n}\ \vec{\tau}]\!]_{\pi',\xi}$.
From $b \in \mathsf{Base} \cup c_1\ [\![\mathsf{Inst}^\imath_{c_1}\ \vec{\tau}]\!]_{\pi',\xi} \cup \ldots \cup c_n\ [\![\mathsf{Inst}^\imath_{c_n}\ \vec{\tau}]\!]_{\pi',\xi}$ we get that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,b \in [\![\theta]\!]_{\pi_0,\xi} \subseteq \mathsf{SN}$. Indeed, if $b \in \mathsf{Base}$, then $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,b \in \mathsf{Base} \subseteq [\![\theta]\!]_{\pi_0,\xi}$, since $(\![e]\!)_{\rho_0} \in \mathsf{SN}$; if $b \in c_k[\![\mathsf{Inst}^\imath_{c_k}\ \vec{\tau}]\!]_{\pi',\xi} \subseteq [\![d^\imath\vec{\tau}]\!]_{\pi_0,\xi}$ for some $k \in 1..n$, then $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,b \rightarrow_k (\![e]\!)_{\rho_0}[f := (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})]\,b = (\![e]\!)_{\rho'}\,b \in [\![\theta]\!]_{\pi_0,\xi}$, and, by Lemma 3.15, $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,b \in \mathsf{SN}$, and hence $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\,b \in [\![\theta]\!]_{\pi_0,\xi}$ as $[\![\theta]\!]_{\pi_0,\xi}$ is saturated.

From $a \twoheadrightarrow_k b$ it follows that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \twoheadrightarrow_k (\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, b$ and, by Lemma 3.14, $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \in \mathsf{SN}$.

Since $[\![\theta]\!]_{\pi_0,\xi}$ is a saturated set, we can conclude that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \in [\![\theta]\!]_{\pi_0,\xi}$.

($\pi_0(\imath) = x$ where x is a limit ordinal) Suppose that

$$a \in [\![d^{\imath}\vec{\tau}]\!]_{\pi_0,\xi} = \bigcup y < x.\ [\![d^{\imath}\vec{\tau}]\!]_{\pi(\imath:=y),\xi}.$$

Then $a \in [\![d^{\imath}\vec{\tau}]\!]_{\pi(\imath:=y),\xi}$ for some $y < x$. By the inner induction hypothesis and by the positivity of $\imath$ in θ, we therefore get that $(\mathsf{letrec}\ f = (\![e]\!)_{\rho_0})\, a \in [\![\theta]\!]_{\pi(\imath:=y),\xi} \subseteq [\![\theta]\!]_{\pi_0,\xi}$.

(sub) Assume the last step is

$$\frac{\Gamma \vdash e : \sigma \quad \sigma \sqsubseteq \sigma'}{\Gamma \vdash e : \sigma'}.$$

Suppose $(\pi, \xi, \rho) \models \Gamma$. We have to show that $(\![e]\!)_\rho \in [\![\sigma']\!]_{\pi,\xi}$. As $(\pi, \xi, \rho) \models \Gamma$, the induction hypothesis gives $(\![e]\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$ and the subtyping soundness gives $[\![\sigma]\!]_{\pi,\xi} \subseteq [\![\sigma']\!]_{\pi,\xi}$. Together, these give $(\![e]\!)_\rho \in [\![\sigma']\!]_{\pi,\xi}$. □

The main result of this subsection follows as an immediate corollary of the soundness of the typing system.

Proposition 3.36 (Strong normalisation). $\rightarrow_{\beta\imath\mu}$ is strongly normalising on typable expressions:

$$\Gamma \vdash e : \sigma \text{ derivable} \quad \Rightarrow \quad e \in \mathsf{SN}.$$

Proof. Assume $\Gamma \vdash e : \sigma$. Then, by Proposition 3.35, $\Gamma \models e : \sigma$. Consider a valuation (π, ξ, ρ) where, for every $x \in \mathscr{V}_{\mathscr{E}}$, $\rho(x) = x$. For every $(x : \tau) \in \Gamma$, we have $(\![x]\!)_\rho = x \in [\![\tau]\!]_{\pi,\xi}$, since $[\![\tau]\!]_{\pi,\xi}$ is saturated, hence $(\pi, \xi, \rho) \models \Gamma$. Therefore $(\![e]\!)_\rho \in [\![\sigma]\!]_{\pi,\xi}$. As $(\![e]\!)_\rho = e$, we have

$$e \in [\![\sigma]\!]_{\pi,\xi} \subseteq \mathsf{SN}.$$ □

4. The System $\lambda_{\mathscr{G}}$

In this section we present the system $\lambda_{\mathscr{G}}$, a simply typed λ-calculus with inductive types. The terms allowed in $\lambda_{\mathscr{G}}$ are the same as those allowed in $\widehat{\lambda}$. In particular, we continue to have the letrec constructor for defining functions recursively, but in $\lambda_{\mathscr{G}}$ (following Giménez (1995)) termination of typable recursively defined functions is ensured by a syntactical condition $\mathscr{G}$ constraining uses of recursive calls in the body of definitions. The condition $\mathscr{G}$ is checked directly on the body of the function and not on its normal form, because of the problem this would raise (as discussed in the introduction).

4.1. *The syntax of $\lambda_{\mathscr{G}}$*

The systems $\lambda_{\mathscr{G}}$ and $\widehat{\lambda}$ allow the same set of terms; they differ at the level of types in the following aspects:

1 Stages are not present in $\lambda_{\mathscr{G}}$, so datatypes are not annotated by stages.
2 There is no subtyping relation in $\lambda_{\mathscr{G}}$.

3 The set of typing rules is different, and $\lambda_{\mathscr{G}}$'s typing rule

$$\frac{\Gamma, f : d\,\vec{\tau} \to \sigma \;\vdash\; e : d\,\vec{\tau} \to \sigma \qquad \mathscr{G}^x_f(\emptyset, a)}{\Gamma \;\vdash\; (\mathsf{letrec}\, f = e) : d\,\vec{\tau} \to \sigma} \quad \text{if } \; e \equiv \lambda x.a$$

for letrec-expressions is complemented by the syntactical condition $\mathscr{G}$.

4 Following Giménez (1995), the datatypes allowed in $\lambda_{\mathscr{G}}$ are slightly more restricted than those of $\widehat{\lambda}$ for, in the argument types of the constructors of a datatype, such a datatype can only have strictly positive occurences; so throughout this section we assume that constructor schemes $(\delta, \vec{\alpha}, \vec{\sigma})$ are as in Definition 2.8 except that condition 1 is replaced by the condition: each σ_i is *strictly positive with respect to* δ, or in other words, if δ occurs in σ_i, σ_i is of the form $\vec{\gamma} \to \delta$ where $\vec{\gamma}$ has no occurrences of δ.

Let us focus on the letrec operator and on the syntactical condition $\mathscr{G}$ it satisfies. This condition complements the reduction rule $\to_\mu$, ensuring that each expansion of the letrec operator consumes (at least) the constructor in the head of its argument. Informally, for a term (letrec $f = e$) we should have the following:

1 f may occur in e only as the head of an application.

2 Any application of f must be protected by a case analysis of the formal argument of e, say x (for this reason f is said to be *guarded by destructors*); therefore f must occur inside e_i's in the following context:

$$\begin{array}{l} \mathsf{case}\; x\; \mathsf{of}\; \{\; c_1 \;\Rightarrow\; \lambda x_{11}.\ldots\lambda x_{1m_1}.\, e_1 \\ \qquad\qquad\quad \vdots \\ \qquad\qquad\quad c_n \;\Rightarrow\; \lambda x_{n1}.\ldots\lambda x_{nm_n}.\, e_n \\ \qquad\qquad \}. \end{array}$$

3 Considering that the *components* of x are the x_{ij} (*direct components*) together with the components of each x_{ij} (*inner components*), f must be applied to a term of the form $z\,\vec{a}$ where z is a *recursive component* of x (that is, z is a component of x whose type has occurrences of the type of x).

To illustrate the above observations, let us consider the examples already given in Section 2, *plus* and *even*, now transposed to $\lambda_{\mathscr{G}}$.

Example 4.1.

— *The addition of two natural numbers.*

$$\begin{array}{ll} (\mathsf{letrec} & plus = \lambda x.\,\lambda y.\,\mathsf{case}\; x\; \mathsf{of}\; \{\mathsf{o} \;\Rightarrow\; y \\ & \qquad\qquad\qquad\qquad\qquad\;\; \mid \mathsf{s} \;\Rightarrow\; \lambda n.\mathsf{s}\,(plus\; n\; y) \\ & \qquad\qquad\qquad\qquad\qquad\;\; \} \\ ): & \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat} \end{array}$$

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

— *A function that indicates whether or not a natural number is even.*

$$\begin{array}{lll}
(\mathsf{letrec} & \mathit{even} = \lambda x.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{true} & \\
 & \qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda y.\,\mathsf{case}\ y\ \mathsf{of}\ \{\mathsf{o} \Rightarrow \mathsf{false} & \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{s} \Rightarrow \lambda z.\,\mathit{even}\,z\,\} & \\
 & \qquad\qquad\qquad\qquad \} & \\
)\,: & \mathrm{Nat} \rightarrow \mathrm{Bool} &
\end{array}$$

In this example the application of *even* is guarded by a case analysis on the argument x. The argument of this application is the pattern variable z, which is an inner component of x that becomes available in the case analysis on the pattern variable y, which is a direct component of x.

The formal description of the guarded-by-destructors condition is provided by the predicate $\mathcal{G}^x_f(V, a)$ defined below. The V argument is a set of variables used to collect the pattern variables in a representing the recursive components of x. In order to identify the recursive components of a variable, we start by characterising the recursive positions of a constructor scheme as follows.

