Constructor subtyping
(extended version)

DRAFT June 1999

Gilles Barthe[1 2] and Maria João Frade[1]

[1] Departamento de Informática, Universidade do Minho, Braga, Portugal
[2] Institutionen för Datavetenskap, Chalmers Tekniska Högskola, Göteborg, Sweden
`{gilles,mjf}@di.uminho.pt`

Abstract. *Constructor subtyping* is a form of subtyping in which an inductive type σ is viewed as a subtype of another inductive type τ if τ has more constructors than σ. As suggested in [5, 12], its (potential) uses include proof assistants and functional programming languages.
In this report, we introduce and study the properties of a simply typed λ-calculus with record types and datatypes, and which supports record subtyping and constructor subtyping. We show that the calculus is confluent and strongly normalizing. We also show that the type-checking is decidable.

1 Introduction

Type systems [3, 8] lie at the core of modern functional programming languages, such as Haskell [26] or ML [24], and proof assistants, such as Coq [4] or PVS [30]. In order to improve the usability of these languages, it is important to devise flexible (and safe) type systems, in which programs and proofs may be written easily. A basic mechanism to enhance the flexibility of type systems is to endorse the set of types with a *subtyping* relation $\leq$ and to enforce a *subsumption* rule

$$\frac{a : A \qquad A \leq B}{a : B}$$

This basic mechanism of subtyping is powerful enough to capture a variety of concepts in computer science, see e.g. [9], and its use is spreading both in functional programming languages, see e.g. [23, 28, 29], and in proof assistants, see e.g. [7, 22, 30].

Constructor subtyping is a basic form of subtyping, suggested in [12] and developed in [5], in which an inductive type σ is viewed as a subtype of another inductive type τ if τ has more constructors than σ. As such, constructor subtyping captures in a type-theoretic context the ubiquitous use of subtyping as inclusion between inductively defined sets. In its simplest instance, constructor subtyping enforces subtyping from odd or even numbers to naturals, as illustrated in the following example, which introduces in a ML-like syntax the mutually recursive datatypes `Odd` and `Even`, and the `Nat` datatype:

Example 1.

```
datatype Odd  = s of Even         datatype Nat = 0
and      Even = 0                              | s of Nat
              | s of Odd ;                     | s of Odd
                                               | s of Even ;
```

Here `Even` and `Odd` are subtypes of `Nat` (i.e. `Even` $\leq$ `Nat` and `Odd` $\leq$ `Nat`), since every constructor of `Even` and `Odd` is also a constructor of `Nat`.

In [5] it was introduced and studied constructor subtyping for one first-order mutually recursive parametric datatype, and showed the calculus to be confluent and strongly normalizing. In the present paper, we improve on this work in several directions:

1. we extend constructor subtyping to the class of strictly positive, mutually recursive and parametric datatypes. In addition, the present calculus supports incremental definitions;

2. following recent trends in the design of proof assistants (and a well-established trend in the design of functional programming languages), we replace the elimination constructors of [5] by case-expressions. This leads to a simpler system, which is easier to use;

The main technical contribution of this report is to show that the calculus enjoys several fundamental meta-theoretical properties including confluence, subject reduction and strong normalization. These results lay the foundations for constructor subtyping and open the possibility of using constructor subtyping in programming languages and proof assistants.

This report is an extended version of [6]. In [6] we also show that the calculus admits a well-behaved theory of canonical inhabitants, provided one adopts expansive extensionality rules, including η-expansion, surjective pairing, and a suitable expansion rule for datatypes.

Acknowledgements We are grateful to T. Altenkirch, P. Dybjer and L. Pinto for useful discussions on constructor subtyping. The first author is partially supported by a TMR fellowship. The second author is partially supported by the LOGCOMP project.

2 An informal account of constructor subtyping

Constructor subtyping formalizes the view that an inductively defined set σ is a subtype of an inductively defined set τ if τ has more constructors than σ. As may be seen from the example of even, odd and natural numbers, the relative generality of constructor subtyping relies on the possibility for constructors to be overloaded and, to a lesser extent, on the possibility for datatypes to be defined in terms of previously introduced datatypes. The following example, which introduces the parametric datatypes `List` of lists and `NeList` of non-empty lists, provides further evidence.

Example 2.

```
datatype 'a List = nil
                 | cons of ('a * 'a List) ;

datatype 'a NeList = cons of ('a * 'a List) ;
```

Here `'a NeList` $\leq$ `'a List` since the only constructor of `'a NeList`, `cons : ('a * 'a List)` $\rightarrow$`'a NeList` is matched by the constructor of `'a List`, `cons : ('a * 'a List)` $\rightarrow$`'a List`.

The above examples reveal a possible pattern of constructor subtyping: for two parametric datatypes d and d' with the same arity, we set $d \leq d'$ if every declaration (`c` in case of a constant, `c of B` otherwise) of d is matched in d'.[1] Another pattern, used in [5], is to take subtyping as a primitive. Here we allow for the subtyping relation to be specified directly in the definition of the datatype. As shown below, such a pattern yields simpler definitions, with less declarations.

Example 3.

```
datatype Odd  = s of Even        datatype Nat  =  s of Nat
and      Even = 0                with     Odd ≤ Nat,
              | s of Odd ;                Even ≤ Nat ;
```

The original datatype may be recovered by adding a declaration of the form $c : \boldsymbol{\sigma} \rightarrow d'$ whenever $c : \boldsymbol{\sigma} \rightarrow d$ and $d \leq d'$. The same technique can be used to define `'a List` and `'a NeList`:

Example 4.

```
datatype 'a List = nil
and      'a NeList = cons of ('a * 'a List)
with     'a NeList ≤ 'a List ;
```

[1] For the sake of simplicity, we gloss over renamings and assume the parameters of d and d' to be identical.

For the clarity of the exposition, we shall adopt the second pattern in examples, whereas we consider the first pattern in the formal definition of $\lambda_{\rightarrow,[],\mathsf{data}}$.

Thus far, the subtyping relation is confined to datatypes. It may be extended to types in the usual (structural) way. In this paper, we force datatypes to be monotonic in their parameters. Hence, we can derive

$$\begin{array}{rcl} \mathtt{Odd\ List} & \leq & \mathtt{Nat\ List} \\ [\mathtt{l1}:\mathtt{Even},\mathtt{l2}:\mathtt{Nat\ List},\mathtt{l3}:\mathtt{Odd}] & \leq & [\mathtt{l1}:\mathtt{Nat},\mathtt{l2}:\mathtt{Nat\ List}] \\ \mathtt{Nat}\rightarrow\mathtt{Even\ NeList} & \leq & \mathtt{Odd}\rightarrow\mathtt{Nat\ NeList} \end{array}$$

from the fact that $\mathtt{Odd} \leq \mathtt{Nat}$, $\mathtt{Even} \leq \mathtt{Nat}$ and `'a NeList` $\leq$ `'a List`. The formal definition of the subtyping relation is presented in the next section.

In order to introduce *strict overloading*, which is a central concept in this paper, let us anticipate on the next section by considering the evaluation rule for case-expressions. Two observations can be made: first, our informal definition of datatype allows for arbitrary overloading of constructors. Second, it is not possible to define a type-independent evaluation rule for case-expressions for arbitrary datatypes. For example, consider the following datatype, where `Sum` is a datatype identifier of arity 2:

```
datatype ('a,'b) Sum = inj of 'a
                     | inj of 'b ;
```

Note that the datatype is obtained from the usual definition of sum types by overloading the constructors inj_1 and inj_2. Now, a case-expression for this datatype should be of the form

```
case a of (inj x) => b1 | (inj x) => b2
```

with evaluation rules

```
case (inj a) of (inj x) => b1 | (inj x) => b2   →   b1{x:=a}
case (inj a) of (inj x) => b1 | (inj x) => b2   →   b2{x:=a}
```

As `b1` and `b2` are arbitrary, the calculus is obviously not confluent. Thus one needs to impose some restrictions on overloading. One drastic solution to avoid non-confluence is to require constructors to be declared at most once in a given datatype, but this solution is too restrictive. A better solution is to require constructors to be declared "essentially" at most once in a given datatype. Here "essentially" consists in allowing a constructor `c` to be multiply defined in a datatype d, but by requiring that for every declaration `c of rho`, we have `rho` $\leq$ `rhom` where `c of rhom` is the first declaration of `c` in d. In other words, the only purpose of repeated declarations is to enforce the desired subtyping constraints but (once subtyping is defined) only the first declaration needs to be used for typing expressions. This notion, which we call strict overloading, is mild enough to be satisfied by most datatypes that occur in the literature, see [5] for a longer discussion on this issue.

We conclude this section with further examples of datatypes.

Example 5. Firstly, we define a datatype of ordinals (or better said of ordinal notations). Note that the datatype is a higher-order one, because of the constructor `lim` which takes a function as input.

```
datatype Ord = s of Ord | lim of (Nat -> Ord)
with     Nat ≤ Ord ;
```

Example 6. Second, we define a datatype of binary integers. These datatypes are part of the Coq library, but Coq does not take advantage of constructor subtyping.

```
datatype positive = xH | xI of positive | xO of positive ;
datatype natural  = ZERO
with     positive ≤ natural ;
datatype integer  = NEG of positive
with     natural ≤ integer ;
```

Example 7. Thirdly, and as pointed out in [5, 12], constructor subtyping provides a suitable framework in which to formalize programming languages, including the object calculi of Abadi and Cardelli [1] and a variety of other languages taken from [27]. Yet another example of language that can be expressed with constructor semantics is mini-ML [21], as shown below. Here we consider four datatypes identifiers: `E` of *expressions*, `I` for *identifiers*, `P` of *patterns* and `N` for the *nullpattern*, all with arity 0.

```
datatype I = ident ;
datatype N = nullpat ;
datatype P = pairpat of (P * P)
with     I ≤ P, N ≤ P ;
datatype E = num | false | true | lamb of (P * E)
           | if of (E * E * E) | mlpair of (E * E)
           | apply of (E * E) | let of (P * E * E)
           | letrec of (P * E * E)
with     I ≤ E, N ≤ E ;
```

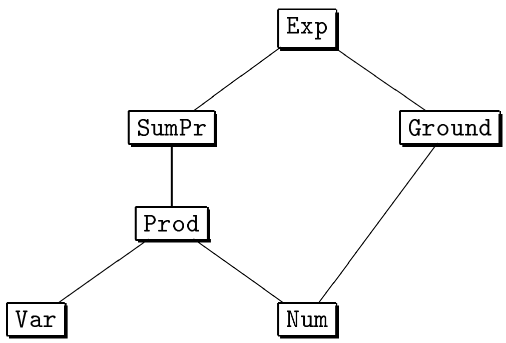
Example 8. In the following we declare datatypes for arithemic expressions. We want to distinguish the expressions that are sums of products, i.e., that do not contain additions as sub-terms of multiplications. Moreover, we want to distinguish ground expressions, i.e., expressions which contain no variables. This example is adapted from [14],

```
datatype Ground  = num of Nat
                 | plus of (Ground * Ground)
                 | times of (Ground * Ground) ;

datatype Prod    = num of Nat
                 | var of String
                 | times of (Prod * Prod) ;

datatype SumPr   = plus of (SumPr * SumPr)
with     Prod ≤ SumPr ;

datatype Exp     = plus of (Exp * Exp)
                 | times of (Exp * Exp)
with     Ground ≤ Exp ,
         SumPr ≤ Exp ;
```

We can have a more refined hierarchy of types if we take a type for numbers and another for variables. In this case we must declare:

```
datatype Num    = num of Nat ;

datatype Var    = var of String ;

datatype Ground = plus of (Ground * Ground)
                | times of (Ground * Ground)
with     Num ≤ Ground ;

datatype Prod   = times of (Prod * Prod)
with     Num ≤ Prod ,
         Var ≤ Prod ;

datatype SumPr  = plus of (SumPr * SumPr)
with     Prod ≤ SumPr ;

datatype Exp    = plus of (Exp * Exp)
                | times of (Exp * Exp)
with     Ground ≤ Exp ,
         SumPr ≤ Exp ;
```

In this case we have the following hierarchy:



Example 9. Lastly, we conclude with a definition of CTL* formulae, see [15]. In this example, we consider two datatypes identifiers `SF` of *state formulae* and `PF` of *path formulae*, both with arity 1.

```
datatype 'a SF = i of ('a * 'a SF) | conj of ('a SF * 'a SF)
               | not of 'a SF | forsomefuture of 'a PF
               | forallfuture of 'a PF
and      'a PF = conj of ('a PF * 'a PF) | not of 'a PF
               | nexttime of 'a PF | until of 'a PF
with     'a SF ≤ 'a PF ;
```

CTL* and related temporal logics provide suitable frameworks in which to verify the correctness of programs and protocols, and hence are interesting calculi to formalize in proof assistants.

You can look the appendix for further examples.

3 A core calculus $\lambda_{\to,[],\mathsf{data}}$

In this section, we introduce the core calculus $\lambda_{\to,[],\mathsf{data}}$. The first subsection is devoted to types, datatypes and subtyping; the second subsection is devoted to expressions, reduction and typing.

3.1 Types and subtyping

Below we assume given some pairwise disjoint sets $\mathcal{L}$ of *labels*, $\mathcal{D}$ of *datatype identifiers*, $\mathcal{C}$ of *constructor identifiers* and $\mathcal{X}$ of *type variables*. Moreover, we let $l, l', l_i, \ldots$ range over $\mathcal{L}$, $d, d', \ldots$ range over $\mathcal{D}$, $c, c', c_i, \ldots$ range over $\mathcal{C}$ and $\alpha, \alpha', \alpha_i, \beta, \ldots$ range over $\mathcal{X}$. In addition, we assume that every datatype identifier d has a fixed *arity* $\mathsf{ar}(d)$ and that $\alpha_1, \alpha_2, \ldots$ is a fixed enumeration of $\mathcal{X}$. Sometimes we will write $\boldsymbol{\tau}$ instead of $\tau_1, \ldots, \tau_n$.

Definition 1 (Types). *The set $\mathcal{T}$ of* types *is given by the abstract syntax:*

$$\sigma, \tau := d[\tau_1, \ldots, \tau_{\mathsf{ar}(d)}] \mid \alpha \mid \sigma \to \tau \mid [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$$

where in the last clause it is assumed that the l_is are pairwise distinct. By convention, we identify record types that only differ in the order of their declarations, such as $[l : \sigma, l' : \tau]$ and $[l' : \tau, l : \sigma]$.

We now turn to the definition of datatype. Informally, a *datatype* is a list of *constructor declarations*, i.e. of pairs (c, τ) where c is a constructor identifier and τ is a *constructor type*, i.e. a type of the form

$$\rho_1 \to \ldots \to \rho_n \to d[\alpha_1, \ldots, \alpha_{\mathsf{ar}(d)}]$$

with $d \in \mathcal{D}$. However not all datatypes are valid. In order for a datatype to be valid, it must satisfy several properties.

1. Constructors must be *strictly positive*, so that datatypes have a direct set-theoretic interpretation. For example, $c_1 : \mathsf{nat} \to d$ and $c_2 : (\mathsf{nat} \to d) \to d$ are strictly positive w.r.t. d, whereas $c_3 : (d \to d) \to d$ is not.
2. Parameters must appear positively in the domains of constructor types, so that datatypes are *monotonic* in their parameters. For example, the parameter α appears positively in the domain of $\alpha \to d[\alpha]$, while it appears negatively in the domain of $(\alpha \to \mathsf{nat}) \to d[\alpha]$.
3. Datatypes that mutually depend on each other must have the same number of parameters, for the sake of simplicity.
4. Constructors must be strictly overloaded, so that case-expressions can be evaluated unambiguously.

In addition, we allow datatypes to depend on previously defined datatypes. This leads us naturally to the notion of *datatype context*. Informally, a datatype context is a finite list of datatypes. Below we let σ, τ range over types, $\aleph$ range over datatype contexts, c range over datatype constructors and d, d' range over datatype identifiers.

Definition 2 (Legal pre-type). *σ is a* legal pre-type *in $\aleph$ with variables in $\{\alpha_1, \ldots, \alpha_k\}$ (or $\emptyset$ if $k = 0$) and a set of datatype identifiers $\mathcal{I}$, written $\aleph \vdash_k \sigma$ $\mathsf{pretype}(\mathcal{F})$, is defined by the rules of Figure 1.*

Note that in ($\mathsf{predata\text{-}pre}$) we do not allow mutually dependent types to appear nested, because we force each σ_i to be a type and not a pre-type.

Definition 3 (Legal type). *σ is a* legal type *in $\aleph$ with variables in $\{\alpha_1, \ldots, \alpha_k\}$ (or $\emptyset$ if $k = 0$), written $\aleph \vdash_k \sigma$ type, is defined by the rules of Figure 2.*

Comparing the rules of the Figures 1 and 2 one can conclude that types and pre-types are very closed notions. Actually, the notion of pre-type is more general than the notion of type. That is, every type is a pre-type, because the form of the rules that define type can be found in the set of rules that define pre-type. There is only an extra rule ($\mathsf{predata\text{-}pre}$) stating that if d is in the set of datatype identifiers that are being defined and all the parameters $\boldsymbol{\sigma}$ are types, then $d[\boldsymbol{\sigma}]$ is a pre-type.

When we are analyzing the construction of a new datatype, to know if it is valid, we have to assure that the type of the constructors satisfy the properties described above. At this point, the datatype it is being constructed is not yet a type (because we don't know if it is valid), but it may appear in the domains of the constructors. However it can be seen as a "candidate" to be a type. The notion of pre-type is introduced to capture this situation.

$$(\mathsf{pre} \to) \qquad \frac{\aleph \vdash_k \sigma \ \mathsf{pretype}(\mathcal{I}) \qquad \aleph \vdash_k \tau \ \mathsf{pretype}(\mathcal{I})}{\aleph \vdash_k \sigma \to \tau \ \mathsf{pretype}(\mathcal{I})}$$

$$(\mathsf{pre}[]) \qquad \frac{\aleph \vdash_k \sigma_i \ \mathsf{pretype}(\mathcal{I}) \quad (1 \leq i \leq n)}{\aleph \vdash_k [l_1 : \sigma_1, \ldots, l_n : \sigma_n] \ \mathsf{pretype}(\mathcal{I})}$$

$$(\mathsf{predata}) \qquad \frac{d \in \aleph \quad \aleph \vdash_k \sigma_i \ \mathsf{pretype}(\mathcal{I}) \quad (1 \leq i \leq \mathsf{ar}(d))}{\aleph \vdash_k d[\boldsymbol{\sigma}] \ \mathsf{pretype}(\mathcal{I})}$$

$$(\mathsf{pre} - \alpha) \qquad \frac{\aleph \ \mathsf{legal}}{\aleph \vdash_k \alpha_i \ \mathsf{pretype}(\mathcal{I})}\,, \quad \text{if } 1 \leq i \leq k$$

$$(\mathsf{predata} - \mathsf{pre}) \qquad \frac{d \in \mathcal{I} \quad \aleph \vdash_k \sigma_i \ \mathsf{type} \quad (1 \leq i \leq \mathsf{ar}(d))}{\aleph \vdash_k d[\boldsymbol{\sigma}] \ \mathsf{pretype}(\mathcal{I})}$$

Fig. 1. Pre-type formation rules

$$(\to) \qquad \frac{\aleph \vdash_k \sigma \ \mathsf{type} \qquad \aleph \vdash_k \tau \ \mathsf{type}}{\aleph \vdash_k \sigma \to \tau \ \mathsf{type}}$$

$$([]) \qquad \frac{\aleph \vdash_k \sigma_i \ \mathsf{type} \quad (1 \leq i \leq n)}{\aleph \vdash_k [l_1 : \sigma_1, \ldots, l_n : \sigma_n] \ \mathsf{type}}$$

$$(\mathsf{data}) \qquad \frac{d \in \aleph \quad \aleph \vdash_k \sigma_i \ \mathsf{type} \quad (1 \leq i \leq \mathsf{ar}(d))}{\aleph \vdash_k d[\boldsymbol{\sigma}] \ \mathsf{type}}$$

$$(\alpha) \qquad \frac{\aleph \ \mathsf{legal}}{\aleph \vdash_k \alpha_i \ \mathsf{type}}\,, \quad \text{if } 1 \leq i \leq k$$

Fig. 2. Type formation rules

Definition 4 (Subtype). *σ is a* subtype *of τ in $\aleph$, written $\aleph \vdash \sigma \leq \tau$, is defined by the rules of Figure 3, where $\aleph \vdash d \leq d'$ if*

- $\mathsf{ar}(d) = \mathsf{ar}(d') = m$*;*
- *every declaration $c : \tau_1 \to \ldots \to \tau_n \to d[\alpha_1, \ldots, \alpha_m]$ in $\aleph$ is matched by another declaration $c : \tau_1 \to \ldots \to \tau_n \to d'[\alpha_1, \ldots, \alpha_m]$ in $\aleph$.*

$$(\leq_{\mathrm{refl}}) \quad \frac{\aleph \vdash_k \sigma \ \mathsf{type}}{\aleph \vdash \sigma \leq \sigma}$$

$$(\leq_{\mathrm{trans}}) \quad \frac{\aleph \vdash \sigma \leq \tau \qquad \aleph \vdash \tau \leq \rho}{\aleph \vdash \sigma \leq \rho}$$

$$(\leq_{\to}) \quad \frac{\aleph \vdash \sigma' \leq \sigma \qquad \aleph \vdash \tau \leq \tau'}{\aleph \vdash \sigma \to \tau \leq \sigma' \to \tau'}$$

$$(\leq_{[]}) \quad \frac{\aleph \vdash \sigma_i \leq \tau_i \quad (1 \leq i \leq n) \quad \aleph \vdash_k \sigma_j \ \mathsf{type} \quad (n+1 \leq j \leq m)}{\aleph \vdash [l_1 : \sigma_1, \ldots, l_{n+m} : \sigma_{n+m}] \leq [l_1 : \tau_1, \ldots, l_n : \tau_n]}$$

$$(\leq_{\mathsf{data}}) \quad \frac{\aleph \vdash d \leq d' \quad \aleph \vdash \sigma_i \leq \tau_i \quad (1 \leq i \leq \mathsf{ar}(d))}{\aleph \vdash d[\boldsymbol{\sigma}] \leq d'[\boldsymbol{\tau}]}$$

Fig. 3. SUBTYPING RULES

To better understand the subtyping relation determined by the subtyping system of Figure 3 let us look carefully to each of its rules. As expected, rules ($\leq_{\mathsf{refl}}$) and ($\leq_{\mathsf{trans}}$) indicate that the subtyping relation is reflexive and transitive.

Rule ($\leq_{\to}$) says that $\sigma \to \tau$ is a subtype of $\sigma' \to \tau'$ if σ' is a subtype of σ, and τ a subtype of τ'. Note that the subtyping is inverted (*contra-variant*) on the domains and *covariant* on the codomains. A function f of type $\sigma \to \tau$ accepts elements of type σ; so it also accepts elements of any subtype σ' of σ. The same function f returns elements if type τ; so, it returns elements that belong to any supertype τ' of τ. Therefore f has also type $\sigma' \to \tau'$.

Rule ($\leq_{[]}$) works componentwise saying that a record type ρ is a subtype of a record type ρ' if each field of ρ has a type that is a subtype of the type of ρ' corresponding field, but operate also lengthwise: a longer record type is a subtype of a shorter record type. This indicate that additional fields can be forgotten by subtyping.

Rule ($\leq_{\mathsf{data}}$) works also componentwise and lengthwise: a datatype $d[\boldsymbol{\sigma}]$ is a subtype of a datatype $d'[\boldsymbol{\tau}]$ if $\sigma_i \leq \tau_i$ for $1 \leq i \leq \mathsf{ar}(d)$ and $d \leq d'$ (i.e., d' has more constructors than d). Thus additional cases can be introduced by subtyping.

Example 10. Consider the following example (adapted from [8])

```
datatype WorkingAge = student | adult ;

datatype Age        = child | student | adult | senior ;

type     Worker     = [ name : String, age : WorkingAge, profession:String ] ;

type     Person     = [ name : String, age : Age ] ;
```

We have: `WorkingAge` $\leq$ `Age` and `Worker` $\leq$ `Person`.

Definition 5 (d-Constructor type). *τ is a d-*constructor type *in $\aleph$ with a set of previously defined datatype identifiers $\mathcal{I}$, written $\aleph \vdash_{\mathcal{I}} \tau \ \mathsf{coty}(d)$, is defined by the rules of Figure 6, where:*

– α appears positively *in* τ, *written* α pos τ, *is defined by the rules of Figure 4;*
– ρ *is* strictly positive *w.r.t.* d, *written* ρ spos d, *is defined by the rules of Figure 5, where* d nocc τ *denotes that* d *does not occur in* τ*;*
– $d \in \aleph$ *if there exists a declaration* $(c : \tau) \in \aleph$ *in which* d *occurs.*

$$\text{(pos0)}\qquad \alpha\ \mathsf{pos}\ \alpha$$

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

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

$$\text{(pos3)}\quad \frac{\alpha\ \mathsf{pos}\ \sigma_i \quad (1 \leq i \leq n)}{\alpha\ \mathsf{pos}\ [l_1 : \sigma_1, \ldots, l_n : \sigma_n]} \qquad\qquad \text{(neg3)}\quad \frac{\alpha\ \mathsf{neg}\ \sigma_i \quad (1 \leq i \leq n)}{\alpha\ \mathsf{neg}\ [l_1 : \sigma_1, \ldots, l_n : \sigma_n]}$$

$$\text{(pos4)}\quad \frac{\alpha\ \mathsf{pos}\ \sigma_i \quad (1 \leq i \leq n)}{\alpha\ \mathsf{pos}\ d[\sigma_1, \ldots, \sigma_n]} \qquad\qquad \text{(neg4)}\quad \frac{\alpha\ \mathsf{neg}\ \sigma_i \quad (1 \leq i \leq n)}{\alpha\ \mathsf{neg}\ d[\sigma_1, \ldots, \sigma_n]}$$

Fig. 4. Positive-Negative rules

$$\text{(spos1)}\quad \frac{d\ \mathsf{nocc}\ \tau}{\tau\ \mathsf{spos}\ d}$$

$$\text{(spos2)}\quad \frac{d\ \mathsf{nocc}\ \rho_i \quad (1 \leq i \leq n)}{\rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}]\ \mathsf{spos}\ d}$$

Fig. 5. Strictly positive rules

$$\text{(coty)}\quad \frac{\aleph \vdash_k \rho_i\ \mathsf{pretype}(\mathcal{I}) \qquad \rho_i\ \mathsf{spos}\ d \qquad \alpha_j\ \mathsf{pos}\ \rho_i \quad (1 \leq i \leq n\ ,\ 1 \leq j \leq k)}{\aleph \vdash_{\mathcal{I}} \rho_1 \to \ldots \to \rho_n \to d[\alpha_1, \ldots, \alpha_k]\ \mathsf{coty}(d)}\ ,\ \text{with } d \notin \aleph$$

Fig. 6. Constructor type rule

The rule (coty) reflect the ideas expressed above at points 1 and 2. The rules defining positive, negative and strictly positive are the usual ones.

Definition 6. *Let* D *be a sequence of constructor declarations.*

1. $\mathsf{di}(D)$ *denotes the set of datatype identifiers of* D*. It can be defined inductively as follows:*
   (a) $\mathsf{di}(.) = \emptyset$
   (b) $\mathsf{di}(c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}]) = \{d\}$
   (c) $\mathsf{di}(D', c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}]) = \mathsf{di}(D') \cup \{d\}$

2. $\mathsf{ci}(D)$ *denotes the set of datatype identifiers of* D*. It can be defined inductively as follows:*

(a) $\mathsf{ci}(.) = \emptyset$
(b) $\mathsf{ci}(c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}]) = \{c\}$
(c) $\mathsf{ci}(D', c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}]) = \mathsf{ci}(D') \cup \{c\}$

Definition 7 (Main d-declaration). *We say $c : \tau$ is a* main d-declaration*, written* $\mathsf{main}_d(c : \tau)$*, if it is the first declaration of c in a datatype declaration.* $\mathsf{main}_d(c : \tau)$ *can be defined by the rules of Figure 7.*

$$(main1)\quad \frac{\aleph; D, c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}]\ \mathsf{ok}(\mathcal{I}) \qquad c \notin \mathsf{ci}(D)}{\mathsf{main}_d(c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}])}$$

$$(main2)\quad \frac{\aleph; D, c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}]\ \mathsf{ok}(\mathcal{I}) \qquad d \notin \mathsf{di}(D)}{\mathsf{main}_d(c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}])}$$

Fig. 7. Main d-declaration rules

Definition 8 (Legal datatype context). $\aleph$ *is a* legal datatype context*, written* $\aleph$ legal*, is defined by the rules of Figure 8, where* $\aleph$ $\mathsf{compatible}(D)$ *if*

1. *for every* $(c : \tau') \in D$,

$$\aleph \vdash_{\mathsf{di}(D)} \tau'\ \mathsf{coty}(d) \ \wedge\ \mathsf{main}_d(c : \tau) \quad \Rightarrow \quad \aleph; D; \vdash\ \tau \leq \tau'$$

2. *for every* $(c : \tau), (c' : \tau') \in D$,

$$\mathsf{main}_d(c : \tau) \ \wedge\ \mathsf{main}_{d'}(c' : \tau') \quad \Rightarrow \quad \mathsf{ar}(d) = \mathsf{ar}(d')$$

$$\text{(empty)}\qquad .;\ \mathsf{legal}$$

$$\text{(close)}\qquad \frac{\aleph; D\ \mathsf{ok}(\mathsf{di}(D))}{\aleph; D;\ \mathsf{legal}}\,, \qquad \aleph\ \mathsf{compatible}(D)$$

$$\text{(add-cons)}\qquad \frac{\aleph; D\ \mathsf{ok}(\mathcal{I}) \qquad \aleph \vdash_{\mathcal{I}} \tau\ \mathsf{coty}(d)}{\aleph; D, c : \tau\ \mathsf{ok}(\mathcal{I})}$$

$$\text{(add-data)}\qquad \frac{\aleph\ \mathsf{legal}}{\aleph\ \mathsf{ok}(\mathcal{I})}$$

Fig. 8. Datatype rules

As you may notice, the rules of Figure 8 introduce a new kind of judgment $\aleph; D\ \mathsf{ok}(\mathcal{I})$ which states that extending the datatype context $\aleph$ we are constructing a new datatype D in a legal way.

The side-condition in the rule (close) synthesize the idea expressed in the points 3 and 4. More precisely, the first condition of the definition of $\aleph$ $\mathsf{compatible}(D)$ assure that constructors are strictly overloaded. The second condition impose that mutually dependent datatypes have the same arity.

Observe that Definitions 2-8 are mutually dependent. Note that Definitions 5 and 8 above are enforced by the side-condition in (close) whereas Definitions 3 and 4 above are enforced by the rule (coty). Also note that in the side-condition for (close), τ' and τ are compared w.r.t. $\aleph; D;$ and not $\aleph$.

Some examples of legal datatype contexts The definitions given in this subsection seem very complex. In order to make clear the concepts involved in the formal definition of datatype contexts, let us take some examples of section 2 and prove that they form legal datatype contexts.

Example 11. Remember the definition of `Odd`, `Even` and `Nat`. Let

$$\begin{aligned}
\aleph_1 &= .; s : \mathrm{Even} \to \mathrm{Odd}, 0 : \mathrm{Even}, s : \mathrm{Odd} \to \mathrm{Even};\\
\mathcal{I}_1 &= \{\mathrm{Even}, \mathrm{Odd}\}\\
\aleph_2 &= \aleph_1\ 0 : \mathrm{Nat}, s : \mathrm{Nat} \to \mathrm{Nat}, s : \mathrm{Odd} \to \mathrm{Nat}, s : \mathrm{Even} \to \mathrm{Nat};\\
\mathcal{I}_2 &= \{\mathrm{Nat}\}
\end{aligned}$$

Let us prove that $\aleph_1$ is a legal datatype context.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{.;\ \mathsf{legal}}{.;\ \mathsf{ok}(\mathcal{I}_1)} \quad \dfrac{.; \vdash_0 \mathrm{Even}\ \mathsf{pretype}(\mathcal{I}_1) \quad \dfrac{\mathrm{Odd}\ \mathsf{nocc}\ \mathrm{Even}}{\mathrm{Even}\ \mathsf{spos}\ \mathrm{Odd}}}{.; \vdash_{\mathcal{I}_1} \mathrm{Even} \to \mathrm{Odd}\ \mathsf{coty}(\mathrm{Odd})}}{.; s : \mathrm{Even} \to \mathrm{Odd}\ \mathsf{ok}(\mathcal{I}_1)}}{.; s : \mathrm{Even} \to \mathrm{Odd}, 0 : \mathrm{Even}\ \mathsf{ok}(\mathcal{I}_1)} \quad \dfrac{.; \vdash_0 \mathrm{Odd}\ \mathsf{pretype}(\mathcal{I}_1) \quad \dfrac{\mathrm{Even}\ \mathsf{nocc}\ \mathrm{Odd}}{\mathrm{Odd}\ \mathsf{spos}\ \mathrm{Even}}}{.; \vdash_{\mathcal{I}_1} \mathrm{Odd} \to \mathrm{Even}\ \mathsf{coty}(\mathrm{Even})}}{\dfrac{.; s : \mathrm{Even} \to \mathrm{Odd}, 0 : \mathrm{Even}, s : \mathrm{Odd} \to \mathrm{Even}\ \mathsf{ok}(\mathcal{I}_1)}{\aleph_1\ \mathsf{legal}}\ (\mathrm{close})}$$

It is easy to see that the side-condition of (close), $.;\ \mathsf{compatible}(s : \mathrm{Even} \to \mathrm{Odd}, 0 : \mathrm{Even}, s : \mathrm{Odd} \to \mathrm{Even})$ is verified.

$$\dfrac{\dfrac{\dfrac{\aleph_1\ \mathsf{legal}}{\aleph_1\ \mathsf{ok}(\mathcal{I}_2)} \quad \aleph_1 \vdash_{\mathcal{I}_2} \mathrm{Nat}\ \mathsf{coty}(\mathrm{Nat})}{\aleph_1\ 0 : \mathrm{Nat}\ \mathsf{ok}(\mathcal{I}_2)} \quad \dfrac{\dfrac{\mathrm{Nat} \in \mathcal{I}_2}{\aleph_1 \vdash_0 \mathrm{Nat}\ \mathsf{pretype}(\mathcal{I}_2)} \quad \mathrm{Nat}\ \mathsf{spos}\ \mathrm{Nat}}{\aleph_1 \vdash_{\mathcal{I}_2} \mathrm{Nat} \to \mathrm{Nat}\ \mathsf{coty}(\mathrm{Nat})}}{\aleph_1\ 0 : \mathrm{Nat}, s : \mathrm{Nat} \to \mathrm{Nat}\ \mathsf{ok}(\mathcal{I}_2)} \tag{1}$$

The following prove that $\aleph_2$, defined incrementally over $\aleph_1$, is also a legal datatype context.

$$\dfrac{\dfrac{\dfrac{\dfrac{(1)}{\aleph_1\ 0 : \mathrm{Nat}, s : \mathrm{Nat} \to \mathrm{Nat}\ \mathsf{ok}(\mathcal{I}_2)} \quad \dfrac{\dfrac{\mathrm{Odd} \in \aleph_1}{\aleph_1 \vdash_0 \mathrm{Odd}\ \mathsf{pretype}(\mathcal{I}_2)} \quad \dfrac{\mathrm{Nat}\ \mathsf{nocc}\ \mathrm{Odd}}{\mathrm{Odd}\ \mathsf{spos}\ \mathrm{Nat}}}{\aleph_1 \vdash_{\mathcal{I}_2} \mathrm{Odd} \to \mathrm{Nat}\ \mathsf{coty}(\mathrm{Nat})}}{\aleph_1\ 0 : \mathrm{Nat}, s : \mathrm{Nat} \to \mathrm{Nat}, s : \mathrm{Odd} \to \mathrm{Nat}\ \mathsf{ok}(\mathcal{I}_2)} \quad \dfrac{\dfrac{\mathrm{Even} \in \aleph_1}{\aleph_1 \vdash_0 \mathrm{Even}\ \mathsf{pretype}(\mathcal{I}_2)} \quad \dfrac{\mathrm{Nat}\ \mathsf{nocc}\ \mathrm{Even}}{\mathrm{Even}\ \mathsf{spos}\ \mathrm{Nat}}}{\aleph_1 \vdash_{\mathcal{I}_2} \mathrm{Even} \to \mathrm{Nat}\ \mathsf{coty}(\mathrm{Nat})}}{\aleph_1\ 0 : \mathrm{Nat}, s : \mathrm{Nat} \to \mathrm{Nat}, s : \mathrm{Odd} \to \mathrm{Nat}, s : \mathrm{Even} \to \mathrm{Nat}\ \mathsf{ok}(\mathcal{I}_2)}}{\aleph_2\ \mathsf{legal}}\ (\mathrm{close})$$

Let us see as the side-condition of (close), $\aleph_1\ \mathsf{compatible}(0 : \mathrm{Nat}, s : \mathrm{Nat} \to \mathrm{Nat}, s : \mathrm{Odd} \to \mathrm{Nat}, s : \mathrm{Even} \to \mathrm{Nat})$, is verified:

1. We have

$$\begin{array}{lcl}
\mathsf{main}_{\mathrm{Nat}}(0 : \mathrm{Nat}) & \text{and} & \aleph_1 \vdash_{\mathcal{I}_2} \mathrm{Nat}\ \mathsf{coty}(\mathrm{Nat})\\
\mathsf{main}_{\mathrm{Nat}}(s : \mathrm{Nat} \to \mathrm{Nat}) & & \aleph_1 \vdash_{\mathcal{I}_2} \mathrm{Nat} \to \mathrm{Nat}\ \mathsf{coty}(\mathrm{Nat})\\
 & & \aleph_1 \vdash_{\mathcal{I}_2} \mathrm{Odd} \to \mathrm{Nat}\ \mathsf{coty}(\mathrm{Nat})\\
 & & \aleph_1 \vdash_{\mathcal{I}_2} \mathrm{Even} \to \mathrm{Nat}\ \mathsf{coty}(\mathrm{Nat})
\end{array}$$

$$\dfrac{\aleph_2 \vdash \mathrm{Odd} \leq \mathrm{Nat} \quad \dfrac{\dfrac{\mathrm{Nat} \in \aleph_2}{\aleph_2 \vdash_0 \mathrm{Nat}\ \mathsf{type}}}{\aleph_2 \vdash \mathrm{Nat} \leq \mathrm{Nat}}}{\aleph_2 \vdash \mathrm{Nat} \to \mathrm{Nat} \leq \mathrm{Odd} \to \mathrm{Nat}} \qquad \dfrac{\aleph_2 \vdash \mathrm{Even} \leq \mathrm{Nat} \quad \dfrac{\dfrac{\mathrm{Nat} \in \aleph_2}{\aleph_2 \vdash_0 \mathrm{Nat}\ \mathsf{type}}}{\aleph_2 \vdash \mathrm{Nat} \leq \mathrm{Nat}}}{\aleph_2 \vdash \mathrm{Nat} \to \mathrm{Nat} \leq \mathrm{Even} \to \mathrm{Nat}}$$

As every declaration of $c : \tau_1 \to \ldots \to \tau_n \to \mathrm{Odd}$ in $\aleph_2$ is matched by another declaration $c : \tau_1 \to \ldots \to \tau_n \to \mathrm{Nat}$ in $\aleph_2$, we have $\aleph_2 \vdash \mathrm{Odd} \leq \mathrm{Nat}$. Similarly, for every $c : \tau_1 \to \ldots \to \tau_n \to \mathrm{Even} \in \aleph_2$ we have $c : \tau_1 \to \ldots \to \tau_n \to \mathrm{Nat} \in \aleph_2$, hence $\aleph_2 \vdash \mathrm{Even} \leq \mathrm{Nat}$.