Definition 4.2. Let c be a $\lambda_{\mathcal{G}}$ constructor such that $\mathsf{D}(c) = (\delta, \vec{\alpha}, \vec{\sigma})$. We say that the number j corresponds to a *recursive position* of $\mathsf{D}(c)$, written $\mathsf{RP}(j, \mathsf{D}(c))$, if σ_j is of the form $\vec{\gamma} \rightarrow \delta$.

The predicate $\mathcal{G}$ is now defined as follows.

Definition 4.3 ($\mathcal{G}$ predicate). Let $U \subseteq \mathcal{V}$, let x and f be distinct variables not in U and let $a \in \mathcal{E}$. The predicate $\mathcal{G}^x_f(U, a)$ is derivable using the rules in Figure 6.

Lemma 4.4. If $f\ \mathsf{nocc}\ a$ then $\mathcal{G}^x_f(U, a)$.

Proof. The proof is by induction on the structure of a. □

One can check that the guard predicate holds on addition.

Example 4.5. The function *plus* of Example 4.1 can be shown guarded as follows

$$\dfrac{\dfrac{\dfrac{\mathit{plus} \neq y}{\mathcal{G}^x_{\mathit{plus}}(\emptyset, y)}\;1 \qquad \dfrac{\dfrac{}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, \mathsf{s})}\;4 \qquad \dfrac{\dfrac{\dfrac{\mathit{plus} \neq n}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, n)}\;1}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, \mathit{plus}\,n)}\;6 \qquad \dfrac{\mathit{plus} \neq y}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, y)}\;1}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, \mathit{plus}\,n\,y)}\;5}{\mathcal{G}^x_{\mathit{plus}}(\{n\}, \mathsf{s}\,(\mathit{plus}\,n\,y))}\;5}{\mathcal{G}^x_{\mathit{plus}}(\emptyset, \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda n.\mathsf{s}\,(\mathit{plus}\,n\,y)\})}\;8}{\mathcal{G}^x_{\mathit{plus}}(\emptyset, \lambda y.\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{o} \Rightarrow y \mid \mathsf{s} \Rightarrow \lambda n.\mathsf{s}\,(\mathit{plus}\,n\,y)\})}\;2$$

As suggested in the introduction, the predicate $\mathcal{G}$ is very sensitive to syntax. This is illustrated by the example below.

1. $$\frac{f \neq y}{\mathcal{G}^x_f(U, y)} \qquad \text{if } y \text{ is a variable}$$

2. $$\frac{\mathcal{G}^x_f(U, a)}{\mathcal{G}^x_f(U, \lambda z.a)}$$

3. $$\frac{\mathcal{G}^x_f(U, e)}{\mathcal{G}^x_f(U, \mathsf{letrec}\ g = e)}$$

4. $$\frac{}{\mathcal{G}^x_f(U, c)}$$

5. $$\frac{\mathcal{G}^x_f(U, a) \quad \mathcal{G}^x_f(U, b)}{\mathcal{G}^x_f(U, a\, b)}$$

6. $$\frac{\mathcal{G}^x_f(U, z\, \vec{a})}{\mathcal{G}^x_f(U, f\, (z\, \vec{a}))} \qquad \text{if } z \in U$$

7. $$\frac{\mathcal{G}^x_f(U, e) \quad \mathcal{G}^x_f(U, b_i) \quad (1 \leqslant i \leqslant n)}{\mathcal{G}^x_f(U, \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})} \qquad \text{if } \begin{cases} e \not\equiv z\, \vec{a} \\ \quad \vee \\ (e \equiv z\, \vec{a}\ \wedge\ z \notin U \cup \{x\}) \end{cases}$$

8. $$\frac{\mathcal{G}^x_f(U, a_j) \quad (1 \leqslant j \leqslant m) \quad \mathcal{G}^x_f(V_i, e_i) \quad (1 \leqslant i \leqslant n)}{\mathcal{G}^x_f(U, \mathsf{case}\ (z\, a_1 \ldots a_m)\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})}$$

$$\text{if } \begin{cases} z \in U \cup \{x\} \\ b_i \equiv \lambda y_1. \ldots \lambda y_{\mathsf{ar}(c_i)}.\, e_i \\ V_i \equiv U \cup \{y_j \mid \mathsf{RP}(j, \mathsf{D}(c_i)) \text{ for } 1 \leqslant j \leqslant \mathsf{ar}(c_i)\} \end{cases}$$

Fig. 6. Guarded-by-destructors rules

Example 4.6. Consider the following expression:

$$\begin{aligned} \mathsf{letrec}\quad plus = \lambda x.\, \lambda y.\, \mathsf{case}\ x\ \mathsf{of}\ \{&\mathsf{o} \Rightarrow y \\ \mid\ &\mathsf{s} \Rightarrow \lambda n.\mathsf{s}\,((\lambda g.\, g\, n\, y)\ plus) \\ \}& \end{aligned}$$

This expression also defines the addition of two natural numbers: it is obtained from the *plus* function defined in Example 4.1 by a β-expansion. However, this definition of *plus* does not satisfy condition $\mathcal{G}$ because the occurrence of *plus* in the letrec body is not the head of an application. So, when trying to prove the condition $\mathcal{G}$ we would have to derive $\mathcal{G}^x_{plus}(\{n\}, plus)$, which, by looking at the rules defining $\mathcal{G}$, we can immediately say is underivable.

Another example of an expression not satisfying the guard condition is the Euclidean division already considered.

(var) $$\frac{}{\Gamma \vdash x : \sigma} \qquad \text{if } (x : \sigma) \in \Gamma$$

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

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

(cons) $$\frac{}{\Gamma \vdash c : \mathsf{Inst}_c\ \vec{\tau} \to d\,\vec{\tau}} \qquad \text{if } c \in \mathsf{C}(d)$$

(case) $$\frac{\Gamma \vdash e : d\,\vec{\tau} \quad \Gamma \vdash e_i : \mathsf{Inst}_{c_i}\ \vec{\tau} \to \theta \quad (1 \leqslant i \leqslant n)}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} : \theta} \qquad \text{if } \mathsf{C}(d) = \{c_1, \ldots, c_n\}$$

(rec) $$\frac{\Gamma, f : d\,\vec{\tau} \to \sigma \vdash e : d\,\vec{\tau} \to \sigma \qquad \mathcal{G}_f^x(\emptyset, a)}{\Gamma \vdash (\mathsf{letrec}\ f = e) : d\,\vec{\tau} \to \sigma} \qquad \text{if } e \equiv \lambda x.a$$

Fig. 7. Typing rules for $\lambda_{\mathcal{G}}$

Example 4.7. The Euclidean division defined in Example 2.21 does not satisfy $\mathcal{G}$ for in the recursive call of the div function (div (minus x' y) y) its argument (minus x' y) is not a recursive component of its formal argument (x), but instead the result of applying the previously defined minus to a recursive component (x') of x.

We now turn to the typing system. First, one needs to define instances of constructors. The definition is almost identical to the one for $\widehat{\lambda}$, the only difference being the absence of stages.

Definition 4.8. Let $d \in \mathcal{D}$, $c \in \mathsf{C}(d)$ and $\vec{\tau} \in \mathcal{T}$ such that $\#\vec{\tau} = \mathsf{ar}(d)$. Assume $\mathsf{D}(c) = (\delta, \vec{\alpha}, \vec{\sigma})$. An *instance of c with respect to* $\vec{\tau}$ is defined as follows:

$$\mathsf{Inst}_c\ \vec{\tau} = \vec{\sigma}[\delta := d\,\vec{\tau}][\vec{\alpha} := \vec{\tau}].$$

Typing of terms is defined in the usual way.

Definition 4.9 (Typing). The typing judgment $\Gamma \vdash e : \sigma$ is derivable if it can be inferred by the rules of Figure 7, where $\mathcal{G}_f^x(\emptyset, a)$ is the *guarded-by-destructors* condition defined in Figure 6.

We will now present some properties of $\lambda_{\mathcal{G}}$ that will be used in the interpretation of $\lambda_{\mathcal{G}}$ into $\widehat{\lambda}$ exhibited in the following section.

Lemma 4.10. $\Gamma_1, \Gamma_2, x : \tau, \Gamma_3 \vdash a : \sigma \quad \Rightarrow \quad \Gamma_1, x : \tau, \Gamma_2, \Gamma_3 \vdash a : \sigma$.

Proof. The proof is by induction on the derivation of $\Gamma_1, \Gamma_2, x : \tau, \Gamma_3 \vdash a : \sigma$. □

Lemma 4.11 (Generation lemma for $\mathcal{G}$). If $\mathcal{G}_f^x(U, a)$ has a derivation D, then only one rule can be applied as the last step of D.

Proof. The proof is by case analysis on a. Note that only the conclusions of the rules 5 and 6 can be matched. Furthermore, in order to match the conclusions of such rules, a must be of the form $f\,(z\,\vec{b})$, in which case rule 5 cannot be applied as last rule since its left premise would be underivable. $\square$

Lemma 4.12. If $\mathscr{G}_f^x(U,a)$ and $U \subseteq V$, then $\mathscr{G}_f^x(V,a)$.

Proof. The proof is by induction on the derivation of $\mathscr{G}_f^x(U,a)$. The interesting case is when the last rule applied is rule 7.

Assume $a \equiv \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$ and that the last step is

$$\frac{\mathscr{G}_f^x(U,e) \quad \mathscr{G}_f^x(U,b_i) \quad (1 \leqslant i \leqslant n)}{\mathscr{G}_f^x(U,\mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})}.$$

— If $e \not\equiv z\,\vec{a}$ or if $e \equiv z\,a_1 \ldots a_m$, $z \notin U \cup \{x\}$ and $z \notin V$, then by induction hypothesis $\mathscr{G}_f^x(V,e)$ and $\mathscr{G}_f^x(V,b_i)$ for $1 \leqslant i \leqslant n$, thus $\mathscr{G}_f^x(V,a)$ can be derived using rule 7.

— Now consider $e \equiv z\,a_1 \ldots a_m$, $z \notin U \cup \{x\}$ and $z \in V$. Each b_i must be of the form $\lambda y_1 . \ldots \lambda y_{\mathsf{ar}(c_i)}. e_i$. Let $Q_i \equiv V \cup \{y_j \mid \mathsf{RP}(j, \mathsf{D}(c_i))$ for $1 \leqslant j \leqslant \mathsf{ar}(c_i)\}$. For $1 \leqslant i \leqslant n$, since $V \subseteq Q_i$, using the induction hypothesis $\mathscr{G}_f^x(Q_i,b_i)$, and then, by Lemma 4.11, $\mathscr{G}_f^x(Q_i,e_i)$. Also, from the induction hypothesis we have $\mathscr{G}_f^x(V,a_j)$ for $1 \leqslant j \leqslant m$, and therefore, applying rule 8, $\mathscr{G}_f^x(V,a)$.

The remaining cases can be easily proved using the induction hypothesis. $\square$

Lemma 4.13 (Generation lemma for $\lambda_{\mathscr{G}}$).

1 $\Gamma \vdash x : \sigma \ \Rightarrow\ (x : \sigma) \in \Gamma$.

2 $\Gamma \vdash e\,e' : \sigma \ \Rightarrow\ \exists \tau \in \mathscr{T}.\, \Gamma \vdash e : \tau \to \sigma \ \wedge\ \Gamma \vdash e' : \tau$.

3 $\Gamma \vdash \lambda x.e : \theta \ \Rightarrow\ \theta \equiv \tau \to \sigma \ \wedge\ \Gamma, x : \tau \vdash e : \sigma$.

4 $\Gamma \vdash c : \theta \ \Rightarrow\ \theta \equiv \mathsf{Inst}_c\ \vec{\tau} \to d\,\vec{\tau}$ with $c \in \mathsf{C}(d)$.

5 $\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow e_1 | \ldots | c_n \Rightarrow e_n\} : \theta \ \Rightarrow\ \exists d \in \mathscr{D}\, \exists \vec{\tau} \in \mathscr{T}.\, \Gamma \vdash e : d\,\vec{\tau} \ \wedge\ \Gamma \vdash e_i : \mathsf{Inst}_{c_i}\ \vec{\tau} \to \theta$ for $1 \leqslant i \leqslant n$ with $c_i \in \mathsf{C}(d)$.

6 $\Gamma \vdash \mathsf{letrec}\ f = e : \theta \ \Rightarrow\ \theta \equiv d\,\vec{\tau} \to \sigma \ \wedge\ \Gamma, f : d\,\vec{\tau} \to \sigma \vdash e : d\,\vec{\tau} \to \sigma \ \wedge\ e \equiv \lambda x.a \ \wedge\ \mathscr{G}_f^x(\varnothing, a)$.

4.2. *From $\lambda_{\mathscr{G}}$ to $\widehat{\lambda}$*

In this section we show that $\widehat{\lambda}$ is a more general system than $\lambda_{\mathscr{G}}$. The Examples 4.6 and 4.7 already illustrated that some terms typable in $\widehat{\lambda}$ cannot be typed in $\lambda_{\mathscr{G}}$. In this section we show that: *if* $\Gamma \vdash_{\lambda_{\mathscr{G}}} a : \sigma$ *then* $\Gamma \vdash_{\widehat{\lambda}} a : \sigma$ (the subscript at the turnstyle sign indicating the type system considered). Naturally, the main difficulty in passing from $\lambda_{\mathscr{G}}$ to $\widehat{\lambda}$ is posed by letrec-expressions because the two systems have different kinds of typing rules for these expressions.

Given $\Gamma \vdash_{\lambda_{\mathscr{G}}} \mathsf{letrec}\ f = \lambda x.a : d\,\vec{\tau} \to \sigma$, by the generation lemma for $\lambda_{\mathscr{G}}$, we have

$$\Gamma, f : d\,\vec{\tau} \to \sigma, x : d\,\vec{\tau} \vdash_{\lambda_{\mathscr{G}}} a : \sigma \quad \wedge \quad \mathscr{G}_f^x(\varnothing, a). \tag{6}$$

However, we would want to have

$$\Gamma, f : d^{\imath}\vec{\tau} \to \sigma, x : d^{\widehat{\imath}}\vec{\tau} \vdash_{\widehat{\lambda}} a : \sigma \qquad (\imath \text{ fresh in } \Gamma, \vec{\tau}) \tag{7}$$

in order to use the $\widehat{\lambda}$ rec-rule and so derive $\Gamma \vdash_{\widehat{\lambda}}$ letrec $f = \lambda x.a : d\,\vec{\tau} \to \sigma$. Intuitively, (6) is sufficient to guarantee (7) because, as we have $\mathscr{G}^x_f(\emptyset, a)$, all the possible occurrences of f in a are of the form $f\,(z\,\vec{a})$, with z being a recursive component of x. In (7) we have $x : \widehat{d^{\imath}}\vec{\tau}$, so, if z is a recursive component of x, we should have $z : \vec{\gamma} \to d^{\imath}\vec{\tau}$. Hence $f\,(z\,\vec{a})$ is also typable in $\widehat{\lambda}$.

The remainder of this subsection is devoted to the embedding of $\lambda_{\mathscr{G}}$ into $\widehat{\lambda}$. In this embedding the Main Lemma below plays a central role. There we present the full construction underlying the lemma because it lays open the details of the relation between the systems $\lambda_{\mathscr{G}}$ and $\widehat{\lambda}$.

In the following we assume that each variable x_i is uniquely associated to a stage variable $\jmath_i$. Recall also that, in $\widehat{\lambda}$, the notation $d\,\vec{\tau}$ abbreviates the datatype $d^{\infty}\vec{\tau}$.

Lemma 4.14 (Main Lemma). Let

$$\begin{array}{lll} \Gamma_0 &=& \Gamma \\ \Gamma_i &=& \Gamma_{i-1}, f_i : d_i\,\vec{\tau}_i \to \sigma_i, x_i : d_i\,\vec{\tau}_i \quad \text{for } 1 \leqslant i \leqslant n \\ \\ \widehat{\Gamma_0} &=& \Gamma_0 \\ \widehat{\Gamma_i} &=& \widehat{\Gamma_{i-1}}, f_i : d_i^{\jmath_i}\vec{\tau}_i \to \sigma_i, x_i : \widehat{d_i^{\jmath_i}}\vec{\tau}_i \quad \text{for } 1 \leqslant i \leqslant n \\ && \qquad \text{where } \jmath_i \text{ is a fresh stage variable} \\ && \qquad \text{associated to } x_i \end{array}$$

and, for $1 \leqslant i \leqslant n$, let U_i be a set of variables such that for each $z \in U_i$, $z : \vec{\gamma} \to d_i\,\vec{\tau}_i \in \Gamma$ and so that all the U_i's are disjoint. Then,

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} a : \sigma \;\wedge\; \big(\forall i \in \{1,\ldots,n\}.\ \mathscr{G}^{x_i}_{f_i}(U_i, a)\big) \;\Rightarrow\; [\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} a : \sigma$$

where $U = \bigcup_{1 \leqslant i \leqslant n} U_i$ and $[\widehat{\Gamma_n}]_U$ is obtained from $\widehat{\Gamma_n}$ by replacing each declaration $z : \vec{\gamma} \to d_i\,\vec{\tau}_i$ (with $z \in U_i$) by $z : \vec{\gamma} \to d_i^{\jmath_i}\,\vec{\tau}_i$. Note that in order to make Γ_n a context, in particular, all the f_i's and x_i's must be distinct and cannot be declared in Γ.

Proof. The proof is by induction on the structure of a.

1 Case $a \equiv x$: the hypothesis is

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} x : \sigma \;\wedge\; \forall i \in \{1,\ldots,n\}.\ \mathscr{G}^{x_i}_{f_i}(U_i, x).$$

— If $x \equiv x_i$ for some $i \in \{1,\ldots,n\}$, then $\sigma \equiv d_i\,\vec{\tau}_i$, so, $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} x : \widehat{d_i^{\jmath_i}}\vec{\tau}_i$. As $\widehat{d_i^{\jmath_i}}\vec{\tau}_i \sqsubseteq d_i^{\infty}\vec{\tau}_i$, using the rule (sub),

$$[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} x : d_i\vec{\tau}_i.$$

— If $x \not\equiv x_i$ for every $i \in \{1,\ldots,n\}$, then, since $\forall i \in \{1,\ldots,n\}.\ \mathscr{G}^{x_i}_{f_i}(U_i, x)$, $x \not\equiv f_i$ for every $i \in \{1,\ldots,n\}$. Therefore, using Lemma 4.13, $(x : \sigma) \in \Gamma$. Hence,

(a) If $x \notin U$, then $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} x : \sigma$.

(b) If $x \in U$, then, $\sigma \equiv \vec{\gamma} \to d_i\vec{\tau}_i$ for some $i \in \{1,\ldots,n\}$. So, $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} x : \vec{\gamma} \to d_i^{\jmath_i}\vec{\tau}_i$, and, since $\vec{\gamma} \to d_i^{\jmath_i}\vec{\tau}_i \sqsubseteq \vec{\gamma} \to d_i^{\infty}\vec{\tau}_i$, by (sub)

$$[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} x : \sigma.$$

2 Case $a \equiv e\,e'$: the hypothesis is

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} e\,e' : \sigma \quad \wedge \quad \forall i \in \{1,\ldots,n\}.\ \mathscr{G}^{x_i}_{f_i}(U_i, e\,e').$$

— If $e \equiv f_i$ for some $i \in \{1,\ldots,n\}$, then, by Lemma 4.11, $e' \equiv z\,\vec{b}$, $\mathscr{G}^{x_i}_{f_i}(U_i, e')$ and $z \in U_i$. Moreover:

(a) $\Gamma_n \vdash_{\lambda_{\mathscr{G}}} f_i : d_i\vec{\tau}_i \to \sigma_i$ and $\sigma \equiv \sigma_i$. So, $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} f_i : d_i^{J_i}\vec{\tau}_i \to \sigma$.