2. Trivial.

Example 12. We now turn to an example with parametric datatypes. Remember the definition of `'a List` and `'a NeList`. Let

$$\begin{aligned} \aleph_1 &= .; nil : \mathrm{List}[\alpha_1], cons : \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{List}[\alpha_1]; \\ \mathcal{I}_1 &= \{\mathrm{List}\} \\ \aleph_2 &= \aleph_1\ cons : \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{NeList}[\alpha_1]; \\ \mathcal{I}_2 &= \{\mathrm{NeList}\} \end{aligned}$$

We have

$$\frac{\dfrac{.;\ \mathsf{legal}}{.;\vdash_1 \alpha_1\ \mathsf{pretype}(\mathcal{I}_1)} \quad \dfrac{\mathrm{List}\in\mathcal{I}_1 \quad \dfrac{.;\ \mathsf{legal}}{.;\vdash_1 \alpha_1\ \mathsf{type}}}{.;\vdash_1 \mathrm{List}[\alpha_1]\ \mathsf{pretype}(\mathcal{I}_1)} \quad \alpha_1\ \mathsf{pos}\ \alpha_1 \quad \dfrac{\alpha_1\ \mathsf{pos}\ \alpha_1}{\alpha_1\ \mathsf{pos}\ \mathrm{List}[\alpha_1]} \quad \dfrac{\mathrm{List}\ \mathsf{nocc}\ \alpha_1}{\alpha_1\ \mathsf{spos}\ \mathrm{List}} \quad \mathrm{List}[\alpha_1]\ \mathsf{spos}\ \mathrm{List}}{.;\vdash_{\mathcal{I}_1} \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{List}[\alpha_1]\ \mathsf{coty}(\mathrm{List})} \tag{2}$$

$$\frac{\dfrac{\dfrac{\dfrac{.;\ \mathsf{legal}}{.;\ \mathsf{ok}(\mathcal{I}_1)} \quad .;\vdash_{\mathcal{I}_1} \mathrm{List}[\alpha_1]\ \mathsf{coty}(\mathrm{List})}{.; nil : \mathrm{List}[\alpha_1]\ \mathsf{ok}(\mathcal{I}_1)} \quad \dfrac{(2)}{.;\vdash_{\mathcal{I}_1} \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{List}[\alpha_1]\ \mathsf{coty}(\mathrm{List})}}{.; nil : \mathrm{List}[\alpha_1], cons : \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{List}[\alpha_1]\ \mathsf{ok}(\mathcal{I}_1)}}{\aleph_1\ \mathsf{legal}}\ (\mathrm{close}) \tag{3}$$

So, $\aleph_1$ is a legal datatype context. The side-condition of (close), $.;\ \mathsf{compatible}(nil : \mathrm{List}[\alpha_1], cons : \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{List}[\alpha_1])$, is trivially verified. In the following we show that $\aleph_2$ is also a legal datatype context.

$$\frac{\dfrac{\dfrac{(3)}{\aleph_1\ \mathsf{legal}}}{\aleph_1 \vdash_1 \alpha_1\ \mathsf{pretype}(\mathcal{I}_2)} \quad \dfrac{\dfrac{\dfrac{(3)}{\aleph_1\ \mathsf{legal}}}{\aleph_1 \vdash_1 \alpha_1\ \mathsf{pretype}(\mathcal{I}_2)}}{\aleph_1 \vdash_1 \mathrm{List}[\alpha_1]\ \mathsf{pretype}(\mathcal{I}_2)} \quad \alpha_1\ \mathsf{pos}\ \alpha_1 \quad \dfrac{\alpha_1\ \mathsf{pos}\ \alpha_1}{\alpha_1\ \mathsf{pos}\ \mathrm{List}[\alpha_1]} \quad \dfrac{\mathrm{NeList}\ \mathsf{nocc}\ \alpha_1}{\alpha_1\ \mathsf{spos}\ \mathrm{NeList}} \quad \mathrm{List}[\alpha_1]\ \mathsf{spos}\ \mathrm{NeList}}{\aleph_1 \vdash_{\mathcal{I}_2} \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{NeList}[\alpha_1]\ \mathsf{coty}(\mathrm{NeList})} \tag{4}$$

$$\frac{\dfrac{\dfrac{\dfrac{(3)}{\aleph_1\ \mathsf{legal}}}{\aleph_1 \vdash_1 \alpha_1\ \mathsf{pretype}(\mathcal{I}_2)} \quad \dfrac{(4)}{\aleph_1 \vdash_{\mathcal{I}_2} \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{NeList}[\alpha_1]\ \mathsf{coty}(\mathrm{NeList})}}{\aleph_1\ cons : \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{NeList}[\alpha_1]\ \mathsf{ok}(\mathcal{I}_2)}}{\aleph_2\ \mathsf{legal}}\ (\mathrm{close})$$

The side-condition of (close), $\aleph_1\ \mathsf{compatible}(cons : \alpha_1 \to \mathrm{List}[\alpha_1] \to \mathrm{NeList}[\alpha_1])$ is trivially verified.

We conclude this subsection by defining the substitution of a type for a type variable and state some of their properties.

Below we assume given a legal datatype context $\aleph$ and let $\mathcal{T}_\aleph$ be the set of legal types in $\aleph$ i.e., $\mathcal{T}_\aleph = \{\sigma | \aleph \vdash_k \sigma\ \mathsf{type}\}$. In the sequel, we often drop the datatype context of the subtyping rules and write $\tau \leq \sigma$ instead of $\aleph\ \vdash\ \tau \leq \sigma$, if it is clear that $\aleph$ is the datatype context in which we are working.

Next we give the definition of substitution of a type for a type variable and show some properties of it.

Definition 9. *Let $\alpha \in \mathcal{X}$ and $\rho, \tau \in \mathcal{T}_\aleph$. The* substitution of τ for α in ρ, *denoted by $\rho\{\alpha := \tau\}$, is defined by induction on the structure of ρ as follows:*

$$\begin{array}{rcll} \alpha\{\alpha := \tau\} & \equiv & \tau & \\ \alpha'\{\alpha := \tau\} & \equiv & \alpha' & \textit{if } \alpha' \neq \alpha \\ (\gamma \to \sigma)\{\alpha := \tau\} & \equiv & \gamma\{\alpha := \tau\} \to \sigma\{\alpha := \tau\} & \\ [l_1 : \sigma_1, \ldots, l_n : \sigma_n]\{\alpha := \tau\} & \equiv & [l_1 : \sigma_1\{\alpha := \tau\}, \ldots, l_n : \sigma_n\{\alpha := \tau\}] & \\ d[\tau_1, \ldots, \tau_{\mathsf{ar}(d)}]\{\alpha := \tau\} & \equiv & d[\tau_1\{\alpha := \tau\}, \ldots, \tau_{\mathsf{ar}(d)}\{\alpha := \tau\}] & \end{array}$$

Lemma 1. *Let $\tau, \sigma, \rho \in \mathcal{T}_\aleph$.*

1. *If $\tau \leq \sigma$ and α pos ρ, then $\rho\{\alpha := \tau\} \leq \rho\{\alpha := \sigma\}$.*
2. *If $\tau \leq \sigma$ and α neg ρ, then $\rho\{\alpha := \sigma\} \leq \rho\{\alpha := \tau\}$.*

Proof. By simultaneous induction on the structure of ρ.

1. Assume $\tau \leq \sigma$ and α pos ρ.

   (a) If $\rho \equiv \alpha$, trivial.

   (b) If $\rho \equiv \alpha'$, trivial.

   (c) If $\rho \equiv \gamma \to \gamma'$, then $\rho\{\alpha := \tau\} \equiv \gamma\{\alpha := \tau\} \to \gamma'\{\alpha := \tau\}$ and $\rho\{\alpha := \sigma\} \equiv \gamma\{\alpha := \sigma\} \to \gamma'\{\alpha := \sigma\}$. Since α pos $\gamma \to \gamma'$, we have α pos γ' and α neg γ. By induction hypothesis $\gamma'\{\alpha := \tau\} \leq \gamma'\{\alpha := \sigma\}$ and $\gamma\{\alpha := \sigma\} \leq \gamma\{\alpha := \tau\}$. Hence, by $(\leq_\to)$, $\rho\{\alpha := \tau\} \leq \rho\{\alpha := \sigma\}$.

   (d) If $\rho \equiv [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$, then $\rho\{\alpha := \tau\} \equiv [l_1 : \sigma_1\{\alpha := \tau\}, \ldots, l_n : \sigma_n\{\alpha := \tau\}]$ and $\rho\{\alpha := \sigma\} \equiv [l_1 : \sigma_1\{\alpha := \sigma\}, \ldots, l_n : \sigma_n\{\alpha := \sigma\}]$. Since α pos $[l_1 : \sigma_1, \ldots, l_n : \sigma_n]$, we have α pos σ_i for $1 \leq i \leq n$. By induction hypothesis $\sigma_1\{\alpha := \tau\} \leq \sigma_i\{\alpha := \sigma\}$ for $1 \leq i \leq n$. Hence, by $(\leq_{[]})$, $\rho\{\alpha := \tau\} \leq \rho\{\alpha := \sigma\}$.

   (e) If $\rho \equiv d[\tau_1, \ldots, \tau_{\mathsf{ar}(d)}]$, then $\rho\{\alpha := \tau\} \equiv d[\tau_1\{\alpha := \tau\}, \ldots, \tau_{\mathsf{ar}(d)}\{\alpha := \tau\}]$ and $\rho\{\alpha := \sigma\} \equiv d[\tau_1\{\alpha := \sigma\}, \ldots, \tau_{\mathsf{ar}(d)}\{\alpha := \sigma\}]$. Since α pos $d[\tau_1, \ldots, \tau_{\mathsf{ar}(d)}]$, we have α pos τ_i for $1 \leq i \leq \mathsf{ar}(d)$. By induction hypothesis $\tau_1\{\alpha := \tau\} \leq \tau_i\{\alpha := \sigma\}$ for $1 \leq i \leq \mathsf{ar}(d)$. Hence, by $(\leq_{\mathsf{data}})$, $\rho\{\alpha := \tau\} \leq \rho\{\alpha := \sigma\}$.

2. Similar.

□

We let $\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}$ denote $\{\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n\}$ which represent the simultaneous substitutions of the types $\tau_1, \ldots, \tau_n$ for the type variables $\alpha_1, \ldots, \alpha_n$, respectively. Moreover, we write $\boldsymbol{\tau} \leq \boldsymbol{\sigma}$ when $\boldsymbol{\tau}$ and $\boldsymbol{\sigma}$ represent sequences of the same lenght, n, and $\tau_1 \leq \sigma_1, \ldots, \tau_n \leq \sigma_n$.

Lemma 2.

1. *If $\boldsymbol{\tau} \leq \boldsymbol{\sigma}$ and α_i pos ρ for $1 \leq i \leq n$, then $\rho\{\boldsymbol{\alpha} := \boldsymbol{\tau}\} \leq \rho\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}$.*
2. *If $\boldsymbol{\tau} \leq \boldsymbol{\sigma}$ and α_i neg ρ for $1 \leq i \leq n$, then $\rho\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\} \leq \rho\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}$.*

Proof. Similar to the proof of Lemma 1. □

3.2 Expressions and typing

In this subsection, we conclude the definition of $\lambda_{\to,[],\mathsf{data}}$ by defining its expressions, specifying their computational behavior and providing them with a typing system. Below we assume given a set $\mathcal{V}$ of *variables* and let $x, x', x_i, y, \ldots$ range over $\mathcal{V}$. Moreover, we assume given a legal datatype context $\aleph$ and $\sigma, \tau, \ldots$ are assumed to range over $\mathcal{T}_\aleph$. Moreover, we use $\boldsymbol{a}$ as an abbreviation for $a_1 \ldots a_n$.

Definition 10 (Expressions). *The set $\mathcal{E}$ of* expressions *is given by the abstract syntax:*

$$a, b := x \mid \lambda x{:}\tau.\ a \mid a\ b \mid [l_1 = a_1, \ldots, l_n = a_n] \mid a.l \mid \\ c[\boldsymbol{\sigma}]\,\boldsymbol{a} \mid \mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$$

Free and bound variables, substitution $.\{. := .\}$ are defined the usual way. Moreover we assume standard variable conventions [2] and identify record expressions which only differ in the order of their components, e.g. $[l = a, l' = a']$ and $[l' = a', l = a]$. All the constructions are the usual ones, except perhaps for case-expressions, which are typed so as to avoid failure of subject reduction, see e.g. [19], and are slightly different from the usual case expressions in that we pattern-match against constructors rather than against patterns.

Definition 11. *The set of* free variables *of an expression M, denoted by $\mathsf{FV}(M)$, is defined by induction on the structure of M as follows:*

$$\begin{aligned}
\mathsf{FV}(x) &= \{x\} \\
\mathsf{FV}(\lambda x{:}\tau.\ a) &= \mathsf{FV}(a) \backslash \{x\} \\
\mathsf{FV}(a\ b) &= \mathsf{FV}(a) \cup \mathsf{FV}(b) \\
\mathsf{FV}([l_1 = a_1, \ldots, l_n = a_n]) &= \mathsf{FV}(a_1) \cup \ldots \cup \mathsf{FV}(a_n) \\
\mathsf{FV}(a.l) &= \mathsf{FV}(a) \\
\mathsf{FV}(c[\boldsymbol{\sigma}]\, a_1 \ldots a_k) &= \mathsf{FV}(a_1) \cup \ldots \cup \mathsf{FV}(a_k) \\
\mathsf{FV}(\mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}) &= \mathsf{FV}(a) \cup \mathsf{FV}(b_1) \cup \ldots \cup \mathsf{FV}(b_n)
\end{aligned}$$

Definition 12. *Let $M, N \in \mathcal{E}$ and $x \in \mathcal{V}$. The* substitution of N for x in M, *denoted by $M\{x := N\}$, is defined by induction on the structure of M as follows:*

$$\begin{aligned}
x\{x := N\} &\equiv N \\
y\{x := N\} &\equiv x & \textit{if } x \neq y \\
(\lambda x{:}\tau.\ a)\{x := N\} &\equiv (\lambda x{:}\tau.\ a) \\
(\lambda y{:}\tau.\ a)\{x := N\} &\equiv (\lambda y{:}\tau.\ a\{x := N\}) \\
(a\ b)\{x := N\} &\equiv (a\{x := N\}\ b\{x := N\}) \\
[l_1 = a_1, \ldots, l_n = a_n]\{x := N\} &\equiv [l_1 = a_1\{x := N\}, \ldots, l_n = a_n\{x := N\}] \\
(a.l)\{x := N\} &\equiv (a\{x := N\}).l \\
(c[\boldsymbol{\sigma}]\, a_1 \ldots a_k)\{x := N\} &\equiv (c[\boldsymbol{\sigma}]\, a_1\{x := N\} \ldots a_k\{x := N\}) \\
(\mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})\{x := N\} &\equiv \mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ a\{x := N\}\ \mathsf{of}\ \{c_1 \Rightarrow b_1\{x := N\} \mid \ldots \\
&\quad\ \mid c_n \Rightarrow b_n\{x := N\}\}
\end{aligned}$$

By the variable convention, in the fourth clause of the previous definition the variable y doesn't occur free in the term N, and $x \neq y$.

Definition 13 (Context).

1. *A* context *Γ is a finite set of assumptions $x_1 : \tau_1, \ldots, x_n : \tau_n$ such that the x_is are pairwise distinct elements of $\mathcal{V}$ and $\tau_i \in \mathcal{T}_{\aleph}$.*

2. *Let $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ be a context. Define $\mathsf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$. We also write $\Gamma(x_1) = \tau_i$. That is, Γ is considered as a partial function.*

Definition 14 (Typing).

1. *A* judgment *is a triple of the form $\Gamma \vdash_{cs} a : \tau$, where Γ is a context, $a \in \mathcal{E}$ and $\tau \in \mathcal{T}_{\aleph}$. Generally, the subscript cs of $\vdash$ is dropped.*

2. *A judgment is* derivable *if it can be inferred from the rules of Figure 9, where in the (case) rule it is assumed that $c_1 : \tau_1, \ldots, c_n : \tau_n$ are the sole main d-declarations and that τ^{σ} denotes $\xi_1 \to \ldots \to \xi_n \to \sigma$ whenever $\tau = \xi_1 \to \ldots \to \xi_n \to d[\boldsymbol{\rho}]$.*

(start)	$\Gamma \vdash x : \tau$,	if $x : \tau \in \Gamma$
(application)	$\dfrac{\Gamma \vdash e : \tau \to \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \sigma}$	
(abstraction)	$\dfrac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \sigma}$	
(record)	$\dfrac{\Gamma \vdash e_i : \tau_i \qquad (1 \leq i \leq n)}{\Gamma \vdash [l_1 = e_1, \ldots, l_n = e_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]}$	
(select)	$\dfrac{\Gamma \vdash e : [l_1 : \tau_1, \ldots, l_n : \tau_n]}{\Gamma \vdash e.l_i : \tau_i}$,	if $1 \leq i \leq n$
(constructor)	$\dfrac{\Gamma \vdash b_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\} \quad (1 \leq i \leq k)}{\Gamma \vdash c[\boldsymbol{\tau}]\, \boldsymbol{b} : d[\boldsymbol{\tau}]}$,	if $c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph$
(case)	$\dfrac{\Gamma \vdash a : d[\boldsymbol{\rho}] \quad \Gamma \vdash b_i : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma} \quad (1 \leq i \leq n)}{\Gamma \vdash \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma}$	
(subsumption)	$\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \sigma}$,	if $\aleph \vdash \tau \leq \sigma$

Fig. 9. Typing rules

3. *An expression $a \in \mathcal{E}$ is* typable *if $\Gamma \vdash a : \sigma$ for some context Γ and type σ.*

Note that the connection between the typing judgment and the subtyping judgment is made by the subsumption rule, which states that if an expression e has type τ, and τ is a subtype of σ, then e also has type σ. That is, subtyping behaves very much like the inclusion, when type membership is seem as set membership.

The computational behavior of $\lambda_{\to,[],\mathsf{data}}$ is drawn from the usual notion of β-reduction, ι-reduction and π-reduction. We begin defining the notion of compatibility.

Definition 15 (Compatibility). *A binary relation $\mathcal{R}$ on $\mathcal{E}$ is called* compatible *if*

$$
\begin{aligned}
a\ \mathcal{R}\ a' \Rightarrow\ & (a\, b)\ \mathcal{R}\ (a'\, b)\ , \\
& (b\, a)\ \mathcal{R}\ (b\, a')\ , \\
& (\lambda x{:}\tau.\, a)\ \mathcal{R}\ (\lambda x{:}\tau.\, a')\ , \\
& [l_1 = a_1, \ldots, l_i = a, \ldots, l_n = a_n]\ \mathcal{R}\ [l_1 = a_1, \ldots, l_i = a', \ldots, l_n = a_n]\ , \\
& (a.l)\ \mathcal{R}\ (a'.l)\ , \\
& (c[\boldsymbol{\sigma}]\ a_1 \ldots a \ldots a_k)\ \mathcal{R}\ (c[\boldsymbol{\sigma}]\ a_1 \ldots a' \ldots a_k)\ , \\
& (\mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})\ \mathcal{R}\ (\mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ a'\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})\ , \\
& (\mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ e\ \mathsf{of}\ \{c_1 \Rightarrow a_1 \mid \ldots \mid c_i \Rightarrow a \mid \ldots \mid c_n \Rightarrow a_n\})\ \mathcal{R} \\
& \qquad\qquad (\mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ e\ \mathsf{of}\ \{c_1 \Rightarrow a_1 \mid \ldots \mid c_i \Rightarrow a' \mid \ldots \mid c_n \Rightarrow a_n\})
\end{aligned}
$$

Definition 16 (Compatible closure). *Let $\mathcal{R}$ be a binary relation on $\mathcal{E}$. The* compatible closure *of $\mathcal{R}$ is the least relation extending $\mathcal{R}$ that is compatible.*

We introduce now the formal definition of the β, π and ι reductions.

Definition 17 (Reductions).

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

$$(\lambda x{:}\sigma.\, a)\ b \to_\beta a\{x := b\}$$

2. *π-reduction $\to_\pi$ is defined as the compatible closure of the rule*

$$[l_1 = a_1, \ldots, l_n = a_n].l_i \to_\pi a_i$$

3. *ι-reduction $\to_\iota$ is defined as the compatible closure of the rule*

$$\mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}']}\ (c_i[\boldsymbol{\tau}]\,\boldsymbol{a})\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\} \to_\iota f_i\,\boldsymbol{a}$$

4. *$\to_{basic}$ is defined as $\to_\beta \cup \to_\pi \cup \to_\iota$.*

5. *$\twoheadrightarrow_{basic}$ and $=_{basic}$ are respectively defined as the reflexive-transitive and the reflexive-symmetric-transitive closures of $\to_{basic}$.*

Note that we do not require $\boldsymbol{\tau}$ and $\boldsymbol{\tau}'$ to coincide in the definition of ι-reduction as it would lead to a too weak equational theory. However, the typing rules will enforce $\boldsymbol{\tau} \leq \boldsymbol{\tau}'$ on legal terms.

4 Meta-theory of the core language

This section is devoted to the study of some important properties of the $\lambda_{\to,[],\mathsf{data}}$ system: confluence, the subject reduction property, the decidability of type-checking, and the property of strong normalization. The majority of the results presented in this section are given in detail. We only omit some straightforward proofs and the proof of confluence.

We assume that each variable has a fixed type and that each type has infinitely many variables. We use $\mathcal{V}^\sigma$ to denote the set of variables of type σ. We also assume that we have a legal datatype context $\aleph$.

4.1 Confluence

In $\lambda_{\to,[],\mathsf{data}}$, the reduction strategy used to compute an expression is not relevant. This is a result of the confluence property of the computation relation, which states that if an expression e can be partially computed into two different expressions e_1 and e_3, then there exists a third expression e' such that both e_1 and e_3 can be computed into e'. Below we only state the principal result without presenting the proof.

Proposition 1 (Confluence). *$\to_{basic}$ is confluent:*

$$a =_{basic} b \quad \Rightarrow \quad \exists\, c \in \mathcal{E}.\ a \twoheadrightarrow_{basic} c \quad \wedge \quad b \twoheadrightarrow_{basic} c$$

Proof. By the standard technique of Tait and Martin-Löf.

4.2 Subject Reduction

In $\lambda_{\to,[],\mathsf{data}}$ the type of an expression is preserved under computation. However, the proof of subject reduction is not trivial and requires some key lemmas which ensure that subtyping is "structurally defined". These key lemmas are hard to establish because of the $(\leq_{\mathsf{trans}})$ rule.

Although the subtyping relation must be transitive, the subtyping system may or may not contain a general trasitivity rule. In the latter case, it shoud be proved that the binary relation derived from this system is transitive. In such case, we say the system has the transitivity elimination proprety.

The transitivity elimination property is also a key step in the decidability of subtyping, since without the transitivity rule the subtyping rules can be directly turned into a subtyping-checking algorithm.

We start proving that our system has the transitivity elimination proprety.

Lemma 3. *Any subtyping derivation containing a sole application of $(\leq_{\mathrm{trans}})$ rule can be transformed into one ended by the same subtyping judgment free from that rule.*

Proof. Consider a subtyping system derivation in $\leq$ with exactly one application of $(\leq_{\text{trans}})$. Consider the application of transitivity in a derivation

$$\frac{T \leq T' \quad T' \leq T''}{T \leq T''}\ (\leq_{\text{trans}})$$

The derivations of $T \leq T'$ and $T' \leq T''$ are transitivity-free. We proceed by case analysis of the last pair of rules used to derive $T \leq T''$. We show that the derivations can be transformed to one in which each transitivity application is "smaller" than the original, i.e. the types involved in the application of $(\leq_{\text{trans}})$ rule are simpler. Thus, by induction, those subderivations can be transformed to transitivity-free derivations.

***Case** $(\leq_{\text{refl}},\ \_\ )$* A derivation of the form

$$\frac{\dfrac{}{\sigma \leq \sigma}\ (\leq_{\text{refl}}) \quad \sigma \leq \tau}{\sigma \leq \tau}\ (\leq_{\text{trans}})$$

can be transformed in $\sigma \leq \tau$.

***Case** $(\ \_\ ,\ \leq_{\text{refl}})$* A derivation of the form

$$\frac{\sigma \leq \tau \quad \dfrac{}{\tau \leq \tau}\ (\leq_{\text{refl}})}{\sigma \leq \tau}\ (\leq_{\text{trans}})$$

can be transformed in $\sigma \leq \tau$.

***Case** $(\leq_{\rightarrow},\ \leq_{\rightarrow})$* A derivation of the form

$$\frac{\dfrac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'}\ (\leq_{\rightarrow}) \quad \dfrac{\sigma'' \leq \sigma' \quad \tau' \leq \tau''}{\sigma' \rightarrow \tau' \leq \sigma'' \rightarrow \tau''}\ (\leq_{\rightarrow})}{\sigma \rightarrow \tau \leq \sigma'' \rightarrow \tau''}\ (\leq_{\text{trans}})$$

can be transformed in

$$\frac{\dfrac{\sigma'' \leq \sigma' \quad \sigma' \leq \sigma}{\sigma'' \leq \sigma}\ (\leq_{\text{trans}}) \quad \dfrac{\tau \leq \tau' \quad \tau' \leq \tau}{\tau \leq \tau''}\ (\leq_{\text{trans}})}{\sigma \rightarrow \tau \leq \sigma'' \rightarrow \tau''}\ (\leq_{\rightarrow})$$

***Case** $(\leq_{[]},\ \leq_{[]})$* A derivation of the form

$$\frac{\dfrac{\sigma_i \leq \rho_i \quad (1 \leq i \leq n+r)}{[l_1 : \sigma_1, \ldots, l_{n+m} : \sigma_{n+m}] \leq [l_1 : \rho_1, \ldots, l_{n+r} : \rho_{n+r}]}\ (\leq_{[]}) \quad \dfrac{\rho_j \leq \tau_j \quad (1 \leq j \leq n)}{[l_1 : \rho_1, \ldots, l_{n+r} : \rho_{n+r}] \leq [l_1 : \tau_1, \ldots, l_n : \tau_n]}\ (\leq_{[]})}{[l_1 : \sigma_1, \ldots, l_{n+m} : \sigma_{n+m}] \leq [l_1 : \tau_1, \ldots, l_n : \tau_n]}\ (\leq_{\text{trans}})$$

with $0 \leq r \leq m$, can be transformed in

$$\frac{\dfrac{\sigma_i \leq \rho_i \quad \rho_i \leq \tau_i}{\sigma_i \leq \tau_i}\ (\leq_{\text{trans}}) \quad (1 \leq i \leq n)}{[l_1 : \sigma_1, \ldots, l_{n+m} : \sigma_{n+m}] \leq [l_1 : \tau_1, \ldots, l_n : \tau_n]}\ (\leq_{[]})$$

***Case** $(\leq_{\mathsf{data}},\ \leq_{\mathsf{data}})$* A derivation of the form

$$\frac{\dfrac{\sigma_i \leq \rho_i \quad (1 \leq i \leq \mathsf{ar}(d))}{d[\boldsymbol{\sigma}] \leq d'[\boldsymbol{\rho}]}\ (\leq_{\mathsf{data}}) \quad \dfrac{\rho_i \leq \tau_i \quad (1 \leq i \leq \mathsf{ar}(d'))}{d'[\boldsymbol{\rho}] \leq d''[\boldsymbol{\tau}]}\ (\leq_{\mathsf{data}})}{d[\boldsymbol{\sigma}] \leq d''[\boldsymbol{\tau}]}\ (\leq_{\text{trans}})$$

with $d \leq d'$ and $d' \leq d''$, can be transformed in

$$\frac{\dfrac{\sigma_i \leq \rho_i \quad \rho_i \leq \tau_i}{\sigma_i \leq \tau_i}\ (\leq_{\text{trans}}) \quad (1 \leq i \leq \mathsf{ar}(d))}{d[\boldsymbol{\sigma}] \leq d''[\boldsymbol{\tau}]}\ (\leq_{\mathsf{data}})$$

because $\mathsf{ar}(d) = \mathsf{ar}(d') = \mathsf{ar}(d'')$ and $d \leq d''$.

So, we can always replace a derivation with one transitivity rule by another without transitivity (the first two cases) or with the transitivity rule applied to simpler types. □

Proposition 2 (Transitivity elimination). *The subtyping system has the transitivity elimination property. In other words, any subtyping derivation containing applications of ($\leq_{\mathrm{trans}}$) rule can be transformed into a transitivity-free derivation ended by the same subtyping judgment.*

Proof. By Lemma 3 we know that a subtyping derivation containing exactly one application of transitivity at last step can be transformed into a transitivity-free derivation. We may then eliminate transitivities from arbitrary derivations one by one, beginning with uppermost application of transitivity. □

Before going into the proof of the subject reduction, let us state the following lemmas:

Lemma 4 (Generation for subtyping).

1. *If $T \leq [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$ then $T \equiv [l_1 : \tau_1, \ldots, l_{n+m} : \tau_{n+m}]$ with $\tau_1 \leq \sigma_i$ for $1 \leq i \leq n$.*
2. *If $\tau \to \sigma \leq T$ then $T \equiv \tau' \to \sigma'$ with $\tau' \leq \tau$ and $\sigma \leq \sigma'$.*
3. *If $T \leq \tau \to \sigma$ then $T \equiv \tau' \to \sigma'$ with $\tau \leq \tau'$ and $\sigma' \leq \sigma$.*

Proof. By inspection, using Propositon 2. □

Lemma 5 (Basis).

1. *Let Γ be a context and $\Gamma' \supseteq \Gamma$ another context. If $\Gamma \vdash M : \sigma$ then $\Gamma' \vdash M : \sigma$.*
2. *If $\Gamma \vdash M : \sigma$ then $\mathsf{FV}(M) \subseteq \mathsf{dom}(\Gamma)$.*

Proof. 1. By induction on the derivation of $\Gamma \vdash M : \sigma$.

Case (start). If M is a variable and $M : \sigma \in \Gamma$ then also $M : \sigma \in \Gamma'$. Hence, $\Gamma' \vdash M : \sigma$.

Case (application). $\Gamma \vdash e\, e' : \sigma$ follows directly from $\Gamma \vdash e : \tau \to \sigma$ and $\Gamma \vdash e' : \tau$. By induction hypothesis $\Gamma' \vdash e : \tau \to \sigma$ and $\Gamma' \vdash e' : \tau$. Then, by the (application) rule, $\Gamma' \vdash e\, e' : \sigma$.

Case (abstraction). $\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \rho$ follows directly from $\Gamma, x{:}\tau \vdash e : \rho$. By the variable convention x does not occur in Γ'. Then, $\Gamma', x{:}\tau$ is also a context which extends $\Gamma, x{:}\tau$. Therefore, by the induction hypothesis we have $\Gamma, x{:}\tau \vdash e : \rho$ and so, by the (abstraction) rule, $\Gamma' \vdash \lambda x{:}\tau.\, e : \tau \to \rho$.

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

2. By induction on the derivation of $\Gamma \vdash M : \sigma$.

Case (start). If M is a variable and $M : \sigma \in \Gamma$ then $\mathsf{FV}(M) = \{M\} \subseteq \mathsf{dom}(\Gamma)$.

Case (application). $\Gamma \vdash e\, e' : \sigma$ follows directly from $\Gamma \vdash e : \tau \to \sigma$ and $\Gamma \vdash e' : \tau$. By induction hypothesis $\mathsf{FV}(e) \subseteq \mathsf{dom}(\Gamma)$ and $\mathsf{FV}(e') \subseteq \mathsf{dom}(\Gamma)$. Hence, $\mathsf{FV}(e\, e') = \mathsf{FV}(e) \cup \mathsf{FV}(e') \subseteq \mathsf{dom}(\Gamma)$.

Case (abstraction). $\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \rho$ follows directly from $\Gamma, x{:}\tau \vdash e : \rho$. By induction hypothesis, $\mathsf{FV}(e) \subseteq \mathsf{dom}(\Gamma, x{:}\tau)$. Let $y \in \mathsf{FV}(\lambda x{:}\tau.\, e)$, then $y \in \mathsf{FV}(e)$ and $y \not\equiv x$. Therefore, $y \in \mathsf{dom}(\Gamma)$.

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

□

Lemma 6 (Generation).

1. *If $\Gamma \vdash x : \sigma$ then $(x : \tau) \in \Gamma$ for some type τ and $\tau \leq \sigma$.*
2. *If $\Gamma \vdash M\, N : T$ then $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash N : \sigma$ for $\tau \leq T$.*

3. *If* $\Gamma \vdash (\lambda x{:}\rho.\ M) : T$ *then* $\Gamma, x : \rho \vdash M : \sigma$ *for some type* σ *and* $T \equiv \rho' \to \sigma'$ *with* $\rho' \leq \rho$ *and* $\sigma \leq \sigma'$.

4. *If* $\Gamma \vdash [l_1 = a_1, \ldots, l_n = a_n] : T$ *then* $\Gamma \vdash a_i : \sigma'_i$ *for* $1 \leq i \leq n$, *and* $T \equiv [l_1 : \sigma_1, \ldots, l_r : \sigma_r]$ *with* $\sigma'_j \leq \sigma_j$ *for* $1 \leq j \leq r \leq n$.

5. *If* $\Gamma \vdash M.l : T$ *then* $\Gamma \vdash M : [l_1 : \sigma_1, \ldots, l : \sigma, \ldots, l_n : \sigma_n]$ *with* $\sigma \leq T$.

6. *If* $\Gamma \vdash c[\boldsymbol{\tau}]\, b_1 \ldots b_k : T$ *then* $\Gamma \vdash b_i : \rho\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}$ *for* $1 \leq i \leq k$ *with* $c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph$ *and* $T \equiv d'[\boldsymbol{\sigma}]$ *with* $d \leq d'$ *and* $\boldsymbol{\tau} \leq \boldsymbol{\sigma}$.

7. *If* $\Gamma \vdash \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 | \ldots | c_n \Rightarrow b_n\} : T$ *then* $\Gamma \vdash a : d[\boldsymbol{\rho}]$, $b_i \vdash (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma} :$ *for* $1 \leq i \leq n$ *and* $\sigma \leq T$.

Proof.

1. By inspection on the derivation of $\Gamma \vdash x : \sigma$.
2. By inspection on the derivation of $\Gamma \vdash M\, N : T$.
3. By inspection on the derivation of $\Gamma \vdash (\lambda x{:}\rho.\ M) : T$. To derive $\Gamma \vdash (\lambda x{:}\rho.\ M) : T$ only two cases can occur: the last rule used is
   Case (abstraction).
   $$\frac{\Gamma, x{:}\rho \vdash M : \sigma}{\Gamma \vdash (\lambda x{:}\rho.\ M) : \rho \to \sigma}$$
   In this case $\Gamma \vdash (\lambda x{:}\rho.\ M) : \rho \to \sigma$, and $T \equiv \rho \to \sigma$ with $\rho \leq \rho$ and $\sigma \leq \sigma$.
   Case (subsumption).
   $$\frac{\Gamma \vdash (\lambda x{:}\rho.\ M) : T'}{\Gamma \vdash (\lambda x{:}\rho.\ M) : T} \quad \text{, with } T' \leq T$$
   By induction hypothesis $\Gamma, x : \rho \vdash M : \sigma$ for some type σ and $T' \equiv \rho' \to \sigma'$ with $\rho' \leq \rho$ and $\sigma \leq \sigma'$. We have $T' \leq T$ thus, using Lemma 4 we have $T \equiv A \to B$, with $A \leq \rho'$ and $\sigma' \leq B$. By transitivity $A \leq \rho'$ and $\sigma' \leq B$. Hence, we have $\Gamma, x : \rho \vdash M : \sigma$ for some type σ and $T \equiv A \to B$ with $A \leq \rho'$ and $\sigma' \leq B$.
4. By inspection on the derivation of $\Gamma \vdash [l_1 = a_1, \ldots, l_n = a_n] : T$.
5. By inspection on the derivation of $\Gamma \vdash M.l : T$.
6. By inspection on the derivation of $\Gamma \vdash c[\boldsymbol{\tau}]\, b_1 \ldots b_k : T$.
7. By inspection on the derivation of $\Gamma \vdash \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 | \ldots | c_n \Rightarrow b_n\} : T$.

□

Lemma 7. *If* $\Gamma \vdash [l_1 = a_1, \ldots, l_n = a_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]$ *then* $\Gamma \vdash a_i : \tau_i$, *for* $1 \leq i \leq n$.

Proof. By induction on the derivation of $\Gamma \vdash [l_1 = a_1, \ldots, l_n = a_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]$. □

Lemma 8 (Substitution). *If* $\Gamma_1, x{:}\rho, \Gamma_2 \vdash M : \sigma$ *and* $\Gamma_1, \Gamma_2 \vdash N : \rho$ *then* $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \sigma$.

Proof. By induction on the generation of $\Gamma_1, x{:}\rho, \Gamma_2 \vdash M : \sigma$ assuming that $\Gamma_1, \Gamma_2 \vdash N : \rho$ is derivable.

Case (start). M is a variable and $M : \sigma \in \Gamma_1, x{:}\rho, \Gamma_2$. Two cases can occur:
- if $M \not\equiv x$ then $M\{x := N\} = M$ and $M \in \mathsf{dom}(\Gamma_1, \Gamma_2)$. Hence, $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \sigma$.
- if $M = x$ then $M\{x := N\} = N$ and $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \rho$. But, if $\Gamma_1, x{:}\rho, \Gamma_2 \vdash x : \sigma$ then $\rho \leq \sigma$ (its easy to see, by induction on the derivation of $\Gamma_1, x{:}\rho, \Gamma_2 \vdash x : \sigma$). Hence, by subsumption $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \sigma$.

Case (application). $M \equiv e\, e'$ and follows directly from $\Gamma_1, x{:}\rho, \Gamma_2 \vdash e : \tau \to \sigma$ and $\Gamma_1, x{:}\rho, \Gamma_2 \vdash e' : \tau$. By induction hypothesis $\Gamma_1, \Gamma_2 \vdash e\{x := N\} : \tau \to \sigma$ and $\Gamma_1, \Gamma_2 \vdash e'\{x := N\} : \tau$. $M\{x := N\} = e\{x := N\}e'\{x := N\}$ therefore, by the (application) rule, $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \sigma$.

Case (abstraction). $M \equiv (\lambda y:\tau.\ e)$ and follows directly from $\Gamma_1, x:\rho, \Gamma_2, y:\tau \vdash e:\sigma$. We have $\Gamma_1, \Gamma_2 \vdash N:\rho$ then, by Lemma 5, $\mathsf{FV}(N) \subseteq \mathsf{dom}(\Gamma_1, \Gamma_2)$. Let $y \notin \mathsf{FV}(N)$, then by Lemma 5, $\Gamma_1, \Gamma_2, y:\tau \vdash N:\rho$. So, by induction hypothesis, $\Gamma_1, \Gamma_2, y:\tau \vdash e\{x := N\}:\sigma$ Hence, by the (abstraction) rule, $\Gamma_1, \Gamma_2 \vdash M\{x := N\}:\sigma$.

Case (record). $M \equiv [l_1 = e_1, \ldots, l_n = e_n]$ and follows directly from $\Gamma_1, x{:}\rho, \Gamma_2 \vdash e_i : \tau_i$ for $1 \leq i \leq n$. By induction hypothesis $\Gamma_1, \Gamma_2 \vdash e_i\{x := N\} : \tau_i$ for $1 \leq i \leq n$. $M\{x := N\} = [l_1 = e_1\{x := N\}, \ldots, l_n = e_n\{x := N\}]$, hence, $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \sigma$ follows using (record).

Case (select). $M \equiv e.l_i$, $\sigma \equiv \sigma_i$ and follows directly from $\Gamma_1, x{:}\rho, \Gamma_2 \vdash e : [l_1 : \tau_1, \ldots, l_n : \tau_n]$. By induction hypothesis $\Gamma_1, \Gamma_2 \vdash e\{x := N\} : [l_1 : \tau_1, \ldots, l_n : \tau_n]$. $M\{x := N\} = (e.l_i)\{x := N\} = e\{x := N\}.l_i$, hence, rule $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \sigma$ follows using (select).

Case (constructor). $M \equiv c[\boldsymbol{\tau}]\,\boldsymbol{b}$, with $c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}]$, $\sigma \equiv d[\boldsymbol{\tau}]$, and follows directly from $\Gamma_1, x{:}\rho, \Gamma_2 \vdash b_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}$ for $1 \leq i \leq k$. By induction hypothesis $\Gamma_1, \Gamma_2 \vdash b_i\{x := N\} : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}$ for $1 \leq i \leq k$. $M\{x := N\} = (c[\boldsymbol{\tau}]\,\boldsymbol{b})\{x := N\} = c[\boldsymbol{\tau}]\ b_i\{x := N\}\ \ldots\ b_k\{x := N\}$ hence, $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \sigma$ follows using (constructor).

Case (case). $M \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$ and follows directly from $\Gamma_1, x{:}\rho, \Gamma_2 \vdash a : d[\boldsymbol{\rho}]$ and $\Gamma_1, x{:}\rho, \Gamma_2 \vdash b_i : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma}$ for $1 \leq i \leq n$. By induction hypothesis $\Gamma_1, \Gamma_2 \vdash a\{x := N\} : d[\boldsymbol{\rho}]$ and $\Gamma_1, \Gamma_2 \vdash b_i\{x := N\} : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma}$ for $1 \leq i \leq n$. $M\{x := N\} = \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a\{x := N\}\ \mathsf{of}\ \{c_1 \Rightarrow b_1\{x := N\} \mid \ \ldots \ \mid c_n \Rightarrow b_n\{x := N\}\}$, hence, $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \sigma$ follows using (case).