(b) $\Gamma_n \vdash_{\lambda_{\mathscr{G}}} z : \vec{\gamma} \to d_i\vec{\tau}_i$. So, $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} z : \vec{\gamma} \to d_i^{J_i}\vec{\tau}_i$ because $z \in U_i$.

(c) $\Gamma_n \vdash_{\lambda_{\mathscr{G}}} \vec{b} : \vec{\gamma}$ (using this notation to abbreviate the list of judgments $\Gamma_n \vdash_{\lambda_{\mathscr{G}}} b_k : \gamma_k$ for each $b_k \in \vec{b}$) and for every $b_k \in \vec{b}$, $\mathscr{G}^{x_i}_{f_i}(U_i, b_k)$ because $z \neq f_i$. For $j \in \{1,\ldots,n\} - \{i\}$, $e \neq f_j$ and, by Lemma 4.11, $\mathscr{G}^{x_j}_{f_j}(U_j, b_k)$ for every $b_k \in \vec{b}$. Therefore, by the induction hypothesis,

$$[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} \vec{b} : \vec{\gamma}.$$

From (a), (b) and (c), using (app), we then get

$$[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} f_i\,(z\,\vec{b}) : \sigma.$$

— If $e \not\equiv f_i$ for every $i \in \{1,\ldots,n\}$, then, using Lemmas 4.11 and 4.13,

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} e : \gamma \to \sigma \quad \wedge \quad \Gamma_n \vdash_{\lambda_{\mathscr{G}}} e' : \gamma$$

and

$$\forall i \in \{1,\ldots,n\}.\ \mathscr{G}^{x_i}_{f_i}(U_i, e) \quad \wedge \quad \mathscr{G}^{x_i}_{f_i}(U_i, e').$$

Hence, by the induction hypothesis,

$$[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} e : \gamma \to \sigma \quad \wedge \quad [\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} e' : \gamma.$$

Using the rule (app), we get $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} e\,e' : \sigma$.

3 Case $a \equiv \lambda y.e$: the hypothesis is

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} \lambda y.e : \sigma \quad \wedge \quad \forall i \in \{1,\ldots,n\}.\ \mathscr{G}^{x_i}_{f_i}(U_i, \lambda y.e).$$

Using Lemmas 4.13 and 4.10, $\sigma \equiv \gamma \to \sigma'$ for some $\gamma, \sigma' \in \mathscr{T}$, and

$$\Gamma, y : \gamma, f_1 : d_1\vec{\tau}_1 \to \sigma_2, x_1 : d_1\vec{\tau}_1, \ldots, f_n : d_n\vec{\tau}_n \to \sigma, x_n : d_n\vec{\tau}_n \vdash_{\lambda_{\mathscr{G}}} e : \sigma'.$$

By Lemma 4.11, $\forall i \in \{1,\ldots,n\}.\ \mathscr{G}^{x_i}_{f_i}(U_i, e)$. Hence, by the induction hypothesis,

$$\left[\Gamma, y : \gamma, f_1 : d_1^{J_1}\vec{\tau}_1 \to \sigma_1, x_1 : d_1^{\widehat{J_1}}\vec{\tau}_1, \ldots, f_n : d_n^{J_n}\vec{\tau}_n \to \sigma_n, x_n : d_n^{\widehat{J_n}}\vec{\tau}_n\right]_U \vdash_{\widehat{\lambda}} e : \sigma'.$$

We know that $y \notin \Gamma$, so $y \notin U$. Therefore, using Lemma 3.6, $[\widehat{\Gamma_n}]_U, y : \gamma \vdash_{\widehat{\lambda}} e : \sigma'$ and the proof of this case is concluded by applying rule (abs).

4 Case $a \equiv c$ and $c \in \mathsf{C}(d)$: we assume $\Gamma_n \vdash_{\lambda_{\mathscr{G}}} c : \mathsf{Inst}_c\ \vec{\tau} \to d\,\vec{\tau}$. Thus in $\widehat{\lambda}$ we can apply (cons) to obtain $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} c : \mathsf{Inst}_c^{\infty}\ \vec{\tau} \to d^{\widehat{\infty}}\vec{\tau}$, and since, $\mathsf{Inst}_c\ \vec{\tau}$ is being used as an abbreviation for $\mathsf{Inst}_c^{\infty}\ \vec{\tau}$ and $d^{\widehat{\infty}}\vec{\tau} \sqsubseteq d^{\infty}\vec{\tau}$, we also have

$$[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} c : \mathsf{Inst}_c\ \vec{\tau} \to d\,\vec{\tau}.$$

5 Case $a \equiv \mathsf{case}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 | \ldots | c_m \Rightarrow b_m\}$: the hypotheses are

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} \mathsf{case}\ e\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} : \sigma \tag{8}$$

$$\forall i \in \{1, \ldots, n\}.\ \mathscr{G}^{x_i}_{f_i}(U_i, \mathsf{case}\ e\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\}) \tag{9}$$

and from (8), applying Lemma 4.13, there exists d, $\vec{\tau}$ such that

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} e : d\,\vec{\tau} \tag{10}$$

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} b_k : \mathsf{Inst}_{c_k}\ \vec{\tau} \to \sigma \tag{11}$$

for each $1 \leqslant k \leqslant m$. Two cases can now occur:

— If $e \not\equiv z\,\vec{a}$ or $e \equiv z\,\vec{a}$ and $z \notin U_i \cup \{x_i\}$ for every $i \in \{1, \ldots, n\}$, then from (9) by Lemma 4.11,

$$\forall i \in \{1, \ldots, n\}. \forall k \in \{1, \ldots, m\}.\ \mathscr{G}^{x_i}_{f_i}(U_i, e) \ \wedge\ \mathscr{G}^{x_i}_{f_i}(U_i, b_k).$$

Thus, applying the induction hypothesis to (10), followed by rule (sub), we have $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} e : d^{\widehat{\infty}}\vec{\tau}$, and applying the induction hypothesis to (11), we get $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} b_k : \mathsf{Inst}^{\infty}_{c_k}\ \vec{\tau} \to \sigma$. Derivations of these judgments can now be put together by means of the rule (case), proving

$$[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} \mathsf{case}\ e\ \mathsf{of}\ \{\vec{c} \Rightarrow \vec{b}\} : \sigma.$$

— Assume now that $e \equiv z\,\vec{a}$ and $z \in U_i \cup \{x_i\}$ for some $i \in \{1, \ldots, n\}$ (recall that such i must be unique since the U_j's are disjoint and contain none of the x_j's, and the x_j's are distinct). Let, for each $1 \leqslant k \leqslant m$,

$$\begin{cases} b_k \equiv \lambda \vec{y}_k . e_k \\ V_{k,i} \equiv U_i \cup \{y_{k,r} \mid \mathsf{RP}(r, \mathsf{D}(c_k)) \text{ for } 1 \leqslant r \leqslant \mathsf{ar}(c_k)\} \\ V_{k,j} \equiv U_j \text{ for } j \in \{1, \ldots, n\} - \{i\} \\ V_k \equiv \bigcup_{1 \leqslant j \leqslant n} V_{k,j} \end{cases}$$

where $y_{k,r}$ denotes the r-th component of vector $\vec{y}_k$. Applying Lemma 4.11 to (9), we can now assume that for each $1 \leqslant j \leqslant n$

$$\forall\, a_s \in \vec{a}\ .\ \mathscr{G}^{x_j}_{f_j}(U_j, a_s) \ \wedge\ \forall\, k \in \{1, \ldots, m\}\ .\ \mathscr{G}^{x_j}_{f_j}(V_{k,j}, e_k). \tag{12}$$

From (11) by Lemmas 4.13 and 4.10, we have

$$\Gamma, \vec{y}_k : \mathsf{Inst}_{c_k}\ \vec{\tau}, \Gamma_n \setminus \Gamma \vdash_{\lambda_{\mathscr{G}}} e_k : \sigma$$

where $\Gamma_n \setminus \Gamma$ is the context Γ_n without the declarations in Γ. Moreover, $y_{k,r} : \vec{\gamma_{y_{k,r}}} \to d_i \vec{\tau}_i \in (\vec{y}_k : \mathsf{Inst}_{c_k}\ \vec{\tau})$ for each $1 \leqslant r \leqslant \mathsf{ar}(c_k)$ such that $\mathsf{RP}(r, \mathsf{D}(c_k))$ and thus, for each $1 \leqslant k \leqslant m$ and $1 \leqslant j \leqslant n$, and for each $z \in V_{k,j}$, we have $z : \vec{\gamma}_z \to d_i\,\vec{\tau}_i \in (\Gamma, \vec{y}_k : \mathsf{Inst}_{c_k}\ \vec{\tau})$. Hence, by the induction hypothesis,

$$[\widehat{\Gamma, \vec{y}_k : \mathsf{Inst}_{c_k}\ \vec{\tau}, \Gamma_n \setminus \Gamma}]_{V_k} \vdash_{\widehat{\lambda}} e_k : \sigma,$$

from which one can show $[\widehat{\Gamma_n}]_U, \vec{y}_k : \mathsf{Inst}^{j_i}_{c_k}\ \vec{\tau} \vdash_{\widehat{\lambda}} e_k : \sigma$ (observe that $V_k = U \cup \{y_{k,r} \mid \mathsf{RP}(r, \mathsf{D}(c_k)) \text{ for } 1 \leqslant r \leqslant \mathsf{ar}(c_k)\}$) and therefore, by the rule (abs), $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} b_k : \mathsf{Inst}^{j_i}_{c_k}\ \vec{\tau} \to \sigma$ holds.

To conclude the proof of this case, it now suffices to show that

$$[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} e : d_i^{\widehat{\jmath_i}}\vec{\tau}_i, \tag{13}$$

and then to use the rule (case). In order to prove (13), we proceed as follows:

(a) Case $e \equiv x_i$, $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} x_i : d_i^{\widehat{\jmath_i}}\vec{\tau}_i$ is derivable.

(b) Case $e \equiv z\,\vec{a}$ with $z \in U_i$, from (10) by Lemma 4.13, $\Gamma_n \vdash_{\lambda_{\mathscr{G}}} z : \vec{\gamma} \to d\,\vec{\tau}$ (thus, $d\,\vec{\tau} \equiv d_i\,\vec{\tau}_i$) and

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} \vec{a} : \vec{\gamma}. \tag{14}$$

Now, since (12) holds, we can apply the induction hypothesis to (14) to obtain $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} \vec{a} : \vec{\gamma}$. It is also true that $z : \vec{\gamma} \to d_i^{\jmath_i}\vec{\tau}_i \in [\widehat{\Gamma_n}]_U$, for $z \in U_i$, and since $\vec{\gamma} \to d_i^{\jmath_i}\vec{\tau}_i \sqsubseteq \vec{\gamma} \to d_i^{\widehat{\jmath_i}}\vec{\tau}_i$, by (sub) and (app), $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} z\,\vec{a} : d_i^{\widehat{\jmath_i}}\vec{\tau}_i$ holds.

6 Case $a \equiv \mathsf{letrec}\ f = \lambda x.a'$: we must have $\sigma \equiv d\,\vec{\tau} \to \sigma'$ for some $d\,\vec{\tau}, \sigma' \in \mathscr{T}$, and the hypothesis is

$$\Gamma_n \vdash_{\lambda_{\mathscr{G}}} \mathsf{letrec}\, f = \lambda x.a' : d\,\vec{\tau} \to \sigma' \quad \wedge \quad \forall i \in \{1,\ldots,n\}.\ \mathscr{G}_{f_i}^{x_i}(U_i, \mathsf{letrec}\, f = \lambda x.a').$$

By Lemma 4.13, we get

$$\Gamma_n, f : d\,\vec{\tau} \to \sigma', x : d\,\vec{\tau} \vdash_{\lambda_{\mathscr{G}}} a' : \sigma'$$

and $\mathscr{G}_f^x(\emptyset, a')$. Again by the hypothesis, by Lemma 4.11, $\forall i \in \{1,\ldots,n\}.\ \mathscr{G}_{f_i}^{x_i}(U_i, a')$, and hence, assuming $U_{n+1} = \emptyset$, $x_{n+1} = x$ and $f_{n+1} = f$, we have

$$\forall i \in \{1,\ldots,n+1\}.\ \mathscr{G}_{f_i}^{x_i}(U_i, a').$$

So, by the induction hypothesis, $[\widehat{\Gamma_{n+1}}]_U \vdash_{\widehat{\lambda}} a' : \sigma'$. Applying (abs) and (rec), we get $[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} \mathsf{letrec}\ f = \lambda x.a' : (d^{\imath}\vec{\tau} \to \sigma')[\imath := \infty]$. Hence, for $\imath$ has no occurrences in either $\vec{\tau}$ or σ',

$$[\widehat{\Gamma_n}]_U \vdash_{\widehat{\lambda}} (\mathsf{letrec}\, f = \lambda x.a') : d\,\vec{\tau} \to \sigma'. \qquad \square$$

We are now ready to prove the main result of this section.

Proposition 4.15.

$$\Gamma \vdash_{\lambda_{\mathscr{G}}} a : \sigma \quad \Rightarrow \quad \Gamma \vdash_{\widehat{\lambda}} a : \sigma.$$

Proof. The proof is by induction on the derivation of $\Gamma \vdash_{\lambda_{\mathscr{G}}} a : \sigma$.

(rec) Assume the last step is

$$\frac{\Gamma, f : d\,\vec{\tau} \to \sigma \vdash_{\lambda_{\mathscr{G}}} e : d\,\vec{\tau} \to \sigma \qquad \mathscr{G}_f^x(\emptyset, a)}{\Gamma \vdash_{\lambda_{\mathscr{G}}} (\mathsf{letrec}\, f = e) : d\,\vec{\tau} \to \sigma} \quad \text{with } e \equiv \lambda x.a$$

By Lemma 4.13, $\Gamma, f : d\,\vec{\tau} \to \sigma, x : d\,\vec{\tau} \vdash_{\lambda_{\mathscr{G}}} a : \sigma$, and since $\mathscr{G}_f^x(\emptyset, a)$, we have the conditions for applying the Main Lemma and conclude $\Gamma, f : d^{\imath}\vec{\tau} \to \sigma, x : d^{\widehat{\imath}}\vec{\tau} \vdash_{\widehat{\lambda}} a : \sigma$. Hence, applying the rules (abs) and (rec), we derive $\Gamma \vdash_{\widehat{\lambda}} (\mathsf{letrec}\ f = e) : (d^{\imath}\vec{\tau} \to \sigma)[\imath := \infty]$, which is the same as

$$\Gamma \vdash_{\widehat{\lambda}} (\mathsf{letrec}\, f = e) : d\,\vec{\tau} \to \sigma$$

for $\imath$ does not occur in σ or $\vec{\tau}$.

All the remaining cases can be easily proved using the induction hypothesis. □

5. Extension to coinductive types

Coinductive types are a mechanism for the introduction of infinite objects into type theory, and are useful in the modelling of perpetual computations, for example, the operation of process systems. The system $\widehat{\lambda}$ is readily extensible to support coinductive types. We shall here outline the syntax of an appropriate extension of $\widehat{\lambda}$ and give some programming examples.

First, the definition of the set $\mathscr{E}$ of terms is extended with corecursive definitions:

$$e ::= \ldots \mid ({}^{\mathrm{co}}\mathsf{letrec}\ f = e).$$

In addition to β-, ι- and μ-reduction, we define ν-reduction as the compatible closure of the rule

$$\begin{aligned}&\mathsf{case}\ ({}^{\mathrm{co}}\mathsf{letrec}\ f = e)\ \vec{a}\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}\\ &\quad\to_\nu\ \mathsf{case}\ e[f := ({}^{\mathrm{co}}\mathsf{letrec}\ f = e)]\ \vec{a}\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\}.\end{aligned}$$

The form of the ν-reduction rule may look unexpected, but it is dual to μ-reduction: while the μ-reduction rule allows unfolding of a recursive definition provided that the argument value is produced by a constructor, the ν-reduction rule allows it, if the result value is consumed by a case-expression.

Second, the definition of the set $\mathscr{T}$ of type expressions is extended with codatatype approximation expressions:

$$\sigma, \tau ::= \ldots \mid {}^{\mathrm{co}}d^{s}\ \vec{\tau}$$

${}^{\mathrm{co}}d^{\infty}\ \vec{\tau}$ will also be written as ${}^{\mathrm{co}}d\ \vec{\tau}$.

The subtyping rules are supplemented by the following (codata) rule, which is dual to the (data) rule:

$$(\text{codata})\ \frac{s \preccurlyeq r \quad \tau_i \sqsubseteq \tau_i' \quad (1 \leqslant i \leqslant \mathsf{ar}(d))}{{}^{\mathrm{co}}d^{r}\vec{\tau} \sqsubseteq {}^{\mathrm{co}}d^{s}\vec{\tau}}\ .$$

The typing rules are supplemented by the rules (cons′), (case′) and (corec). The first two of these are essentially the same as the rules (cons), (case), but they are used for the construction and destruction of values of coinductive, not inductive types. Below ${}^{\mathrm{co}}\mathsf{Inst}_c^s\ \vec{\tau}$ stands for $\vec{\sigma}[\delta := {}^{\mathrm{co}}d^{s}\vec{\tau}][\vec{\alpha} := \vec{\tau}]$.

$$(\text{cons}')\quad \frac{}{\Gamma \vdash c : {}^{\mathrm{co}}\mathsf{Inst}_c^s\ \vec{\tau} \to {}^{\mathrm{co}}d^{\widehat{s}}\vec{\tau}} \quad \text{if } c \in \mathsf{C}(d)$$

$$(\text{case}')\quad \frac{\Gamma \vdash e' : {}^{\mathrm{co}}d^{\widehat{s}}\vec{\tau} \qquad \Gamma \vdash e_i : {}^{\mathrm{co}}\mathsf{Inst}_{c_i}^s\ \vec{\tau} \to \theta \quad (1 \leqslant i \leqslant n)}{\Gamma \vdash \mathsf{case}\ e'\ \mathsf{of}\ \{c_1 \Rightarrow e_1 \mid \ldots \mid c_n \Rightarrow e_n\} : \theta} \quad \text{if } \mathsf{C}(d) = \{c_1\ \ldots, c_n\}$$

$$(\text{corec})\quad \frac{\Gamma, f : \vec{\sigma} \to {}^{\mathrm{co}}d^{\imath}\vec{\tau} \vdash e : (\vec{\sigma} \to {}^{\mathrm{co}}d^{\imath}\vec{\tau})[\imath := \widehat{\imath}] \qquad \imath\ \mathsf{pos}\ \vec{\sigma}}{\Gamma \vdash ({}^{\mathrm{co}}\mathsf{letrec}\ f = e) : (\vec{\sigma} \to {}^{\mathrm{co}}d^{\imath}\vec{\tau})[\imath := s]} \quad \text{if } \imath \text{ not in } \Gamma, \vec{\tau}.$$

Below are some programming examples that illustrate the use of co-recursive functions.