Case (subsumption). $\Gamma_1, x:\rho, \Gamma_2 \vdash M:\sigma$ follows directly from $\Gamma_1, x:\rho, \Gamma_2 \vdash M:\tau$ with $\tau \leq \sigma$. By induction hypothesis $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \tau$. Hence, by the (constructor) rule, $\Gamma_1, \Gamma_2 \vdash M\{x := N\} : \sigma$.

□

We arrive now to the main result of this subsection.

Proposition 3 (Subject reduction). *Typing is closed under* $\to_{basic}$*:*

$$\Gamma \vdash a : \sigma \quad \wedge \quad a \to_{basic} b \quad \Rightarrow \quad \Gamma \vdash b : \sigma$$

Proof. By induction on the derivations of $\Gamma \vdash a : \sigma$, considering the last rule:

Case (start). In this case a is a variable so, it can't be reduced.

Case (application). In this case $a \equiv e\,e'$, $\Gamma \vdash e : \tau \to \sigma$ and $\Gamma \vdash e' : \tau$. The expression $e\,e'$ can be reduced if:

$e \to_{basic} e''$**.** In this case $e\ e' \to_{basic} e''\ e'$. By the induction hypothesis and the premises we get $\Gamma \vdash e'' : \tau \to \sigma$. The result $\Gamma \vdash e''\,e' : \sigma$ follows using (application).

$e' \to_{basic} e''$**.** In this case $e\ e' \to_{basic} e\ e''$. By the induction hypothesis and the premises we get $\Gamma \vdash e'' : \tau$. The result $\Gamma \vdash e\,e'' : \sigma$ follows using (application).

$e \equiv (\lambda x{:}\rho.\ M)$**.** In this case we have $\Gamma \vdash \lambda x{:}\rho.\ M : \tau \to \sigma$ and $(\lambda x{:}\rho.\ M)\,e' \to_{basic} M\{x := e'\}$. By Lemma 6 we have $\Gamma, x:\rho \vdash M:\sigma$ and $\tau \leq \rho$. As we have $\Gamma \vdash e' : \tau$ and $\tau \leq \rho$, $\Gamma \vdash e' : \rho$ follows using (subsumption). We have $\Gamma, x:\rho \vdash M:\sigma$ and $\Gamma \vdash e':\rho$ then, by the Substitution Lemma, $\Gamma \vdash M\{x := e'\} : \sigma$.

Case (abstraction). In this case $a \equiv (\lambda x{:}\tau.\, e)$ and $\Gamma, x{:}\tau \vdash e : \sigma$. The expression $(\lambda x{:}\tau.\, e)$ can be reduced only if $e \to_{basic} e'$ and, in this case, $(\lambda x{:}\tau.\ e) \to_{basic} (\lambda x{:}\tau.\ e')$. By induction hypothesis $\Gamma, x{:}\tau \vdash e' : \sigma$. Hence, $\Gamma \vdash (\lambda x{:}\tau.\ e') : \tau \to \sigma$ follows using (abstraction).

Case (record). In this case $a \equiv [l_1 = e_1, \ldots, l_n = e_n]$ and $\Gamma \vdash e_i : \tau_i$ for $1 \leq i \leq n$. The expression $[l_1 = e_1, \ldots, l_n = e_n]$ can only be reduced if, for some e_i, $e_i \rightarrow_{basic} e'_i$, and in this case, $[l_1 = e_1, \ldots, l_i = e_i, \ldots, l_n = e_n] \rightarrow_{basic} [l_1 = e_1, \ldots, l_i = e'_i, \ldots, l_n = e_n]$. By induction hypothesis $\Gamma \vdash e'_i : \tau_i$, for $1 \leq i \leq n$. Hence, $\Gamma \vdash [l_1 = e_1, \ldots, l_i = e'_i, \ldots, l_n = e_n] : [l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n]$ follows using (record).

Case (select). In this case $a \equiv e.l_i$, for $1 \leq i \leq n$, and $\Gamma \vdash e : [l_1 : \tau_1, \ldots, l_n : \tau_n]$. The expression $e.l_i$ can be reduced if:

$e \rightarrow_{basic} e'$. In this case $e.l_i \rightarrow_{basic} e'.l_i$. By induction hypothesis $\Gamma \vdash e' : [l_1 : \tau_1, \ldots, l_n : \tau_n]$. Hence, $\Gamma \vdash e'.l_i : \tau_i$ follows using (select).

$e \equiv [l_1 = a_1, \ldots, l_n = a_n]$. In this case $[l_1 = a_1, \ldots, l_n = a_n].l_i \rightarrow_{basic} a_i$. We have $\Gamma \vdash e : [l_1 : \tau_1, \ldots, l_n : \tau_n]$, hence, $\Gamma \vdash a_i : \tau_i$ follows using Lemma 7.

Case (constructor). In this case $a \equiv c[\boldsymbol{\tau}]\ \boldsymbol{b}$ with $c : \rho_1 \rightarrow \ldots \rightarrow \rho_k \rightarrow d[\boldsymbol{\alpha}] \in \aleph$, $\boldsymbol{b} \equiv b_1 \ldots b_k$ and $\Gamma \vdash b_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}$ for $1 \leq i \leq k$. The expression $c[\boldsymbol{\tau}]\ \boldsymbol{b}$ can only be reduced if for some i, $1 \leq i \leq k$, $b_i \rightarrow_{basic} b'_i$, and in this case $c[\boldsymbol{\tau}]\ b_1 \ldots b_i \ldots b_k \rightarrow_{basic} c[\boldsymbol{\tau}]\ b_1 \ldots b'_i \ldots b_k$. By induction hypothesis $\Gamma \vdash b'_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}$, and $\Gamma \vdash b_j : \rho_j\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}$ for $1 \leq j \leq k$, $j \neq i$. Hence, $\Gamma \vdash c[\boldsymbol{\tau}]\ b_1 \ldots b'_i \ldots b_k : d[\boldsymbol{\tau}]$ follows using (constructor).

Case (case). In this case $a \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_n \Rightarrow b_n\}$, $\Gamma \vdash e : d[\boldsymbol{\rho}]$ and $\Gamma \vdash b_i : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma}$ with $1 \leq i \leq n$. The expression $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ \ldots \ \mid c_n \Rightarrow b_n\}$ can be reduced if:

$e \rightarrow_{basic} e'$. In this case $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 | \ldots | c_n \Rightarrow b_n\} \rightarrow_{basic} \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ e'\ \mathsf{of}\ \{c_1 \Rightarrow b_1 | \ldots | c_n \Rightarrow b_n\}$. By induction hypothesis $\Gamma \vdash e' : d[\boldsymbol{\rho}]$. Hence, $\Gamma \vdash \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ e'\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma$ follows using (case).

$e \equiv (c_i[\boldsymbol{\rho}']\ \boldsymbol{a})$. In this case $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ (c_i[\boldsymbol{\rho}']\ \boldsymbol{a})\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} \rightarrow_{basic} b_i\ \boldsymbol{a}$. As $\Gamma \vdash b_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}$ and $\boldsymbol{a} \equiv a_1 \ldots a_n$ then, by applying n times the (application) rule we get $\Gamma \vdash b_i\ \boldsymbol{a} : \sigma$.

$b_i \rightarrow_{basic} b'_i$. In this case $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_i \Rightarrow b_i \mid \ldots \mid c_n \Rightarrow b_n\} \rightarrow_{basic} \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_i \Rightarrow b'_i \mid \ldots \mid c_n \Rightarrow b_n\}$. By induction hypothesis $\Gamma \vdash b'_i : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma}$. Hence, $\Gamma \vdash \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ e\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_i \Rightarrow b'_i \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma$ follows using (case).

Case (subsumption). In this case we have $\Gamma \vdash a : \tau$ with $\tau \leq \sigma$. If $a \rightarrow_{basic} b$ then, by induction hypothesis, $\Gamma \vdash b : \tau$. Hence, $\Gamma \vdash b : \sigma$ follows using (subsumption).

□

4.3 Decidability of Type-checking

The decidability of type-checking is a fundamental property of typed λ-calculus. Indeed, program correctness on a typed programming language and proof checking in a proof-development system are often reduced to type-checking itself. Thus it is important to be able to decide whether or not a typing judgement is derivable.

In this subsection, we will give an algorithm to decide whether a judgement is derivable in $\lambda_{\rightarrow,[],\mathsf{data}}$. The algorithm relies on:

1. the construction of a set of minimal types of an expression in a given context;
2. an algorithm to decide whether a subtyping judgement is derivable.

One can not rely on the existence of minimal types, as they may not exist (for minimal types to exist, one must require datatypes to be pre-regular, see e.g. [5, 18]). Instead, we can define for every context Γ and expression a a finite set $\mathsf{min}_{\Gamma}(a)$ of minimal types such that

$$\sigma \in \mathsf{min}_{\Gamma}(a) \quad \Rightarrow \quad \Gamma \vdash a : \sigma \tag{5}$$

$$\Gamma \vdash a : \sigma \quad \Rightarrow \quad \exists \tau \in \mathsf{min}_\Gamma(a).\ \tau \leq \sigma \tag{6}$$

The following definition introduces this auxiliary notion of $\mathsf{min}_\Gamma(a)$. In the sequence we state two lemmas proving that (5) and (6) are verified.

Definition 18 (Minimal types). *Let e be an expression and Γ a context. We define the set $\mathsf{min}_\Gamma(e)$ of* minimal types *of e in Γ as follows:*

$$\begin{aligned}
\mathsf{min}_\Gamma(x) &= \{\sigma \mid\ x : \sigma \in \Gamma\}\\
\mathsf{min}_\Gamma(\lambda x{:}\tau.\ a) &= \{\tau \to \sigma \mid\ \sigma \in \mathsf{min}_{\Gamma, x:\tau}(a)\}\\
\mathsf{min}_\Gamma(a\,b) &= \{\sigma \mid\ \exists \tau, \tau' \in \mathcal{T}_\aleph.\ \tau \to \sigma \in \mathsf{min}_\Gamma(a)\ \wedge\ \tau' \in \mathsf{min}_\Gamma(b)\ \wedge\ \tau' \leq \tau\}\\
\mathsf{min}_\Gamma([l_1 = a_1, \ldots, l_n = a_n]) &= \{[l_1 : \tau_1, \ldots, l_n : \tau_n] \mid\ \tau_i \in \mathsf{min}_\Gamma(a_i)\ ,\ for\ 1 \leq i \leq n\}\\
\mathsf{min}_\Gamma(a.l) &= \{\tau \mid\ [l_1 : \tau_1, \ldots, l : \tau, \ldots, l_n : \tau_n] \in \mathsf{min}_\Gamma(a)\}\\
\mathsf{min}_\Gamma(c[\boldsymbol{\sigma}]\boldsymbol{a}) &= \{d[\boldsymbol{\sigma}] \mid\ c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph\ \wedge\ \boldsymbol{\sigma'} \leq \boldsymbol{\sigma}\\
&\qquad \wedge\ \rho_i'\{\boldsymbol{\alpha} := \boldsymbol{\sigma'}\} \in \mathsf{min}_\Gamma(a_i)\ \wedge\ \rho_i' \leq \rho_i\ , for 1 \leq i \leq k\}\\
\mathsf{min}_\Gamma(\mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}) &= \{\tau \mid\ \exists \gamma \in \mathsf{min}_\Gamma(a). \exists \tau_i \in \mathsf{min}_\Gamma(b_i).\ \gamma \leq d[\boldsymbol{\sigma}]\ \wedge\\
&\qquad \tau_i \leq (\boldsymbol{\rho_i}\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\} \to d[\boldsymbol{\sigma}])^\tau\ ,\ with\ c_i : \boldsymbol{\rho_i} \to d[\alpha] \in \aleph,\\
&\qquad for\ 1 \leq i \leq n\}
\end{aligned}$$

The set $\mathsf{min}_\Gamma(e)$ is finite because there are only finitely many declarations for each constructor.

Lemma 9. *If $T \in \mathsf{min}_\Gamma(e)$ then $\Gamma \vdash e : T$.*

Proof. By induction on $T \in \mathsf{min}_\Gamma(e)$.

$T \in \mathsf{min}_\Gamma(x)$. In this case $x : T \in \Gamma$. Hence $\Gamma \vdash x : T$.

$T \in \mathsf{min}_\Gamma(\lambda x{:}\tau.\ a)$. In this case $T \equiv \tau \to \sigma$ and $\sigma \in \mathsf{min}_{\Gamma,x:\tau}(a)$. By induction hypothesis $\Gamma, x : \tau \vdash a : \sigma$. Hence, $\Gamma \vdash (\lambda x{:}\tau.\ a) : \tau \to \sigma$ follows from (abstraction).

$T \in \mathsf{min}_\Gamma(a\,b)$. In this case $T \equiv \sigma$ with $\tau \to \sigma \in \mathsf{min}_\Gamma(a)$, $\tau' \in \mathsf{min}_\Gamma(b)$ and $\tau' \leq \tau$. By induction hypothesis $\Gamma \vdash a : \tau \to \sigma$ and $\Gamma \vdash b : \tau'$. By subsumption $\Gamma \vdash b : \tau$. Hence, $\Gamma \vdash a\,b : \sigma$ follows from (application).

$T \in \mathsf{min}_\Gamma([l_1 = a_1, \ldots, l_n = a_n])$. In this case $T \equiv [l_1 : \tau_1, \ldots, l_n : \tau_n]$ with $\tau_i \in \mathsf{min}_\Gamma(a_i)$ for $1 \leq i \leq n$. By induction hypothesis $\Gamma \vdash a_i : \tau_i$ for $1 \leq i \leq n$. Hence, $\Gamma \vdash [l_1 = a_1, \ldots, l_n = a_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]$ follows from (record).

$T \in \mathsf{min}_\Gamma(a.l)$. In this case $T \equiv \tau$ with $[l_1 : \tau_1, \ldots, l : \tau, \ldots, l_n : \tau_n] \in \mathsf{min}_\Gamma(a)$. By induction hypothesis $\Gamma \vdash a : [l_1 : \tau_1, \ldots, l : \tau, \ldots, l_n : \tau_n]$. Hence, $\Gamma \vdash a.l : \tau$ follows from (select).

$T \in \mathsf{min}_\Gamma(c[\boldsymbol{\sigma}]\boldsymbol{a})$. In this case $T \equiv d[\boldsymbol{\sigma}]$ with $c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\sigma}] \in \aleph$, $\rho_i'\{\boldsymbol{\alpha} := \boldsymbol{\sigma'}\} \in \mathsf{min}_\Gamma(a_i)$, $\boldsymbol{\sigma'} \leq \boldsymbol{\sigma}$ and $\rho_i' \leq \rho_i$. By induction hypothesis $\Gamma \vdash a_i : \rho_i'\{\boldsymbol{\alpha} := \boldsymbol{\sigma'}\}$. Then, by subsumption, $\Gamma \vdash a_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}$, since $\rho_i'\{\boldsymbol{\alpha} := \boldsymbol{\sigma'}\} \leq \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}$ by Lemma 2. Hence, $\Gamma \vdash c[\boldsymbol{\sigma}]\boldsymbol{a} : d[\boldsymbol{\sigma}]$ follows from (constructor).

$T \in \mathsf{min}_\Gamma(\mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})$. In this case $T \equiv \tau$ and $\exists \gamma \in \mathsf{min}_\Gamma(a). \exists \tau_i \in \mathsf{min}_\Gamma(b_i).\ \gamma \leq d[\boldsymbol{\sigma}]\ \wedge\ \tau_i \leq (\boldsymbol{\rho_i}\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\} \to d[\boldsymbol{\sigma}])^\tau$, with $c_i : \boldsymbol{\rho_i} \to d[\alpha] \in \aleph$, for $1 \leq i \leq n$. By induction hypothesis $\Gamma \vdash a : \gamma$ and $\Gamma \vdash b_i : \tau_i$ thus, by subsumption, $\Gamma \vdash a : d[\boldsymbol{\sigma}]$ and $\Gamma \vdash b_i : (\boldsymbol{\rho_i}\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\} \to d[\boldsymbol{\sigma}])^\tau$. Hence, $\Gamma \vdash \mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \tau$ follows from (case).

□

Lemma 10. *If $\Gamma \vdash e : \sigma$ then $\exists \tau \in \mathsf{min}_\Gamma(e).\ \tau \leq \sigma$.*

Proof. By induction on the derivation of $\Gamma \vdash e : \sigma$.

Case (start). In this case $e \equiv x$ and $x : \sigma \in \Gamma$. Hence, by Definition 18, $\sigma \in \mathsf{min}_\Gamma(x)$ and $\sigma \leq \sigma$ by $(\leq_{\mathsf{refl}})$.

Case (application). In this case $e \equiv a\, b$, $\Gamma \vdash a : \rho \to \sigma$ and $\Gamma \vdash b : \rho$. By induction hypothesis $\exists \tau_1 \in \mathsf{min}_\Gamma(a).\ \tau_1 \leq \rho \to \sigma$ and $\exists \tau_2 \in \mathsf{min}_\Gamma(b).\ \tau_2 \leq \rho$. By Lemma 4 $\tau_1 \leq \rho \to \sigma$ implies that $\tau_1 \equiv \tau_1' \to \tau_1''$, with $\rho \leq \tau_1'$ and $\tau_1 \leq \sigma$. Then, by transitivity, $\tau_2 \leq \tau_1'$. We have $\tau_1' \to \tau_1'' \in \mathsf{min}_\Gamma(a)$, $\tau_2 \in \mathsf{min}_\Gamma(b)$ and $\tau_2 \leq \tau_1'$, therefore, by Definition 18, $\tau_1'' \in \mathsf{min}_\Gamma(a\, b)$ and $\tau_1'' \leq \sigma$.

Case (abstraction). In this case $e \equiv \lambda x{:}\tau.\, a$, $\sigma \equiv \tau \to \rho$ and $\Gamma, x : \tau \vdash a : \rho$. By induction hypothesis $\exists \gamma \in \mathsf{min}_{\Gamma, x:\tau}(a).\ \gamma \leq \rho$. Therefore, by Definition 18, $\tau \to \gamma \in \mathsf{min}_\Gamma(\lambda x{:}\tau.\, a)$ and $\tau \to \gamma \leq \tau \to \rho$ follows using $(\leq_\to)$ and $(\leq_{\mathsf{refl}})$.

Case (record). In this case $e \equiv [l_1 = a_1, \ldots, l_n = a_n]$, $\sigma \equiv [l_1 : \tau_1, \ldots, l_n : \tau_n]$ and $\Gamma \vdash a_i : \tau_i$ for $1 \leq i \leq n$. By induction hypothesis $\exists \rho_i \in \mathsf{min}_\Gamma(a_i).\ \rho_i \leq \tau_i$, for $1 \leq i \leq n$. Let $\tau \equiv [l_1 : \rho_1, \ldots, l_n : \rho_n]$, we have $\tau \in \mathsf{min}_\Gamma(e)$ and $\tau \leq \sigma$, by $(\leq_{[]})$.

Case (select). In this case $e \equiv a.l$ and $\Gamma \vdash a : [l_1 : \sigma_1, \ldots, l : \sigma, \ldots, l_n : \sigma_n]$. By induction hypothesis $\exists \gamma \in \mathsf{min}_\Gamma(a).\ \gamma \leq [l_1 : \sigma_1, \ldots, l : \sigma, \ldots, l_n : \sigma_n]$, then by Lemma 4 $\gamma \equiv [l_1 : \rho_1, \ldots, l_r : \rho_r]$ with $r \leq n$ and $\rho_i \leq \sigma_i$ for $1 \leq i \leq r$. As $\gamma \in \mathsf{min}_\Gamma(a)$ then, by Definition 18, $l \equiv l_j$ for some $j \leq r$. Hence, by Definition 18, $\rho_j \in \mathsf{min}_\Gamma(a.l)$ and $\rho_j \leq \sigma$.

Case (constructor). In this case $e \equiv c[\boldsymbol{\tau}]\boldsymbol{a}$, $\sigma \equiv d[\boldsymbol{\tau}]$ and $\Gamma \vdash a_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}$ for $1 \leq i \leq k$ with $c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph$. By induction hypothesis $\exists \gamma_i \in \mathsf{min}_\Gamma(a_i).\ \gamma_i \leq \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}$ for $1 \leq i \leq k$. From Definition 18 follows that $d[\boldsymbol{\tau}] \in \mathsf{min}_\Gamma(c[\boldsymbol{\tau}]\boldsymbol{a})$ and $d[\boldsymbol{\tau}] \leq d[\boldsymbol{\tau}]$ follows using $(\leq_{\mathsf{refl}})$.

Case (case). In this case $e \equiv \mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$, $\sigma \equiv \tau$, $\Gamma \vdash a : d[\boldsymbol{\sigma}]$ and $\Gamma \vdash b_i : (\boldsymbol{\rho_i}\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\} \to d[\boldsymbol{\sigma}])^\tau$ for $1 \leq i \leq n$. By induction hypothesis, $\exists \gamma \in \mathsf{min}_\Gamma(a).\gamma \leq d[\boldsymbol{\sigma}]$ and $\exists \tau_i \in \mathsf{min}_\Gamma(b_i).\tau_i \leq (\boldsymbol{\rho_i}\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\} \to d[\boldsymbol{\sigma}])^\tau$ for $1 \leq i \leq n$. So, trivially, $\tau \in \mathsf{min}_\Gamma(e)$ and $\tau \leq \tau$.

Case (subsumption). In this case $\Gamma \vdash e : \sigma$ follows directly from $\Gamma \vdash e : \tau$ with $\tau \leq \sigma$. By induction hypothesis $\exists \rho \in \mathsf{min}_\Gamma(e).\ \rho \leq \tau$. Hence, $\rho \leq \sigma$ follows from transitivity.

□

The decidability of subtyping can be obtained directly from the subtyping rules excluding the $(\leq_{\mathsf{trans}})$ rule, as the transitivity elimination property holds. The resulting subtyping algorithm is:

$$
\begin{array}{l}
\mathsf{check}(\sigma \leq \tau) = \\
\quad \text{if } \ \sigma \equiv \tau \ \text{ then } \ True \\
\quad \text{else if } \ \sigma \equiv \sigma_1 \to \sigma_2 \ \wedge \ \tau \equiv \tau_1 \to \tau_2 \\
\qquad \text{then } \ \mathsf{check}(\tau_1 \leq \sigma_1) \ \wedge \ \mathsf{check}(\sigma_2 \leq \tau_2) \\
\quad \text{else if } \ \sigma \equiv [l_1 : \sigma_1, \ldots, l_{n+m} : \sigma_{n+m}] \ \wedge \ \tau \equiv [l_1 : \tau_1, \ldots, l_n : \tau_n] \\
\qquad \text{then } \ \bigwedge_{1 \leq i \leq n} \mathsf{check}(\sigma_i \leq \tau_i) \\
\quad \text{else if } \ \sigma \equiv d[\boldsymbol{\sigma}] \ \wedge \ \tau \equiv d'[\boldsymbol{\tau}] \ \wedge \ d \leq d' \\
\qquad \text{then } \ \bigwedge_{1 \leq i \leq \mathsf{ar}(d)} \mathsf{check}(\sigma_i \leq \tau_i) \\
\quad \text{else } \ False
\end{array}
$$

Now we show the correctness of this algorithm.

Proposition 4 (Correcteness of the subtyping algorithm).

$$\sigma \leq \tau \quad \Leftrightarrow \quad \mathsf{check}(\sigma \leq \tau) = True$$

Proof.

$\Leftarrow$) It's easy to construct the derivation from the algorithm. Basically, each case corresponds to the application of a rule.

$\Rightarrow$) By induction on the derivation $\sigma \leq \tau$. By case analysis on the last rule applied.

Case $(\leq_{\mathsf{refl}})$. In this case $\sigma \equiv \tau$ so, $\mathsf{check}(\sigma \leq \tau) = True$.

Case $(\leq_{\rightarrow})$. In this case $\sigma \equiv \sigma_1 \rightarrow \sigma_2$, $\tau \equiv \tau_1 \rightarrow \tau_2$, with $\tau_1 \leq \sigma_1$ and $\sigma_2 \leq \tau_2$. By induction hypothesis $\mathsf{check}(\tau_1 \leq \sigma_1) = True$ and $\mathsf{check}(\sigma_2 \leq \tau_2) = True$ thus, $\mathsf{check}(\sigma \leq \tau) = True$.

Case $(\leq_{[]})$. In this case $\sigma \equiv [l_1 : \sigma_1, \ldots, l_{n+m} : \sigma_{n+m}]$ and $\tau \equiv [l_1 : \tau_1, \ldots, l_n : \tau_n]$. By induction hypothesis $\mathsf{check}(\sigma_i \leq \tau_i) = True$ for $1 \leq i \leq n$ thus, $\mathsf{check}(\sigma \leq \tau) = True$.

Case $(\leq_{\mathsf{data}})$. In this case $\sigma \equiv d[\boldsymbol{\sigma}]$, $\tau \equiv d'[\boldsymbol{\tau}]$ and $\sigma_i \leq \tau_i$, with $1 \leq i \leq \mathsf{ar}(d)$ and $d \leq d'$. By induction hypothesis $\mathsf{check}(\sigma_i \leq \tau_i) = True$ for $1 \leq i \leq \mathsf{ar}(d)$ thus, $\mathsf{check}(\sigma \leq \tau) = True$.

□

We are now ready to state the following proposition.

Proposition 5 (Decidability of subtyping).
Let σ and τ be types. The subtyping judment $\sigma \leq \tau$ is decidable.

Proof. The algorithm $\mathsf{check}(\sigma \leq \tau)$ always terminate because the types involved in the recursive calls of check are always smaller than the types passed initially. □

The set $\mathsf{min}_\Gamma(a)$ is always finite. So, we can decide if $\Gamma \vdash a : \sigma$ is derivable or not with the following algorithm (let $m = \mathsf{min}_\Gamma(a)$) :

$$
\begin{array}{l}
\mathsf{find}(m, \sigma) = \\
\quad \text{if } \ m = \emptyset \ \text{ then } \ False \\
\quad \text{else let } \ \tau \in m \\
\qquad \text{in } \ \text{if } \ \mathsf{check}(\tau \leq \sigma) \ \text{ then } \ True \\
\qquad\quad \text{else } \ \mathsf{find}(m \backslash \{\tau\}, \sigma)
\end{array}
$$

Lemma 11 (Properties of find).

1. $\exists \tau \in m.\, \tau \leq \sigma \quad \Leftrightarrow \quad \mathsf{find}(m, \sigma) = True$
2. *The execution of* $\mathsf{find}(m, \sigma)$ *always terminate.*

Proof.

1. $\Rightarrow$) If $\exists \tau \in m.\, \tau \leq \sigma$ then $m \neq \emptyset$. So, when we pick up τ from m we will have $\mathsf{check}(\tau \leq \sigma) = True$, by Proposition 4. Therefore, $\mathsf{find}(m, \sigma) = True$.
   $\Leftarrow$) If $\mathsf{find}(m, \sigma) = True$ then $m \neq \emptyset$ and for some $\tau \in m$, $\mathsf{check}(\tau \leq \sigma) = True$. So, using Proposition 4, $\exists \tau \in m.\, \tau \leq \sigma$.
2. The algorithm $\mathsf{find}(m, \sigma)$ always terminate because:
   (a) The set m is finite.
   (b) The set of minimal types passed in the recursive call of find is smaller than the initial set.
   (c) $\mathsf{check}(\tau \leq \sigma)$ always terminates.

□

Proposition 6 (Correcteness of find). $\Gamma \vdash a : \sigma \quad \Leftrightarrow \quad \mathsf{find}(\mathsf{min}_\Gamma(a), \sigma) = True$

Proof.

$\Rightarrow$) If $\Gamma \vdash a : \sigma$ then, by Lemma 10, $\exists \tau \in \mathsf{min}_\Gamma(a).\, \tau \leq \sigma$. Hence, by Lemma 11, $\mathsf{find}(\mathsf{min}_\Gamma(a), \sigma) = True$.

$\Leftarrow$) If $\mathsf{find}(\mathsf{min}_\Gamma(a), \sigma) = True$ then, $\exists \tau \in \mathsf{min}_\Gamma(a).\, \tau \leq \sigma$. Thus, by Lemma 10, $\Gamma \vdash a : \tau$. Therefore, $\Gamma \vdash a : \sigma$ follows from (subsumption).

□

Proposition 7 (Decidability of type-checking). *Type-checking is decidable: there exists an algorithm to decide whether a given judgment $\Gamma \vdash a : \sigma$ is derivable.*

Proof. Proceed in two steps: first compute $\mathsf{min}_\Gamma(a)$, second check whether there exists $\tau \in \mathsf{min}_\Gamma(a)$ such that $\tau \leq \sigma$, using $\mathsf{find}(\mathsf{min}_\Gamma(a), \sigma)$. □

Pre-regularity

Definition 19. *Let $\aleph$ be a legal datatype context. For every $\boldsymbol{\rho} \in \mathcal{T}_\aleph^*$ and for every constructor identifier $c \in \aleph$, the set $\mathsf{codom}^\aleph_{(c,\boldsymbol{\rho})}$ of the* codomains *of c is defined as follows:*

$$\mathsf{codom}^\aleph_{(c,\boldsymbol{\rho})} = \{d[\boldsymbol{\alpha}] \in \mathcal{T}_\aleph \mid \exists \boldsymbol{\rho'} \in \mathcal{T}_\aleph^* .\ \boldsymbol{\rho} \leq \boldsymbol{\rho'} \ \wedge\ c : \boldsymbol{\rho'} \to d[\boldsymbol{\alpha}] \in \aleph\}$$

Note that $c : \boldsymbol{\rho} \to \tau$ denote $c : \rho_1 \to \ldots \to \rho_k \to \tau$.

Definition 20. *A legal datatype context $\aleph$ is* pre-regular *if for every constructor $c \in \aleph$ and $\rho \in \mathcal{T}_\aleph^*$, the set $\mathsf{codom}^\aleph_{(c,\boldsymbol{\rho})}$ is either empty or has a minimal element. If it exists, the minimal element of $\mathsf{codom}^\aleph_{(c,\boldsymbol{\rho})}$ is denoted by $\mathsf{mincod}^\aleph_{(c,\boldsymbol{\rho})}$.*

Lemma 12. *Let $\aleph$ be a pre-regular legal datatype context. The set $\mathsf{min}_\Gamma(e)$ of minimal types of e in Γ, has a minimum, denoted by $\mathsf{Min}_\Gamma(e)$.*

Proof. By induction on the structure of e.

$e \equiv x$. In this case $\mathsf{min}_\Gamma(e) = \{\sigma \mid\ x : \sigma \in \Gamma\}$. Since x can only occur once in Γ, $\mathsf{min}_\Gamma(e)$ is a singular set. Hence, it has a minimum.

$e \equiv (\lambda x{:}\tau.\ a)$. In this case $\mathsf{min}_\Gamma(e) = \{\tau \to \sigma \mid\ \sigma \in \mathsf{min}_{\Gamma,x:\tau}(a)\}$. By IH, $\mathsf{min}_{\Gamma,x:\tau}(a)$ has a minimum, thus $\forall \tau \to \sigma \in \mathsf{min}_\Gamma(\lambda x{:}\tau.a).\ \tau \to \mathsf{Min}_{\Gamma,x:\tau}(a) \leq \tau \to \sigma$. Hence, $\mathsf{min}_\Gamma(\lambda x{:}\tau.a)$ has a minimum $\tau \to \mathsf{Min}_{\Gamma,x:\tau}(a)$.

$e \equiv a\ b$. In this case $\mathsf{min}_\Gamma(e) = \{\sigma \mid\ \exists \tau, \tau' \in \mathcal{T}_\aleph.\ \tau \to \sigma \in \mathsf{min}_\Gamma(a) \ \wedge\ \tau' \in \mathsf{min}_\Gamma(b) \ \wedge\ \tau' \leq \tau\}$. By IH, $\mathsf{min}_\Gamma(a)$ and $\mathsf{min}_\Gamma(b)$ have a minimum. Let $\mathsf{Min}_\Gamma(a) = \rho_1 \to \rho_2$ and $\mathsf{Min}_\Gamma(b) = \rho_1'$, with $\rho_1' \leq \rho_1$, then $\forall \sigma \in \mathsf{min}_\Gamma(a\ b).\rho_2 \leq \sigma$. Hence, $\mathsf{min}_\Gamma(a\ b)$ has a minimum, ρ_2.

$e \equiv [l_1 = a_1, \ldots, l_n = a_n]$. In this case $\mathsf{min}_\Gamma(e) = \{[l_1 : \tau_1, \ldots, l_n : \tau_n] \mid\ \tau_i \in \mathsf{min}_\Gamma(a_i)\ ,\ for\ 1 \leq i \leq n\}$. By IH, $\mathsf{min}_\Gamma(a_i)$ has a minimum, for $1 \leq i \leq n$. Thus, $\forall [l_1 : \tau_1, \ldots, l_n : \tau_n] \in \mathsf{min}_\Gamma([l_1 = a_1, \ldots, l_n = a_n]).[l_1 : \tau_1, \ldots, l_n : \tau_n] \leq [l_1 : \tau_1, \ldots, l_n : \tau_n]$. Hence, $\mathsf{min}_\Gamma([l_1 = a_1, \ldots, l_n = a_n])$ has a mimimum.

$e \equiv a.l$. In this case $\mathsf{min}_\Gamma(e) = \{\tau \mid\ [l_1 : \tau_1, \ldots, l : \tau, \ldots, l_n : \tau_n] \in \mathsf{min}_\Gamma(a)\}$. By IH, $\mathsf{min}_\Gamma(a.l)$ has a minimum. Thus, $\forall \sigma \in \mathsf{min}_\Gamma(a.l).\ (\mathsf{Min}_\Gamma(a)).l \leq \sigma$. Hence, $\mathsf{min}_\Gamma(a.l)$ has a minimum.

$e \equiv c[\boldsymbol{\sigma}]\,\boldsymbol{a}$. In this case $\mathsf{min}_\Gamma(e) = \{d[\boldsymbol{\sigma}] \mid\ c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph \ \wedge\ \boldsymbol{\sigma'} \leq \boldsymbol{\sigma} \ \wedge\ \rho_i'\{\boldsymbol{\alpha} := \boldsymbol{\sigma'}\} \in \mathsf{min}_\Gamma(a_i) \ \wedge\ \rho_i' \leq \rho_i\ , for 1 \leq i \leq k\}$. By IH, $\mathsf{min}_\Gamma(a_i)$ has a minimum, for $1 \leq i \leq k$. Thus, $\mathsf{min}_\Gamma(c[\boldsymbol{\sigma}]\,\boldsymbol{a}) = \{d[\boldsymbol{\sigma}] \mid\ c : \rho_1 \to \ldots \rho_k \to d[\boldsymbol{\alpha}] \in \aleph \ \wedge\ \mathsf{Min}_\Gamma(a_i) \leq \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\},\ for\ 1 \leq i \leq k\}$. Since $\aleph$ is pre-regular, $\mathsf{min}_\Gamma(c[\boldsymbol{\sigma}]\,\boldsymbol{a})$ has a minimum because $\mathsf{codom}^\aleph_{(c,\boldsymbol{\rho})}$, with $\boldsymbol{\rho} = \mathsf{Min}_\Gamma(a_1) \ldots \mathsf{Min}_\Gamma(a_k)$, has a minimum.

$e \equiv \mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ a$ of $\{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$. In this case $\mathsf{min}_\Gamma(e) = \{\tau \mid\ \exists \gamma \in \mathsf{min}_\Gamma(a).\exists \tau_i \in \mathsf{min}_\Gamma(b_i).\ \gamma \leq d[\boldsymbol{\sigma}] \ \wedge\ \tau_i \leq (\boldsymbol{\rho_i}\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\} \to d[\boldsymbol{\sigma}])^\tau\ ,\ with\ c_i : \boldsymbol{\rho_i} \to d[\alpha] \in \aleph,\ for\ 1 \leq i \leq n\}$. By IH, $\mathsf{min}_\Gamma(a)$ and $\mathsf{min}_\Gamma(b_i)$ for $1 \leq i \leq n$ have a minimum, $\mathsf{Min}_\Gamma(a)$ and $\mathsf{Min}_\Gamma(b_i)$. So, obviously, $\mathsf{min}_\Gamma(\mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ a$ of $\{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$ has a minimum, τ.

□

Proposition 8 (Minimal typing). *Let $\aleph$ be a pre-regular legal datatype context.*

$$\Gamma \vdash t : \sigma \quad \Leftrightarrow \quad \Gamma \vdash t : \mathsf{Min}_\Gamma(t) \ \wedge\ \mathsf{Min}_\Gamma(t) \leq \sigma$$

Proof.

$\Rightarrow$) Assume $\Gamma \vdash t : \sigma$. Then, by Lemma 10, $\exists \tau \in \mathsf{min}_\Gamma(t).\tau \leq \sigma$. By Lemma 12, $\mathsf{Min}_\Gamma(t) \leq \tau$. So, by transitivity, $\mathsf{Min}_\Gamma(t) \leq \sigma$. Hence, by Lemma 9, $\Gamma \vdash t : \mathsf{Min}_\Gamma(t)$, because $\mathsf{Min}_\Gamma(t) \in \mathsf{min}_\Gamma(t)$.

$\Leftarrow$) Assume $\Gamma \vdash t : \mathsf{Min}_\Gamma(t)$ and $\mathsf{Min}_\Gamma(t) \leq \sigma$. Then, by subsumption, $\Gamma \vdash t : \sigma$.

□

4.4 Strong Normalization

Any strategy for computing a well-typed term of $\lambda_{\rightarrow,[],\mathsf{data}}$ leads to an expression that can not be further computed. This property is known as *strong normalization*. As usual, we say that an expression e is *strongly normalizing* with respect to a relation $\rightarrow$ if all reduction sequences starting with e terminate. That is, if there is no infinite sequence $e \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots$ We let $\mathsf{SN}(\rightarrow)$ denote the set of expressions that are strongly normalizing with respect to $\rightarrow$.

Definition 21. *Let $a \in \mathcal{E}$.*

1. *The term a is in basic-normal form if there is no term b such that $a \rightarrow_{basic} b$.*
2. *The term a basic-strongly normalizes if there is no infinite basic-reduction starting with a.*
3. $\mathsf{SN}(\rightarrow_{basic})$ *denote the set of terms that basic-strongly normalize.* $\mathsf{SN}(\sigma)$ *denote the set of expressions of type σ that basic-strongly normalize.*

The strong normalization is usually one of the most difficult meta-theoretical properties to prove. In this subsection we will prove that strong normalization holds for $\lambda_{\rightarrow,[],\mathsf{data}}$. Our method consists in developing an interpretation of the typing calculus in set theory, where types are viewed as sets of terms satisfying some closure conditions, called saturated sets. The interpretation of terms is made using a valuation map $\rho : \mathcal{V} \rightarrow \mathcal{E}$, more precisely we define a term interpretation $(\![.]\!)_\rho : \mathcal{E} \rightarrow \mathcal{E}$. We then show that under this interpretation and suitable conditions, the judgment $\Gamma \vdash M : \sigma$ is read as the set membership $(\![M]\!)_\rho \in [\![\sigma]\!]$, where $[\![\sigma]\!]$ is the saturated set associated with the type σ. Since saturated sets only contain strongly normalizing terms, the proof of strong normalization of the type theory is reduced to verifying the soundness of the interpretation, i.e., that $\Gamma \vdash M : \sigma$ implies $\Gamma \vDash M : \sigma$.