Example 5.1.

— *The colist of all natural numbers starting from a given one (in the ascending order).*

$$({}^{\mathrm{co}}\mathsf{letrec}\ \mathit{from} = \lambda n.\ \mathsf{cons}\ n\ (\mathit{from}\ (\mathsf{s}\ n)))\ :\ \mathrm{Nat}\to{}^{\mathrm{co}}\mathrm{List}\ \mathrm{Nat}$$

— *The infinite colist consisting of zeros.*

$$({}^{\mathrm{co}}\mathsf{letrec}\ \mathit{zeros} = \mathsf{cons}\ \mathsf{o}\ \mathit{zeros})\ :\ {}^{\mathrm{co}}\mathrm{List}\ \mathrm{Nat}$$

— *Concatenation of two colists.* This program admits a type containing the information that the concatenation of two colists is in the same approximation of the colist type as the two individual colists are.

$$\begin{array}{ll}
\mathrm{append} \equiv & ({}^{\mathrm{co}}\mathsf{letrec}\ \mathit{append}_{:{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau\to{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau\to{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau} = \\
& \quad \lambda x_{:{}^{\mathrm{co}}\mathrm{List}^{\widehat{\imath}}\,\tau}.\ \lambda y_{:{}^{\mathrm{co}}\mathrm{List}^{\widehat{\imath}}\,\tau}.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \overbrace{y}^{:{}^{\mathrm{co}}\mathrm{List}^{\widehat{\imath}}\,\tau} \\
& \qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda a_{:\tau}.\ \lambda x'_{:{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau}.\ \underbrace{\mathsf{cons}\ a\ \underbrace{(\mathit{append}\ x'\ y)}_{:{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau}}_{:{}^{\mathrm{co}}\mathrm{List}^{\widehat{\imath}}\tau} \\
& \qquad\qquad \} \\
)\ : & {}^{\mathrm{co}}\mathrm{List}^{\mathsf{s}}\ \tau\to{}^{\mathrm{co}}\mathrm{List}^{\mathsf{s}}\ \tau\to{}^{\mathrm{co}}\mathrm{List}^{\mathsf{s}}\ \tau
\end{array}$$

— *Exchange of every first and second element in a given colist.*

$$\begin{array}{ll}
({}^{\mathrm{co}}\mathsf{letrec} & \mathit{exch}_{:{}^{\mathrm{co}}\mathrm{List}\tau\to{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau} = \lambda l_{:{}^{\mathrm{co}}\mathrm{List}\tau}. \\
& \mathsf{case}\ l\ \mathsf{of}\ \{ \\
& \quad \mathsf{nil} \Rightarrow \mathsf{nil} \\
& \mid \mathsf{cons} \Rightarrow \lambda a_{:\tau}.\ \lambda l'_{:{}^{\mathrm{co}}\mathrm{List}\tau}.\mathsf{case}\ l'\ \mathsf{of}\ \{ \\
& \qquad\qquad \mathsf{nil} \Rightarrow \mathsf{cons}\ a\ \mathsf{nil} \\
& \qquad\qquad \mid \mathsf{cons} \Rightarrow \lambda a'_{:\tau}.\ \lambda l''_{:{}^{\mathrm{co}}\mathrm{List}\tau}.\ \underbrace{\mathsf{cons}\ a'\ (\mathsf{cons}\ a\ \underbrace{(\mathit{exch}\ l'')}_{:{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau})}_{:{}^{\mathrm{co}}\mathrm{List}^{\widehat{\widehat{\imath}}}\tau\,\sqsubseteq\,{}^{\mathrm{co}}\mathrm{List}^{\widehat{\imath}}\tau}\ \} \\
& \quad \} \\
)\ : & {}^{\mathrm{co}}\mathrm{List}\ \tau\to{}^{\mathrm{co}}\mathrm{List}^{s}\ \tau
\end{array}$$

Although the exchange function does not alter the length of a colist, the type we have given to the program above is the best possible in our setting.

— *Given a colist, compute its infinite repetition.* The typability of this program is a consequence of the typing we have given for append.

$$\begin{array}{ll}
({}^{\mathrm{co}}\mathsf{letrec} & \mathit{rep}_{:{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau\to{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau} = \lambda l_{:{}^{\mathrm{co}}\mathrm{List}^{\widehat{\imath}}\tau}. \\
& \mathsf{case}\ l\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil} \\
& \qquad \mid \mathsf{cons} \Rightarrow \lambda a_{:\tau}.\ \lambda l'_{:{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau}.\ \underbrace{\mathsf{cons}\ a\ (\mathrm{append}\ l'\ \underbrace{(\mathit{rep}\ l)}_{:{}^{\mathrm{co}}\mathrm{List}^{\imath}\tau})}_{:{}^{\mathrm{co}}\mathrm{List}^{\widehat{\imath}}\tau} \\
& \qquad \} \\
)\ : & {}^{\mathrm{co}}\mathrm{List}^{\mathsf{s}}\tau\to{}^{\mathrm{co}}\mathrm{List}^{\mathsf{s}}\tau
\end{array}$$

The extended $\widehat{\lambda}$ enjoys the same properties of subject reduction and strong normalisability of typable terms as the original $\widehat{\lambda}$. The proof for $\widehat{\lambda}$ extends readily; we omit it for space reasons. We believe that $\lambda_{\mathcal{G}}$ extended with guarded by constructors corecursion is embeddable in the extended $\widehat{\lambda}$ as the original $\lambda_{\mathcal{G}}$ is embeddable in $\widehat{\lambda}$.

6. Related work

For the sake of clarity, we will divide the existing systems into five categories:

(1) Those based on traditional-style terminating recursors.
(2) Those based on a fixpoint operator controlled by a syntactic guard predicate.
(3) Those exploiting pattern matching.
(4) Those based on a fixpoint operator controlled by an unusual typing ensuring that the recursion actually terminates.
(5) Those relying on other type-based techniques for ensuring termination.

Comparison with Martin-Löf (1971) and other work on traditional-style terminating recursors

Most formalisations of inductive types in type theory only support recursive definitions indirectly *via* eliminators behaving as iterators or primitive recursors (Martin-Löf 1971; Leivant 1983; Pierce *et al.* 1989; Pfenning and Paulin-Mohring 1990; Coquand and Paulin 1990; Paulin-Mohring 1993; Dybjer 1994; Geuvers 1992; Altenkirch 1999; Matthes 1999; Spławski and Urzyczyn 1999). Such systems are well understood metatheoretically and enjoy good properties, but are hard to use in practical programming: this requires the programmer to translate all recursive definitions he would like to make into explicit definitions involving primitive recursion.

It is possible to devise similar eliminators capturing more sophisticated schemes of terminating recursion such as course-of-value iteration or course-of-value primitive recursion (Uustalu 1998; McBride 1999), but the resulting systems are even clumsier to use in practice.

Comparison with Coquand (1994) and other work relying on a fixpoint operator controlled by an external guard predicate

Coquand (1994) introduces a simple guard predicate to ensure termination of fixpoint expressions in a calculus of infinite objects. Building on Coquand's work, Giménez (1995) defines a more liberal guarded-by-constructors predicate for terminating corecursion as well as a guarded-by-destructors predicate for terminating recursion. Giménez shows that primitive recursor expressions can be rendered as fixpoint expressions guarded by one destructor. In the reverse direction, a fixpoint expression guarded by destructors can be coded as an expression involving primitive recursors, but the translation is not uniform. The predicates defined by Giménez form the basis of the mechanism for (co)inductive types in Coq. More recently, Blanqui *et al.* (2002), building on Jouannaud and Okada (1997), proposed another definition of the guard predicate for inductive types that allows for yet more expressions to be typed. Following a similar line of research, Abel and Altenkirch (2002) proposed a basic framework for studying and comparing the different termination conditions that have been proposed so far, focusing their attention on what conditions should be fulfilled for a checking to be sound. An application of such framework to a particular condition can be found in Abel (2000).

One possible objection to this line of work is that the system becomes more unpredictable to the user as the complexity of the guard predicate builds up. Besides, the guard predicate remains purely syntactic, which is not appropriate for a number of applications, including separate compilation or interactive proof construction.

Comparison with Coquand (1992) and other work on pattern-matching

Coquand (1992) pioneered the use of pattern-matching in type theory. While pattern-matching yields leaner definitions, its proof-theoretical status in the context of dependent types remains unclear. Unlike guarded-by-destructors recursion, general pattern-matching is not a conservative improvement over primitive recursors: Hofmann and Streicher (1994) proved the derivability of uniqueness of equality proofs in a type theory with pattern-matching, while equality proofs cannot be shown to be unique in the usual Calculus of Inductive Constructions. To our knowledge, there is no complete account of the meta-theoretical properties of pattern-matching in dependent type theory. McBride (1999) showed that, under the uniqueness of equality proofs as an extra axiom, pattern matching is admissible. Giménez has remarked (Giménez 1996) that in a typing system with dependent pattern matching the computation rule used in this article for corecursive definitions only satisfies a weak form of the subject reduction property. Ongoing work on checking the termination of recursive function definitions in functional languages (see, for example, Telford and Turner (1997), Abel and Altenkirch (2002), Giesl *et al.* (1998), and Manoury and Simonot (1994)) is relevant for this direction of type-theoretic developments. Of particular interest for the future type-theoretic formalisations might be recent work (Lee *et al.* 2001) on the size-change principle for program termination.

As for implementations, restricted forms of pattern-matching have been implemented in Coq (Cornes 1997) and Lego (Elbers 1998). Both implementations take advantage of translations to recursors. Pattern-matching has also been consistently supported in Alf and its subsequent versions, although no mechanism for termination checking was ever implemented. In order to simplify the proof engine, Agda, which is the latest incarnation of Alf, only supports a limited form of pattern-matching in which variables are only allowed to occur once in the type of a constructor. This restriction rules out, for example, inductive definitions such as equality.

Comparison with Giménez (1998) and other work on guarded types

This line of work is really about non-traditional-style terminating recursors that look like fixpoint operators, but where the computation is guaranteed to terminate by an unusual (stronger) typing system. Such a system involves introducing some kind of annotations on recursive types, a notion of sub-typing enabling the transformation of such annotations, and a typing rule for the term $\mathsf{letrec}\ f = e$ where the type of f and the type of e are marked differently. In this sense, some of the systems mentioned in this section are not far from the so called *abstract interpretation* techniques (Cousot and Cousot 1996), even though they are formulated from a type-theoretical point of view. The exact relation of such typing systems with respect to abstract interpretation techniques has not been studied in detail yet, and could be a subject for further research.

Mendler (1987) was, as far as we are aware, the first author to propose a formalisation of inductive and coinductive types in a simply typed lambda calculus where primitive recursion and primitive corecursion were formulated in a fixpoint-like style. In Mendler's system, type annotations on the fixpoint rule correspond to type variables. Mendler (1991) considered a system supporting only iteration and coiteration. Work that comments on these two papers include Leivant (1990), Geuvers (1992), Uustalu and Vene (1997), Uustalu (1998), Matthes (1998; 2002), and Spławski and Urzyczyn (1999). Of these, Leivant (1990) and Geuvers (1992) were the first papers to contrast and compare traditional-style and Mendler-style terminating recursors. Uustalu and Vene (1997; 2002) and Uustalu (1998) showed that Mendler's approach is readily generalisable for course-of-value (co)recursion (in other words, full structural (co)recursion).

Giménez (1996) introduced an extension of the Calculus of Constructions with inductive and coinductive types, called CC^{∞}. The fixpoint rules in CC^{∞} make use of three kinds of marks, corresponding, using the notation of this article, to the stages ∞, $\imath$ and $\hat{\imath}$. This means that in CC^{∞} the hat operator cannot be applied to another stage, but only to stage variables. In Giménez (1996), marks also have a second component, which specifies whether the recursive type is inductive or coinductive. There is no stage polymorphism, and hence the function *div* of Example 2.21 cannot be typed.

One of the main disadvantages of Giménez (1996) is that it tried to tackle too many problems at once, rendering the typing calculus less clear. Among the extra features introduced in CC^{∞} that are not considered in this article we may cite the following:

— Inductive lists are considered a subtype of coinductive ones, so that a function defined on the type ^{co}List can also be used on an element of type List.

— Annotations are placed on typing judgments, writing $x :^{s}$ List instead of $x :$ List^{s}. One of the original motivations for this notation was to enable the description of abstract recursion schema, where the type of the decreasing argument of the function is abstracted away using a term of the form $\lambda A : Set \cdot \mathsf{letrec}\ f = \lambda x :^{s} A \cdot e$. Also, the choice of having two different universal quantifiers renders unnecessary the introduction of two types of lists (one for inductive lists and the other for coinductive ones) with the same constructors. On the other hand, it is less clear how an ordinal based semantics like the one proposed in this paper could be used to make sense of a term of the form $\lambda A : Set \cdot \lambda x :^{s} A \cdot e$. This is why, even though annotated quantifications were kept, the calculus in Giménez (1996) forces A in a term of the form $\lambda x :^{s} A \cdot e$ to have a recursive type at its rightmost position.

— C^{∞} is built on top of the Calculus of Constructions, so it uses Church's style for variable binding, where the type of the abstracted variable is explicitly mentioned. Thus, types – and hence marks – may appear in the terms. As a consequence, the reduction rule for fixpoints has to replace all mark variables by the ∞ mark in order to avoid having residual unbound mark identifiers in the definiens. Note that this problem does not arise in $\widehat{\lambda}$, where variable binding is '*à la* Curry'.

Giménez (1998) introduced CCR, a different extension of the Calculus of Constructions with inductive and coinductive types, based on (not fully general) sub- and supertyping and bounded universal quantification over types. In CCR, marks are represented as type variables (as in Mendler's work), the hat operator is a type constructor, and stages are just types. Since stage variables are type variables, stage replacement just corresponds to the ordinary substitution operation of the calculus. The calculus in Giménez (1998) was the first calculus to introduce stage polymorphism, enabling us to type definitions like the function *div* of Example 2.21 and the stream *rep* of Example 5.1. The calculus of the present paper is very much inspired by Giménez (1998), but replaces sub- and supertypes with approximating types, and bounded type quantification with stage quantification – the change allows the structure of stages to be uniform over all datatypes and simplifies the introduction of recursive definitions on mutually dependent inductive types. No studies have yet been made of the meta-theory of CCR or its connection to implemented extensions of a calculus of (co)inductive constructions like the system Coq. The detailed study of the main meta-theoretical properties of $\widehat{\lambda}$ presented in this paper can be seen as a basic stage for developing the meta-theory of an extension of the Calculus of Construction where the termination of functions is ensured by typing constraints.

Amadio and Coupet-Grimal (1998) defines a simply typed λ-calculus '*à la* Curry' featuring guarded coinductive types. Starting from Coquand's guardedness condition, they propose a semantics for such an extension of lambda calculus based on partial equivalent relations and ordinal iteration to interpret coinductive types. From that semantics, they derive a typing rule for corecursive definitions using a mark system with three kinds of marks, which correspond in our notation to ∞, $\imath$ and $\hat{\imath}$. The semantic interpretation used to study the meta-theory of $\widehat{\lambda}$ in this article is actually an extension of the one introduced in Amadio and Coupet-Grimal (1998) for coinductive types. Also, the need for the constraint $\imath$ pos σ in the typing rule for recursive definitions has already been noticed by Amadio and Coupet. Their calculus introduces an extra rule enabling the treatment of nested fixpoint definitions of the form ($\mathsf{letrec}\ f = (\mathsf{letrec}\ g = e)$) by reusing the mark introduced in the definition of f as the mark for the variable g. However, the calculus described in Amadio and Coupet-Grimal (1998) lacks full stage polymorphism, so definitions like the function *rep* in Example 5.1 cannot be typed in their system. Their calculus does not consider inductive types. On the positive side, their calculus is shown to have decidable type inference in Bac (1998).

Barras (1999) formalises in Coq a variant of Giménez' calculus CC^{∞}, with the aim of proving the decidability of its typing judgment and extracting a type-checker from the proof. In Barras' calculus, inductive types are annotated with lists of marks, each corresponding to the stages ∞, $\imath$ and $\hat{\imath}$ of our system. The use of lists of marks enables the typing of nested recursive function definitions like the ones considered in Amadio and Coupet-Grimal (1998), but for inductive types. He does not consider coinductive types nor stage polymorphism. As the underlying lambda calculus is '*à la* Church', Barras introduces a distinguished primitive type $\mathcal{M}$ for marks, and marks are just variables of that type. Mark variables are bound in fixpoint terms, so mark erasure in fixpoint reductions just corresponds to ordinary variable substitution. The complete meta-theory of Barras' system has not been studied yet, but his system is the only mark based one for which a type-checking algorithm has been developed.

Other type-based approaches to termination analysis

Xi (2001) proposes a system of restricted dependent types, built upon DML (Xi and Pfenning 1999), to ensure program termination. In essence, his system is closely related to ours since it uses stage information to ensure termination. However, Xi's system differs from ours in its expressiveness and complexity: while we focus on the weakest calculus that uses type-based termination and extends other calculi based on a simple syntactic guard predicate, Xi presents a very rich system with stage arithmetic, and a notion of metric that is very useful for handling functions in several arguments. Of course, expressiveness is achieved to the detriment of simplicity and Xi's system is much more complex than ours. Grobauer (2001) uses DML to find cost recurrences for first-order recursive definitions: a cost recurrence is an upper bound to the running time of the program with respect to the size of its input, and hence a witness that the recursive definition is terminating. In his work, Grobauer exploits complex features of DML, including stage arithmetic, so his techniques do not seem directly applicable to $\widehat{\lambda}$. Closely related is the recent work on sized types (Hughes *et al.* 1996; Pareto 2000; Chin and Khoo 2001).

7. Conclusion

We have introduced $\widehat{\lambda}$ as a novel type system for terminating recursive functions. The salient features of $\widehat{\lambda}$ are its type-based approach to ensure termination through the notion of stage, and its support for stage polymorphism. The calculus is powerful enough to encode many recursive definitions rejected by existing type systems, scales up easily to mutually inductive types and supports separate compilation. In comparison to $\lambda_{\mathcal{G}}$, it has a much clearer syntax and admits a clean semantics; the strong normalisation can be proved by means of a standard method. In practice, this means that $\widehat{\lambda}$ is less difficult to implement (implementing the guard condition of $\lambda_{\mathcal{G}}$ is error-prone) and the code written in it is more easily maintainable. This makes $\widehat{\lambda}$ a good candidate base system for type theory based proof-assistants such as Coq.

In order to validate this claim, the following steps need to be taken:

— Scale up $\widehat{\lambda}$ to dependent types and explicit polymorphism as in Barras (1999) and Giménez (1998).
— Develop type checking and type inference algorithms for $\widehat{\lambda}$. For the purpose of proof assistants, it may be of interest to study a calculus where type annotations are given and stage annotations are inferred.
— Provide mechanisms to support mutually inductive datatypes, mutually recursive definitions and recursive functions in several parameters. For the latter, some form of stage arithmetic might be needed.

For a different line of work, it may be of interest to give a precise characterisation of the functions from ℕ to ℕ that are representable in $\widehat{\lambda}$.