Before introducing the notion of saturated sets, we are going to give some preliminary definitions and basic properties of them. We start with the following:

Definition 22 (Prebase terms). *The set $\mathcal{B}a(\sigma)$ of* σ-prebase terms *is defined inductively as follows:*

1. $\mathcal{V}^\sigma \subseteq \mathcal{B}a(\sigma)$.
2. *If $a \in \mathcal{B}a(\tau \rightarrow \sigma)$ and $b : \tau$ then $a\, b \in \mathcal{B}a(\sigma)$.*
3. *If $a \in \mathcal{B}a(d[\boldsymbol{\rho}])$ and $b_i : \tau_i^\sigma$ for $1 \leq i \leq n$, assuming that $c_i : \tau_i$, then* $\mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} \in \mathcal{B}a(\sigma)$.
4. *If $a \in \mathcal{B}a([l_1 : \sigma_1, \ldots, l : \sigma, \ldots, l_n : \sigma_n])$ then $a.l \in \mathcal{B}a(\sigma)$.*
5. *If $a \in \mathcal{B}a(\tau)$ and $\tau \leq \sigma$, then $a \in \mathcal{B}a(\sigma)$.*

Definition 23 (Base terms). *The set $\mathsf{Base}(\sigma)$ of* σ-base terms *is defined inductively as follows:*

1. $\mathcal{V}^\sigma \subseteq \mathsf{Base}(\sigma)$.
2. *If $a \in \mathsf{Base}(\tau \rightarrow \sigma)$ and $b \in \mathsf{SN}(\tau)$ then $a\, b \in \mathsf{Base}(\sigma)$.*
3. *If $a \in \mathsf{Base}(d[\boldsymbol{\rho}])$ and $b_i \in \mathsf{SN}(\tau_i^\sigma)$ for $1 \leq i \leq n$, assuming that $c_i : \tau_i$, then* $\mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} \in \mathsf{Base}(\sigma)$.
4. *If $a \in \mathsf{Base}([l_1 : \sigma_1, \ldots, l : \sigma, \ldots, l_n : \sigma_n])$ then $a.l \in \mathsf{Base}(\sigma)$.*
5. *If $a \in \mathsf{Base}(\tau)$ and $\tau \leq \sigma$, then $a \in \mathsf{Base}(\sigma)$.*

Informally, a base term is a term whose reduction is stopped by the occurrence of a variable , and whose subterms are strongly normalizing. For example, the terms $x\,(\lambda y\!:\!\tau.\ M)\,N$ and $\mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\ (x\,M)\ \mathsf{of}\ \{c_1 \Rightarrow N_1 \mid \ldots \mid c_n \Rightarrow N_n\}$ are blocked by the variable x.

Lemma 13. *If $\tau \leq \sigma$ then $\mathsf{Base}(\tau) \subseteq \mathsf{Base}(\sigma)$.*

Proof. Immediate form the definition of base terms. □

Lemma 14. *If $a \in \mathsf{Base}(\sigma)$, then $a \not\equiv [l_1 = e_1, \ldots, l_n = e_n]$, $a \not\equiv \lambda x\!:\!\tau.\ b$ and $a \not\equiv (c[\boldsymbol{\tau}]\ \boldsymbol{b})$.*

Proof. Directly form the definition of base terms. □

As it is expected, a set of base terms is closed under $\rightarrow_{basic}$.

Lemma 15. *If $a \in \mathsf{Base}(\sigma)$ and $a \rightarrow_{basic} b$, then $b \in \mathsf{Base}(\sigma)$.*

Proof. Assume $a \in \mathsf{Base}(\sigma)$ and $a \rightarrow_{basic} b$. We will prove, by induction on the derivation of $a \in \mathsf{Base}(\sigma)$, that $b \in \mathsf{Base}(\sigma)$.

1. If $a \in \mathcal{V}^{\sigma}$, then $a \not\rightarrow_{basic} b$.

2. If $a \equiv a_1\, a_2$ with $a_1 \in \mathsf{Base}(\tau \rightarrow \sigma)$ and $a_2 \in \mathsf{SN}(\tau)$ then, using Lemma 14, b can only be of the following forms:
   - $b \equiv b_1\, a_2$ with $a_1 \rightarrow_{basic} b_1$. By induction hypothesis $b_1 \in \mathsf{Base}(\tau \rightarrow \sigma)$. Hence, $b \in \mathsf{Base}(\sigma)$.
   - $b \equiv a_1\, b_2$ with $a_2 \rightarrow_{basic} b_2$. Since $b_2 \in \mathsf{SN}(\tau)$, $b \in \mathsf{Base}(\sigma)$.

3. If $a \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a_1$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$ with $a_1 \in \mathsf{Base}(d[\boldsymbol{\rho}])$ and $f_i \in \mathsf{SN}(\tau_1 \rightarrow ... \rightarrow \tau_k \rightarrow \sigma)$ for $1 \leq i \leq n$ then, using Lemma 14, b can only be of the following forms:
   - $b \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ b_1$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$ with $a_1 \rightarrow_{basic} b_1$. By induction hypothesis $b_1 \in \mathsf{Base}(d[\boldsymbol{\rho}])$. Hence, $b \in \mathsf{Base}(\sigma)$.
   - $b \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a_1$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_j \Rightarrow f'_j \mid \ldots \mid c_n \Rightarrow f_n\}$ for some $j \in \{1, \ldots, n\}$, with $f_j \rightarrow_{basic} f'_j$. Since $f'_j \in \mathsf{SN}(\tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow \sigma)$, $b \in \mathsf{Base}(\sigma)$.

4. If $a \equiv a_1.l$ with $a_1 \in \mathsf{Base}([\ldots, l : \sigma, \ldots])$ then, using Lemma 14, b can only be of the form $b_1.l$ with $a_1 \rightarrow_{basic} b_1$. By induction hypothesis $b_1 \in \mathsf{Base}([\ldots, l : \sigma, \ldots])$. Hence, $b \in \mathsf{Base}(\sigma)$.

5. If $a \in \mathsf{Base}(\tau)$ with $\tau \leq \sigma$, then by induction hypothesis $b \in \mathsf{Base}(\tau)$. Hence, $b \in \mathsf{Base}(\sigma)$.

□

The following state that every base term is strongly normalizing.

Lemma 16. *For every type σ, $\mathsf{Base}(\sigma) \subseteq \mathsf{SN}(\sigma)$.*

Proof. Assume $t \in \mathsf{Base}(\sigma)$. By induction on the derivation of $t \in \mathsf{Base}(\sigma)$, we will prove that $t \in \mathsf{SN}(\sigma)$.

1. If $t \in \mathcal{V}^{\sigma}$, then $t \in \mathsf{SN}(\sigma)$.

2. If $t \equiv a\, b$ with $a \in \mathsf{Base}(\tau \rightarrow \sigma)$ and $b \in \mathsf{SN}(\tau)$, then by induction hypothesis $a \in \mathsf{SN}(\tau \rightarrow \sigma)$. Since a is a base term, $a \not\twoheadrightarrow_{basic} \lambda x{:}\tau.\, N$. Hence, $t \in \mathsf{SN}(\sigma)$.

3. If $t \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$ with $a \in \mathsf{Base}(d[\boldsymbol{\rho}])$ and $f_i \in \mathsf{SN}(\tau_1 \rightarrow \ldots \rightarrow \tau_k \rightarrow \sigma)$ for $1 \leq i \leq n$, then by induction hypothesis $a \in \mathsf{SN}(d[\boldsymbol{\rho}])$. Since a is a base term, $a \not\twoheadrightarrow_{basic} (c_i[\boldsymbol{\rho}]\, \boldsymbol{b})$. Hence, $t \in \mathsf{SN}(\sigma)$.

4. If $t \equiv a.l$ with $a \in \mathsf{Base}([\ldots, l : \sigma, \ldots])$, then by induction hypothesis $a \in \mathsf{SN}([\ldots, l : \sigma, \ldots])$. Since a is a base term, $a \not\twoheadrightarrow_{basic} [\ldots, l = b, \ldots]$. Hence, $t \in \mathsf{SN}(\sigma)$.

5. If $t \in \mathsf{Base}(\tau)$ with $\tau \leq \sigma$, then by induction hypothesis $t \in \mathsf{SN}(\tau)$. Hence, $t \in \mathsf{SN}(\sigma)$.

□

Definition 24.

1. *Let $X, Y \subseteq \mathcal{E}$. Define $X \rightarrow Y$ a subset of $\mathcal{E}$ by*

$$X \rightarrow Y = \{F \in \mathcal{E} \mid \forall x \in X.\, F\, x \in Y\}$$

2. *Let $X_i \subseteq \mathcal{E}$ for $1 \leq i \leq n$. Define $[l_i : X_i, \ldots, l_n : X_n]$ a subset of $\mathcal{E}$ by*

$$[l_i : X_i, \ldots, l_n : X_n] = \{a \in \mathcal{E} \mid a.l_i \in X_i \text{ for } 1 \leq i \leq n\}$$

The following definition is required in order to formulate the notion of saturated set.

Definition 25.

1. *The* key-redex *of $M \in \mathcal{E}$ is defined inductively as follows:*
   (a) *If M is a redex, then M is its own key-redex.*
   (b) *If M has key-redex N, then $M\,P$ has key-redex N.*
   (c) *If M has key-redex N, then $M.l$ has key-redex N.*
   (d) *If M has key-redex N, then $\mathsf{case}^{\sigma}_{d[\tau]}\ M\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$ has key-redex N.*
2. Key-reduction $\to_k$ *is defined as the smallest relation such that $M \to_k M'$ if M' is obtained from M by contracting the unique (is it exists) key-redex of M.*

When M has a key-redex, let $\mathsf{red}_\mathsf{k}(M)$ be the term obtained performing the key-reduction of M. One of the interests of this reduction strategy is that whenever M has a key-redex and $M \to M'$ by a non-key reduction, then M' has still a key-redex and $\mathsf{red}_\mathsf{k}(M) \to \mathsf{red}_\mathsf{k}(M')$

$$\begin{tikzcd} M \arrow[r] \arrow[d, "k"] & M' \arrow[d, "k"] \\ \mathsf{red}_\mathsf{k}(M) \arrow[r] & \mathsf{red}_\mathsf{k}(M') \end{tikzcd}$$

In other words, whenever a term has a key-redex, performing a key-reduction is an unavoidable step for computing the normal form of the term.

In order to prove some properties of the saturated sets, we need to have a characterization of strongly normalizing terms. We will now characterize strongly normalizing expressions by an inductively defined set. We start by giving a characterization of the set of normal forms, also as an inductively defined set.

Definition 26. *The set $\mathcal{NF}(\sigma)$ of* normal forms *of type σ is the smallest set of expressions of type σ that satisfies the following:*

1. $$\mathcal{V}^\sigma \subseteq \mathcal{NF}(\sigma)$$

2. $$\frac{P_i \in \mathcal{NF}(\sigma_i) \quad (1 \le i \le n)}{[l_1 = P_1, \ldots, l_n = P_n] \in \mathcal{NF}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])}$$

3. $$\frac{P_i \in \mathcal{NF}(\rho_i) \quad (1 \le i \le n)}{(c[\boldsymbol{\tau}]P_1 ... P_n) \in \mathcal{NF}(d[\boldsymbol{\tau}])}, \qquad \textit{with } c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}] \in \aleph$$

4. $$\frac{P \in \mathcal{NF}([l_1 : \sigma_1, \ldots, l_n : \sigma_n]) \quad P \not\equiv [l_1 = P_1, \ldots, l_m = P_m],\, m \ge n}{P.l_i \in \mathcal{NF}(\sigma_i)}, \qquad \textit{for some } i \in \{1, \ldots, n\}$$

5. $$\frac{M \in \mathcal{NF}(d[\boldsymbol{\rho}]) \quad M \not\equiv (c_i[\boldsymbol{\rho}]P_1 \ldots P_{k_i}) \quad f_i \in \mathcal{NF}(\rho_1 \to \ldots \to \rho_{k_i} \to \tau) \quad (1 \le i \le n)}{\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ M \ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\} \in \mathcal{NF}(\tau)}$$

6. $$\frac{M \in \mathcal{NF}(\sigma)}{\lambda x{:}\tau.\, M \in \mathcal{NF}(\tau \to \sigma)}$$

7. $$\frac{P \in \mathcal{NF}(\tau_1 \to ... \to \tau_n \to \sigma) \quad M_i \in \mathcal{NF}(\tau_i) \quad P \not\equiv (\lambda x{:}\tau_1.\, M) \quad (1 \le i \le n)}{P\, M_1 ... M_n \in \mathcal{NF}(\sigma)}$$

8. $$\frac{M \in \mathcal{NF}(\tau) \quad \tau \le \sigma}{M \in \mathcal{NF}(\sigma)}$$

Proposition 9. *An expression M of type σ is a normal form if and only if $M \in \mathcal{NF}(\sigma)$.*

Proof.

$\Rightarrow$) By induction on the structure of M. Assume M is a normal form of type σ.

1. If $M \in \mathcal{V}^\sigma$, then $M \in \mathcal{NF}(\sigma)$, by rule 1.
2. If $M \equiv \lambda x{:}\tau.N$ and $\sigma \equiv \tau \to \gamma$, then N must be a normal form. By induction hypothesis $N \in \mathcal{NF}(\gamma)$. Hence, by rule 6, $M \in \mathcal{NF}(\sigma)$.
3. If $M \equiv P\,Q$ and $\sigma \equiv \tau \to \gamma$, then P and Q must be normal forms and $P \not\equiv \lambda x{:}\tau.\, N$. Thus, by induction hypothesis $P \in \mathcal{NF}(\tau \to \gamma)$ and $Q \in \mathcal{NF}(\tau)$. Hence, by rule 7, $M \in \mathcal{NF}(\sigma)$.
4. If $M \equiv [l_1 = P_1, \ldots, l_n = P_n]$ and $\sigma \equiv [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$, then $P_1, \ldots, P_n$ are normal forms. By induction hypothesis $P_i \in \mathcal{NF}(\sigma_i)$ for $1 \le i \le n$. Hence, by rule 2, $M \in \mathcal{NF}(\sigma)$.
5. If $M \equiv P.l$, then P must be a normal form and $P \not\equiv [\ldots, l = P_0, \ldots]$. By induction hypothesis $P \in \mathcal{NF}([\ldots, l = \sigma, \ldots])$. Hence, by rule 4, $M \in \mathcal{NF}(\sigma)$.
6. If $M \equiv (c[\boldsymbol{\tau}]\, P_1..P_n)$, then $P_1, \ldots, P_n$ are normal forms. By induction hypothesis $P_i \in \mathcal{NF}(\rho_i)$ for $1 \le i \le n$. Hence, by rule 3, $M \in \mathcal{NF}(\sigma)$.
7. If $M \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ P\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$, then $P, f_1, \ldots, f_n$ are normal forms and $P \not\equiv (c_i[\boldsymbol{\tau}]\, P_1..P_{k_i})$ for $1 \le i \le n$. By induction hypothesis $P \in \mathcal{NF}(d[\boldsymbol{\rho}])$ and $f_i \in \mathcal{NF}(\tau_i^\sigma)$ for $1 \le i \le n$. Hence, by rule 5, $M \in \mathcal{NF}(\sigma)$.

8. If M is normal form of type σ because $M:\tau$ and $\tau \leq \sigma$, then by induction hypothesis $M \in \mathcal{NF}(\tau)$. Hence, by rule 8, $M \in \mathcal{NF}(\sigma)$.

$\Leftarrow$) Suppose $M \in \mathcal{NF}(\sigma)$. By induction on the derivation of $M \in \mathcal{NF}(\sigma)$.

1. Assume $M \in \mathcal{V}^\sigma$. Then M is a normal form of type σ.

2. Assume $M \equiv [l_1 = P_1, \ldots, l_n = P_n]$ and $\sigma \equiv [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$, with $P_i \in \mathcal{NF}(\sigma_i)$ for $1 \leq i \leq n$. By induction hypothesis every P_i is a normal form of type σ_i. Hence, M is a normal form of type σ.

3. Assume $M \equiv (c[\boldsymbol{\tau}]\, P_1..P_n)$, with $P_i \in \mathcal{NF}(\rho_i)$ for $1 \leq i \leq n$. By induction hypothesis every P_i is a normal form of type ρ_i. Hence, M is a normal form of type σ.

4. Assume $M \equiv P.l$ with $P \not\equiv [\ldots, l = P_0, \ldots]$ and $P \in \mathcal{NF}([\ldots, l : \sigma, \ldots])$. By induction hypothesis P is a normal form of type $[\ldots, l : \sigma, \ldots]$. On the other hand, we can't have a π-reduction. Hence, M is a normal form of type σ.

5. Assume $M \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ P\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$ with $P \in \mathcal{NF}(d[\boldsymbol{\rho}])$, $P \not\equiv (c_i[\boldsymbol{\tau}]\, P_1..P_{k_i})$ and $f_i \in \mathcal{NF}(\rho_1 \to \ldots \to \rho_{k_i} \to \sigma)$ for $1 \leq i \leq n$. By induction hypothesis $P, f_1, \ldots, f_n$ are normal forms. On the other hand, we can't have a ι-reduction. Hence, M is a normal form of type σ.

6. Assume $M \equiv \lambda x{:}\tau.\, P$ and $\sigma \equiv \tau \to \gamma$, with $P \in \mathcal{NF}(\gamma)$. By induction hypothesis P is a normal form of type γ. Hence, M is a normal form of type σ.

7. Assume $M \equiv P\, M_1, \ldots, M_n$ with $P \in \mathcal{NF}(\tau_1 \to \ldots \to \tau_n \to \sigma)$, $M_i \in \mathcal{NF}(\tau_i)$ for $1 \leq i \leq n$ and $P \not\equiv \lambda x{:}\tau_1.\, N$. By induction hypothesis $P, M_1, \ldots, M_m$ are normal forms. On the other hand, we can't have a β-reduction. Hence, M is a normal form of type σ.

8. Assume $M \equiv \mathcal{NF}(\tau)$ and $\tau \leq \sigma$. By induction hypothesis M is a normal form of type τ. Hence, by subsumption, M is a normal form of type σ.

□

Lemma 17.

1. *If $P \in \mathcal{NF}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$ and $P \not\equiv [l_1 = P_1, \ldots, l_m = P_m]$ for $n \leq m$, then $P \in \mathsf{Base}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$.*
2. *If $P \in \mathcal{NF}(d[\boldsymbol{\tau}])$ and $P \not\equiv (c[\boldsymbol{\tau}]\, Q_1...Q_n)$, then $P \in \mathsf{Base}(d[\boldsymbol{\tau}])$.*
3. *If $P \in \mathcal{NF}(\tau \to \sigma)$ and $P \not\equiv (\lambda x{:}\tau.\, N)$, then $P \in \mathsf{Base}(\tau \to \sigma)$.*

Proof. By simultaneous induction on the derivation of $P \in \mathcal{NF}(\psi)$.

1. Assume $P \in \mathcal{NF}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$ and $P \not\equiv [l_1 = P_1, \ldots, l_m = P_m]$ for $n \leq m$. Then the following cases can occur:

   – $P \in \mathcal{V}^{[l_1:\sigma_1,\ldots,l_n:\sigma_n]}$. In this case, $P \in \mathsf{Base}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$.

   – $P \equiv M.l_i$ with $M \in \mathcal{NF}([l_1 : \gamma_1, \ldots, l_w : \gamma_w])$, $M \not\equiv [l_1 = M_1, \ldots, l_k = M_k]$, $i \in \{1, \ldots, w\}$ and $w \leq k$. In this case, $M \in \mathsf{Base}([l_1 : \gamma_1, \ldots, l_w : \gamma_w])$ by induction hypothesis. Thus, $P \in \mathsf{Base}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$.

   – $P \equiv \mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ M\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\} \in \mathcal{NF}(\gamma)$ with $\gamma \equiv [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$, $M \in \mathcal{NF}(d[\boldsymbol{\rho}])$, $M \not\equiv (c_i[\boldsymbol{\rho}] P_1...P_{k_i})$ and $f_i \in \mathcal{NF}(\rho_1 \to \ldots \to \rho_{k_i} \to \gamma)$ for $1 \leq i \leq n$. In this case, $M \in \mathsf{Base}(d[\boldsymbol{\rho}])$ by induction hypothesis. By Proposition 9 $f_i \in \mathsf{SN}(\rho_1 \to \ldots \to \rho_{k_i} \to \gamma)$. Thus, $P \in \mathsf{Base}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$.

– $P \equiv M\, M_1 ... M_k$ with $M \in \mathcal{NF}(\gamma_1 \to ... \to \gamma_k \to [l_1 : \sigma_1, \dots, l_n : \sigma_n])$, $M_i \in \mathcal{NF}(\gamma_i)$ for $1 \le i \le k$ and $M \not\equiv (\lambda x{:}\gamma_1.\, N)$. In this case, $M \in \mathsf{Base}(\gamma_1 \to ... \to \gamma_k \to [l_1 : \sigma_1, \dots, l_n : \sigma_n])$ by induction hypothesis. By Proposition 9 $M_i \in \mathsf{SN}(\gamma_i)$ for $1 \le i \le k$ Thus, $P \in \mathsf{Base}([l_1 : \sigma_1, \dots, l_n : \sigma_n])$.

2. Assume $P \in \mathcal{NF}(d[\boldsymbol{\rho}])$ and $P \not\equiv c[\boldsymbol{\rho}]\, Q_1 ... Q_n$. Then the following cases can occur:

– $P \in \mathcal{V}^{d[\boldsymbol{\rho}]}$. In this case, $P \in \mathsf{Base}(d[\boldsymbol{\tau}])$.

– $P \equiv M.l_i$ with $M \in \mathcal{NF}([l_1 : \gamma_1, \dots, l_w : \gamma_w])$, $M \not\equiv [l_1 = M_1, \dots, l_k = M_k]$, $i \in \{1, \dots, w\}$ and $w \le k$. In this case, $M \in \mathsf{Base}([l_1 : \gamma_1, \dots, l_w : \gamma_w])$ by induction hypothesis. Thus, $P \in \mathsf{Base}(d[\boldsymbol{\tau}])$.

– $P \equiv \mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ M\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \dots \mid c_n \Rightarrow f_n\} \in \mathcal{NF}(\gamma)$ with $\gamma \equiv d[\boldsymbol{\tau}]$, $M \in \mathcal{NF}(d[\boldsymbol{\rho}])$, $M \not\equiv (c_i[\boldsymbol{\rho}] P_1 ... P_{k_i})$ and $f_i \in \mathcal{NF}(\rho_1 \to ... \to \rho_{k_i} \to \gamma)$ for $1 \le i \le n$. In this case, $M \in \mathsf{Base}(d[\boldsymbol{\rho}])$ by induction hypothesis. By Proposition 9 $f_i \in \mathsf{SN}(\rho_1 \to ... \to \rho_{k_i} \to \gamma)$. Thus, $P \in \mathsf{Base}(d[\boldsymbol{\tau}])$.

– $P \equiv M\, M_1 ... M_k$ with $M \in \mathcal{NF}(\gamma_1 \to ... \to \gamma_k \to d[\boldsymbol{\tau}])$, $M_i \in \mathcal{NF}(\gamma_i)$ for $1 \le i \le k$ and $M \not\equiv (\lambda x{:}\gamma_1.\, N)$. In this case, $M \in \mathsf{Base}(\gamma_1 \to ... \to \gamma_k \to d[\boldsymbol{\tau}])$ by induction hypothesis. By Proposition 9 $M_i \in \mathsf{SN}(\gamma_i)$ for $1 \le i \le k$ Thus, $P \in \mathsf{Base}(d[\boldsymbol{\tau}])$.

3. Assume $P \in \mathcal{NF}(\tau \to \sigma)$ and $P \not\equiv (\lambda x{:}\tau.\, N)$. Then the following cases can occur:

– $P \in \mathcal{V}^{\tau \to \sigma}$. In this case, $P \in \mathsf{Base}(\tau \to \sigma)$.

– $P \equiv M.l_i$ with $M \in \mathcal{NF}([l_1 : \gamma_1, \dots, l_w : \gamma_w])$, $M \not\equiv [l_1 = M_1, \dots, l_k = M_k]$, $i \in \{1, \dots, w\}$ and $w \le k$. In this case, $M \in \mathsf{Base}([l_1 : \gamma_1, \dots, l_w : \gamma_w])$ by induction hypothesis. Thus, $P \in \mathsf{Base}(\tau \to \sigma)$.

– $P \equiv \mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ M\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \dots \mid c_n \Rightarrow f_n\} \in \mathcal{NF}(\gamma)$ with $\gamma \equiv \tau$
$rightarrow \sigma$, $M \in \mathcal{NF}(d[\boldsymbol{\rho}])$, $M \not\equiv (c_i[\boldsymbol{\rho}] P_1 ... P_{k_i})$ and $f_i \in \mathcal{NF}(\rho_1 \to ... \to \rho_{k_i} \to \gamma)$ for $1 \le i \le n$. In this case, $M \in \mathsf{Base}(d[\boldsymbol{\rho}])$ by induction hypothesis. By Proposition 9 $f_i \in \mathsf{SN}(\rho_1 \to ... \to \rho_{k_i} \to \gamma)$. Thus, $P \in \mathsf{Base}(\tau \to \sigma)$.

– $P \equiv M\, M_1 ... M_k$ with $M \in \mathcal{NF}(\gamma_1 \to ... \to \gamma_k \to \tau \to \sigma)$, $M_i \in \mathcal{NF}(\gamma_i)$ for $1 \le i \le k$ and $M \not\equiv (\lambda x{:}\gamma_1.\, N)$. In this case, $M \in \mathsf{Base}(\gamma_1 \to ... \to \gamma_k \to \tau \to \sigma)$ by induction hypothesis. By Proposition 9 $M_i \in \mathsf{SN}(\gamma_i)$ for $1 \le i \le k$ Thus, $P \in \mathsf{Base}(\tau \to \sigma)$.

□

Lemma 18. *Every $M \in \mathcal{E}$ can be written in the following form:*

$$\lambda x_1{:}\tau_1. \dots \lambda x_m{:}\tau_m.\, N\, Q_1 \dots Q_n \quad , \textit{with } m \ge 0 \textit{ and } n \ge 0$$

where N is of one of the following forms:

1. y
2. $(\lambda y{:}\sigma.\, P)$
3. $(P.l)$
4. $\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ P\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \dots \mid c_n \Rightarrow f_n\}$
5. $(c[\boldsymbol{\sigma}]\, P_1 ... P_k)$
6. $[l_1 = P_1, \dots, l_k = P_k]$

Proof. By case analysis on the structure of M.

1. If M is a variable, then we have $m = n = 0$ and N a variable (case 1).
2. If $M \equiv (\lambda x{:}\tau.\, P)$, then we have case 2 with $m = n = 0$.
3. If $M \equiv P\, Q$, then we have to consider two cases:

– If P is not an application, then we have $m = 0$, $n = 1$, $N = P$ and $Q_1 = Q$.
– If P is an application, then P may be written as $P_1 \dots P_k$ in which P_1 is not an application and $k > 1$. In this case we have $m = 0$, $n = k$, $N = P_1$, $Q_i = P_i + 1$ for $1 \le i \le k-1$ and $Q_k = Q$.

The rest of the cases are proved similarly. □

In the following we characterize the strongly normalizing terms.

Definition 27. *The set $\mathcal{SN}(\sigma)$ is the smallest set of expressions of type σ that satisfies the following:*

(base)
$$\mathsf{Base}(\sigma) \subseteq \mathcal{SN}(\sigma)$$

(lam)
$$\frac{M \in \mathcal{SN}(\sigma)}{(\lambda x{:}\tau.\ M) \in \mathcal{SN}(\tau \to \sigma)}$$

(app)
$$\frac{N \in \mathcal{SN}(\tau) \quad M\{x := N\}\, P_1 ... P_m \in \mathcal{SN}(\sigma),\ m \ge 0}{(\lambda x{:}\tau.\ M) N P_1 ... P_m \in \mathcal{SN}(\sigma)}$$

(rec)
$$\frac{P_i \in \mathcal{SN}(\sigma_i) \quad (1 \le i \le n)}{[l_1 = P_1, \dots, l_n = P_n] \in \mathcal{SN}([l_1 : \sigma_1, \dots, l_n : \sigma_n])}$$

(cons)
$$\frac{Q_i \in \mathcal{SN}(\rho_i) \quad (1 \le i \le n)}{(c[\boldsymbol{\tau}]\, Q_1 ... Q_n) \in \mathcal{SN}(d[\boldsymbol{\tau}])}\ ,$$
with $c : \rho_1 \to \dots \to \rho_n \to d[\boldsymbol{\alpha}] \in \aleph$

(sel1)
$$\frac{Q_i \in \mathcal{SN}(\tau_i) \quad (1 \le i \le n) \quad Q_j P_1 ... P_m \in \mathcal{SN}(\sigma),\ m \ge 0}{[l_1 = Q_1, \dots, l_n = Q_n].l_j\, P_1 ... P_m \in \mathcal{SN}(\sigma)}\ ,$$
for some $j \in \{1, \dots, n\}$

(sel2)
$$\frac{M \in \mathcal{SN}([\dots, l : \tau, \dots]) \quad M \to_k M' \quad M'.l\, P_1 ... P_m \in \mathcal{SN}(\sigma),\ m \ge 0}{M.l\, P_1 ... P_m \in \mathcal{SN}(\sigma)}$$

(case1)
$$\frac{f_i \in \mathcal{SN}(\rho_1 \to ... \to \rho_k \to \tau) \quad (f_j\, \boldsymbol{a})\, P_1 ... P_m \in \mathcal{SN}(\sigma),\ m \ge 0 \quad (1 \le i \le n)}{\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ (c_j[\boldsymbol{\rho}]\, \boldsymbol{a})\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \dots \mid c_n \Rightarrow f_n\}\, P_1 ... P_m \in \mathcal{SN}(\sigma)}\ ,$$
with $\tau \equiv \tau_1 \to ... \to \tau_m \to \sigma$, *for some* $j \in \{1, \dots, n\}$

(case2)
$$\frac{M \in \mathcal{SN}(d[\boldsymbol{\rho}]) \quad M \to_k M' \quad \mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ M'\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \dots \mid c_n \Rightarrow f_n\}\, P_1 ... P_m \in \mathcal{SN}(\sigma),\ m \ge 0}{\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ M\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \dots \mid c_n \Rightarrow f_n\}\, P_1 ... P_m \in \mathcal{SN}(\sigma)}\ ,$$
with $\tau \equiv \tau_1 \to ... \to \tau_m \to \sigma$

(sub)
$$\frac{M \in \mathcal{SN}(\tau) \quad \tau \le \sigma}{M \in \mathcal{SN}(\sigma)}$$

Note that every subterm of a term in $\mathcal{SN}(\sigma)$ is in $\mathcal{SN}(\tau)$, for some appropriate τ.

Lemma 19. *For every type σ, $\mathcal{NF}(\sigma) \subseteq \mathcal{SN}(\sigma)$*

Proof. Assume $M \in \mathcal{NF}(\sigma)$. By induction on the derivation of $M \in \mathcal{NF}(\sigma)$.

1. If $M \in \mathcal{V}^{\sigma}$ then $M \in \mathsf{Base}(\sigma) \subseteq \mathcal{SN}(\sigma)$.
2. If $M \equiv [l_1 = P_1, ..., l_n = P_n]$ and $\sigma \equiv [l_1 : \sigma_1, ..., l_n : \sigma_n]$ with $P_i \in \mathcal{NF}(\sigma_i)$ for $1 \le i \le n$, then by induction hypothesis $P_i \in \mathcal{SN}(\sigma_i)$ for $1 \le i \le n$. Hence, by *(rec)*, $M \in \mathcal{SN}(\sigma)$.

3. If $M \equiv (c[\boldsymbol{\tau}]\, P_1 ... P_n)$ and $\sigma \equiv d[\boldsymbol{\tau}]$ with $P_i \in \mathcal{NF}(\rho_i)$ for $1 \leq i \leq n$ and $c : \rho_1 \to ... \to \rho_n \to d[\boldsymbol{\alpha}] \in \aleph$, then by induction hypothesis $P_i \in \mathcal{SN}(\rho_i)$ for $1 \leq i \leq n$. Hence, by *(cons)*, $M \in \mathcal{SN}(\sigma)$.

4. If $M \equiv P.l_i$ and $\sigma \equiv \sigma_i$ for some $i \in \{1, \ldots, n\}$ with $P \in \mathcal{NF}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$ and $P \not\equiv [l_1 = P_1, ..., l_m = P_m]$, $n \leq m$ then, by Lemma 17, $P \in \mathsf{Base}([l_1 : \sigma_1, ..., l_n : \sigma_n])$. Hence, $M \in \mathsf{Base}(\sigma) \subseteq \mathcal{SN}(\sigma)$.

5. If $M \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ P$ of $\{c_1 \Rightarrow f_1 | \ldots | c_n \Rightarrow f_n\}$ with $P \in \mathcal{NF}(d[\boldsymbol{\rho}])$, $P \not\equiv (c_i[\boldsymbol{\rho}]\, P_1 ... P_k)$ and $f_i \in \mathcal{NF}(\rho_1 \to ... \to \rho_k \to \sigma)$ for $1 \leq i \leq n$ then, by Lemma 17, $P \in \mathsf{Base}(d[\boldsymbol{\rho}])$. Hence, $M \in \mathsf{Base}(\sigma) \subseteq \mathcal{SN}(\sigma)$.

6. If $M \equiv \lambda x{:}\tau.\, N$ and $\sigma \equiv \tau \to \rho$ with $N \in \mathcal{NF}(\rho)$, then by induction hypothesis $N \in \mathcal{SN}(\rho)$. Hence, by *(lam)*, $M \in \mathcal{SN}(\sigma)$.

7. If $M \equiv P M_1 ... M_n$ with $P \in \mathcal{NF}(\tau_1 \to ... \to \tau_n \to \sigma)$, $M_i \in \mathcal{NF}(\tau_i)$ for $1 \leq i \leq n$ and $P \not\equiv \lambda x{:}\tau_1.\, N$ then, by Lemma 17, $P \in \mathsf{Base}(\tau_1 \to ... \to \tau_n \to \sigma)$. By induction hypothesis $M_i \in \mathcal{SN}(\tau_i)$. Hence, $M \in \mathsf{Base}(\sigma) \subseteq \mathcal{SN}(\sigma)$.

8. If $M \in \mathcal{NF}(\sigma)$ because $M \in \mathcal{NF}(\tau)$ with $\tau \leq \sigma$, then by induction hypothesis $M \in \mathcal{SN}(\tau)$. Hence, by *(sub)*, $M \in \mathcal{SN}(\sigma)$.

□

Lemma 20. *If $M : \sigma$ cannot be reduced by key-reduction, written $M \not\to_k$, and $M \not\equiv \lambda x{:}\tau.\, N$, $M \not\equiv (c[\boldsymbol{\tau}]\, \boldsymbol{a})$ and $M \not\equiv [l_1 = P_1, \ldots, l_n = P_n]$, then $M \in \mathcal{B}a(\sigma)$.*

Proof. Assume $M \not\to_k$, then by induction on the structure of M:

1. If $M \in \mathcal{V}^{\sigma}$, then $M \in \mathcal{B}a(\sigma)$.

2. If $M \equiv M_1\, M_2$ then $M_1 \not\to_k$ and $M_1 \not\equiv \lambda x:\tau.\, N$, because $M \not\to_k$. Also $M_1 \not\equiv (c[\boldsymbol{\tau}]\, \boldsymbol{a})$ and $M_1 \not\equiv [l_1 = P_1, \ldots, l_n = P_n]$ because M is well typed. So, by induction hypothesis $M_1 \in \mathcal{B}a(\tau \to \sigma)$. Hence, $M \in \mathcal{B}a(\sigma)$.

3. If $M \equiv P.l$, then $P \not\to_k$ and $P \not\equiv [\ldots, l = Q, \ldots]$, because $M \not\to_k$. Also $P \not\equiv \lambda x{:}\tau.\, N$ and $P \not\equiv (c[\boldsymbol{\tau}]\, \boldsymbol{a})$, because M is well typed. So, by induction hypothesis $P \in \mathcal{B}a([\ldots, l : \sigma, \ldots])$. Hence, $M \in \mathcal{B}a(\sigma)$.

4. If $M \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ P$ of $\{c_1 \Rightarrow f_1 | \ldots | c_n \Rightarrow f_n\}$, then $P \not\to_k$ and $P \not\equiv (c[\boldsymbol{\tau}]\, \boldsymbol{a})$, because $M \not\to_k$. Also $M \not\equiv \lambda x{:}\tau.\, N$ and $M \not\equiv [l_1 = P_1, \ldots, l_n = P_n]$, because M is well typed. So, by induction hypothesis $P \in \mathcal{B}a(d[\boldsymbol{\rho}])$. Hence, $M \in \mathcal{B}a(\sigma)$.

□

Now we will prove that, for some type σ, the set $\mathcal{SN}(\sigma)$ characterizes the strongly normalizing expressions of type σ.

Proposition 10. *An expression M of type σ is strongly normalizing if and only if $M \in \mathcal{SN}(\sigma)$, i.e.,*

$$M \in \mathsf{SN}(\sigma) \ \Leftrightarrow\ M \in \mathcal{SN}(\sigma)$$

Proof.

$\Rightarrow$) Suppose that M is a strongly normalizing expression of type σ. Let $\mathsf{maxred}(M)$ denote the maximum length of a rewrite sequence starting in M and ending in the normal form of M. We will prove by induction on $(\mathsf{maxred}(M), M)$, ordered by the lexicographic product of the usual ordering on $\mathbb{N}$ and the subterm ordering, that $M \in \mathcal{SN}(\sigma)$.
If $\mathsf{maxred}(M) = 0$ then M is a normal form. Then $M \in \mathcal{NF}(\sigma) \subseteq \mathcal{SN}(\sigma)$. Suppose $\mathsf{maxred}(M) > 0$. Let $M \equiv \lambda x_1{:}\tau_1. ... \lambda x_m{:}\tau_m.\, N\, Q_1 ... Q_n$ with $m \geq 0$ and $n \geq 0$, and $\sigma \equiv \tau_1 \to \ldots \to \tau_m \to \tau$. Six cases are distinguished:

1. $N \equiv y$. In this case, we have $M \equiv \lambda x_1{:}\tau_1 \ldots \lambda x_m{:}\tau_m . y\, Q_1 \ldots Q_n$. By induction hypothesis $Q_i \in \mathcal{SN}(\sigma_i)$ for $1 \le i \le n$. So, $y\, Q_1 \ldots Q_n \in \mathsf{Base}(\tau) \subseteq \mathcal{SN}(\tau)$. Hence, by *(lam)*, $M \in \mathcal{SN}(\sigma)$.

2. $N \equiv \lambda y{:}\sigma_1 . P$. In this case, we have $M \equiv \lambda x_1{:}\tau_1 . \ldots \lambda x_m{:}\tau_m . (\lambda y{:}\sigma_1 . P)\, Q_1 \ldots Q_n \rightarrow_{basic} \lambda x_1{:}\tau_1 . \ldots \lambda x_m{:}\tau_m . P\{y := Q_1\}\, Q_2 \ldots Q_n$. By induction hypothesis $\lambda x_1{:}\tau_1 . \ldots \lambda x_m{:}\tau_m . P\{y := Q_1\}\, Q_2 \ldots Q_n \in \mathcal{SN}(\sigma)$. Also by induction hypothesis we have that $Q_1 \in \mathcal{SN}(\sigma_1)$. Hence, by *(app)*, $M \in \mathcal{SN}(\sigma)$.

3. $N \equiv (P.l)$. In this case, we have $M \equiv \lambda x_1{:}\tau_1 . \ldots \lambda x_m{:}\tau_m . (P.l)\, Q_1 \ldots Q_n$.

   (a) If $P \rightarrow_k P'$ then $M \rightarrow_{basic} \lambda x_1{:}\tau_1 \ldots \lambda x_m{:}\tau_m .(P'.l)\, Q_1 \ldots Q_n$. By induction hypothesis $(P'.l)\, Q_1 \ldots Q_n \in \mathcal{SN}(\tau)$. Also by induction hypothesis we have that $P \in \mathcal{SN}([\ldots, l : \gamma, \ldots]))$. Hence, by *(sel2)* and *(lam)*, $M \in \mathcal{SN}(\sigma)$.

   (b) If $P \not\rightarrow_k$ then, by Lemma 20 and because N is well typed, $P \in \mathcal{B}a([\ldots, l : \gamma, \ldots])$ or $P \equiv [\ldots, l = P_0, \ldots]$.

   – If $P \in \mathcal{B}a([\ldots, l : \gamma, \ldots])$ then, as $P \in \mathcal{SN}([\ldots, l : \gamma, \ldots])$ by induction hypothesis, $P \in \mathsf{Base}([\ldots, l : \gamma, \ldots])$. So, $(P.l)\, Q_1 \ldots Q_n \in \mathsf{Base}(\tau) \subseteq \mathcal{SN}(\tau)$. Hence, by *(lam)*, $M \in \mathcal{SN}(\sigma)$.

   – If $P \equiv [l_1 = P_1, \ldots, l = P_0, \ldots, l_k = P_k]$, then $M \rightarrow_{basic} \lambda x_1{:}\tau_1 . \ldots \lambda x_m{:}\tau_m . P_0\, Q_1 \ldots Q_n$. By induction hypothesis $\lambda x_1{:}\tau_1 . \ldots \lambda x_m{:}\tau_m . P_0\, Q_1 \ldots Q_n \in \mathcal{SN}(\sigma)$ so, $P_0\, Q_1 \ldots Q_n \in \mathcal{SN}(\tau)$. Also by induction hypothesis $P_i \in \mathcal{SN}(\gamma_i)$ for $0 \le i \le k$, Hence, by *(sel1)* and *(lam)*, $M \in \mathcal{SN}(\sigma)$.

4. $N \equiv \mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ P\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\}$. In this case, we have $M \equiv \lambda x_1 : \tau_1 . \ldots \lambda x_m : \tau_m . (\mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ P\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\})\, Q_1 \ldots Q_n$.

   (a) If $P \rightarrow_k P'$, then $M \rightarrow_{basic} \lambda x_1{:}\tau_1 \ldots \lambda x_m{:}\tau_m .(\mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ P'\ \mathsf{of}\ \{c_1 \Rightarrow f_1 | \ldots | c_k \Rightarrow f_k\})\, Q_1 \ldots Q_n$. By induction hypothesis $\lambda x_1{:}\tau_1 \ldots \lambda x_m{:}\tau_m .(\mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ P'\ \mathsf{of}\ \{c_1 \Rightarrow f_1 | \ldots | c_k \Rightarrow f_k\})\, Q_1 \ldots Q_n \in \mathcal{SN}(\sigma)$. So $(\mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ P'\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\})\, Q_1 \ldots Q_n \in \mathcal{SN}(\tau)$. Also by induction hypothesis $P \in \mathcal{SN}(d[\boldsymbol{\rho}])$. Hence, by *(case2)* and *(lam)*, $M \in \mathcal{SN}(\sigma)$.

   (b) If $P \not\rightarrow_k$ then, by Lemma 20 and because N is well typed, $P \in \mathcal{B}a(d[\boldsymbol{\rho}])$ or $P \equiv (c_j\, [\boldsymbol{\rho}]\, \boldsymbol{a})$.

   – If $P \in \mathcal{B}a(d[\boldsymbol{\rho}])$ then, as $P \in \mathcal{SN}(d[\boldsymbol{\rho}])$ by induction hypothesis, $P \in \mathsf{Base}(d[\boldsymbol{\rho}])$. By induction hypothesis $f_i \in \mathcal{SN}(\phi_i)$ for $1 \le i \le k$. So, $\mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ P\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\} \in \mathsf{Base}(\sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau)$. Also by induction hypothesis $Q_j \in \mathcal{SN}(\sigma_j)$ for $1 \le j \le n$. So, $(\mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ P\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\})\, Q_1 \ldots Q_n \in \mathsf{Base}(\tau)$. Hence, by *(base)* and *(lam)*, $M \in \mathcal{SN}(\sigma)$.

   – If $P \equiv (c_j\, [\boldsymbol{\rho}]\, \boldsymbol{a})$, then $M \rightarrow_{basic} \lambda x_1{:}\tau_1 . \ldots \lambda x_m{:}\tau_m . (f_j\, \boldsymbol{a})\, Q_1 \ldots Q_n$. By induction hypothesis $\lambda x_1{:}\tau_1 . \ldots \lambda x_m{:}\tau_m . (f_j\, \boldsymbol{a})\, Q_1 \ldots Q_n \in \mathcal{SN}(\sigma)$. So, $(f_j\, \boldsymbol{a})\, Q_1 \ldots Q_n \in \mathcal{SN}(\tau)$. Also by induction hypothesis $f_i \in \mathcal{SN}(\psi_i)$ for $1 \le i \le k$. Hence, by *(case1)* and *(lam)*, $M \in \mathcal{SN}(\sigma)$.

5. $N \equiv (c[\boldsymbol{\sigma}]\, P_1 \ldots P_k)$. In this case, we have $M \equiv \lambda x_1{:}\tau_1 . \ldots \lambda x_m{:}\tau_m . (c[\boldsymbol{\sigma}]\, P_1 \ldots P_k)\, Q_1 \ldots Q_n$. Since M is well typed we must have $n = 0$. By induction hypothesis $P_i \in \mathcal{SN}(\rho_i)$ for $1 \le i \le k$. Hence, by *(cons)* and *(lam)*, $M \in \mathcal{SN}(\sigma)$.

6. $N \equiv [l_1 = P_1, \ldots, l_k = P_k]$. In this case, we have $M \equiv \lambda x_1 : \tau_1 . \ldots \lambda x_m : \tau_m . [l_1 = P_1, \ldots, l_k = P_k]\, Q_1 \ldots Q_n$. Since M is well typed we must have $n = 0$. By induction hypothesis $P_i \in \mathcal{SN}(\rho_i)$ for $1 \le i \le k$. Hence, by *(rec)* and *(lam)*, $M \in \mathcal{SN}(\sigma)$.

$\Leftarrow$) Suppose that $M \in \mathcal{SN}(\sigma)$. We will prove by induction on the derivation of $M \in \mathcal{SN}(\sigma)$ that M is strongly normalizing, $M \in \mathsf{SN}(\sigma)$. Consider an arbitrary rewrite sequence, ζ, starting in M:

$$M \rightarrow_{basic} M_1 \rightarrow_{basic} M_2 \rightarrow_{basic} \ldots$$

Case (base) If $M \in \mathsf{Base}(\sigma)$, then by Lemma 16 $M \in \mathsf{SN}(\sigma)$.

Case (lam) If $M \equiv \lambda x{:}\tau.\, P$ and $\sigma \equiv \tau \rightarrow \gamma$ with $P \in \mathcal{SN}(\gamma)$. By induction hypothesis $P \in \mathsf{SN}(\gamma)$. Therefore $M \in \mathsf{SN}(\sigma)$.

Case (app) If $M \equiv (\lambda x{:}\tau.\, Q)\, N\, P_1 ... P_m$, with $N \in \mathcal{SN}(\tau)$ and $Q\{x := N\}\, P_1 ... P_m \in \mathcal{SN}(\sigma)$, $m \geq 0$, then there are two possibilities: the redex $(\lambda x{:}\tau.\, Q)\, N$ is contracted in ζ or not.

1. In the first case, there is a rewrite step $M_n \rightarrow_{basic} M_{n+1}$ in the rewrite sequence ζ with $M_n \equiv (\lambda x{:}\tau.\, Q')\, N'\, P_1' ... P_m'$ and $M_{n+1} \equiv Q'\{x := N'\}\, P_1' ... P_m'$ such that $Q \twoheadrightarrow_{basic} Q'$, $N \twoheadrightarrow_{basic} N'$ and $P_i \twoheadrightarrow_{basic} P_i'$ for $1 \leq i \leq m$. Since by induction hypothesis $Q\{x := N\}\, P_1 ... P_m \in \mathsf{SN}(\sigma)$, we have that its reduct $M_{n+1} \in \mathsf{SN}(\sigma)$ too. Hence, ζ is finite.

2. In the second case, we have that all the expressions in ζ are of the form $(\lambda x{:}\tau.\, Q')\, N'\, P_1' ... P_m'$, with $Q \twoheadrightarrow_{basic} Q'$, $N \twoheadrightarrow_{basic} N'$ and $P_i \twoheadrightarrow_{basic} P_i'$ for $1 \leq i \leq m$. Since we have by induction hypothesis that $N \in \mathsf{SN}(\tau)$ and $Q\{x := N\}\, P_1 ... P_m \in \mathsf{SN}(\sigma)$, all expressions in ζ are strongly normalizing. Hence, ζ is finite.

Case (rec) If $M \equiv [l_1 = P_1, ..., l_k = P_k]$ and $\sigma \equiv [l_1 : \sigma_1, \ldots, l_k : \sigma_k]$, with $P_i \in \mathcal{SN}(\sigma_i)$ for $1 \leq i \leq k$, then we have that all the expressions in ζ are of the form $[l_1 = P_1', ..., l_k = P_k']$ with $P_i \twoheadrightarrow_{basic} P_i'$ for $1 \leq i \leq k$. Since we have by induction hypothesis that $P_i \in \mathsf{SN}(\sigma_i)$ for $1 \leq i \leq k$, ζ is finite.

Case (cons) If $M \equiv (c[\boldsymbol{\tau}]\, Q_1 ... Q_k)$ and $\sigma \equiv d[\boldsymbol{\tau}]$, with $Q_i \in \mathcal{SN}(\rho_i)$ for $1 \leq i \leq k$, $c : \rho_1 \rightarrow \ldots \rightarrow \rho_k \rightarrow d[\boldsymbol{\alpha}] \in \aleph$, then we have that all the expressions in ζ are of the form $(c[\boldsymbol{\tau}]\, Q_1' ... Q_k')$ with $Q_i \twoheadrightarrow_{basic} Q_i'$ for $1 \leq i \leq k$. Since we have by induction hypothesis that $Q_i \in \mathsf{SN}(\rho_i)$ for $1 \leq i \leq k$, ζ is finite.

Case (sel1) If $M \equiv [l_1 = Q_1, \ldots, l_k = Q_k].l_j\, P_1 ... P_m$, with $Q_i \in \mathcal{SN}(\tau_i)$ for $1 \leq i \leq k$, $Q_j\, P_1 ... P_m \in \mathcal{SN}(\sigma)$, $m \geq 0$ and $j \in \{1, \ldots, k\}$, then there are two possibilities: the redex $[l_1 = Q_1, \ldots, l_k = Q_k].l_j$ is contracted in ζ or not.

1. In the first case, there is a rewrite step $M_n \rightarrow_{basic} M_{n+1}$ in the rewrite sequence ζ with $M_n \equiv [l_1 = Q_1', \ldots, l_k = Q_k'].l_j\, P_1' ... P_m'$ and $M_{n+1} \equiv Q_j'\, P_1' ... P_m'$ such that $Q_i \twoheadrightarrow_{basic} Q_i'$ for $1 \leq i \leq k$ and $P_r \twoheadrightarrow_{basic} P_r'$ for $1 \leq r \leq m$. Since by induction hypothesis $Q_j\, P_1 ... P_m \in \mathsf{SN}(\sigma)$, we have $M_{n+1} \in \mathsf{SN}(\sigma)$ too. Hence ζ is finite.

2. In the second case, we have that all the expressions in ζ are of the form $[l_1 = Q_1', \ldots, l_k = Q_k'].l_j\, P_1' ... P_m'$ with $Q_i \twoheadrightarrow_{basic} Q_i'$ for $1 \leq i \leq k$, and $P_r \twoheadrightarrow_{basic} P_r'$ for $1 \leq r \leq m$. By induction hypothesis $Q_j\, P_1 ... P_m \in \mathsf{SN}(\sigma)$. So, $P_r \in \mathsf{SN}(\sigma_r)$ for $1 \leq r \leq m$. Also by induction hypothesis $Q_i \in \mathsf{SN}(\tau_i)$ for $1 \leq i \leq k$. Therefore, we have that all expressions in ζ are strongly normalizing. Hence, ζ is finite.

Case (sel2) If $M \equiv Q.l\, P_1 ... P_m$, with $Q \in \mathcal{SN}([\ldots, l : \tau, \ldots])$, $Q \rightarrow_k Q'$, $Q'.l\, P_1 ... P_m \in \mathcal{SN}(\sigma)$, $m \geq 0$, then there are two possibilities: the key-reduction step $Q \rightarrow_k Q'$ is in the sequence ζ or not.

1. In the first case, there is a rewrite step $M_n \rightarrow_{basic} M_{n+1}$ in the rewrite sequence ζ with $M_n \equiv Q.l\, P_1' ... P_m'$ and $M_{n+1} \equiv Q'.l\, P_1' ... P_m'$ such that $Q \rightarrow_k Q'$ and $P_i \twoheadrightarrow_{basic} P_i'$ for $1 \leq i \leq m$. Since by induction hypothesis $Q'.l\, P_1 ... P_m \in \mathsf{SN}(\sigma)$, we have $M_{n+1} \in \mathsf{SN}(\sigma)$ too. Hence ζ is finite.

2. In the second case, we have that all the expressions in ζ are of the form $Q''.l\, P'_1 ... P'_m$ with $P_i \twoheadrightarrow_{basic} P'_i$ for $1 \leq i \leq m$, and $Q \twoheadrightarrow_{basic} Q'$ without contracting the key-redex of Q. This is only possible if $Q \equiv Q_1\, Q_2$ or $Q \equiv \mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ Q_1\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\}$.

   - If $Q \equiv Q_1\, Q_2$, then all the expressions in ζ are of the form $(Q_1\, Q''_2).l\, P'_1 ... P'_m$ with $P_i \twoheadrightarrow_{basic} P'_i$ for $1 \leq i \leq m$, and $Q_2 \twoheadrightarrow_{basic} Q''_2$. By induction hypothesis Q and $Q'.l\, P_1 ... P_m$ are strongly normalizing, so P_i for $1 \leq i \leq m$ are strongly normalizing too. Also Q_2 is strongly normalizing because, by induction hypothesis, Q is strongly normalizing. Hence ζ is finite.

   - If $Q \equiv \mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ Q_1\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\}$, then all the expressions in ζ are of the form $(\mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ Q_1\ \mathsf{of}\ \{c_1 \Rightarrow f'_1 \mid \ldots \mid c_k \Rightarrow f'_k\}).l\, P'_1 ... P'_m$ with $P_i \twoheadrightarrow_{basic} P'_i$ for $1 \leq i \leq m$, and $f_j \twoheadrightarrow_{basic} f'_j$ for $1 \leq j \leq k$. Since Q is strongly normalizing by induction hypothesis, f_j is strongly normalizing for $1 \leq j \leq k$. By induction hypothesis $Q'.l\, P_1 ... P_m$ is strongly normalizing, so P_i is strongly normalizing for $1 \leq i \leq m$. Hence ζ is finite.

Case (case1) If $M \equiv \mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ (c_j[\boldsymbol{\rho}]\, Q_1 ... Q_{k_j})\ \mathsf{of}\ \{c_1 \Rightarrow f_1 | \ldots | c_w \Rightarrow f_w\}\, P_1 ... P_m$, with $f_i \in \mathcal{SN}(\rho_1 \to \ldots \to \rho_{k_i} \to \tau)$, $(f_j\, Q_1 ... Q_{k_j})\, P_1 ... P_m \in \mathcal{SN}(\sigma)$, $m \geq 0$, for $1 \leq i \leq w$ and for some $j \in \{1, \ldots, w\}$, then there are two possibilities: the redex $\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ (c_j[\boldsymbol{\rho}]\, Q_1 ... Q_{k_j})\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_w \Rightarrow f_w\}$ is contracted in ζ or not.

1. In the first case, there is a rewrite step $M_n \to_{basic} M_{n+1}$ in the rewrite sequence ζ with $M_n \equiv \mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ (c_j[\boldsymbol{\rho}]\, Q'_1 ... Q'_{k_j})\ \mathsf{of}\ \{c_1 \Rightarrow f'_1 \mid \ldots \mid c_w \Rightarrow f'_w\}\, P'_1 ... P'_m$ and $M_{n+1} \equiv (f_j\, Q'_1 ... Q'_{k_j})\, P'_1 ... P'_m$ such that $Q_i \twoheadrightarrow_{basic} Q'_i$, $P_j \twoheadrightarrow_{basic} P'_j$ and $f_r \twoheadrightarrow_{basic} f'_r$ for $1 \leq i \leq k_j$, $1 \leq j \leq m$ and $1 \leq r \leq w$. By induction hypothesis $(f_j\, Q_1 ... Q_{k_j})\, P_1 ... P_m \in \mathsf{SN}(\sigma)$. So, we have $M_{n+1} \in \mathsf{SN}(\sigma)$ too. Hence ζ is finite.

2. In the second case, we have that all the expressions in ζ are of the form $\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ (c_j\, [\boldsymbol{\rho}]\, Q'_1 ... Q'_{k_j})\ \mathsf{of}\ \{c_1 \Rightarrow f'_1 \mid \ldots \mid c_w \Rightarrow f'_w\}\, P'_1 ... P'_m$ with $Q_i \twoheadrightarrow_{basic} Q'_i$, $P_s \twoheadrightarrow_{basic} P'_s$ and $f_r \twoheadrightarrow_{basic} f'_r$ for $1 \leq i \leq k$, $1 \leq s \leq m$ and $1 \leq r \leq w$. Since we have by induction hypothesis that $f_i \in \mathsf{SN}(\rho_1 \to \ldots \to \rho_{k_i} \to \tau)$ and $(f_j\, Q_1 ... Q_{k_j})\, P_1 ... P_m \in \mathsf{SN}(\sigma)$ for $1 \leq i \leq w$, we have that all expressions in ζ are strongly normalizing. Hence, ζ is finite.

Case (case2) If $M \equiv \mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ Q\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\}\, P_1 ... P_m$, with $Q \in \mathcal{SN}(d[\boldsymbol{\rho}])$, $Q \to_k Q'$, $\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ Q'\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\}\, P_1 ... P_m \in \mathcal{SN}(\sigma)$, $m \geq 0$, then there are two possibilities: the key-redex of Q is reduced in ζ or not.

1. In the first case, there is a rewrite step $M_n \to_{basic} M_{n+1}$ in the rewrite sequence ζ with $M_n \equiv \mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ Q\ \mathsf{of}\ \{c_1 \Rightarrow f'_1 \mid \ldots \mid c_k \Rightarrow f'_k\}\, P'_1 ... P'_m$ and $M_{n+1} \equiv \mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ Q'\ \mathsf{of}\ \{c_1 \Rightarrow f'_1 \mid \ldots \mid c_k \Rightarrow f'_k\}\, P'_1 ... P'_m$ such that $Q \to_k Q'$, $f_i \twoheadrightarrow_{basic} f'_i$ and $P_j \twoheadrightarrow_{basic} P'_j$ for $1 \leq i \leq k$ and $1 \leq j \leq m$. Since by induction hypothesis $\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ Q'\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\}\, P_1 ... P_m \in \mathsf{SN}(\sigma)$, then ζ is finite.

2. In the second case, we have that all the expressions in ζ are of the form $\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ Q''\ \mathsf{of}\ \{c_1 \Rightarrow f'_1 \mid \ldots \mid c_k \Rightarrow f'_k\}\, P'_1 ... P'_m$ with $f_i \twoheadrightarrow_{basic} f'_i$ for $1 \leq i \leq k$, $P_j \twoheadrightarrow_{basic} P'_j$ for $1 \leq j \leq m$, and $Q \twoheadrightarrow_{basic} Q'$ without contracting the key-redex of Q. This is only possible if $Q \equiv Q_1\, Q_2$ or $Q \equiv \mathsf{case}^{\gamma}_{d[\boldsymbol{\delta}]}\ Q_1\ \mathsf{of}\ \{c'_1 \Rightarrow e_1 \mid \ldots \mid c'_w \Rightarrow e_w\}$.

   - If $Q \equiv Q_1\, Q_2$, then every expression is of the form $\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ (Q_1\, Q''_2)\ \mathsf{of}\ \{c_1 \Rightarrow f'_1 \mid \ldots \mid c_k \Rightarrow f'_k\}\, P'_1 ... P'_m$ with $f_i \twoheadrightarrow_{basic} f'_i$ for $1 \leq i \leq k$, $P_j \twoheadrightarrow_{basic} P'_j$ for $1 \leq j \leq m$, and $Q_2 \twoheadrightarrow_{basic} Q''_2$. Since by induction hypothesis $\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ Q'\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\}\, P_1 ... P_m \in \mathsf{SN}(\sigma)$ and Q is strongly normalizing too, then ζ is finite.

- If $Q \equiv \mathsf{case}^{\psi}_{d[\boldsymbol{\delta}]}\ Q_1$ of $\{c'_1 \Rightarrow e_1 \mid \ldots \mid c'_w \Rightarrow e_w\}$, then all the expressions in ζ are of the form $(\mathsf{case}^{\gamma}_{d[\boldsymbol{\rho}]}\ (\mathsf{case}^{\psi}_{d[\boldsymbol{\delta}]}\ Q_1$ of $\{c'_1 \Rightarrow e_1 \mid \ldots \mid c'_w \Rightarrow e_w\})$ of $\{c_1 \Rightarrow f'_1 \mid \ldots \mid c_k \Rightarrow f'_k\}).l\, P'_1 ... P'_m$ with $P_r \twoheadrightarrow_{basic} P'_r$ for $1 \leq r \leq m$, $f_i \twoheadrightarrow_{basic} f'_i$ for $1 \leq i \leq k$, and $e_j \twoheadrightarrow_{basic} e'_j$ for $1 \leq j \leq w$. Since Q is strongly normalizing by induction hypothesis, e_j is strongly normalizing for $1 \leq j \leq w$. By induction hypothesis $\mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ Q'$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_k \Rightarrow f_k\}\, P_1 ... P_m \in \mathsf{SN}(\sigma)$. Hence ζ is finite.

Case (sub) If $M \in \mathcal{SN}(\sigma)$ because $M \in \mathcal{SN}(\tau)$ with $\tau \leq \sigma$, then by induction hypothesis M is strongly normalizing, and by subsumption $M : \sigma$. Hence $M \in \mathsf{SN}(\sigma)$.

□

The following result is useful for proving one of the closure properties of saturated sets.

Lemma 21. *If $t \in \mathsf{SN}(\sigma_1 \to ... \to \sigma_n \to \tau)$, $t \to_k u$ and $u\, \boldsymbol{a} \in \mathsf{SN}(\tau)$, then $t\, \boldsymbol{a} \in \mathsf{SN}(\tau)$.*

Proof. We will prove that $t \in \mathcal{SN}(\sigma_1 \to ... \to \sigma_n \to \tau)$, $t \to_k u$ and $u\, \boldsymbol{a} \in \mathcal{SN}(\tau)$, implies $t\, \boldsymbol{a} \in \mathcal{SN}(\tau)$. Then the result we want to prove follows directly from Proposition 10.

Suppose $t \in \mathcal{SN}(\sigma_1 \to ... \to \sigma_n \to \tau)$, $t \to_k u$ and $u\, \boldsymbol{a} \in \mathcal{SN}(\tau)$. By induction on the structure of t. Since $t \to_k u$, t can be of the following forms:

1. t is a redex.
   - If $t \equiv (\lambda x : \gamma.\ M)\, N$, then $u \equiv M\{x := N\}$. By assumption, $M\{x := N\}\, \boldsymbol{a} \in \mathsf{SN}(\tau)$. Since $t \in \mathcal{SN}(\sigma_1 \to ... \to \sigma_n \to \tau)$ then $N \in \mathcal{SN}(\gamma)$. Hence, by *(app)*, $t\, \boldsymbol{a} \in \mathcal{SN}(\tau)$.
   - If $t \equiv [\ldots, l = P, \ldots].l$, then $u \equiv P$. Since $t \in \mathcal{SN}(\sigma_1 \to ... \to \sigma_n \to \tau)$, $P_i \in \mathcal{SN}(\gamma_i)$. By assumption $P\, \boldsymbol{a} \in \mathcal{SN}(\tau)$. Hence, by *(sel1)*, $t\, \boldsymbol{a} \in \mathcal{SN}(\tau)$.
   - If $t \equiv \mathsf{case}^{\tau}_{d[\boldsymbol{\rho}]}\ (c_j[\boldsymbol{\rho}]\, Q_1 ... Q_{k_j})$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$, then $u \equiv (f_j\, Q_1 ... Q_{k_j})$. Since $t \in \mathcal{SN}(\sigma_1 \to ... \to \sigma_n \to \tau)$, $f_i \in \mathcal{SN}(\rho_1 \to ... \to \rho_{k_i} \to \tau)$. By assumption $(f_j\, Q_1 ... Q_{k_j})\, \boldsymbol{a} \in \mathcal{SN}(\tau)$. Hence, by *(case 1)*, $t\, \boldsymbol{a} \in \mathcal{SN}(\tau)$.
2. $t \equiv M\, P$. In this case $M \in \mathcal{SN}(\gamma)$, $M \to_k M'$, $M\, P \to_k M'\, P$, $u \equiv M'\, P$ and $M'\, P\, \boldsymbol{a} \in \mathcal{SN}(\tau)$. By induction hypothesis $M\, P\, \boldsymbol{a} \in \mathcal{SN}(\tau)$. Hence $t\, \boldsymbol{a} \in \mathcal{SN}(\tau)$.
3. $t \equiv M.l$. In this case $M \in \mathcal{SN}([\ldots, l : (\sigma_1 \to ... \to \sigma_n \to \tau), \ldots])$, $M.l \to_k M'.l$, $u \equiv M'.l$ and $(M'.l)\, \boldsymbol{a} \in \mathcal{SN}(\tau)$. Hence, by *(sel2)*, $t\, \boldsymbol{a} \in \mathcal{SN}(\tau)$.
4. $t \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ M$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$. In this case $M \in \mathcal{SN}(d[\boldsymbol{\rho}])$, $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ M$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\} \to_k \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ M'$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$, $u \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ M'$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$ and $(\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ M'$ of $\{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\})\, \boldsymbol{a} \in \mathcal{SN}(\tau)$. Hence, by *(case2)*, $t\, \boldsymbol{a} \in \mathcal{SN}(\tau)$.

□

Now we define saturated sets and state some of their closure properties.

Definition 28 (Saturated sets). *Let σ be a type. A set $X \subseteq \mathsf{SN}(\sigma)$ is a σ-*saturated set *if*

1. $\mathsf{Base}(\sigma) \subseteq X$
2. $a \in \mathsf{SN}(\sigma)\ \wedge\ a \to_k b\ \wedge\ b \in X\ \Rightarrow\ a \in X$

The collection of σ-saturated sets is denoted by $\mathsf{SAT}(\sigma)$

$$\mathsf{SAT}(\sigma) = \{X \subseteq \mathsf{SN}(\sigma) \mid X \text{ is } \sigma\text{-saturated }\}$$

Lemma 22.

1. *$\mathsf{SN}(\sigma) \in \mathsf{SAT}(\sigma)$*

2. *If $X \in \mathsf{SAT}(\sigma)$ and $Y \in \mathsf{SAT}(\tau)$, then $X \to Y \in \mathsf{SAT}(\sigma \to \tau)$.*

3. *Let $I \neq \emptyset$. If $X_i \in \mathsf{SAT}(\sigma)$ for all $i \in I$, then $\bigcap_{i\in I} X_i \in \mathsf{SAT}(\sigma)$.*

4. *If $X_i \in \mathsf{SAT}(\sigma_i)$ for $1 \leq i \leq n$, then $[l_1 : X_1, \ldots, l_n : X_n] \in \mathsf{SAT}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$.*

Proof.

1. Immediate, using Lemma 16.

2. Suppose $X \in \mathsf{SAT}(\sigma)$ and $Y \in \mathsf{SAT}(\tau)$. Let $F \in X \to Y$ and $x \in X$ then, $F\,x \in Y$. As $F\,x \in \mathsf{SN}(\tau)$ (because $Y \subseteq \mathsf{SN}(\tau)$) then, $F \in \mathsf{SN}(\sigma \to \tau)$. Hence, $X \to Y \subseteq \mathsf{SN}(\sigma \to \tau)$. Now we are going to prove that $X \to Y$ satisfies the two conditions of Definition 28.

   (a) Assume $b \in \mathsf{Base}(\sigma \to \tau)$ and $x \in X$. Then $x \in \mathsf{SN}(\sigma)$ and $b\,x \in \mathsf{Base}(\tau) \subseteq Y$. Hence $b \in X \to Y$ and so, $\mathsf{Base}(\sigma \to \tau) \subseteq X \to Y$.

   (b) Assume $t \in \mathsf{SN}(\sigma \to \tau)$, $t \to_k u$ and $u \in X \to Y$, to show $t \in X \to Y$, i.e., for every $x \in X$, $t\,x \in Y$. As we have $t \to_k u$, then $t\,x \to_k u\,x$. Since $t \in \mathsf{SN}(\sigma \to \tau)$ and $u\,x \in Y \subseteq \mathsf{SN}(\tau)$, then by Lemma 21 $t\,x \in \mathsf{SN}(\tau)$. Hence, as $Y \in \mathsf{SAT}(\tau)$, we have $t\,x \in Y$.

3. Suppose $X_i \in \mathsf{SAT}(\sigma)$ for all $i \in I$. As we have $X_i \subseteq \mathsf{SN}(\sigma)$ for all $i \in I$, obviously $\bigcap_{i\in I} X_i \in \mathsf{SN}(\sigma)$. Now we are going to prove that $\bigcap_{i\in I} X_i$ satisfies the two conditions of Definition 28.

   (a) We have $\mathsf{Base}(\sigma) \subseteq X_i$ for every $i \in I$. Therefore, $\mathsf{Base}(\sigma) \subseteq \bigcap_{i\in I} X_i$.

   (b) Assume $a \in \mathsf{SN}(\sigma)$, $a \to_k b$ and $b \in \bigcap_{i\in I} X_i$, to show $a \in \bigcap_{i\in I}$. As $b \in \bigcap_{i\in I} X_i$, then $b \in X_i$ for every $i \in I$. Since $X_i \in \mathsf{SAT}(\sigma)$ for all $i \in I$, we have $a \in X_i$ for every $i \in I$. Hence, $a \in \bigcap_{i\in I} X_i$.

4. Suppose $X_i \in \mathsf{SAT}(\sigma)$ for $1 \leq i \leq n$. As we have $X_i \subseteq \mathsf{SN}(\sigma)$ for $1 \leq i \leq n$ then every $a \in \mathcal{E}$, such that $a.l_i \in X_i$ $(1 \leq i \leq n)$, is strongly normalizing. Hence, $[l_1 : X_1, \ldots, l_n : X_n] \subseteq \mathsf{SN}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$. Now we are going to prove that $[l_1 : X_1, \ldots, l_n : X_n]$ satisfies the two conditions of Definition 28.

   (a) Assume $b \in \mathsf{Base}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$. Then $b.l_i \in \mathsf{Base}(\sigma_i)$ for $1 \leq i \leq n$. Since $X_i \in \mathsf{SAT}(\sigma_i)$ then, $\mathsf{Base}(\sigma_i) \subseteq X_i$. Thus, $b.l_i \in X_i$ for $1 \leq i \leq n$ and, therefore $b \in [l_1 : X_1, \ldots, l_n : X_n]$, by Definition 24. Hence, $\mathsf{Base}([l_1 : \sigma_1, \ldots, l_n : \sigma_n]) \subseteq [l_1 : X_1, \ldots, l_n : X_n]$.

   (b) Assume $a \in \mathsf{SN}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$, $a \to_k b$ and $b \in [l_1 : X_1, \ldots, l_n : X_n]$, to show $a \in [l_1 : X_1, \ldots, l_n : X_n]$ i.e., $a.l_i \in X_i$ for $1 \leq i \leq n$. As $b \in [l_1 : X_1, \ldots, l_n : X_n]$, then $b.l_i \in X_i$ for $1 \leq i \leq n$. By assumption $a \to_k b$, hence $a.l_i \to_k b.l_i$. By assumption $a \in \mathsf{SN}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$, then $a.l_i \in \mathsf{SN}(\sigma_i)$ for $1 \leq i \leq n$. Since $X_i \in \mathsf{SAT}(\sigma)$, $a.l_i \in X_i$ for $1 \leq i \leq n$. Therefore, $a \in [l_1 : X_1, \ldots, l_n : X_n]$.

□

We now turn to the definition of the interpretation function of types.

Definition 29 (Interpretation of types). *The interpretation of types $[\![.]\!]$ is defined as follows:*

$$
\begin{aligned}
[\![\sigma \to \tau]\!] &= [\![\sigma]\!] \to [\![\tau]\!] \\
[\![[l_1 : \sigma_1, \ldots, l_n : \sigma_n]]\!] &= [l_1 : [\![\sigma_1]\!], \ldots, l_n : [\![\sigma_n]\!]] \\
[\![d[\boldsymbol{\sigma}]]\!] &\quad \text{is defined inductively as follows:}
\end{aligned}
$$

1. $\mathsf{Base}(d[\boldsymbol{\sigma}]) \subseteq [\![d[\boldsymbol{\sigma}]]\!]$
2. $c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph \ \wedge \ b_i \in [\![\rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}]\!]$ *for* $1 \le i \le k \ \wedge \ \boldsymbol{\tau} \le \boldsymbol{\sigma} \ \Rightarrow \ c[\boldsymbol{\tau}]\, b_1 \ldots b_k \in [\![d[\boldsymbol{\sigma}]]\!]$
3. $M \in \mathsf{SN}(d[\boldsymbol{\sigma}]) \ \wedge \ M \to_k M' \ \wedge \ M' \in [\![d[\boldsymbol{\sigma}]]\!] \ \Rightarrow \ M \in [\![d[\boldsymbol{\sigma}]]\!]$

The folowing states that the interpretation of every type is a saturated set.

Lemma 23. *For every type* σ, $[\![\sigma]\!] \in \mathsf{SAT}(\sigma)$.

Proof. By induction on the generation of σ.

1. Suppose $\sigma \equiv \tau \to \rho$, then $[\![\tau \to \rho]\!] = [\![\tau]\!] \to [\![\rho]\!]$. By induction hypothesis $[\![\tau]\!] \in \mathsf{SAT}(\tau)$ and $[\![\rho]\!] \in \mathsf{SAT}(\rho)$. Thus, by Lemma 22, $[\![\tau]\!] \to [\![\rho]\!] \in \mathsf{SAT}(\tau \to \rho)$.
2. Suppose $\sigma \equiv [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$, then $[\![[l_1 : \sigma_1, \ldots, l_n : \sigma_n]]\!] = [l_1 : [\![\sigma_1]\!], \ldots, l_n : [\![\sigma_n]\!]]$. By induction hypothesis $[\![\sigma_i]\!] \in \mathsf{SAT}(\sigma_i)$ for $1 \le i \le n$. Therefore, by Lemma 22, $[\![[l_1 : \sigma_1, \ldots, l_n : \sigma_n]]\!] \in \mathsf{SAT}([l_1 : \sigma_1, \ldots, l_n : \sigma_n])$.
3. Suppose $\sigma \equiv d[\boldsymbol{\tau}]$. Then $[\![d[\boldsymbol{\tau}]]\!] \in \mathsf{SAT}(d[\boldsymbol{\tau}])$ follows directly from the definition of $[\![d[\boldsymbol{\tau}]]\!]$.

□

Next we state some auxiliary lemmas.

Lemma 24. *If* $d \le d'$ *then* $[\![d[\boldsymbol{\sigma}]]\!] \subseteq [\![d'[\boldsymbol{\sigma}]]\!]$.

Proof. Assume $d \le d'$ and $M \in [\![d[\boldsymbol{\sigma}]]\!]$. By induction on the derivation of $M \in [\![d[\boldsymbol{\sigma}]]\!]$.

1. Suppose $M \in \mathsf{Base}(d[\boldsymbol{\sigma}])$. Since $d \le d'$ we have $d[\boldsymbol{\sigma}] \le d'[\boldsymbol{\sigma}]$. Then, by Lemma 13, $M \in \mathsf{Base}(d'[\boldsymbol{\sigma}])$. Hence, $M \in [\![d'[\boldsymbol{\sigma}]]\!]$.
2. Suppose $M \equiv c[\boldsymbol{\tau}]b_1...b_k$, with $c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph$, $b_i \in [\![\rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}]\!]$ for $1 \le i \le k$ and $\boldsymbol{\tau} \le \boldsymbol{\sigma}$. Since $d \le d'$, we have $c : \rho_1 \to \ldots \to \rho_k \to d'[\boldsymbol{\alpha}] \in \aleph$. Hence, by definition, $M \in [\![d'[\boldsymbol{\sigma}]]\!]$.
3. Suppose $M \in \mathsf{SN}(d[\boldsymbol{\sigma}])$, $M \to_k M'$ and $M' \in [\![d[\boldsymbol{\sigma}]]\!]$. By induction hypothesis $M' \in [\![d'[\boldsymbol{\sigma}]]\!]$. Since $d[\boldsymbol{\sigma}] \le d'[\boldsymbol{\sigma}]$ we have $M \in \mathsf{SN}(d'[\boldsymbol{\sigma}])$. Hence, $M \in [\![d'[\boldsymbol{\sigma}]]\!]$.

□

Lemma 25. *If* $\boldsymbol{\tau} \le \boldsymbol{\sigma}$ *then* $[\![d[\boldsymbol{\tau}]]\!] \subseteq [\![d[\boldsymbol{\sigma}]]\!]$.

Proof. Assume $\boldsymbol{\tau} \le \boldsymbol{\sigma}$ and $M \in [\![d[\boldsymbol{\tau}]]\!]$. By induction on the derivation of $M \in [\![d[\boldsymbol{\tau}]]\!]$ we will prove that $M \in [\![d[\boldsymbol{\sigma}]]\!]$.

1. Suppose $M \in \mathsf{Base}(d[\boldsymbol{\tau}])$. Since $\boldsymbol{\tau} \le \boldsymbol{\sigma}$ we have $d[\boldsymbol{\tau}] \le d[\boldsymbol{\sigma}]$. Then, by Lemma 13, $M \in \mathsf{Base}(d[\boldsymbol{\sigma}])$. Hence, $M \in [\![d[\boldsymbol{\sigma}]]\!]$.
2. Suppose $M \equiv c[\boldsymbol{\gamma}]b_1...b_k$, with $c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph$, $b_i \in [\![\rho_i\{\boldsymbol{\alpha} := \boldsymbol{\gamma}\}]\!]$ for $1 \le i \le k$ and $\boldsymbol{\gamma} \le \boldsymbol{\tau}$. By transitivity, $\boldsymbol{\gamma} \le \boldsymbol{\sigma}$. Hence, $M \in [\![d[\boldsymbol{\sigma}]]\!]$.
3. Suppose $M \in \mathsf{SN}(d[\boldsymbol{\tau}])$, $M \to_k M'$ and $M' \in [\![d[\boldsymbol{\tau}]]\!]$. By induction hypothesis $M' \in [\![d[\boldsymbol{\sigma}]]\!]$. Since $d[\boldsymbol{\tau}] \le d[\boldsymbol{\sigma}]$ we have $M \in \mathsf{SN}(d[\boldsymbol{\sigma}])$. Hence, $M \in [\![d[\boldsymbol{\sigma}]]\!]$.

□

Lemma 26. *If* $\tau \le \sigma$ *then* $[\![\tau]\!] \subseteq [\![\sigma]\!]$.

Proof. By induction on the derivation $\tau \le \sigma$. By case analysis on the last rule applied.

Case ($\le_{\mathsf{refl}}$). In this case $\tau \equiv \sigma$ so, $[\![\tau]\!] = [\![\sigma]\!]$.

Case ($\le_{\to}$). In this case $\tau \equiv \tau_1 \to \tau_2$ and $\sigma \equiv \sigma_1 \to \sigma_2$, with $\sigma_1 \le \tau_1$ and $\tau_2 \le \sigma_2$. Assume $t \in [\![\tau]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!]$ to show that $t \in [\![\sigma]\!] = [\![\sigma_1]\!] \to [\![\sigma_2]\!]$ i.e., for every $a \in [\![\sigma_1]\!]$, $t\,a \in [\![\sigma_2]\!]$.
Let $a \in [\![\sigma_1]\!]$. By induction hypothesis $a \in [\![\tau_1]\!]$ so, $t\,a \in [\![\tau_2]\!]$. Hence, by induction hypothesis, $t\,a \in [\![\sigma_2]\!]$. Therefore, $t \in [\![\sigma]\!]$.

Case ($\leq_{[]}$). In this case $\tau \equiv [l_1 : \tau_1, \ldots, l_{n+m} : \tau_{n+m}]$ and $\sigma \equiv [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$, with $\tau_i \leq \sigma_i$ for $1 \leq i \leq n$. Assume $t \in [\![\tau]\!] = [l_1 : [\![\tau_1]\!], \ldots, l_{n+m} : [\![\tau_{n+m}]\!]]$ to show that $t \in [\![\sigma]\!] = [l_1 : [\![\sigma_1]\!], \ldots, l_n : [\![\sigma_n]\!]]$ i.e., $t.l_i \in [\![\sigma_i]\!]$ for $i \leq i \leq n$.
By assumption $t.l_i \in [\![\tau_i]\!]$ for $1 \leq i \leq n+m$. As $\tau_i \leq \sigma_i$ for $1 \leq i \leq n$, by induction hypothesis we have $t.l_i \in [\![\sigma_i]\!]$ for $i \leq i \leq n$. Hence, $t \in [\![\sigma]\!]$.

Case ($\leq_{\mathsf{data}}$). In this case $\tau \equiv d[\boldsymbol{\tau}]$, $\sigma \equiv d'[\boldsymbol{\sigma}]$ and $\tau_i \leq \sigma_i$, with $1 \leq i \leq \mathsf{ar}(d)$ and $d \leq d'$. Assume $t \in [\![d[\boldsymbol{\tau}]]\!]$ to show that $t \in [\![d'[\boldsymbol{\sigma}]]\!]$. By case analysis.