Acknowledgments

We are grateful to our anonymous referees for the very constructive feedback we received and to Roberto Amadio for making Bac (1998) available to us.

The work by Gilles Barthe, Maria João Frade, Luís Pinto, and Tarmo Uustalu was partially supported by the Portuguese Foundation for Science and Technology under grant number PRAXIS XXI/C/EEI/14172/98, by the INRIA-ICCTI collaboration and by the FP5 IST project TYPES. Luís Pinto was partially supported by Centro de Matemática da Universidade do Minho. Tarmo Uustalu also received support from the Estonian Science Foundation under grant number 4155.

References

Abel, A. (2000) Specification and verification of a formal system for structurally recursive functions. In: Coquand, T., Dybjer, P., Nordström, B. and Smith, J. (eds.) Proceedings of TYPES'99. *Springer-Verlag Lecture Notes in Computer Science* **1956** 1–20.

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

Altenkirch, T. (1999) Logical relations and inductive/coinductive types. In: Gottlob, G., Grandjean, E. and Seyr, K. (eds.) Proceedings of CSL'98. *Springer-Verlag Lecture Notes in Computer Science* **1584** 343–354.

Amadio, R. M. and Coupet-Grimal, S. (1998) Analysis of a guard condition in type theory (extended abstract). In: Nivat, M. (ed.) Proceedings of FoSSaCS'98. *Springer-Verlag Lecture Notes in Computer Science* **1378** 48–62.

Bac, A. (1998) Un algorithme d'inférence de types pour les types coinductifs. Memoire de DEA, École Normale Supérieure de Lyon.

Barras, B. (1999) *Auto-validation d'un système de preuves avec familles inductives*, Ph.D. thesis, Université Paris 7.

Blanqui, F., Jouannaud, J.-P. and Okada, M. (2002) Inductive data type systems. *Theoretical Computer Science* **272** (1-2) 41–68.

Chin, W.-N. and Khoo, S.-C. (2001) Calculating sized types. *Higher-Order and Symbolic Computation* **14** (2-3) 261–300.

Coquand, T. (1992) Pattern matching with dependent types. In: B. Nordström, K. Pettersson, and G. Plotkin (eds.) *Informal Proceedings of TYPES'92*, Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ. 71–84. (Available at ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.ps.Z.)

Coquand, T. (1994) Infinite objects in type theory. In: Barendregt, H. and Nipkow, T. (eds.) Proceedings of TYPES'93. *Springer-Verlag Lecture Notes in Computer Science* **806** 62–78.

Coquand, T. and Paulin, C. (1990) Inductively defined types (preliminary version). In: Martin-Löf, P. and Mints, G. (eds.) Proceedings of COLOG'88. *Springer-Verlag Lecture Notes in Computer Science* **417** 50–66.

Cornes, C. (1997) *Conception d'un langage de haut niveau de representation de preuves: Récurrence par filtrage de motifs; Unification en présence de types inductifs primitifs; Synthèse de lemmes d'inversion*, Ph.D. thesis, Université de Paris 7.

Cousot, P. and Cousot, R. (1996) Abstract interpretation. *ACM Computing Surveys* **28** (2) 324–328.

Dybjer, P. (1994) Inductive families. *Formal Aspects of Computing* **6** (4) 440–465.

Elbers, H. (1998) *Connecting formal and informal mathematics*, Ph.D. thesis, Technische Universiteit Eindhoven.

Geuvers, H. (1992) Inductive and coinductive types with iteration and recursion. In: Nordström, B., Pettersson, K. and Plotkin, G. (eds.) *Informal Proceedings of TYPES'92*, Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ. 193–217. (Available at ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.ps.Z.)

Giesl, J., Walther, C. and Brauburger, J. (1998) Termination analysis for functional programs. In: Bibel, W. and Schmitt, P. (eds.) Automated Deduction: A Basis for Applications, Vol. 3: Applications. *Applied Logic Series*, Kluwer Academic Publishers **10** 135–164.

Giménez, E. (1995) Codifying guarded definitions with recursion schemes. In: Dybjer, P. and Nordström, B. (eds.) Proceedings of TYPES'94. *Springer-Verlag Lecture Notes in Computer Science* **996** 39–59.

Giménez, E. (1996) *A calculus of infinite constructions and its application to the verification of reactive systems*, Ph.D. thesis, Ecole Normale Supérieure de Lyon.

Giménez, E. (1998) Structural recursive definitions in Type Theory. In: Larsen, K. G., Skyum, S. and Winskel, G. (eds.) Proceedings of ICALP'98. *Springer-Verlag Lecture Notes in Computer Science* **1443** 397–408.

Grobauer, B. (2001) Cost recurrences for DML programs. In: Proceedings of ICFP'01. *SIGPLAN Notices* **36** (10) 253–264.

Hofmann, M. and Streicher, T. (1994) The groupoid model refutes uniqueness of identity proofs. In: *Proceedings of LICS'94*, IEEE CS Press 208–212.

Hughes, J., Pareto, L. and Sabry, A. (1996) Proving the correctness of reactive systems using sized types. In: *Proceedings of POPL'96*, ACM Press 410–423.

Jouannaud, J. P. and Okada, M. (1997) Abstract data type systems. *Theoretical Computer Science* **173** (2) 349–391.

Lee C.-S., Jones, N. D. and Ben-Amram, A. M. (2001) The size-change principle for program termination. In: Proceedings of POPL'01. *SIGPLAN Notices* **36** (3) 81–92.

Leivant, D. (1983) Reasoning about functional programs and complexity classes associated with type disciplines. In: *Proceedings of FOCS'83*, IEEE Computer Society Press 460–469.

Leivant, D. (1990) Contracting proofs to programs. In: Odifreddi, P. (ed.) Logic and Computer Science. *APIC Studies in Data Processing*, Academic Press **31** 279–327.

Luo, Z. (1994) Computation and Reasoning: A Type Theory for Computer Science. *Int. Series of Monographs in Computer Science*, Clarendon Press **11**.

Manoury, P. and Simonot, M. (1994) Automatizing termination proofs of recursively defined functions. *Theoretical Computer Science* **135** (2) 319–343.

Martin-Löf, P. (1971) Hauptsatz for the intuitionistic theory of iterated inductive definitions. In: Fenstad, J. E. (ed.) Proceedings of 2nd Scandinavian Logic Symp. *Studies in Logic and the Foundations of Mathematics*, North-Holland **63** 179–216.

Matthes, R. (1998) *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*, Ph.D. thesis, Fachbereich Mathematik, Ludwig-Maximilians-Universität München.

Matthes, R. (1999) Monotone fixed-point types and strong normalization. In: Gottlob, G., Grandjean, E. and Seyr, K. (eds.) Proceedings of CSL'98. *Springer-Verlag Lecture Notes in Computer Science* **1584** 298–312.

Matthes, R. (2002) Tarski's fixed-point theorem and lambda calculi with monotone inductive types. *Synthese* **133** (1) 107–129.

McBride, C. (1999) *Dependently Typed Functional Programs and Their Proofs*, Ph.D. thesis, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh.

Mendler, N. P. (1987) Recursive types and type constraints in second-order lambda-calculus. In: *Proceedings of LICS'87*, IEEE Computer Society Press 30–36.

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

Nordström, B., Petersson, K. and Smith, J. (1990) Programming in Martin-Löf's Type Theory: An Introduction. *Int. Series of Monographs on Computer Science*, Clarendon Press **7**.

Pareto, L. (2000) *Types for crash prevention*, Ph.D. thesis, Chalmers Univ. of Techn., Göteborg.

Paulin-Mohring, C. (1993) Inductive definitions in the system Coq: Rules and properties. In: Bezem, M. and Groote, J. F. (eds.) Proceedings of TLCA'93. *Springer-Verlag Lecture Notes in Computer Science* **664** 328–345.

Pfenning, F. and Paulin-Mohring, C. (1990) Inductively defined types in the calculus of constructions. In: Main, M., Melton, A., Mislove, M. and Schmidt, D. (eds.) Proceedings of MFPS'89. *Springer-Verlag Lecture Notes in Computer Science* **442** 209–228.

Pierce, B., Dietzen, S. and Michaylov, S. (1989) Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, School of Computer Science, Carnegie-Mellon Univ.

Spławski, Z. and Urzyczyn, P. (1999) Type fixpoints: Iteration vs. recursion. In: Proceedings of ICFP'99. *SIGPLAN Notices* **34** (9) 102–113.

Telford, A. and Turner, D. (1997) Ensuring streams flow. In: Johnson, M. (ed.) Proceedings of AMAST'97. *Springer-Verlag Lecture Notes in Computer Science* **1349** 509–523.

Uustalu, T. (1998) *Natural Deduction for Intuitionistic Least and Greatest Fixedpoint Logics, with an Application to Program Construction*, Ph.D. thesis (Dissertation TRITA-IT AVH 98:03), Dept. of Teleinformatics, Royal Inst. of Technology, Stockholm.

Uustalu, T. and Vene, V. (1997) A cube of proof systems for the intuitionistic predicate μ, ν-logic. In: Haveraaen, M. and Owe, O. (eds.) Proceedings of NWPT'96. Research Report 248, Dept. of Informatics, University of Oslo 237–246.

Uustalu, T. and Vene, V. (2002) Least and greatest fixedpoints in intuitionistic natural deduction. *Theoretical Computer Science* **272** (1-2) 315–339.

Xi, H. and Pfenning, F. (1998) Eliminating array bound checking through dependent types. In: Proceedings of PLDI'98. *SIGPLAN Notices* **33** (5) 249–257.

Xi, H. and Pfenning, F. (1999) Dependent types in practical programming. In: *Proceedings of POPL'99*, ACM Press 214–227.

Xi, H. (2001) Dependent types for program termination verification. In: *Proceedings of LICS'01*, IEEE CS Press 231–242.