1. Suppose $t \in \mathsf{Base}(d[\boldsymbol{\tau}])$. By Lemma 13, $t \in [\![d'[\boldsymbol{\sigma}]]\!]$.

2. Suppose $t \equiv c\,[\boldsymbol{\tau}]\,b_1 \ldots b_k$ with $c : \rho_1 \to \ldots \to \rho_k \to d\,[\boldsymbol{\alpha}] \in \aleph$ and $b_j \in [\![\rho_j\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}]\!]$ for $1 \leq j \leq k$. As $d \leq d'$ then there is another declaration $c : \rho_1 \to \ldots \to_k \to d'[\boldsymbol{\alpha}] \in \aleph$. As $\tau_i \leq \sigma_i$ for $1 \leq i \leq \mathsf{ar}(d)$ then, by Lemma 2, $\rho_j\{\boldsymbol{\alpha} := \boldsymbol{\tau}\} \leq \rho_j\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}$ for $1 \leq j \leq k$. Thus, by induction hypothesis, $b_j \in [\![\rho_j\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}]\!]$. Hence, $c[\boldsymbol{\tau}]b_1 ... b_k \in [\![d'[\boldsymbol{\sigma}]]\!]$.

3. Suppose $t \in \mathsf{SN}(d[\boldsymbol{\sigma}])$, $t \to_k t'$ and $t' \in [\![d[\boldsymbol{\sigma}]]\!]$. We have $t \in [\![d[\boldsymbol{\tau}]]\!]$, the, by Lemma 24, $t \in [\![d'[\boldsymbol{\tau}]]\!]$. Thus, by Lemma 25, $t \in [\![d'[\boldsymbol{\sigma}]]\!]$.

□

We now turn to the interpretation of terms.

Definition 30 (Interpretation of expressions).

1. *A* valuation *in $\mathcal{E}$ is a map $\rho : \mathcal{V} \to \mathcal{E}$ such that $\rho(x) \in [\![\sigma]\!]$ whenever $x \in \mathcal{V}^\sigma$.*

2. *For every valuation ρ, $e \in \mathcal{E}$ and $x \in \mathcal{V}$, the valuation $\rho(x := e)$ is defined as follows:*

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

3. *Let ρ be a valuation in $\mathcal{E}$ and $M \in \mathcal{E}$. Then*

$$(\![M]\!)_\rho = M\{\boldsymbol{x} := \rho(\boldsymbol{x})\}$$

   where $\boldsymbol{x}$ is the set of free variables in M.

4. *Let ρ be a valuation in $\mathcal{E}$. Then ρ* satisfies *$M : \sigma$, written $\rho \vDash M : \sigma$, if $(\![M]\!)_\rho \in [\![\sigma]\!]$.*

5. *Let ρ be a valuation in $\mathcal{E}$ and Γ a context. Then ρ* satisfies *Γ, written $\rho \vDash \Gamma$, if $\rho \vDash x : \sigma$ for every $x : \sigma \in \Gamma$.*

6. *A context Γ* satisfies *$M : \sigma$, written $\Gamma \vDash M : \sigma$, if for every valuation, ρ, in $\mathcal{E}$*

$$\rho \vDash \Gamma \quad \Rightarrow \quad \rho \vDash M : \sigma$$

We are now ready to prove the soundness of the interpretation.

Proposition 11 (Soundness).

$$\Gamma \vdash M : \sigma \quad \Rightarrow \quad \Gamma \vDash M : \sigma$$

Proof. By induction on the derivation of $\Gamma \vdash M : \sigma$

Case (start). In this case $M \equiv x$ and $x : \sigma \in \Gamma$. Then, trivially $\Gamma \vDash x : \sigma$.

Case (application). In this case $M \equiv a\,b$ and $\Gamma \vdash a\,b : \sigma$ is a direct consequence of $\Gamma \vdash a : \tau \to \sigma$ and $\Gamma \vdash b : \tau$. Assume $\rho \models \Gamma$ in order to show $\rho \models a\,b : \sigma$. Then, by induction hypothesis $\rho \models a : \tau \to \sigma$ and $\rho \models b : \tau$. That is, $(\!|a|\!)_\rho \in [\![\tau \to \sigma]\!] = [\![\tau]\!] \to [\![\sigma]\!]$ and $(\!|b|\!)_\rho \in [\![\tau]\!]$. Thus, $(\!|a\,b|\!)_\rho = (\!|a|\!)_\rho (\!|b|\!)_\rho \in [\![\sigma]\!]$, that is, $\rho \models a\,b : \sigma$. Hence, $\Gamma \models a\,b : \sigma$.

Case (abstraction). In this case $M \equiv \lambda x{:}\tau.\ a$, $\sigma \equiv \tau \to \gamma$ and $\Gamma \vdash \lambda x{:}\tau.\ a : \tau \to \gamma$ comes directly from $\Gamma, x : \tau \vdash a : \gamma$. By induction hypothesis we have

$$\Gamma, x : \tau \models a : \gamma \tag{7}$$

Suppose $\rho \models \Gamma$ in order to show $\rho \models \lambda x{:}\tau.\ a : \tau \to \gamma$. That is, we want to show

$$(\!|\lambda x{:}\tau.\ a|\!)_\rho\ N \in [\![\gamma]\!] \quad ,\ \text{for all } N \in [\![\tau]\!].$$

So, assume that $N \in [\![\tau]\!]$. Then, $\rho(x := N) \models \Gamma, x : \tau$ and hence, by (7),

$$(\!|a|\!)_{\rho(x:=N)} \in [\![\gamma]\!]$$

Since, $(\!|\lambda x{:}\tau.\ a|\!)_\rho\ N = (\lambda x{:}\tau.\ a)\{\boldsymbol{y} := \rho(\boldsymbol{y})\}N \to_k a\{\boldsymbol{y} := \rho(\boldsymbol{y})\}\{x := N\} = (\!|a|\!)_{\rho(x:=N)}$, and $[\![\gamma]\!]$ is saturated (see Lemma 23), we have $(\!|\lambda x{:}\tau.\ a|\!)_\rho \in (\!|a|\!)$.

Case (record). In this case $M \equiv [l_1 = a_1, \ldots, l_n = a_n]$, $\sigma \equiv [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$ and $\Gamma \vdash [l_1 = a_1, \ldots, l_n = a_n] : [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$ is a direct consequence of $\Gamma \vdash a_i : \sigma_i$ for $1 \le i \le n$.
Suppose $\rho \models \Gamma$ in order to show $\rho \models [l_1 = a_1, \ldots, l_n = a_n] : [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$. That is, we want to show

$$(\!|[l_1 = a_1, \ldots, l_n = a_n]|\!)_\rho \in [l_1 : [\![\sigma_1]\!], \ldots, l_n : [\![\sigma_n]\!]]$$

By induction hypothesis we have $\Gamma \models a_i : \sigma_i$ for $1 \le i \le n$ so, $(\!|a_i|\!)_\rho \in [\![\sigma_i]\!]$ for $1 \le i \le n$. Thus,

$$(\!|[l_1 = a_1, \ldots, l_n = a_n]|\!)_\rho = [l_1 = (\!|a_1|\!)_\rho, \ldots, l_n = (\!|a_n|\!)_\rho] \in [l_1 : [\![\sigma_1]\!], \ldots, l_n : [\![\sigma_n]\!]]$$

Hence, $\Gamma \models [l_1 = a_1, \ldots, l_n = a_n] : [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$.

Case (select). In this case $M \equiv a.l$ and $\Gamma \vdash a.l : \sigma$ comes directly from $\Gamma \vdash a : [l_1 : \sigma_1, \ldots, l : \sigma, \ldots, l_n : \sigma_n]$.
Suppose $\rho \models \Gamma$ in order to show $\rho \models a.l : \sigma$. That is, we want to show $(\!|a.l|\!)_\rho \in [\![\sigma]\!]$. By induction hypothesis $\Gamma \models a : [l_1 : \sigma_1, \ldots, l_\sigma, \ldots, l_n : \sigma_n]$, that is

$$(\!|a|\!)_\rho \in [l_1 : [\![\sigma_1]\!], \ldots, l : [\![\sigma]\!], \ldots, l_n : [\![\sigma_n]\!]]$$

Thus,

$$(\!|a.l|\!)_\rho = (\!|a|\!)_\rho.l \in [\![\sigma]\!]$$

Hence, $\Gamma \models a.l : \sigma$.

Case (constructor). In this case $M \equiv c[\boldsymbol{\tau}]\,b_1 ... b_k$ with $c : \rho_1 \to ... \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph$ is a direct consequence of $\Gamma \vdash b_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}$ for $1 \le i \le k$. Assume $\rho \models \Gamma$ in order to show $\rho \models c[\boldsymbol{\tau}]\,b_1 ... b_k : d[\boldsymbol{\tau}]$. By induction hypothesis $\rho \models b_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}$ for $1 \le i \le k$, that is

$$(\!|b_i|\!)_\rho \in [\![\rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\}]\!] \ ,\quad \text{for } 1 \le i \le k$$

Thus,

$$(\!|c\,[\boldsymbol{\tau}]\,b_1 ... b_k|\!)_\rho = c\,[\boldsymbol{\tau}]\,(\!|b_1|\!)_\rho \ldots (\!|b_k|\!)_\rho \in [\![d[\boldsymbol{\tau}]]\!]$$

Hence, $\Gamma \models c[\boldsymbol{\tau}]\,b_1 ... b_k : d[\boldsymbol{\tau}]$.

Case (case). In this case $M \equiv \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}$ is a direct consequence of $\Gamma \vdash a : d[\boldsymbol{\rho}]$ and $\Gamma \vdash f_i : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma}$ for $1 \leq i \leq n$ with $c_i : \tau_i$. Assume $\rho \vDash \Gamma$ in order to show $\rho \vDash \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\} : \sigma$. By induction hypothesis $\rho \vDash a : d[\boldsymbol{\rho}]$ and $\rho \vDash f_i : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma}$ for $1 \leq i \leq n$. That is,

$$(\!|a|\!)_\rho \in [\![d[\boldsymbol{\rho}]]\!] \quad \text{and} \quad (\!|f_i|\!)_\rho \in [\![(\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma}]\!] \;\text{ for }\; 1 \leq i \leq n$$

We know that

$$(\!|\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}|\!)_\rho = \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ (\!|a|\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\}$$

Now we will prove that $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ N\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\} \in [\![\sigma]\!]$ for every $N \in [\![d[\boldsymbol{\rho}]]\!]$, by induction on the derivation of $N \in [\![d[\boldsymbol{\rho}]]\!]$.

1. If $N \in \mathsf{Base}(d[\boldsymbol{\rho}])$ then, since $(\!|f_i|\!)_\rho \in \mathsf{SN}((\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma})$ for $1 \leq i \leq n$, we have $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ N\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\} \in \mathsf{Base}(\sigma) \subseteq [\![\sigma]\!]$ because $[\![\sigma]\!]$ is a σ-saturated set.

2. If $N \equiv c_j\,[\boldsymbol{\tau}]\, b_1 ... b_{k_j}$ with $c_j : \rho_1 \to ... \to \rho_{k_j} \to d[\boldsymbol{\alpha}] \in \aleph$ and $\boldsymbol{\tau} \leq \boldsymbol{\rho}$, then

$$\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ N\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\} \to_k (\!|f_j|\!)_\rho\, b_1 ... b_{k_j}$$

   Since $(\!|f_j|\!)_\rho\, b_1 ... b_{k_j} \in [\![\sigma]\!]$ and $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ N\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\} \in \mathsf{SN}(\sigma)$ (because of *(case1)* rule), we have $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ M\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\} \in [\![\sigma]\!]$ because $[\![\sigma]\!]$ is a σ-saturated set.

3. If $N \in [\![d[\boldsymbol{\rho}]]\!]$ because $N \in \mathsf{SN}(d[\boldsymbol{\rho}])$, $N \to_k N'$ and $N' \in [\![d[\boldsymbol{\rho}]]\!]$, then $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ N\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\} \to_k \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ N'\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\}$. By induction hypothesis $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ N'\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\} \in [\![\sigma]\!]$. Since $[\![\sigma]\!]$ is saturated, it is enought to show $\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ N\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\} \in \mathsf{SN}(\sigma)$, which follows from Proposition 10.

Thus,

$$\mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ (\!|a|\!)_\rho\ \mathsf{of}\ \{c_1 \Rightarrow (\!|f_1|\!)_\rho \mid \ldots \mid c_n \Rightarrow (\!|f_n|\!)_\rho\} \in [\![\sigma]\!]$$

Hence, $\Gamma \vDash \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\} : \sigma$.

Case (subsumption). In this case $\Gamma \vdash M : \sigma$ is a direct consequence of $\Gamma \vdash M : \tau$ with $\tau \leq \sigma$. By induction hypothesis we have $\Gamma \vDash M : \tau$. Suppose $\rho \vDash \Gamma$ then $\rho \vDash M : \tau$ that is. $(\!|M|\!)_\rho \in [\![\tau]\!]$. Thus,

$$(\!|M|\!)_\rho \in [\![\sigma]\!]$$

follows using Lemma 26. Hence, $\Gamma \vDash M : \sigma$.

□

We arrive now to the main result of this subsection.

Proposition 12 (Strong normalization). $\to_{basic}$ *is strongly normalizing on typable expressions:*

$$\Gamma \vdash e : \sigma \quad \Rightarrow \quad e \in \mathsf{SN}(\to_{basic})$$

Proof. Suppose $\Gamma \vdash e : \sigma$. Then $\Gamma \vDash e : \sigma$. Define $\rho_0(x) = x$, for all $x \in \mathcal{V}$. Then, for every $x : \tau \in \Gamma$, $\rho_0 \vDash x : \tau$ because $(\!|x|\!)_{\rho_0} \in [\![\tau]\!]$, since $[\![\tau]\!]$ is saturated. So, $\rho_0 \vDash \Gamma$ and therefore, by Proposition 11, $\rho_0 \vDash e : \sigma$. Hence,

$$e \equiv (\!|e|\!)_{\rho_0} \in [\![\sigma]\!] \in \mathsf{SAT}(\sigma)$$

Thus, by the definition of $\mathsf{SAT}(\sigma)$, $e \in \mathsf{SN}(\sigma) \subseteq \mathsf{SN}(\to_{basic})$. □

As you may see the proof of strong normalization is totally supported by the soundness of interpretation.

5 Adding definitions

In this section, we study an extension of $\lambda_{\rightarrow,[],\mathsf{data}}$ with definitions, named $\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def}}$, and show the resulting calculus preserves the good meta-theoretical properties of $\lambda_{\rightarrow,[],\mathsf{data}}$.

A definition $x = a : \tau$ introduces x as an abbreviation of the term a of type τ. Definitions will be allowed both in contexts, e.g. $\Gamma, x = a : \tau, \ldots$, and in terms $\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b$. Definitions in contexts are called *global* definitions, and definitions in terms are called *local* definitions. We introduce also a new reduction: the *δ-reduction*, for unfolding definitions.

Next we extend the set $\mathcal{E}$ of terms to include local definitions.

Definition 31 (Expressions). *The set of* expressions $\mathcal{E}_\delta$ *is the extension of* $\mathcal{E}$ *with the clause*

$$\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b$$

Definition 32. *Let* $M \in \mathcal{E}_\delta$. *The set of* free variables *of* M, *denoted by* $\mathsf{FV}(M)$, *is defined by extending the Definition 11 with the following clause:*

$$\mathsf{FV}(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) = \mathsf{FV}(a) \cup (\mathsf{FV}(b) \backslash \{x\})$$

Like λ-abstractions, definitions introduce bound variables. Therefore we can use α-conversion when necessary. In $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b)$, x is a bound variable.

Next we extend the notion of context to include global definitions.

Definition 33. *The set* $\mathcal{C}_\delta$ *of* contexts *is defined inductively as follows:*

1. $<> \in \mathcal{C}_\delta$;
2. $< \Gamma, x : \tau > \in \mathcal{C}_\delta$ *if* $\Gamma \in \mathcal{C}_\delta, x \in \mathcal{V}, \tau \in \mathcal{T}_\aleph$ *and* x *is* Γ*-fresh;*
3. $< \Gamma, x = a : \tau > \in \mathcal{C}_\delta$ *if* $\Gamma \in \mathcal{C}_\delta, x \in \mathcal{V}, a \in \mathcal{E}_\delta, \tau \in \mathcal{T}_\aleph, x$ *is* Γ*-fresh and* $x \notin \mathsf{FV}(a)$.

In the third clause, we require that $x \notin \mathsf{FV}(a)$ in order that definitions are not recursive. It will be used the following convention: the expression $\Gamma, x{:}\tau$ stands for the context $< \Gamma, x{:}\tau >$, and $\Gamma, x = a{:}\tau$ stands for $< \Gamma, x = a{:}\tau >$.

Next we define a mapping dom that gives the set of variables declared in a context.

Definition 34. *The mapping* $\mathsf{dom} : \mathcal{C}_\delta \rightarrow \mathcal{P}(\mathcal{V})$ *is defined as follows:*

$$\begin{aligned} \mathsf{dom}(<>) &= \emptyset \\ \mathsf{dom}(\Gamma, x : \tau) &= \mathsf{dom}(\Gamma) \cup \{x\} \\ \mathsf{dom}(\Gamma, x = a : \tau) &= \mathsf{dom}(\Gamma) \cup \{x\} \end{aligned}$$

Definition 35 (Γ-fresh). *Let* x *be a variable.*

1. x *is* $<>$*-fresh*
2. x *is* Γ*-fresh* $\wedge\ x \neq y\ \Rightarrow\ x$ *is* $< \Gamma, y{:}\sigma >$*-fresh*
3. x *is* Γ*-fresh* $\wedge\ x \neq y\ \wedge\ x \neq \mathsf{FV}(a)\ \Rightarrow\ x$ *is* $< \Gamma, y = a{:}\sigma >$*-fresh*

Substitution is extended to terms of $\mathcal{E}_\delta$.

Definition 36. *Let* $M, N \in \mathcal{E}_\delta$ *and* $x \in \mathcal{V}$. *The* substitution of N for x in M, *denoted by* $M\{x := N\}$, *is defined by extending the Definition 12 with the following clauses:*

$$\begin{aligned} (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b)\{x := N\} &\equiv (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \\ (\mathsf{let}\ y = a : \tau\ \mathsf{in}\ b)\{x := N\} &\equiv (\mathsf{let}\ y = a\{x := N\} : \tau\ \mathsf{in}\ b\{x := N\}) \end{aligned}$$

By the variable convention, in the second clause of the previous definition the variable y doesn't occur free in the term N, and $x \neq y$.

Definition 37. *The result of substituting N for (the free occurrences of) a variable x in Γ such that $x \notin \mathsf{dom}(\Gamma)$ is denoted as $\Gamma\{x := N\}$ and is defined as follows:*

$$\begin{aligned} <>\{x := N\} &\equiv <> \\ <\Gamma, y : \tau>\{x := N\} &\equiv <\Gamma\{x := N\}, y : \tau> \\ <\Gamma, y = a : \tau>\{x := N\} &\equiv <\Gamma\{x := N\}, y = a\{x := N\} : \tau> \end{aligned}$$

Next we define a mapping $[.]_{\mathsf{def}}$ that gives the sequence of definitions being in a context.

Definition 38. *Let $\Gamma \in \mathcal{C}_\delta$. We define $[\Gamma]_{\mathsf{def}}$ inductively as follows:*

1. $[<>]_{\mathsf{def}} = <>$
2. $[\Gamma, x : \tau]_{\mathsf{def}} = [\Gamma]_{\mathsf{def}}$
3. $[\Gamma, x = a : \tau]_{\mathsf{def}} = [\Gamma]_{\mathsf{def}}, x = a : \tau$

We let $[\mathcal{C}_\delta]_{\mathsf{def}} = \{[\Gamma]_{\mathsf{def}} \mid \Gamma \in \mathcal{C}_\delta\}$.

the intended meaning of a definition ($\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b$) is that x can be substituted by a in the expression b. So, the expression ($\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b$) can be considered as having a similar behavior than $(\lambda x{:}\tau.\ b)\, a$. In order to perform the unfolding of a definition, let's introduce a relation called δ-reduction.

Definition 39 (Reductions). *Let Δ be a list of definitions, $\Delta \equiv x_1 = a_1 : \tau_1, \ldots, x_n = a_n : \tau_n$.*

1. *The δ-reduction, $\to_{\delta(\Delta)}$, is defined as the compatible closure of the rules:*

$$\begin{aligned} x &\to_{\delta(\Delta)} a \qquad , \textit{if } x = a{:}\tau \in \Delta \\ (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) &\to_{\delta(\Delta)} b\{x := a\} \end{aligned}$$

2. *We let $\to_{basic+\delta(\Delta)}$ denote $\to_{basic} \cup \to_{\delta(\Delta)}$.*

Let $\to_?$ be a binary relation. We let $\to_?^{\equiv}$ denote the reflexive closure of $\to_?$, $\to_?^{+}$ denote the transitive closure of $\to_?$ and $\twoheadrightarrow_?$ denote the reflexive-transitive closure of $\to_?$. The relation $=_?$ is the equivalence relation generated by $\twoheadrightarrow_?$.

Definition 40 (Typing).

1. *A* judgment *is a triple of the form $\Gamma \vdash_{cs_\delta} a : \tau$, where $\Gamma \in \mathcal{C}_\delta$, $a \in \mathcal{E}_\delta$ and $\tau \in \mathcal{T}_{\aleph}$. Generally, the subscript cs_δ of $\vdash$ is dropped.*
2. *A judgment is* derivable *if it can be inferred from the set of typing rules that result of adding the following rules to the rules of Figure 9 (Definition 14).*

$$(\text{global}) \qquad \frac{\Gamma \vdash a : \tau}{\Gamma, x = a : \tau \vdash x : \tau} \quad , \text{ if } x \text{ is } \Gamma\text{-fresh}$$

$$(\text{local}) \qquad \frac{\Gamma \vdash a : \tau \qquad \Gamma, x : \tau \vdash b : \sigma}{\Gamma \vdash (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) : \sigma}$$

3. *An expression $a \in \mathcal{E}_\delta$ is* typable *if $\Gamma \vdash a : \sigma$ for some $\Gamma \in \mathcal{C}_\delta$ and type σ.*

5.1 Basic properties

Here we state some lemmas that are useful in the proof of some results given below.

The following lemma show that the δ-reduction step remains invariant if we enlarge the sequences of definitions.

Lemma 27. *Let* $<\Delta_1, \Delta_2, \Delta_3> \in [\mathcal{C}_\delta]_{\mathsf{def}}$ *and* $e, e' \in \mathcal{E}_\delta$. *Then*

$$e \ \rightarrow_{\delta(\Delta_1,\Delta_3)} \ e' \quad \Rightarrow \quad e \ \rightarrow_{\delta(\Delta_1,\Delta_2,\Delta_3)} \ e'$$

Proof. By induction on the definition of $\rightarrow_{\delta(\Delta)}$. □

Lemma 28. *Let* $a, b, e \in \mathcal{E}_\delta$. *Suppose* $x \neq y$ *and* $x \notin \mathsf{FV}(e)$. *Then*

$$a\{x := b\}\{y := e\} \ \equiv \ a\{y := e\}\{x := b\{y := e\}\}$$

Proof. By induction on the structure of a. □

The following lemma shows that *basic* is substitutive.

Lemma 29. *Let* $a, b, b' \in \mathcal{E}_\delta$.

$$a \ \rightarrow_{basic} \ a' \quad \Rightarrow \quad a\{x := b\} \ \rightarrow_{basic} \ a'\{x := b\}$$

Proof. By induction on the generation of $\rightarrow_{basic}$. □

Lemma 30. *Let* $a, b, b' \in \mathcal{E}_\delta$.

$$b \ \rightarrow_{basic} \ b' \quad \Rightarrow \quad a\{x := b\} \ \rightarrow_{basic} \ a\{x := b'\}$$

Proof. By induction on the structure of a. □

5.2 Subject reduction

In order to prove that subject reduction holds for $\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def}}$, let us give first the following lemmas.

Lemma 31 (Basis). *Let* $x \in \mathcal{V}, \Gamma \in \mathcal{C}_\delta, e, a \in \mathcal{E}_\delta$ *and* $\sigma, \tau \in \mathcal{T}_\aleph$.

1. *If* x *is* Γ*-fresh, then*
$$\Gamma \ \vdash \ e : \sigma \quad \Rightarrow \quad \Gamma, x : \tau \ \vdash \ e : \sigma$$
2. *If* x *is* Γ*-fresh and* $x \notin \mathsf{FV}(a)$*, then*
$$\Gamma \ \vdash \ e : \sigma \quad \Rightarrow \quad \Gamma, x = a : \tau \ \vdash \ e : \sigma$$

Proof. By induction on the derivation of $\Gamma \ \vdash \ e : \sigma$. □

Lemma 32 (Generation). *All the results presented in Lemma 6 for system* $\lambda_{\rightarrow,[],\mathsf{data}}$ *are valid in* $\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def}}$. *In* $\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def}}$ *we extend Lemma 6 with the following result:*

8. If $\Gamma \ \vdash \ (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) : T$*, then* $\Gamma \ \vdash \ a : \rho$ *and* $\Gamma, x : \tau \ \vdash \ b : T$ *with* $\rho \leq \tau$.

Proof. By inspection on th derivation of $\Gamma \ \vdash \ (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) : T$. □

Lemma 33. *If* $\Gamma_1 \ \vdash \ a : \tau$ *and* $\Gamma_1, x : \tau, \Gamma_2 \ \vdash \ b : \sigma$*, then* $\Gamma_1, x = a : \tau, \Gamma_2 \ \vdash \ b : \sigma$.

Proof. By induction on the derivation of $\Gamma_1, x : \tau, \Gamma_2 \ \vdash \ b : \sigma$. □

Lemma 34 (Substitution).

1. *If* $\Gamma_1, x = a : \tau, \Gamma_2 \ \vdash \ b : \sigma$*, then* $\Gamma_1, \Gamma_2\{x := a\} \ \vdash \ b\{x := a\} : \sigma$.
2. *If* $\Gamma_1 \ \vdash \ a : \tau$ *and* $\Gamma_1, x : \tau, \Gamma_2 \ \vdash \ b : \sigma$*, then* $\Gamma_1, \Gamma_2\{x := a\} \ \vdash \ b\{x := a\} : \sigma$.

Proof.

1. By induction on the derivation of $\Gamma_1, x = a : \tau, \Gamma_2 \ \vdash \ b : \sigma$.

2. Using Lemma 33 and part 1.

□

We arrive now to the proof of subject reduction.

Proposition 13 (Subject reduction). *Typing is closed under* $\rightarrow_{basic+\delta}$*:*

$$\Gamma \vdash e : \sigma \quad \wedge \quad e \rightarrow_{basic+\delta([\Gamma]_{\mathsf{def}})} e' \quad \Rightarrow \quad \Gamma \vdash e' : \sigma$$

Proof. By induction on the derivation of $\Gamma \vdash e : \sigma$. Here we only give the proof for some cases. Let $\Delta \equiv [\Gamma]_{\mathsf{def}}$.

(application) $\dfrac{\Gamma \vdash a : \tau \rightarrow \sigma \quad \Gamma \vdash b : \tau}{\Gamma \vdash a\, b : \sigma}$ There are three possibilities:

- $(a\, b) \rightarrow_{basic+\delta(\Delta)} (a'\, b)$ under the hypothesis $a \rightarrow_{basic+\delta(\Delta)} a'$. By induction hypothesis it follows that $\Gamma \vdash a' : \tau \rightarrow \sigma$. Hence, $\Gamma \vdash a'\, b : \sigma$ follows from (application).

- $(a\, b) \rightarrow_{basic+\delta(\Delta)} (a\, b')$ under the hypothesis $b \rightarrow_{basic+\delta(\Delta)} b'$. By induction hypothesis it follows that $\Gamma \vdash b' : \tau$. Hence, $\Gamma \vdash a'\, b : \sigma$ follows from (application).

- Assume $a \equiv (\lambda x{:}\rho. a_1)$ and $(\lambda x{:}\rho. a_1) \rightarrow_{basic+\delta(\Delta)} a_1\{x := b\}$. By Lemma 31 we have $\Gamma, x : \rho \vdash a_1 : \sigma$ with $\tau \leq \rho$. By (subsumption) $\Gamma \vdash b : \rho$, then $\Gamma \vdash a_1\{x := b\} : \sigma$ follows using Lemma 34 part 2.

(select) $\dfrac{\Gamma \vdash a : [l_1 : \tau_1, \ldots, l_n : \tau_n]}{\Gamma \vdash a.l_i : \tau_i}$,if $1 \leq i \leq n$. There are two possibilities:

- $a.l \rightarrow_{basic+\delta(\Delta)} a'.l$ under the hypothesis $a \rightarrow_{basic+\delta(\Delta)} a'$. By induction hypothesis it follows that $\Gamma \vdash a' : [l_1 : \tau_1, \ldots, l_n : \tau_n]$. Hence, $\Gamma \vdash a'.l_i : \tau_i$ follows from (select).

- Assume $a \equiv [l_1 = a_1, \ldots, l_n = a_n]$ and $[l_1 = a_1, \ldots, l_n = a_n].l_i \rightarrow_{basic+\delta(\Delta)} a_1$, with $1 \leq i \leq n$. By Lemma 32 $\Gamma \vdash a_i : \tau_i'$ with $\tau_i' \leq \tau_i$. Then, by (subsumption) $\Gamma \vdash a_i : \tau_i$.

(case) $\dfrac{\Gamma \vdash a : d[\boldsymbol{\rho}] \quad \Gamma \vdash b_i : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^\sigma \quad (1 \leq i \leq n)}{\Gamma \vdash \mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\, a \;\mathsf{of}\; \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma}$. There are three possibilities:

- $\mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\, a \;\mathsf{of}\; \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} \rightarrow_{basic+\delta(\Delta)} \mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\, a' \;\mathsf{of}\; \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}$ under the hypothesis $a \rightarrow_{basic+\delta(\Delta)} a'$. By induction hypothesis $\Gamma \vdash a' : d[\boldsymbol{\rho}]$. Hence, $\Gamma \vdash \mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\, a' \;\mathsf{of}\; \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma$ follows using (case).

- $\mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\, a \;\mathsf{of}\; \{c_1 \Rightarrow b_1 | \ldots | c_i \Rightarrow b_i | \ldots | c_n \Rightarrow b_n\} \rightarrow_{basic} \mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\, a \;\mathsf{of}\; \{c_1 \Rightarrow b_1 | \ldots | c_i \Rightarrow b_i' | \ldots | c_n \Rightarrow b_n\}$ under the hypothesis $b_i \rightarrow_{basic+\delta(\Delta)} b_i'$. By induction hypothesis $\Gamma \vdash b_i' : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^\sigma$. Hence, $\Gamma \vdash \mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\, a \;\mathsf{of}\; \{c_1 \Rightarrow b_1 \mid \ldots \mid c_i \Rightarrow b_i' \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma$ follows using (case).

- $\mathsf{case}^\sigma_{d[\boldsymbol{\rho}]}\, (c_i[\boldsymbol{\rho}']\, \boldsymbol{a}) \;\mathsf{of}\; \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} \rightarrow_{basic+\delta(\Delta)} b_i\, \boldsymbol{a}$. As $\Gamma \vdash b_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\sigma}\}$ and $\boldsymbol{a} \equiv a_1 \ldots a_n$ then, by applying n times the (application) rule we get $\Gamma \vdash b_i\, \boldsymbol{a} : \sigma$.

(global) $\dfrac{\Gamma \vdash a : \tau}{\Gamma, x = a : \tau \vdash x : \tau}$ with x being Γ-fresh. In this case $x \rightarrow_{basic+\delta(\Delta)} a$. We have $\Gamma \vdash a : \tau$ and x is Γ-fresh. Then, by Lemma 31, $\Gamma, x = a : \tau \vdash a : \tau$.

(local) $\dfrac{\Gamma \vdash a : \tau \qquad \Gamma, x : \tau \vdash b : \sigma}{\Gamma \vdash (\mathsf{let}\; x = a : \tau \;\mathsf{in}\; b) : \sigma}$ There are three possibilities:

- $(\mathsf{let}\; x = a : \tau \;\mathsf{in}\; b) \rightarrow_{basic+\delta(\Delta)} (\mathsf{let}\; x = a' : \tau \;\mathsf{in}\; b)$ under the hypothesis $a \rightarrow_{basic+\delta(\Delta)} a'$. By induction hypothesis it follows that $\Gamma \vdash a' : \tau$. Hence, $\Gamma \vdash (\mathsf{let}\; x = a' : \tau \;\mathsf{in}\; b) : \sigma$ follows from (local).

– $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \rightarrow_{basic+\delta(\Delta)} (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b')$ under the hypothesis $b \rightarrow_{basic+\delta(\Delta)} b'$. By induction hypothesis it follows that $\Gamma, x : \tau \vdash b' : \sigma$. Hence, $\Gamma \vdash (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b') : \sigma$ follows from (local).

– $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \rightarrow_{basic+\delta(\Delta)} b\{x := a\}$. We have $\Gamma \vdash a : \tau$ and $\Gamma, x : \tau \vdash b : \sigma$, then by Lemma 34 part 2 comes $\Gamma \vdash b\{x := a\} : \sigma$.

The remaining cases are easy to prove using the induction hypothesis. $\square$

5.3 Confluence

Here we will make the proof of Church Rosser for the δ-reduction and for the $basic+\delta$-reduction. First, let's define a projection map $|.|_.$. The projection of a term a, $|a|_\Delta$, is a term that is obtained from a by unfolding all the definitions occuring in Δ and in a.

Definition 41. *The mapping* $|.|_. : \mathcal{E}_\delta \times [\mathcal{C}_\delta]_{\mathsf{def}} \rightarrow \mathcal{E}$ *is defined as follows:*

$$\begin{array}{rcl}
|x|_\Delta & = & \begin{cases} |a|_{\Delta_1} & \textit{if } \Delta \equiv \Delta_1, x = a : \tau, \Delta_2 \\ x & \textit{otherwise} \end{cases} \\
|a\,b|_\Delta & = & |a|_\Delta\,|b|_\Delta \\
|\lambda x{:}\tau.\ a|_\Delta & = & \lambda x{:}\tau.\ |a|_\Delta \\
|[l_1 = a_1, \ldots, l_n = a_n]|_\Delta & = & [l_1 = |a_1|_\Delta, \ldots, l_n = |a_n|_\Delta] \\
|a.l|_\Delta & = & |a|_\Delta.l \\
|c[\boldsymbol{\sigma}]\,a_1 \ldots a_k|_\Delta & = & c[\boldsymbol{\sigma}]\,|a_1|_\Delta \ldots |a_k|_\Delta \\
|\mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}|_\Delta & = & \mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ |a|_\Delta\ \mathsf{of}\ \{c_1 \Rightarrow |b_1|_\Delta \mid \ldots \mid c_n \Rightarrow |b_n|_\Delta\} \\
|\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b|_\Delta & = & |b|_\Delta\{x := |a|_\Delta\}\ \textit{where } x \textit{ is } \Delta\textit{-fresh}
\end{array}$$

The value $|a|_\Delta$ is the unfolding of all definitions occuring in Δ and all the local definitions of a. Global definitions are unfolded in the first line and local definitions in the last line.

Now we introduce a notion of free variable with respect to a context.

Definition 42. *Let* $M \in \mathcal{E}_\delta$ *and* $\Gamma \in \mathcal{C}_\delta$. *The set of* free variable of M with respect to Γ, *denoted by* $\mathsf{FV}_\Gamma(M)$, *is defined inductively as follows:*

$$\begin{array}{rcl}
\mathsf{FV}_{<>}(b) & = & \mathsf{FV}(b) \\
\mathsf{FV}_{\Gamma, x:\tau}(b) & = & \mathsf{FV}_\Gamma(b) \backslash \{x\} \\
\mathsf{FV}_{\Gamma, x=a:\tau}(b) & = & \mathsf{FV}_\Gamma(a) \cup (\mathsf{FV}_\Gamma(b) \backslash \{x\})
\end{array}$$

Note that variables in Γ are in fact bound variables.

Lemma 35.

1. *If* $x \notin \mathsf{FV}(a)$ *and* x *is* Δ*-fresh, then* $x \in \mathsf{FV}(|a|_\Delta)$.
2. *Let* $< \Delta_1, \Delta_2, \Delta_3 > \in [\mathcal{C}]_{\mathsf{def}}$ *and* $a \in \mathcal{E}_\delta$ *such that* $\mathsf{FV}_{\Delta_3}(a) \cap \mathsf{dom}(\Delta_3) = \emptyset$. *Then* $|a|_{\Delta_1, \Delta_2, \Delta_3} = |a|_{\Delta_1, \Delta_3}$.
3. *Let* $< \Delta_1, x = a : \tau, \Delta_3 > \in [\mathcal{C}]_{\mathsf{def}}$ *and* $a \in \mathcal{E}_\delta$. *Then* $|a|_{\Delta_1, x=a:\tau, \Delta_3} = |a|_{\Delta_1}$

Proof.

1. By induction on the number of symbols occuring in Δ and a.
2. By induction on the structure of a.
3. None of the variable in $\mathsf{dom}(x = a : \tau, \Delta_2)$ can occur in a. Hence, by 2., $|a|_{\Delta_1, x=a:\tau, \Delta_2} = |a|_{\Delta_1}$.

$\square$

The following lemma states that $|.|_.$ preserves substitution, $|b|_\Delta\{x := |a|_\Delta\} = |b\{x := a\}|_\Delta$. Also it shows that $|.|_.$ yields the same value for global and local definitions and this value is given by substitution, $|\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b|_\Delta = |b|_\Delta\{x := |a|_\Delta\} = |b\{x := a\}|_\Delta = |b|_{\Delta, x=a:\tau}$

Lemma 36. *Let $\Delta, x = a : \tau \in [\mathcal{C}_\delta]_{\mathsf{def}}$ and $b \in \mathcal{E}_\delta$. Then*

$$|b|_\Delta\{x := |a|_\Delta\} \;=\; |b\{x := a\}|_\Delta \;=\; |b|_{\Delta, x=a:\tau}$$

Proof. By induction on the structure of b. Only some cases are considered here.

– Suppose $b \equiv x$. Then

$$\begin{array}{l} |x|_\Delta\{x := |a|_\Delta\} = x\{x := |a|_\Delta\} = |a|_\Delta \\ |x\{x := |a|_\Delta\}|_\Delta = |a|_\Delta \\ |x|_{\Delta,x=a:\tau} = |a|_\Delta \end{array}$$

– Suppose $b \equiv y \not\equiv x$. Then

$$\begin{array}{lll} |y|_\Delta\{x := |a|_\Delta\} \;=\; y\{x := |a|_\Delta\} \;=\; |y|_\Delta & & \text{by Lemma 35} \\ |y\{x := |a|_\Delta\}|_\Delta \;=\; |y|_\Delta & & \\ |y|_{\Delta,x=a:\tau} \;=\; |y|_\Delta & & \text{by Lemma 35} \end{array}$$

– Suppose $b \equiv \mathsf{let}\ y = f : \sigma\ \mathsf{in}\ e$. Then

$$\begin{array}{rcll} |b|_\Delta\{x := |a|_\Delta\} & = & |\mathsf{let}\ y = f : \sigma\ \mathsf{in}\ e|_\Delta\{x := |a|_\Delta\} & \\ & = & |e|_\Delta\{y := |f|_\Delta\}\{x := |a|_\Delta\} & \\ & = & |e|_\Delta\{x := |a|_\Delta\}\{y := |f|_\Delta\{x := |a|_\Delta\}\} & \text{by Lemma 28} \\ & = & |e|_{\Delta,x=a:\tau}\{y := |f|_{\Delta,x=a:\tau}\} & \text{by induction hypothesis} \\ & = & |\mathsf{let}\ y = f : \sigma\ \mathsf{in}\ e|_{\Delta,x=a:\tau} & \\ & = & |b|_{\Delta,x=a:\tau} & \end{array}$$

$$\begin{array}{rcll} |b|_\Delta\{x := |a|_\Delta\} & = & |\mathsf{let}\ y = f : \sigma\ \mathsf{in}\ e|_\Delta\{x := |a|_\Delta\} & \\ & = & |e|_\Delta\{y := |f|_\Delta\}\{x := |a|_\Delta\} & \\ & = & |e|_\Delta\{x := |a|_\Delta\}\{y := |f|_\Delta\{x := |a|_\Delta\}\} & \text{by Lemma 28} \\ & = & |e\{x := a\}|_\Delta\{y := |f\{x := a\}|_\Delta\} & \text{by induction hypothesis} \\ & = & |\mathsf{let}\ y = f\{x := a\} : \sigma\ \mathsf{in}\ e\{x := a\}|_\Delta & \\ & = & |(\mathsf{let}\ y = f : \sigma\ \mathsf{in}\ e)\{x := a\}|_\Delta & \text{by Definition 36} \\ & = & |b\{x := a\}|_\Delta & \end{array}$$

□

Next we show that a term reduces to its projection.

Lemma 37. *Let $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$ and $a \in \mathcal{E}_\delta$. Then*

$$e \;\twoheadrightarrow_{\delta(\Delta)}\; |e|_\Delta$$

Proof. By induction on the number of the symbols occuring in Δ and in e. Here, we only consider the following cases:

– Suppose $e \equiv x$.
  - If $\Delta \equiv \Delta_1, x = a : \tau, \Delta_2$, then $|x|_\Delta = |a|_{\Delta_1}$. By induction hypothesis $a \twoheadrightarrow_{\delta(\Delta_1)} |a|_{\Delta_1}$ then, by Lemma 27, $a \twoheadrightarrow_{\delta(\Delta)} |a|_{\Delta_1}$. Hence, $x \rightarrow_{\delta(\Delta)} a \twoheadrightarrow_{\delta(\Delta)} |a|_{\Delta_1} = |x|_\Delta$.
  - If $\Delta \not\equiv \Delta_1, x = a : \tau, \Delta_2$, then $|x|_\Delta = x$. Hence, $x \twoheadrightarrow_{\delta(\Delta)} |x|_\Delta$.

– Suppose $e \equiv \mathsf{let}\ x = a : \tau\ \mathsf{in}\ b$. By induction hypothesis $a \twoheadrightarrow_{\delta(\Delta)} |a|_\Delta$ and $b \twoheadrightarrow_{\delta(\Delta)} |b|_\Delta$. Hence,

$$\begin{array}{rcll} \mathsf{let}\ x = a : \tau\ \mathsf{in}\ b & \twoheadrightarrow_{\delta(\Delta)} & \mathsf{let}\ x = |a|_\Delta : \tau\ \mathsf{in}\ |b|_\Delta & \text{by induction hypothesis and the definition of } \twoheadrightarrow_{\delta(\Delta)} \\ & \rightarrow_{\delta(\Delta)} & |b|_\Delta\{x := |a|_\Delta\} & \\ & = & |\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b|_\Delta & \end{array}$$

The rest of the cases are easy to prove. □

The following lemma states that the projection of two terms that are in $\rightarrow_{\delta(\Delta)}$ are equal.

Lemma 38. *Let $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$ and $e_1, e_2 \in \mathcal{E}_\delta$. Then*

$$e_1 \ \rightarrow_{\delta(\Delta)} \ e_2 \quad \Rightarrow \quad |e_1|_\Delta = |e_2|_\Delta$$

Proof. By induction on the definition of $\rightarrow_{\delta(\Delta)}$. Here only some cases are considered.

– Assume $e_1 \rightarrow_{\delta(\Delta)} e_2$ is $x \rightarrow_{\delta(\Delta_1, x=a:\tau, \Delta_2)} a$.

$$\begin{aligned} |e_1|_\Delta &= |x|_{\Delta_1, x=a:\tau, \Delta_2} \\ &= |a|_{\Delta_1} \\ &= |a|_{\Delta_1, x=a:\tau, \Delta_2} && \text{by Lemma 35} \\ &= |e_2|_\Delta \end{aligned}$$

– Assume $e_1 \rightarrow_{\delta(\Delta)} e_2$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \rightarrow_{\delta(\Delta)} b\{x := a\}$

$$\begin{aligned} |e_1|_\Delta &= |b|_\Delta\{x := |a|_\Delta\} \\ &= |b\{x := a\}|_\Delta && \text{by Lemma 36} \\ &= |e_2|_\Delta \end{aligned}$$

– Assume $e_1 \rightarrow_{\delta(\Delta)} e_2$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \rightarrow_{\delta(\Delta)} (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b')$

$$\begin{aligned} |e_1|_\Delta &= |b|_\Delta\{x := |a|_\Delta\} \\ &= |b|_{\Delta, x=a:\tau} && \text{by Lemma 36} \\ &= |b'|_{\Delta, x=a:\tau} && \text{by induction hypothesis} \\ &= |b'|_\Delta\{x := |a|_\Delta\} && \text{by Lemma 36} \\ &= |e_2|_\Delta \end{aligned}$$

– Assume $e_1 \rightarrow_{\delta(\Delta)} e_2$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \rightarrow_{\delta(\Delta)} (\mathsf{let}\ x = a' : \tau\ \mathsf{in}\ b)$

$$\begin{aligned} |e_1|_\Delta &= |b|_\Delta\{x := |a|_\Delta\} \\ &= |b|_\Delta\{x := |a'|_\Delta\} && \text{by induction hypothesis} \\ &= |e_2|_\Delta \end{aligned}$$

The rest of the cases are easy to prove. □

Proposition 14 (Church Rosser for δ-reduction). *Let $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$ and $a, a_1, a_2 \in \mathcal{E}_\delta$ such that $a \twoheadrightarrow_{\delta(\Delta)} a_1$ and $a \twoheadrightarrow_{\delta(\Delta)} a_2$. Then there exists a term a_3 such that $a_1 \twoheadrightarrow_{\delta(\Delta)} a_3$ and $a_2 \twoheadrightarrow_{\delta(\Delta)} a_3$.*

Proof. By Lemma 37, $a \twoheadrightarrow_{\delta(\Delta)} |a|_\Delta$, $a_1 \twoheadrightarrow_{\delta(\Delta)} |a_1|_\Delta$ and $a_2 \twoheadrightarrow_{\delta(\Delta)} |a_2|_\Delta$. By Lemma 38, it follows that $|a|_\Delta = |a_1|_\Delta = |a_2|_\Delta$.

$$\begin{tikzcd} & a \arrow[dl, two heads, "\delta(\Delta)"'] \arrow[dr, two heads, "\delta(\Delta)"] \arrow[dd, two heads, "\delta(\Delta)"] & \\ a_1 \arrow[d, two heads, "\delta(\Delta)"] & & a_2 \arrow[d, two heads, "\delta(\Delta)"] \\ |a_1|_\Delta \arrow[r, equal] & |a|_\Delta \arrow[r, equal] & |a_2|_\Delta \end{tikzcd}$$

□

The following lemma states that the projection preserves β-reduction.

Lemma 39. *Given $e, e' \in \mathcal{E}_\delta$ and $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$.*

$$e \ \rightarrow_{basic} \ e' \quad \Rightarrow \quad |e|_\Delta \ \rightarrow_{basic} \ |e'|_\Delta$$

Proof. By induction on the definition of $e \to_{basic} e'$. Most cases follows immediately from induction hypothesis and Definitions 39 and 41. Here are only the non-immediate cases:

– Assume $e \to_{basic} e'$ is $(\lambda x{:}\tau.\, a)\, b \to_{basic} a\{x := b\}$.

$$\begin{array}{rcll} |e|_\Delta & = & |(\lambda x{:}\tau.\, a)\, b|_\Delta & \\ & = & |(\lambda x{:}\tau.\, a)|_\Delta\, |b|_\Delta & \\ & \to_{basic} & |a|_\Delta\{x := |b|_\Delta\} & \\ & = & |b\{x := a\}|_\Delta & \text{by Lemma 36} \\ & = & |e'|_\Delta & \end{array}$$

– Assume $e \to_{basic} e'$ is $[\ldots, l = a, \ldots].l \to_{basic} a$.

$$\begin{array}{rcl} |e|_\Delta & = & |[\ldots, l = a, \ldots].l|_\Delta \\ & = & [\ldots, l = |a|_\Delta, \ldots].l \\ & \to_{basic} & |a|_\Delta \\ & = & |e'|_\Delta \end{array}$$

– Assume $e \to_{basic} e'$ is $\mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}']}\ (c_i[\boldsymbol{\tau}]\, a_1 \ldots a_{k_i})\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\} \to_{basic} f_i\, a_1 \ldots a_{k_i}$.

$$\begin{array}{rcl} |e|_\Delta & = & |\mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}']}\ (c_i[\boldsymbol{\tau}]\, a_1 \ldots a_{k_i})\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}|_\Delta \\ & = & \mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}']}\ (c_i[\boldsymbol{\tau}]\, |a_1|_\Delta \ldots |a_{k_i}|_\Delta)\ \mathsf{of}\ \{c_1 \Rightarrow |f_1|_\Delta \mid \ldots \mid c_n \Rightarrow |f_n|_\Delta\} \\ & \to_{basic} & |f_i|_\Delta\, |a_1|_\Delta \ldots |a_{k_i}|_\Delta \\ & = & |f_i\, a_1 \ldots a_{k_i}|_\Delta \\ & = & |e'|_\Delta \end{array}$$

– Assume $e \to_{basic} e'$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \to_{basic} (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b')$, with $b \to_{basic} b'$.

$$\begin{array}{rcll} |e|_\Delta & = & |b|_\Delta\{x := |a|_\Delta\} & \\ & \to_{basic} & |b'|_\Delta\{x := |a|_\Delta\} & \text{by induction hypothesis and Lemma 29} \\ & = & |\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b'|_\Delta & \\ & = & |e'|_\Delta & \end{array}$$

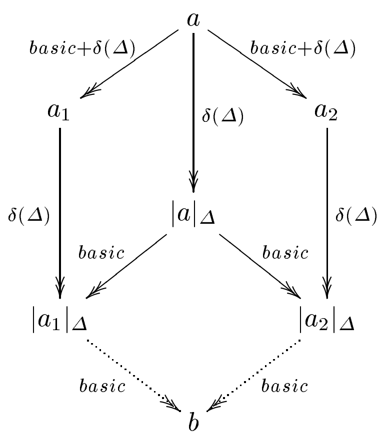
– Assume $e \to_{basic} e'$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \to_{basic} (\mathsf{let}\ x = a' : \tau\ \mathsf{in}\ b)$, with $a \to_{basic} a'$.

$$\begin{array}{rcll} |e|_\Delta & = & |b|_\Delta\{x := |a|_\Delta\} & \\ & \to_{basic} & |b|_\Delta\{x := |a'|_\Delta\} & \text{by induction hypothesis and Lemma 30} \\ & = & |\mathsf{let}\ x = a' : \tau\ \mathsf{in}\ b|_\Delta & \\ & = & |e'|_\Delta & \end{array}$$

□

Proposition 15 (Church Rosser for $basic + \delta$-reduction).
Let $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$ and $a \in \mathcal{E}_\delta$ be such that $a \twoheadrightarrow_{basic+\delta(\Delta)} a_1$ and $a \twoheadrightarrow_{basic+\delta(\Delta)} a_2$. Then there exists a term b such that $a_1 \twoheadrightarrow_{basic+\delta(\Delta)} b$ and $a_2 \twoheadrightarrow_{basic+\delta(\Delta)} b$.

Proof. By Lemma 37, $a \twoheadrightarrow_{\delta(\Delta)} |a|_\Delta$, $a_1 \twoheadrightarrow_{\delta(\Delta)} |a_1|_\Delta$ and $a_2 \twoheadrightarrow_{\delta(\Delta)} |a_2|_\Delta$. By Lemmas 39 and 38, it follows that $|a|_\Delta \twoheadrightarrow_{basic} |a_1|_\Delta$ and $|a|_\Delta \twoheadrightarrow_{basic} |a_2|_\Delta$. By Proposition 1, there exists a term $b \in \mathcal{E}$ such that $|a_1|_\Delta \twoheadrightarrow_{basic} b$ and $|a_2|_\Delta \twoheadrightarrow_{basic} b$. Hence, by composition, $a_1 \twoheadrightarrow_{basic+\delta(\Delta)} b$ and $a_2 \twoheadrightarrow_{basic+\delta(\Delta)} b$.



□

5.4 Strong Normalization

Let us begin with the proof of strong normalization for δ-reduction.

Definition 43. *Let $a \in \mathcal{E}_\delta$ and $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$. The term a* is in δ-normal form (or δ-nf) in Δ, *if there is no term b such that $a \to_{\delta(\Delta)} b$.*

A term can be in δ-normal form but not in *basic*-normal form.

Lemma 40. *Let $a \in \mathcal{E}_\delta$.*

$$a \textit{ is in } \delta\textit{-normal form in } \Delta \quad \textit{if and only if} \quad a \in \mathcal{E} \quad \textit{and} \quad \mathsf{FV}(a) \cap \mathsf{dom}(\Delta) = \emptyset$$

Proof. By induction on the structure of a. □

This states that a term a is in δ-normal form in Δ if and only if a does not contain either local or global definitions.

Lemma 41. *Let $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$ and $a \in \mathcal{E}_\delta$. Te term $|a|_\Delta$ is the δ-normal form of a in Δ.*

Proof. It is easy to proof using Lemma 40

Definition 44. *Let $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$ and $a \in \mathcal{E}_\delta$.*

1. *The term a* δ-strongly normalizes in Δ *if there is no infinite δ-reduction starting with a in Δ.*

2. *The reduction δ is* strongly normalizing *if for all pairs $(\Delta, b) \in [\mathcal{C}_\delta]_{\mathsf{def}} \times \mathcal{E}_\delta$, the term b δ-strongly normalizes in Δ.*

By the Lemma 41 the normal form of for an arbitrary δ-term exists, but there is no guarantees that all δ-paths starting at the term are finite.

In order to prove strong normalization for δ we are going to define a function $\mathsf{nat}_{\cdot}(.) : [\mathcal{C}_\delta]_{\mathsf{def}} \times \mathcal{E}_\delta \to \mathbb{N}$ taht decreses with δ-reduction.

Definition 45. *The mapping $\mathsf{nat}_{\cdot}(.) : [\mathcal{C}_\delta]_{\mathsf{def}} \times \mathcal{E}_\delta \to \mathbb{N}$ is defined as follows:*

$$
\begin{array}{rcll}
\mathsf{nat}_{\Delta_1, x=a:\tau, \Delta_2}(x) & = & \mathsf{nat}_{\Delta_1}(a) + 1 & \\
\mathsf{nat}_{\Delta}(x) & = & 0 & \textit{if } x \notin \mathsf{dom}(\Delta) \\
\mathsf{nat}_{\Delta}(a\, b) & = & \mathsf{nat}_{\Delta}(a) + \mathsf{nat}_{\Delta}(b) & \\
\mathsf{nat}_{\Delta}(\lambda x{:}\tau.\, a) & = & \mathsf{nat}_{\Delta}(a) & \\
\mathsf{nat}_{\Delta}([l_1 = a_1, \ldots, l_n = a_n]) & = & \mathsf{nat}_{\Delta}(a_1) + \ldots + \mathsf{nat}_{\Delta}(a_n) & \\
\mathsf{nat}_{\Delta}(a.l) & = & \mathsf{nat}_{\Delta}(a) & \\
\mathsf{nat}_{\Delta}(c[\boldsymbol{\sigma}]\, a_1 \ldots a_k) & = & \mathsf{nat}_{\Delta}(a_1) + \ldots + \mathsf{nat}_{\Delta}(a_k) & \\
\mathsf{nat}_{\Delta}(\mathsf{case}^{\tau}_{d[\sigma]}\ a \ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}) & = & \mathsf{nat}_{\Delta}(a) + \mathsf{nat}_{\Delta}(b_1) + \ldots + \mathsf{nat}_{\Delta}(b_n) & \\
\mathsf{nat}_{\Delta}(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) & = & \mathsf{nat}_{\Delta}(a) + \mathsf{nat}_{\Delta, x=a:\tau}(b) + 1 &
\end{array}
$$

Lemma 42. *Let* $< \Delta_1, \Delta_2, \Delta_3 > \in [\mathcal{C}_\delta]_{\mathsf{def}}$ *and* $a \in \mathcal{E}_\delta$. *Then*

$$\mathsf{FV}_{\Delta_3}(a) \cap \mathsf{dom}(\Delta_2) = \emptyset \quad \Rightarrow \quad \mathsf{nat}_{\Delta_1,\Delta_2,\Delta_3}(a) = \mathsf{nat}_{\Delta_1,\Delta_3}(a)$$

Proof. By induction on the number of symbols in $< \Delta_1, \Delta_2, \Delta_3 >$ and in a. □

Definition 46. *Let* $\Delta, \Delta' \in [\mathcal{C}_\delta]_{\mathsf{def}}$. *We define* $\Delta \to_\delta \Delta'$ *as follows:*

$$\frac{a \ \to_{\delta(\Delta_1)} \ a'}{\Delta_1, x = a : \tau, \Delta_2 \ \to_\delta \ \Delta_1, x = a' : \tau, \Delta_2}$$

Lemma 43. *Let* $\Delta, \in [\mathcal{C}_\delta]_{\mathsf{def}}$ *and* $a, a' \in \mathcal{E}_\delta$.

1. *If* $a \to_{\delta(\Delta)} a'$ *then* $\mathsf{nat}_\Delta(a) > \mathsf{nat}_\Delta(a')$

2. *If* $\Delta \to_\delta \Delta'$ *then* $\mathsf{nat}_\Delta(a) \geq \mathsf{nat}_\Delta(a)$

Proof. By simultaneous induction on the number of symbols in Δ and in a.
Here we only present the case that $a \equiv x$

1. In this case $a \to_{\delta(\Delta)} a'$ is $x \to_{\delta(\Delta_1, x=e:\tau, \Delta_2)} e$. We have $\mathsf{nat}_{\Delta_1,x=e:\tau,\Delta_2}(x) = \mathsf{nat}_{\Delta_1}(e) + 1$. By Lemma 42 $\mathsf{nat}_{\Delta_1}(e) = \mathsf{nat}_\Delta(e)$. Hence, $\mathsf{nat}_\Delta(x) = \mathsf{nat}_\Delta(e) + 1 > \mathsf{nat}_\Delta(e)$.
2. Assume $\Delta \to_\delta \Delta'$. According to the Definition 46, suppose $\Delta \equiv \Delta_1, y = e : \tau, \Delta_2$ with $e \to_{\delta(\Delta_1)} e'$ and $\Delta' \equiv \Delta_1, y = e' : \tau, \Delta_2$.
   - If $x \in \mathsf{dom}(\Delta_1)$ then $\mathsf{nat}_\Delta(x) = \mathsf{nat}_{\Delta'}(x)$.
   - If $x \equiv y$ then, by induction hypothesis, $\mathsf{nat}_{\Delta_1}(e) > \mathsf{nat}_{\Delta_1}(e')$. Hence, $\mathsf{nat}_\Delta(x) = \mathsf{nat}_{\Delta_1}(e) + 1 > \mathsf{nat}_{\Delta_1}(e') + 1 = \mathsf{nat}_{\Delta'}(x)$.
   - If $x \in \mathsf{dom}(\Delta_2)$ then, by induction hypothesis, $\mathsf{nat}_\Delta(x) \geq \mathsf{nat}_{\Delta'}(x)$.
   - If $x \notin \mathsf{dom}(\Delta)$ then $\mathsf{nat}_\Delta(x) = 0 = \mathsf{nat}_{\Delta'}(x)$.

The rest of the cases are easy to prove. □

Proposition 16 (Strong normalization for δ). *The reduction δ is strongly normalizing.*

Proof. Follows immediately form Lemma 43, part 1. □

$\mathsf{nat}_\Delta(a)$ is an upper bound for the number of reduction steps in a δ-reduction sequence starting at a in Δ.

Let us now turn to the proof of strong normalization for the $basic + \delta$ reduction.

Definition 47. *Let* $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$ *and* $e \in \mathcal{E}_\delta$.

1. *The term* e *basic*+δ-strongly normalizes in Δ *if there is no infinite basic*+δ*-reduction starting with* e *in* Δ.

2. *The reduction basic* + δ *is* strongly normalizing *if for all pairs* $(\Delta, b) \in [\mathcal{C}_\delta]_{\mathsf{def}} \times \mathcal{E}_\delta$, *the term* b *basic* + δ*-strongly normalizes in* Δ.

The function $|.|_.$ may not map infinite *basic*-reduction sequences to infinite *basic*-reduction sequences. For example: suppose there is an infinite *basic*-reduction sequence starting at a and hence at the expression $\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b$. If b is a $basic + \delta$-normal form and $x \notin \mathsf{FV}(b)$, then there is no *basic*-reduction sequence sarting at $|\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b|_{<>} = |b|_{<>}\{x := |a|_{<>}\} = |b|_{<>} = b$.

The mapping $|.|_.$ erases the term a which could contain an infinite reduction sequence. So, this function $|.|_.$ can not be used to study strong normalization for $basic + \delta$. Let's define a new function $._.$.

Definition 48. *The mapping* $\{.\}_. : \mathcal{E}_\delta \times [\mathcal{C}_\delta]_{\mathsf{def}} \to \mathcal{E}$ *is defined as follows:*

$$\begin{array}{rcl}
\{x\}_\Delta &=& \begin{cases} \{a\}_{\Delta_1} & \textit{if } \Delta \equiv \Delta_1, x = a : \tau, \Delta_2 \\ x & \textit{otherwise} \end{cases} \\
\{a\, b\}_\Delta &=& \{a\}_\Delta \, \{b\}_\Delta \\
\{\lambda x{:}\tau.\, a\}_\Delta &=& \lambda x{:}\tau.\, \{a\}_\Delta \\
\{[l_1 = a_1, \ldots, l_n = a_n]\}_\Delta &=& [l_1 = \{a_1\}_\Delta, \ldots, l_n = \{a_n\}_\Delta] \\
\{a.l\}_\Delta &=& \{a\}_\Delta.l \\
\{c[\boldsymbol{\sigma}]\, a_1 \ldots a_k\}_\Delta &=& c[\boldsymbol{\sigma}]\, \{a_1\}_\Delta \ldots \{a_k\}_\Delta \\
\{\mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ a \ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\}\}_\Delta &=& \mathsf{case}^\tau_{d[\boldsymbol{\sigma}]}\ \{a\}_\Delta \ \mathsf{of}\ \{c_1 \Rightarrow \{b_1\}_\Delta \mid \ldots \mid c_n \Rightarrow \{b_n\}_\Delta\} \\
\{\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b\}_\Delta &=& (\lambda x{:}\tau.\, \{b\}_{\Delta, x = a:\tau})\{a\}_\Delta
\end{array}$$

The mapping $\{.\}_.$ unfolds all global definitions and transfer every local definitions into a β-redex. The redex $(\lambda x{:}\tau.\, \{b\}_{\Delta,x=a:\tau})\{a\}_\Delta$ is a special kind of β-redex since $x \notin \mathsf{FV}(\{b\}_{\Delta,x=a:\tau})$.

The mapping $\{.\}_.$ is extended to contexts.

Definition 49. *The mapping* $\{.\} : \mathcal{C}_\delta \to \mathcal{C}$ *is defined as follows:*

$$\begin{array}{rcl}
\{<>\} &=& <> \\
\{\Gamma, x : \tau\} &=& \{\Gamma\}, x : \tau \\
\{\Gamma, x = a : \tau\} &=& \{\Gamma\}, x : \tau
\end{array}$$

Similar properties proved for the projection $|.|_.$ hold for the function $\{.\}_.$.

Lemma 44.

1. *If* $x \notin \mathsf{FV}(a)$ *and* x *is* Δ*-fresh, then* $x \in \mathsf{FV}(\{a\}_\Delta)$.
2. *Let* $< \Delta_1, \Delta_2, \Delta_3 > \in [\mathcal{C}]_{\mathsf{def}}$ *and* $a \in \mathcal{E}_\delta$ *such that* $\mathsf{FV}_{\Delta_3}(a) \cap \mathsf{dom}(\Delta_3) = \emptyset$. *Then* $\{a\}_{\Delta_1,\Delta_2,\Delta_3} = \{a\}_{\Delta_1,\Delta_3}$.
3. *Let* $< \Delta_1, x = a : \tau, \Delta_3 > \in [\mathcal{C}]_{\mathsf{def}}$ *and* $a \in \mathcal{E}_\delta$. *Then* $\{a\}_{\Delta_1, x = a:\tau, \Delta_3} = \{a\}_{\Delta_1}$

Proof.

1. By induction on the number of symbols occuring in Δ and a.
2. By induction on the structure of a.
3. None of the variable in $\mathsf{dom}(x = a : \tau, \Delta_2)$ can occur in a. Hence, by 2., $\{a\}_{\Delta_1, x=a:\tau, \Delta_2} = \{a\}_{\Delta_1}$.

□

Lemma 45. *Let* $\Delta, x = a : \tau \in [\mathcal{C}_\delta]_{\mathsf{def}}$ *and* $b \in \mathcal{E}_\delta$. *Then*

$$\{b\}_\Delta\{x := \{a\}_\Delta\} \;=\; \{b\{x := a\}\}_\Delta \;=\; \{b\}_{\Delta, x = a:\tau}$$

Proof. By induction on the structure of b.

By the following lemma, $\{.\}_.$ maps an infinite $\mathit{basic} + \delta$-reduction sequence to an infinite basic-reduction sequence.

Lemma 46. *Given* $e, e' \in \mathcal{E}_\delta$ *and* $\Delta \in [\mathcal{C}_\delta]_{\mathsf{def}}$.

1. $e \to_{basic} e' \quad \Rightarrow \quad \{e\}_\Delta \to^+_{basic} \{e'\}_\Delta$
2. $e \to_{\delta(\Delta)} e' \quad \Rightarrow \quad \{e\}_\Delta \to^\equiv_{basic} \{e'\}_\Delta$

Proof.

1. By induction on the definition of $e \to_{basic} e'$. Most cases follows immediately from induction hypothesis and Definitions 39 and 48. Here are only the non-immediate cases:

– Assume $e \rightarrow_{basic} e'$ is $(\lambda x{:}\tau.\, a)\, b \rightarrow_{basic} a\{x := b\}$.

$$\begin{array}{rcll} \{e\}_\Delta & = & \{(\lambda x{:}\tau.\, a)\, b\}_\Delta & \\ & = & (\lambda x{:}\tau.\, \{a\}_\Delta)\, \{b\}_\Delta & \\ & \rightarrow_{basic} & \{a\}_\Delta\{x := \{b\}_\Delta\} & \\ & = & \{a\{x := b\}\}_\Delta & \text{by Lemma 45} \\ & = & \{e'\}_\Delta & \end{array}$$

– Assume $e \rightarrow_{basic} e'$ is $[\ldots, l = a, \ldots].l \rightarrow_{basic} a$.

$$\begin{array}{rcl} \{e\}_\Delta & = & \{[\ldots, l = a, \ldots].l\}_\Delta \\ & = & [\ldots, l = \{a\}_\Delta, \ldots].l \\ & \rightarrow_{basic} & \{a\}_\Delta \\ & = & \{e'\}_\Delta \end{array}$$

– Assume $e \rightarrow_{basic} e'$ is $\mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}']}\ (c_i[\boldsymbol{\tau}]\, a_1 \ldots a_{k_i})\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\} \rightarrow_{basic} f_i\, a_1 \ldots a_{k_i}$.

$$\begin{array}{rcl} \{e\}_\Delta & = & \{\mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}']}\ (c_i[\boldsymbol{\tau}]\, a_1 \ldots a_{k_i})\ \mathsf{of}\ \{c_1 \Rightarrow f_1 \mid \ldots \mid c_n \Rightarrow f_n\}\}_\Delta \\ & = & \mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}']}\ (c_i[\boldsymbol{\tau}]\, \{a_1\}_\Delta \ldots \{a_{k_i}\}_\Delta)\ \mathsf{of}\ \{c_1 \Rightarrow \{f_1\}_\Delta \mid \ldots \mid c_n \Rightarrow \{f_n\}_\Delta\} \\ & \rightarrow_{basic} & \{f_i\}_\Delta\, \{a_1\}_\Delta \ldots \{a_{k_i}\}_\Delta \\ & = & \{f_i\, a_1 \ldots a_{k_i}\}_\Delta \\ & = & \{e'\}_\Delta \end{array}$$

– Assume $e \rightarrow_{basic} e'$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \rightarrow_{basic} (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b')$, with $b \rightarrow_{basic} b'$.

$$\begin{array}{rcll} \{e\}_\Delta & = & (\lambda x{:}\tau.\, \{b\}_{\Delta, x=a:\tau})\, \{a\}_\Delta & \\ & \rightarrow^{+}_{basic} & (\lambda x{:}\tau.\, \{b'\}_{\Delta, x=a:\tau})\, \{a\}_\Delta & \text{by induction hypothesis} \\ & = & \{\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b'\}_\Delta & \\ & = & \{e'\}_\Delta & \end{array}$$

– Assume $e \rightarrow_{basic} e'$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \rightarrow_{basic} (\mathsf{let}\ x = a' : \tau\ \mathsf{in}\ b)$, with $a \rightarrow_{basic} a'$.

$$\begin{array}{rcll} \{e\}_\Delta & = & (\lambda x{:}\tau.\, \{b\}_{\Delta, x=a:\tau})\, \{a\}_\Delta & \\ & = & (\lambda x{:}\tau.\, \{b\}_\Delta\{x := \{a\}_\Delta\})\, \{a\}_\Delta & \text{by Lemma 45} \\ & \rightarrow^{+}_{basic} & (\lambda x{:}\tau.\, \{b\}_\Delta\{x := \{a'\}_\Delta\})\, \{a'\}_\Delta & \text{by induction hypothesis and Lemma 30} \\ & = & (\lambda x{:}\tau.\, \{b\}_{\Delta, x=a':\tau})\, \{a'\}_\Delta & \text{by Lemma 45} \\ & = & \{\mathsf{let}\ x = a' : \tau\ \mathsf{in}\ b\}_\Delta & \\ & = & \{e'\}_\Delta & \end{array}$$

2. By induction on the definition of $e \rightarrow_{basic} e'$. Here are only the non-immediate cases:

   – Assume $e \rightarrow_{\delta(\Delta)} e'$ is $x \rightarrow_{\delta(\Delta_1, x=a:\tau, \Delta_2)} a$. We have $\{x\}_{\delta(\Delta_1, x=a:\tau, \Delta_2)} = \{a\}_{\Delta_1}$ and, by Lemma 44 part 3, $\{a\}_{\Delta_1} = \{a\}_{\delta(\Delta_1, x=a:\tau, \Delta_2)}$.

   – Assume $e \rightarrow_{\delta(\Delta)} e'$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \rightarrow_{\delta(\Delta)} (\mathsf{let}\ x = a' : \tau\ \mathsf{in}\ b)$, with $a \rightarrow_{\delta(\Delta)} a'$.

$$\begin{array}{rcll} \{e\}_\Delta & = & (\lambda x{:}\tau.\, \{b\}_{\Delta, x=a:\tau})\, \{a\}_\Delta & \\ & = & (\lambda x{:}\tau.\, \{b\}_\Delta\{x := \{a\}_\Delta\})\, \{a\}_\Delta & \text{by Lemma 45} \\ & \rightarrow^{+}_{basic} & (\lambda x{:}\tau.\, \{b\}_\Delta\{x := \{a'\}_\Delta\})\, \{a'\}_\Delta & \text{by induction hypothesis and Lemma 30} \\ & = & (\lambda x{:}\tau.\, \{b\}_{\Delta, x=a':\tau})\, \{a'\}_\Delta & \text{by Lemma 45} \\ & = & \{\mathsf{let}\ x = a' : \tau\ \mathsf{in}\ b\}_\Delta & \\ & = & \{e'\}_\Delta & \end{array}$$

   – Assume $e \rightarrow_{\delta(\Delta)} e'$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \rightarrow_{\delta(\Delta)} (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b')$, with $b \rightarrow_{\delta(\Delta)} b'$.

$$\begin{array}{rcll} \{e\}_\Delta & = & (\lambda x{:}\tau.\, \{b\}_{\Delta, x=a:\tau})\, \{a\}_\Delta & \\ & \rightarrow^{\equiv}_{basic} & (\lambda x{:}\tau.\, \{b'\}_{\Delta, x=a:\tau})\, \{a\}_\Delta & \text{by induction hypothesis} \\ & = & \{\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b'\}_\Delta & \\ & = & \{e'\}_\Delta & \end{array}$$

– Assume $e \to_{\delta(\Delta)} e'$ is $(\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) \to_{\delta(\Delta)} b$, because $x \notin \mathsf{FV}(b)$.

$$\begin{array}{rcll} \{e\}_\Delta & = & (\lambda x{:}\tau.\ \{b\}_{\Delta,x=a:\tau})\,\{a\}_\Delta & \\ & \to_{basic} & \{b\}_{\Delta,x=a:\tau}\{x := \{a\}_\Delta\} & \\ & = & \{b\{x := a\}\}_\Delta\{x := \{a\}_\Delta\} & \text{by Lemma 44} \\ & = & \{b\}_\Delta & \\ & = & \{e'\}_\Delta & \end{array}$$

The rest of the cases are easy to prove. □

Lemma 47.

1. $a \to_{\delta(\Delta)} a' \quad \Rightarrow \quad \{a\}_\Delta \twoheadrightarrow_{basic} \{a'\}_\Delta$
2. $a =_{basic+\delta(\Delta)} a' \quad \Rightarrow \quad \{a\}_\Delta =_{basic} \{a'\}_\Delta$

Proof. Immediate, by Lemma 46. □

The following lemma states that $\{.\}_.$ mapps expressions that are typeble in $\lambda_{\to,[],\mathsf{data},\mathsf{def}}$ to expressions that are typeble in $\lambda_{\to,[],\mathsf{data}}$.

Lemma 48. *Let $\Gamma \in \mathcal{C}_\delta$, $e \in \mathcal{E}_\delta$ and $\sigma \in \mathcal{T}_\aleph$.*

$$\Gamma \vdash_{cs_\delta} e : \sigma \quad \Rightarrow \quad \{\Gamma\} \vdash_{cs} \{e\}_{[\Gamma]_{\mathsf{def}}} : \sigma$$

Proof. By induction on the derivation of $\Gamma \vdash_{cs_\delta} e : \sigma$. Suppose the last step in the derivation is

(start) $\Gamma \vdash_{cs_\delta} e : \sigma$ is $\Gamma \vdash_{cs_\delta} x : \sigma$, with $x : \sigma \in \Gamma$. Then $x : \sigma \in \{\Gamma\}$ and $\{x\}_{[\Gamma]_{\mathsf{def}}} = x$. Hence $\{\Gamma\} \vdash_{cs} \{x\}_{[\Gamma]_{\mathsf{def}}} : \sigma$.

(application) $\dfrac{\Gamma \vdash_{cs_\delta} a : \tau \to \sigma \qquad \Gamma \vdash_{cs_\delta} a' : \tau}{\Gamma \vdash_{cs_\delta} a\,a' : \sigma}$
By induction hypothesis $\{\Gamma\} \vdash_{cs} \{a\}_{[\Gamma]_{\mathsf{def}}} : \tau \to \sigma$ and $\{\Gamma\} \vdash_{cs} \{a'\}_{[\Gamma]_{\mathsf{def}}} : \tau$. By (application), $\{\Gamma\} \vdash_{cs} \{a\}_{[\Gamma]_{\mathsf{def}}}\,\{a'\}_{[\Gamma]_{\mathsf{def}}} : \sigma$. Hence, $\{\Gamma\} \vdash_{cs} \{a\,a'\}_{[\Gamma]_{\mathsf{def}}} : \sigma$.

(global) $\dfrac{\Gamma \vdash_{cs_\delta} a : \tau}{\Gamma, x = a : \tau \vdash_{cs_\delta} x : \tau}$ if x is Γ-fresh.
We have $\{x\}_{[\Gamma]_{\mathsf{def}},x=a:\tau} = \{a\}_{[\Gamma]_{\mathsf{def}}}$ and $\{\Gamma, x = a : \tau\} = \{\Gamma\}, x : \tau$. By induction hypothesis, $\{\Gamma\} \vdash_{cs} \{a\}_{[\Gamma]_{\mathsf{def}}} : \tau$. Hence, by Lemma 5, $\{\Gamma\}, x : \tau \vdash_{cs} \{a\}_{[\Gamma]_{\mathsf{def}}} : \tau$.

(local) $\dfrac{\Gamma \vdash_{cs_\delta} a : \tau \qquad \Gamma, x : \tau \vdash_{cs_\delta} b : \sigma}{\Gamma \vdash_{cs_\delta} (\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b) : \sigma}$
We have $\{\mathsf{let}\ x = a : \tau\ \mathsf{in}\ b\}_{[\Gamma]_{\mathsf{def}}} = (\lambda x{:}\tau.\ \{b\}_{[\Gamma]_{\mathsf{def}},x=a:\tau})\,\{a\}_{[\Gamma]_{\mathsf{def}}}$.
By induction hypothesis, $\{\Gamma\} \vdash_{cs} \{a\}_{[\Gamma]_{\mathsf{def}}} : \tau$ and $\{\Gamma, x : \tau\} \vdash_{cs} \{b\}_{[\Gamma]_{\mathsf{def}}} : \sigma$. So, by (abstraction), $\{\Gamma\} \vdash_{cs} (\lambda x{:}\tau.\ \{b\}_{[\Gamma]_{\mathsf{def}}}) : \tau \to \sigma$. Hence, by (application), $\{\Gamma\} \vdash_{cs} (\lambda x{:}\tau.\ \{b\}_{[\Gamma]_{\mathsf{def}}})\,\{a\}_{[\Gamma]_{\mathsf{def}}} : \sigma$.

The remaining cases are easy to prove using the induction hypothesis and Definition 48. □

Proposition 17 (Strong normalization for $basic + \delta$**).** *The reduction $basic + \delta$ is strongly normalizing on typeble expressions:*

$$\Gamma \vdash_{cs_\delta} a : \sigma \quad \Rightarrow \quad a \in \mathsf{SN}(\to_{basic+\delta([\Gamma]_{\mathsf{def}})})$$

Proof. We know, by Proposition 12, that $\to_{basic}$ is strongly normalizing on typeble expressions. Supposed (towards a contradiction) that $\Gamma \vdash_{cs_\delta} a : \sigma$ and that there is an infinite $basic + \delta([\Gamma]_{\mathsf{def}})$-reduction sequence, ζ, starting at a.

Since δ is strongly normalizing, ζ must have an infinite sequence of *basic*-reductions. Otherwise it would follow that there is a $n_0 \in \mathbb{N}$ such that $\forall m \geq n_0.\ a_m \to_{\delta(\Delta)} a_{m+1}$. Hence we would have the infinite sequence

$a_{n_0+1} \rightarrow_{\delta(\Delta)} a_{n_0+2} \rightarrow_{\delta(\Delta)} \ldots$. But, as δ is strongly normalizing this can not happend. Hence, the number of *basic*-reduction steps in the sequence $a \rightarrow_{basic+\delta(\Delta)} a_1 \rightarrow_{basic+\delta(\Delta)} a_1 \rightarrow_{basic+\delta(\Delta)} \ldots$ is infinite. Then this sequence is of the form

$$a \twoheadrightarrow_{\delta(\Delta)} a_{n_1} \rightarrow_{basic} a_{n_2} \twoheadrightarrow_{\delta(\Delta)} a_{n_3} \rightarrow_{basic} a_{n_4} \twoheadrightarrow_{\delta(\Delta)} \cdots$$

By Lemmas 46 and 47 there is an infinite *basic*-reduction sequence starting at $\{a\}_\Delta$

$$\{a\}_\Delta \twoheadrightarrow_{basic} \{a_{n_1}\}_\Delta \rightarrow^+_{basic} \{a_{n_2}\}_\Delta \twoheadrightarrow_{basic} \{a_{n_3}\}_\Delta \rightarrow^+_{basic} \{a_{n_4}\}_\Delta \twoheadrightarrow_{basic} \cdots$$

By Lemma 48 $\{\Gamma\} \vdash_{cs} \{a\}_{[\Gamma]_{\mathsf{def}}} : \sigma$ which contradicts the assumption that $\rightarrow_{basic}$ is strongly normalizing. □

6 Adding fixpoints

$\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def}}$ has a very restricted computational power. In particular, it does not support recursion. In this section, we present an extension of $\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def}}$ with fixpoints, $\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def},\mathsf{fix}}$, and we study some properties of the resulting calculus.

In $\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def}}$ we may have recursive datatypes but we don't have yet a way of expressing recursive definitions, so that we can program and reason with the elements of these datatypes.

A possible way of expressing recursion is to associate an elimination rule to each inductive type. But in this approach, the only way of describing a function on a recursive datatype is to codify it using the elimination constant for that datatype.

Sometimes it is difficult (not evident) to describe certain functions using this technique. Instead of using elimination constants to define recursive functions, we extend $\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def}}$ with a fixed point operator for doing recursive definitions named letrec. This technique is closer to the functional programming languages, where a recursive function is defined directly by a set of pattern matching equations.

Definition 50 (Expressions). *The set of* expressions $\mathcal{E}_{\delta,\mu}$ *is the extension of* $\mathcal{E}_\delta$ *with the clause*

$$\mathsf{letrec}_i(x_1 : \tau_1 =_{k_1} a_1, \ldots, x_n : \tau_n =_{k_n} a_n)$$

The operator letrec is used to define functions over recursive datatypes. As it is suggested by this expression scheme, we may have mutually recursive definitions. The integer i in the letrec_i indicate which function is being defined and the integers $k_1, \ldots, k_n$ point out on which inductive argument of the function the recursion is done.

Example 13. Remember the following legal datatype context, already presented in previous sections, defining the Nat, Odd and Even datatypes:

$$\begin{array}{l} \aleph_1 = .; s : \mathrm{Even} \rightarrow \mathrm{Odd}, 0 : \mathrm{Even}, s : \mathrm{Odd} \rightarrow \mathrm{Even}; \\ \aleph_2 = \aleph_1\ 0 : \mathrm{Nat}, s : \mathrm{Nat} \rightarrow \mathrm{Nat}, s : \mathrm{Odd} \rightarrow \mathrm{Nat}, s : \mathrm{Even} \rightarrow \mathrm{Nat}; \end{array}$$

The addition and multiplication of two natural numbers can be defined as follows:

$$\begin{array}{rll} + = \mathsf{letrec}_1\ (p & : & \mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat} \\ & =_1 & \lambda x{:}\mathrm{Nat}.\ \lambda y{:}\mathrm{Nat}.\ \mathsf{case}^{\mathrm{Nat}}_{\mathrm{Nat}}\ x\ \mathsf{of}\ \{\ 0 \Rightarrow y\ \mid \\ & & \qquad\qquad\qquad\qquad\qquad\qquad s \Rightarrow \lambda n{:}\mathrm{Nat}.\ s\,(p\,n\,y)\} \\ ) & : & \mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat} \end{array}$$

$$\begin{array}{rll} \times = \mathsf{letrec}_1\ (t & : & \mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat} \\ & =_1 & \lambda x{:}\mathrm{Nat}.\ \lambda y{:}\mathrm{Nat}.\ \mathsf{case}^{\mathrm{Nat}}_{\mathrm{Nat}}\ x\ \mathsf{of}\ \{\ 0 \Rightarrow 0\ \mid \\ & & \qquad\qquad\qquad\qquad\qquad\qquad s \Rightarrow \lambda n{:}\mathrm{Nat}.\ +\,y\,(t\,n\,y)\} \\ ) & : & \mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat} \end{array}$$

Example 14. The addition of two odd numbers and the addition of an odd and an even number, declared in a mutually dependent way.

$$
\begin{array}{rl}
+_{\mathsf{OO}} = \mathsf{letrec}_1\,(\,+_{oo} & : \ \text{Odd} \to \text{Odd} \to \text{Even} \\
& =_1 \lambda x{:}\text{Odd}.\ \lambda y{:}\text{Odd}.\ \mathsf{case}^{\text{Even}}_{\text{Odd}}\ x\ \mathsf{of}\ \{\ \ s \Rightarrow \lambda n{:}\text{Even}.\ s\,(+_{eo}\,n\,y)\}\quad, \\
+_{eo} & : \ \text{Even} \to \text{Odd} \to \text{Odd} \\
& =_1 \lambda x{:}\text{Even}.\ \lambda y{:}\text{Odd}.\ \mathsf{case}^{\text{Odd}}_{\text{Even}}\ x\ \mathsf{of}\ \{\ 0 \Rightarrow y\ \mid \\
& \qquad\qquad s \Rightarrow \lambda n{:}\text{Odd}.\ s\,(+_{oo}\,n\,y)\} \\
) & : \ \text{Odd} \to \text{Odd} \to \text{Even}
\end{array}
$$

$$
\begin{array}{rl}
+_{\mathsf{OE}} = \mathsf{letrec}_2\,(\,+_{ee} & : \ \text{Even} \to \text{Even} \to \text{Even} \\
& =_1 \lambda x{:}\text{Even}.\ \lambda y{:}\text{Even}.\ \mathsf{case}^{\text{Even}}_{\text{Even}}\ x\ \mathsf{of}\ \{\ 0 \Rightarrow y\ \mid \\
& \qquad\qquad s \Rightarrow \lambda n{:}\text{Odd}.\ s\,(+_{oe}\,n\,y)\}\quad, \\
+_{oe} & : \ \text{Odd} \to \text{Even} \to \text{Odd} \\
& =_1 \lambda x{:}\text{Odd}.\ \lambda y{:}\text{Even}.\ \mathsf{case}^{\text{Odd}}_{\text{Odd}}\ x\ \mathsf{of}\ \{\ \ s \Rightarrow \lambda n{:}\text{Even}.\ s\,(+_{ee}\,n\,y)\} \\
) & : \ \text{Even} \to \text{Even} \to \text{Even}
\end{array}
$$

Example 15. The mutually dependent definition of the multiplication of two odd numbers.

$$
\begin{array}{rl}
\times_{\mathsf{OO}} = \mathsf{letrec}_1\,(\,\times_{oo} & : \ \text{Odd} \to \text{Odd} \to \text{Odd} \\
& =_1 \lambda x{:}\text{Odd}.\ \lambda y{:}\text{Odd}.\ \mathsf{case}^{\text{Odd}}_{\text{Odd}}\ x\ \mathsf{of}\ \{\ \ s \Rightarrow \lambda n{:}\text{Even}.\ +_{\mathsf{OE}}\,y\,(\times_{eo}\,n\,y)\}\quad, \\
\times_{eo} & : \ \text{Even} \to \text{Odd} \to \text{Even} \\
& =_1 \lambda x{:}\text{Even}.\ \lambda y{:}\text{Odd}.\ \mathsf{case}^{\text{Even}}_{\text{Even}}\ x\ \mathsf{of}\ \{\ 0 \Rightarrow 0\ \mid \\
& \qquad\qquad s \Rightarrow \lambda n{:}\text{Odd}.\ +_{\mathsf{OO}}\,y\,(\times_{oo}\,n\,y)\} \\
) & : \ \text{Odd} \to \text{Odd} \to \text{Odd}
\end{array}
$$

letrec expressions introduce bound variables. Therefore we can use α-conversion when necessary. In $\mathsf{letrec}_i(x_1 : \tau_1 =_{k_1} a_1, \ldots, x_n : \tau_n =_{k_n} a_n)$, the variables $x_1, \ldots, x_n$ are bound.

Definition 51. *Let $M \in \mathcal{E}_{\delta,\mu}$. The set of* free variables *of M, denoted by $\mathsf{FV}(M)$, is defined by extending the Definition 32 with the following clause:*

$$\mathsf{FV}(\mathsf{letrec}_i(x_1 : \tau_1 =_{k_1} a_1, \ldots, x_n : \tau_n =_{k_n} a_n)) = (\mathsf{FV}(a_1) \cup \ldots \cup \mathsf{FV}(a_n)) \backslash \{x_1, \ldots, x_n\}$$

Definition 52 (Substitution). *Let $M, N \in \mathcal{E}_{\delta,\mu}$ and $x \in \mathcal{V}$. The* substitution of N for x in M*, denoted by $M\{x := N\}$, is defined by extending the Definition 36 with the following clauses:*

$$
\begin{array}{rcll}
\mathsf{letrec}_i(x_1 : \tau_1 =_{k_1} a_1, \ldots, x_n : \tau_n =_{k_n} a_n)\{x_j := N\} & \equiv & \mathsf{letrec}_i(x_1 : \tau_1 =_{k_1} a_1, \ldots, x_n : \tau_n =_{k_n} a_n) & \textit{if } 1 \leq j \leq n \\
\mathsf{letrec}_i(x_1 : \tau_1 =_{k_1} a_1, \ldots, x_n : \tau_n =_{k_n} a_n)\{x := N\} & \equiv & \mathsf{letrec}_i(x_1 : \tau_1 =_{k_1} a_1\{x := N\}, \ldots, x_n : \tau_n =_{k_n} a_n\{x := N\}) &
\end{array}
$$

By the variable convention, in the second clause of the previous definition the variables $x_1, \ldots, x_n$ don't occur free in the term N, and $x_i \neq x$ for $1 \leq i \leq n$.

Recursive objects are defined by fixpoint definitions as in functional programming languages but, to avoid the introduction of non-normalizable terms, some syntactical checking must be done on the recursive definitions before being accepted. In the typing rule of the letrec-expressions there will be a syntactical condition (called $\mathcal{G}$) that should be satisfied. This kind of mechanism was introduced in [17]. Let us first present the typing rule and then discuss the condition $\mathcal{G}$.

Definition 53 (Typing).

1. *A* judgment *is a triple of the form $\Gamma \ \vdash_{cs_{\delta,\mu}} \ a : \tau$, where $\Gamma \in \mathcal{C}_\delta$, $a \in \mathcal{E}_{\delta,\mu}$ and $\tau \in \mathcal{T}_\aleph$. In order to enhance readability, we will often omit the subscript $cs_{\delta,\mu}$ of $\vdash$.*

2. *A judgment is* derivable *if it can be inferred from the set of typing rules that result of adding the following rule to the rules of system* $\lambda_{\rightarrow,[],\mathsf{data},\mathsf{def}}$.

$$(\text{letrec}) \quad \frac{\Gamma,\, f_1 : \tau_1, \ldots,\, f_n : \tau_n \;\vdash\; a_i : \tau_i \qquad \mathcal{G}_{\mathsf{M}}(\emptyset, f_i, x_i, b_i) \qquad (1 \leq i \leq n)}{\Gamma \;\vdash\; \mathsf{letrec}_j(f_1 : \tau_1 =_{k_1} a_1, \ldots, f_n : \tau_n =_{k_n} a_n) : \tau_j} \;,$$

$$\begin{array}{ll} \textit{with} & 1 \leq j \leq n \\ & \tau_i \equiv \rho_1 \rightarrow \ldots \rightarrow \rho_{k_i-1} \rightarrow d_i[\boldsymbol{\sigma}_i] \rightarrow \gamma_i \\ & a_i \equiv \lambda z_1{:}\rho_1. \ldots \lambda z_{k_i-1}{:}\rho_{k_i-1}.\, \lambda x_i{:}d_i[\boldsymbol{\sigma}_i].\, b_i \\ & \mathsf{M} \equiv \{(f_1, k_1), \ldots, (f_n, k_n)\} \end{array}$$

3. *An expression* $a \in \mathcal{E}_{\delta,\mu}$ *is* typable *if* $\Gamma \vdash a : \sigma$ *for some* $\Gamma \in \mathcal{C}_\delta$ *and type* $\sigma \in \mathcal{T}_{\aleph}$.

We introduce now the formal definition of the μ-reduction.

Definition 54 (Reduction).

1. *The μ-reduction, $\rightarrow_\mu$, is defined as the compatible closure of the rule:*

$$\begin{array}{l} \mathsf{letrec}_i(\boldsymbol{x} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a})\; p_1 \ldots p_{k_i-1}\; (c[\boldsymbol{\sigma}]\, \boldsymbol{b}) \;\rightarrow_\mu \\ \quad a_i\{x_1 := \mathsf{letrec}_1(\boldsymbol{x} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a}), \ldots, x_n := \mathsf{letrec}_n(\boldsymbol{x} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a})\}\; p_1 \ldots p_{k_i-1}\; (c[\boldsymbol{\sigma}]\, \boldsymbol{b}) \end{array}$$

2. *Let Δ be a list of definitions.* $\rightarrow_{basic+\delta(\Delta)+\mu}$ *is defined as* $\rightarrow_{basic} \cup \rightarrow_{\delta(\Delta)} \cup \rightarrow_\mu$. *We let* $\rightarrow_{total} \equiv \rightarrow_{basic+\delta(\Delta)+\mu}$.

In order to keep the strong normalization property, the μ-reduction will only be performed when the argument in position of the recursive variable (whose type should be a datatype) starts with a constructor.

In the (letrec) rule there is a syntactical condition $\mathcal{G}_{\mathsf{M}}(\emptyset, f_i, x_i, b_i)$ for each f_i defined in the letrec-expression. It is so, because we have mutual dependent definitions. These conditions complement the μ-reduction rule, ensuring that each expansion of the letrec operator consumes (at least) the constructor in the head of the k^{th} argument.

Informally, the term $\mathsf{letrec}_1(f : \tau =_k a)$ should satisfy the following constraints (note that we don't have here a mutually dependent definition):

- f may occur in a only at the head of an application;
- any application of f must be protected by a case analysis of the k^{th} abstraction of a, say x;
- the k^{th} argument in the application of f must be a component of x;
- only recursive components of x are authorized to be the argument of a recursive call.

For example, the definition of $+$ over Nat in example 13 verifies these requisites: p occur only at the head of the application $(p\,n\,y)$; the application of p, $(p\,n\,y)$, is protected by a case analysis of x (the first variable abstracted) and n is a component of x (since for this case we have $x = s\,n$).

When we have a mutually dependent definition $\mathsf{letrec}_j(f_1 : \tau_1 =_{k_1} a_1, \ldots, f_n : \tau_n =_{k_n} a_n)$ we have to make some adjustments on this ideas, because f_i may not occur directly in a_i, but indirectly through another $f_{i'}$ (with $1 \leq i' \leq n$) such that f_i occur (directly or indirectly) in $f_{i'}$. So we can say that the term $\mathsf{letrec}_j(f_1 : \tau_1 =_{k_1} a_1, \ldots, f_n : \tau_n =_{k_n} a_n)$ should satisfy the following requirements:

- $f_{i'}$ may occur in a_i only at the head of an application;
- any application of $f_{i'}$ must be protected by a case analysis of the k_i^{th} abstraction of a_i, say x_i;
- the $k_{i'}^{th}$ argument in the application of $f_{i'}$ must be a component of x_i;
- only recursive components of x_i are authorized to be the argument of a recursive call.

See in example 14 the definition of $+_{\mathsf{OO}}$: $+_{oo}$ and $+_{eo}$ occur at the had of applications $(+_{oo}\,n\,y)$ and $+_{eo}\,n\,y$, respectively; the application of $+_{eo}$ is protected by a case analysis of x, the recursive argument of $+_{oo}$; the application of $+_{oo}$ is protected by a case analysis of x, the recursive argument of $+_{eo}$; n is a component of x (in bought cases).

The direct components of x_i are represented by terms of the form $(z\, p_1 \ldots p_r)$ with z being a pattern variable of the case analysis which protect the application. Further matchings on these terms allow to access to deeper components of x.

A pattern variable y_i representing a component of x_i is recursive if the type of x_i occurs in the type of y_i

In order to know which are the recursive components of a given object, we define an auxiliary predicate RP that indicates the recursive positions of a given d-constructor type.

Definition 55 (Recursive position).

Let $\tau \equiv \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}]$ be a d-constructor type. We say that the number j corresponds to a recursive position of τ if $d[\boldsymbol{\alpha}]$ appears in the type ρ_j. We write this property as $\mathsf{RP}(j, \tau)$.

As we already mentioned there is a syntactical condition $\mathcal{G}_{\mathsf{M}}(\emptyset, f_i, x_i, b_i)$ $(1 \leq i \leq n)$ that should be satisfied as part of the typing rule (letrec). This condition $\mathcal{G}$ constrains the occurrences of f_i allowed in the body of the operator letrec and it will be called *to be guarded by destructors* because, as we said before, any application of $f_{i'}$ must be protected by a case analysis of the k_i^{th} abstraction of a_i.

The formal description of the guarded-by-destructors condition is given by a predicate $\mathcal{G}_{\mathsf{M}}(V, f, x, b)$ defined by induction on the term b. M is a set of pairs that to each function identifier associate the position of the inductive argument on which the recursion is done. V is a set of identifiers used to collect the pattern variable in b which represent the recursive components of x. f is the name of the function we want to test it is guarded by destructors in b. x is the recursive argument of f.

Definition 56 (Guarded by destructors).

Let M be the set of pairs of identifiers and positive integers, V a set of identifiers, f and x identifiers, and b a term. The predicate $\mathcal{G}_{\mathsf{M}}(V, f, x, b)$ is defined inductively by the following rules (where f nocc e denotes that f does not occur in e):

1. $$\frac{f \ \mathsf{nocc} \ e}{\mathcal{G}_{\mathsf{M}}(V,f,x,e)}$$

2. $$\frac{\mathcal{G}_{\mathsf{M}}(V,f,x,e)}{\mathcal{G}_{\mathsf{M}}(V,f,x,\lambda z{:}\tau.\ e)}$$

3. $$\frac{\mathcal{G}_{\mathsf{N}}(V,f,x,a_i) \qquad (1 \leq i \leq n)}{\mathcal{G}_{\mathsf{M}}(V,f,x,\mathsf{letrec}_i(g_1 : \tau_1 =_{k_1} a_1, \ldots, g_n : \tau_n =_{k_n} a_n))}\ , \quad \textit{if} \ \ \mathsf{N} \equiv \mathsf{M} \cup \{(g_1,k_1),\ldots,(g_n,k_n)\}$$

4. $$\frac{\mathcal{G}_{\mathsf{M}}(U,f,x,e_i) \qquad (1 \leq i \leq n)}{\mathcal{G}_{\mathsf{M}}(V,f,x,\mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ z \ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})}\ , \quad \textit{if} \begin{cases} z \in V \cup \{x\} \\ b_i \equiv \lambda y_1{:}\rho_1.\ \ldots \lambda y_{h_i}{:}\rho_{h_i}.\, e_i \\ \tau_i \equiv \rho_1 \to \ldots \to \rho_{h_i} \to \tau \\ U \equiv V \cup \{y_j \mid \mathsf{RP}(j,\tau_i),\ \textit{for}\ 1 \leq j \leq h_i\} \end{cases}$$

5. $$\frac{\mathcal{G}_{\mathsf{M}}(V,f,x,e) \qquad \mathcal{G}_{\mathsf{M}}(U,f,x,e_i) \qquad (1 \leq i \leq n)}{\mathcal{G}_{\mathsf{M}}(V,f,x,\mathsf{case}^{\tau}_{d[\boldsymbol{\sigma}]}\ e \ \mathsf{of}\ \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\})}\ , \quad \textit{if} \begin{cases} e \notin V \cup \{x\} \\ b_i \equiv \lambda y_1{:}\rho_1.\ \ldots \lambda y_{h_i}{:}\rho_{h_i}.\, e_i \\ \tau_i \equiv \rho_1 \to \ldots \to \rho_{h_i} \to \tau \\ U \equiv V \cup \{y_j \mid \mathsf{RP}(j,\tau_i),\ \textit{for}\ \ 1 \leq j \leq h_i\} \end{cases}$$

6. $$\frac{\mathcal{G}_{\mathsf{M}}(V,f,x,p_i) \qquad (1 \leq i \leq n)}{\mathcal{G}_{\mathsf{M}}(V,f,x,(g\,p_1 \ldots p_n))}\ , \qquad \textit{if} \begin{cases} (g,k) \in \mathsf{M} \\ n \geq k \\ p_k \equiv (z\,q_1 \ldots q_m) \\ z \in V \end{cases}$$

7. $$\frac{\mathcal{G}_{\mathsf{M}}(V,f,x,g) \qquad \mathcal{G}_{\mathsf{M}}(V,f,x,p_i) \qquad (1 \leq i \leq n)}{\mathcal{G}_{\mathsf{M}}(V,f,x,(g\,p_1 \ldots p_n))}\ , \qquad \textit{if} \ \ (g,r) \notin \mathsf{M}$$

8. $$\frac{\mathcal{G}_{\mathsf{M}}(V,f,x,a_i) \qquad (1 \leq i \leq n)}{\mathcal{G}_{\mathsf{M}}(V,f,x,[l_1 = a_1,\ldots,l_n = a_n])}$$

9. $$\frac{\mathcal{G}_{\mathsf{M}}(V,f,x,e)}{\mathcal{G}_{\mathsf{M}}(V,f,x,e.l)}$$

10. $$\frac{\mathcal{G}_{\mathsf{M}}(V,f,x,a_i) \qquad (1 \leq i \leq n)}{\mathcal{G}_{\mathsf{M}}(V,f,x,(c[\boldsymbol{\sigma}]\,a_1 \ldots a_n))}$$

11. $$\frac{\mathcal{G}_{\mathsf{M}}(V,f,x,a) \qquad \mathcal{G}_{\mathsf{M}}(V,f,x,b)}{\mathcal{G}_{\mathsf{M}}(V,f,x,(\mathsf{let}\ y = a : \tau\ \mathsf{in}\ b))}$$

comentar

(...)

Example 16. Let us expand the datatype context $\aleph_2$ with the declarations of the datatypes List$[\alpha]$, NeList$[\alpha]$, Bool and BinTree$[\alpha]$.

$$\begin{array}{l}
\aleph_3 = \aleph_2\ \mathit{nil} : \mathrm{List}[\alpha], \mathit{cons} : \alpha \to \mathrm{List}[\alpha] \to \mathrm{List}[\alpha]; \\
\aleph_4 = \aleph_3\ \mathit{cons} : \alpha \to \mathrm{List}[\alpha] \to \mathrm{NeList}[\alpha]; \\
\aleph_5 = \aleph_4\ \mathit{true} : \mathrm{Bool}, \mathit{false} : \mathrm{Bool}; \\
\aleph_6 = \aleph_5\ \mathit{empty} : \mathrm{BinTree}[\alpha], \mathit{node} : \mathrm{BinTree}[\alpha] \to \mathrm{BinTree}[\alpha] \to \mathrm{BinTree}[\alpha];
\end{array}$$

The predicate that indicates if a natural number is even or not may be defined as follows:

$$
\begin{array}{lll}
\mathsf{even} = \mathsf{letrec}_1\,(\,e & : & \mathrm{Nat} \to \mathrm{Bool} \\
& =_1 & \lambda x{:}\mathrm{Nat}.\,\mathsf{case}^{\mathrm{Bool}}_{\mathrm{Nat}}\ \mathrm{x}\ \mathsf{of}\ \{\ 0 \Rightarrow \mathrm{true}\ \mid \\
& & \qquad s \Rightarrow \lambda y{:}\mathrm{Nat}.\,\mathsf{case}^{\mathrm{Bool}}_{\mathrm{Nat}}\ \mathrm{y}\ \mathsf{of}\ \{\ 0 \Rightarrow \mathrm{false}\ \mid \\
& & \qquad\qquad s \Rightarrow \lambda z{:}\mathrm{Nat}.\,(\mathrm{e\,z})\}\} \\
) & : & \mathrm{Nat} \to \mathrm{Bool}
\end{array}
$$

If we look at the typing tree of the function even we see the condition

$$\mathcal{G}_{\mathsf{M}}(\emptyset, e, x, \mathsf{case}^{\mathrm{Bool}}_{\mathrm{Nat}}\ x\ \mathsf{of}\ \{0 \Rightarrow \mathrm{true}\ \mid s \Rightarrow \lambda y{:}\mathrm{Nat}.\,\mathsf{case}^{\mathrm{Bool}}_{\mathrm{Nat}}\ \mathrm{y}\ \mathsf{of}\ \{0 \Rightarrow \mathrm{false}\ \mid \mathrm{s} \Rightarrow \lambda \mathrm{z}{:}\mathrm{Nat}.\,(\mathrm{e\,z})\}\})$$

with $\mathsf{M} = \{(e, 1)\}$, must be verified. Let us present its proof tree

$$
\dfrac{
\dfrac{e\ \mathsf{nocc}\ true}{\mathcal{G}_{\mathsf{M}}(\emptyset, e, x, true)}\,1
\quad
\dfrac{
\dfrac{e\ \mathsf{nocc}\ false}{\mathcal{G}_{\mathsf{M}}(\{y\}, e, x, false)}\,1
\quad
\dfrac{\dfrac{e\ \mathsf{nocc}\ z}{\mathcal{G}_{\mathsf{M}}(\{y,z\}, e, x, z)}\,1}{\mathcal{G}_{\mathsf{M}}(\{y,z\}, e, x, (e,z))}\,6
}{\mathcal{G}_{\mathsf{M}}(\{y\}, e, x, \mathsf{case}^{\mathrm{Bool}}_{\mathrm{Nat}}\ y\ \mathsf{of}\ \{0 \Rightarrow \mathrm{false}\ \mid s \Rightarrow \lambda z{:}\mathrm{Nat}.\,(\mathrm{e\,z})\})}\,4
}{\mathcal{G}_{\mathsf{M}}(\emptyset, e, x, \mathsf{case}^{\mathrm{Bool}}_{\mathrm{Nat}}\ x\ \mathsf{of}\ \{0 \Rightarrow \mathrm{true}\ \mid s \Rightarrow \lambda y{:}\mathrm{Nat}.\,\mathsf{case}^{\mathrm{Bool}}_{\mathrm{Nat}}\ \mathrm{y}\ \mathsf{of}\ \{0 \Rightarrow \mathrm{false}\ \mid \mathrm{s} \Rightarrow \lambda \mathrm{z}{:}\mathrm{Nat}.\,(\mathrm{e\,z})\}\})}\,4
$$

Example 17. Here we present the addition of two even numbers.

$$
\begin{array}{rll}
+_{\mathsf{EE}} = \mathsf{letrec}_1\,(\,+_{ee} & : & \mathrm{Even} \to \mathrm{Even} \to \mathrm{Even} \\
& =_1 & \lambda x{:}\mathrm{Even}.\,\lambda y{:}\mathrm{Even}.\,\mathsf{case}^{\mathrm{Even}}_{\mathrm{Even}}\ x\ \mathsf{of}\ \{\ 0 \Rightarrow y\ \mid \\
& & \qquad s \Rightarrow \lambda n{:}\mathrm{Odd}.\,s\,(+_{eo}\,y\,n)\} \quad , \\
\\
+_{eo} & : & \mathrm{Even} \to \mathrm{Odd} \to \mathrm{Odd} \\
& =_2 & \lambda x{:}\mathrm{Even}.\,\lambda y{:}\mathrm{Odd}.\,\mathsf{case}^{\mathrm{Odd}}_{\mathrm{Odd}}\ y\ \mathsf{of}\ \{\quad s \Rightarrow \lambda n{:}\mathrm{Even}.\,s\,(+_{ee}\,n\,x)\} \\
) & : & \mathrm{Even} \to \mathrm{Even} \to \mathrm{Even}
\end{array}
$$

The definition of $+_{\mathsf{EE}}$ is done in a mutually deppendent way. In $+_{ee}$ the inductive argument is the first one and $+_{eo}$ has as inductive argument the second one.

To show $+_{\mathsf{EE}}$ is well defined we have to verify its definition satisfies the guarded condition. For that, we need to demonstrate that predicates

$$
\begin{array}{l}
\mathcal{G}_{\mathsf{M}}(\emptyset, +_{ee}, 1, x, \lambda y{:}\mathrm{Even}.\,\mathsf{case}^{\mathrm{Even}}_{\mathrm{Even}}\ \mathrm{x}\ \mathsf{of}\ \{0 \Rightarrow \mathrm{y} \mid \mathrm{s} \Rightarrow \lambda \mathrm{n}{:}\mathrm{Odd}.\,\mathrm{s}\,(+_{\mathrm{eo}}\,\mathrm{y\,n})\}) \\
\mathcal{G}_{\mathsf{M}}(\emptyset, +_{eo}, 2, y, \mathsf{case}^{\mathrm{Odd}}_{\mathrm{Odd}}\ y\ \mathsf{of}\ \{s \Rightarrow \lambda n{:}\mathrm{Even}.\,s\,(+_{ee}\,n\,x)\})
\end{array}
$$

are valid, with

$$\mathsf{M} = \{(+_{ee}, 1), (+_{eo}, 2)\}$$

In the following we present the proof for the first one. The remaining predicate can be proved in a similar way.

$$
\dfrac{
\dfrac{
\dfrac{+_{ee}\ \mathsf{nocc}\ y}{\mathcal{G}_{\mathsf{M}}(\emptyset, +_{ee}, x, y)}\,1
\quad
\dfrac{
\dfrac{
\dfrac{+_{ee}\ \mathsf{nocc}\ n}{\mathcal{G}_{\mathsf{M}}(\{n\}, +_{ee}, x, n)}\,1
\quad
\dfrac{+_{ee}\ \mathsf{nocc}\ n}{\mathcal{G}_{\mathsf{M}}(\{n\}, +_{ee}, x, y)}\,1
}{\mathcal{G}_{\mathsf{M}}(\{n\}, +_{ee}, x, +_{oe}\,n\,y)}\,6
}{\mathcal{G}_{\mathsf{M}}(\{n\}, +_{ee}, x, s\,(+_{eo}\,y\,n))}\,10
}{\mathcal{G}_{\mathsf{M}}(\emptyset, +_{ee}, x, \mathsf{case}^{\mathrm{Even}}_{\mathrm{Even}}\ x\ \mathsf{of}\ \{0 \Rightarrow y \mid s \Rightarrow \lambda n{:}\mathrm{Odd}.\,\mathrm{s}\,(+_{\mathrm{eo}}\,\mathrm{y\,n})\})}\,4
}{\mathcal{G}_{\mathsf{M}}(\emptyset, +_{ee}, x, \lambda y{:}\mathrm{Even}.\,\mathsf{case}^{\mathrm{Even}}_{\mathrm{Even}}\ \mathrm{x}\ \mathsf{of}\ \{0 \Rightarrow \mathrm{y} \mid \mathrm{s} \Rightarrow \lambda \mathrm{n}{:}\mathrm{Odd}.\,\mathrm{s}\,(+_{\mathrm{eo}}\,\mathrm{y\,n})\})}\,2
$$

In the following examples we declare a serie of well defined recursive functions. In the proof of the guarded conditions for these functions almost all cases of Definition 56 are used.

Example 18. The lenght of a list.

$$\begin{array}{rcl}\mathsf{lenght} = \mathsf{letrec}_1\,(\,\mathsf{len} & : & \mathrm{List}[\alpha] \to \mathrm{Nat} \\ & =_1 & \lambda l{:}\mathrm{List}[\alpha].\ \mathsf{case}^{\mathrm{Nat}}_{\mathrm{List}[\alpha]}\ \mathrm{l}\ \mathsf{of}\ \{\ nil\ \Rightarrow\ 0\ \mid \\ & & \qquad cons\ \Rightarrow\ \lambda h{:}\alpha.\ \lambda t{:}\mathrm{List}[\alpha].\ s\,(\mathsf{len}\,t)\} \\ ) & : & \mathrm{List}[\alpha] \to \mathrm{Nat}\end{array}$$

Example 19. The maximum of a list of naturals.

$$\begin{array}{rcl}\mathsf{bigger} = \mathsf{letrec}_1\,(\,\mathsf{big} & : & \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat} \\ & =_1 & \lambda x{:}\mathrm{Nat}.\ \lambda \mathrm{y}{:}\mathrm{Nat}.\ \mathsf{case}^{\mathrm{Nat}}_{\mathrm{Nat}}\ \mathrm{x}\ \mathsf{of}\ \{\ 0\ \Rightarrow\ y\ \mid \\ & & \qquad s\ \Rightarrow\ \lambda m{:}\mathrm{Nat}.\ (\mathsf{big}\,\mathrm{n}\,\mathrm{m})\} \\ ) & : & \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat}\end{array}$$

$$\begin{array}{rcl}\mathsf{maximum} = \mathsf{letrec}_1\,(\,\mathsf{max} & : & \mathrm{NeList}[\mathrm{Nat}] \to \mathrm{Nat} \\ & =_1 & \lambda l{:}\mathrm{NeList}[\mathrm{Nat}].\ \mathsf{case}^{\mathrm{Nat}}_{\mathrm{NeList}[\mathrm{Nat}]}\ \mathrm{l}\ \mathsf{of}\ \{\ cons\ \Rightarrow\ \lambda h{:}\mathrm{Nat}.\ \lambda \mathrm{t}{:}\mathrm{List}[\mathrm{Nat}].\ (\mathsf{bigger}\,\mathrm{h}\,(\mathsf{maxl}\,\mathrm{t}))\} \\ & & \\ \mathsf{maxl} & : & \mathrm{List}[\mathrm{Nat}] \to \mathrm{Nat} \\ & =_1 & \lambda l{:}\mathrm{List}[\mathrm{Nat}].\ \mathsf{case}^{\mathrm{Nat}}_{\mathrm{List}[\mathrm{Nat}]}\ \mathrm{l}\ \mathsf{of}\ \{\qquad nil\ \Rightarrow\ 0 \\ & & \qquad cons\ \Rightarrow\ \lambda h{:}\mathrm{Nat}.\ \lambda \mathrm{t}{:}\mathrm{List}[\mathrm{Nat}].\ (\mathsf{bigger}\,\mathrm{h}\,(\mathsf{maxl}\,\mathrm{t}))\} \\ & & \\ ) & : & \mathrm{NeList}[\mathrm{Nat}] \to \mathrm{Nat}\end{array}$$

Example 20. The function map.

$$\begin{array}{rcl}\mathsf{map} = \mathsf{letrec}_1\,(\,m & : & (\alpha_1 \to \alpha_2) \to \mathrm{List}[\alpha_1] \to \mathrm{List}[\alpha_2] \\ & =_2 & \lambda f{:}\alpha_1 \to \alpha_2.\ \lambda l{:}\mathrm{List}[\alpha_1].\ \mathsf{case}^{\mathrm{List}[\alpha_2]}_{\mathrm{List}[\alpha_1]}\ l\ \mathsf{of}\ \{\ nil\ \Rightarrow\ nil[\alpha_2] \mid \\ & & \qquad cons\ \Rightarrow\ \lambda x{:}\alpha_1.\ \lambda t{:}\mathrm{List}[\alpha_1].\ (cons[\alpha_2]\,(f\,x)(m\,f\,t))\} \\ ) & : & (\alpha_1 \to \alpha_2) \to \mathrm{List}[\alpha_1] \to \mathrm{List}[\alpha_2]\end{array}$$

Example 21. The function append that makes the concatenation of two lists.

$$\begin{array}{rcl}\mathsf{append} = \mathsf{letrec}_1\,(\,\mathsf{app} & : & \mathrm{List}[\alpha] \to \mathrm{List}[\alpha] \to \mathrm{List}[\alpha] \\ & =_1 & \lambda a{:}\mathrm{List}[\alpha].\ \lambda b{:}\mathrm{List}[\alpha].\ \mathsf{case}^{\mathrm{List}[\alpha]}_{\mathrm{List}[\alpha]}\ a\ \mathsf{of}\ \{\ nil\ \Rightarrow\ b \mid \\ & & \qquad cons\ \Rightarrow\ \lambda h{:}\alpha.\ \lambda t{:}\mathrm{List}[\alpha].\ (cons[\alpha]\,h\,(\mathsf{app}\,t\,b))\} \\ ) & : & \mathrm{List}[\alpha] \to \mathrm{List}[\alpha] \to \mathrm{List}[\alpha]\end{array}$$

Example 22. The function inorder.

$$\begin{array}{rcl}\mathsf{inorder} = \mathsf{letrec}_1\,(\,\mathsf{io} & : & \mathrm{BinTree}[\alpha] \to \mathrm{List}[\alpha] \\ & =_1 & \lambda b{:}\mathrm{BinTree}[\alpha].\ \mathsf{case}^{\mathrm{List}[\alpha]}_{\mathrm{BinTree}[\alpha]}\ b\ \mathsf{of}\ \{\ empty\ \Rightarrow\ nil[\alpha] \mid \\ & & \qquad node\ \Rightarrow \\ & & \lambda x{:}\alpha.\ \lambda l{:}\mathrm{List}[\alpha].\ \lambda r{:}\mathrm{List}[\alpha].\ (\mathsf{append}\,(\mathsf{io}\,l)(cons[\alpha]\,x\,(\mathsf{io}\,r)))\} \\ ) & : & \mathrm{BinTree}[\alpha] \to \mathrm{List}[\alpha]\end{array}$$

Example 23. The function reverse that makes the invertion of a list.

$$\begin{array}{l}\mathsf{reverse} = \lambda l{:}\mathrm{List}[\alpha].\ (\mathsf{let} \\ \quad \begin{array}{rcl}\mathsf{rev} = \mathsf{letrec}_1\,(\,r & : & \mathrm{List}[\alpha] \to \mathrm{List}[\alpha] \to \mathrm{List}[\alpha] \\ & =_2 & \lambda x{:}\mathrm{List}[\alpha].\ \lambda y{:}\mathrm{List}[\alpha].\ \mathsf{case}^{\mathrm{List}[\alpha]}_{\mathrm{List}[\alpha]}\ y\ \mathsf{of}\ \{\ nil\ \Rightarrow\ x \mid \\ & & \qquad cons\ \Rightarrow\ \lambda h{:}\alpha.\ \lambda t{:}\mathrm{List}[\alpha].\ r\,(cons[\alpha]\,h\,x)\,t\} \\ ) & : & \mathrm{List}[\alpha] \to \mathrm{List}[\alpha] \to \mathrm{List}[\alpha]\end{array} \\ \quad \mathsf{in}\ (\mathsf{rev}\,(nil[\alpha])\,l))\end{array}$$

Example 24. The function neglist that negates all the elements of a list of booleans.

$$
\begin{array}{l}
\mathsf{neglist} = \mathsf{letrec}_1\,(\,\mathsf{neg} \;:\; \mathrm{List}[\mathrm{Bool}] \to \mathrm{List}[\mathrm{Bool}] \\
\quad =_1 \lambda l{:}\mathrm{List}[\mathrm{Bool}].\ \mathsf{case}^{\mathrm{List}[\mathrm{Bool}]}_{\mathrm{List}[\mathrm{Bool}]}\ l\ \mathsf{of}\ \{\ \ nil \Rightarrow nil[\mathrm{Bool}]\ | \\
\qquad\qquad cons \Rightarrow \lambda h{:}\mathrm{Bool}.\ \lambda \mathrm{t}{:}\mathrm{List}[\mathrm{Bool}]. \\
\qquad (\mathsf{let\ not} = (\lambda x{:}\mathrm{Bool}.\ \mathsf{case}^{\mathrm{Bool}}_{\mathrm{Bool}}\ \mathrm{x}\ \mathsf{of}\ \{\ true \Rightarrow false\ | \\
\qquad\qquad false \Rightarrow true\}\,) : \mathrm{Bool} \to \mathrm{Bool} \\
\qquad\ \mathsf{in}\ (cons[\mathrm{Bool}]\,(\mathsf{not}\,h)(\mathsf{neg}\,t)) \\
\qquad) \\
\qquad\qquad \} \\
)\ \ :\ \ \mathrm{List}[\mathrm{Bool}] \to \mathrm{List}[\mathrm{Bool}]
\end{array}
$$

Example 25. The function filter that takes a list of records of type $[\mathrm{n} : \mathrm{Nat}, \mathrm{f} : \mathrm{Bool}]$ and removes all the records where $\mathrm{f} = false$.

$$
\begin{array}{l}
\mathsf{filter} = \mathsf{letrec}_1\,(\,p \;:\; \mathrm{List}[\,[\mathrm{n} : \mathrm{Nat}, \mathrm{f} : \mathrm{Bool}]\,] \to \mathrm{List}[\mathrm{Nat}] \\
\quad =_1 \lambda l{:}\mathrm{List}[\,[\mathrm{n} : \mathrm{Nat}, \mathrm{f} : \mathrm{Bool}]\,].\ \mathsf{case}^{\mathrm{List}[\mathrm{Nat}]}_{\mathrm{List}[\,[\mathrm{n:Nat,f:Bool}]\,]}\ \mathrm{l}\ \mathsf{of}\ \{\ nil \Rightarrow nil[\mathrm{Nat}]\ | \\
\qquad\qquad cons \Rightarrow \lambda h{:}[\mathrm{n} : \mathrm{Nat}, \mathrm{f} : \mathrm{Bool}]. \\
\qquad \lambda t{:}\mathrm{List}[\,[\mathrm{n} : \mathrm{Nat}, \mathrm{f} : \mathrm{Bool}]\,].\ \mathsf{case}^{\mathrm{List}[\mathrm{Nat}]}_{\mathrm{Bool}}\ (\mathrm{h.f})\ \mathsf{of}\ \{\ false \Rightarrow p\,t\ | \\
\qquad\qquad true \Rightarrow cons[\mathrm{Nat}]\ h.\mathrm{n}\,(p\,t)\} \\
\quad \} \\
)\ \ :\ \ \mathrm{List}[\,[\mathrm{n} : \mathrm{Nat}, \mathrm{f} : \mathrm{Bool}]\,] \to \mathrm{List}[\mathrm{Nat}]
\end{array}
$$

Example 26. The function mev that takes a list of naturals and marks each number with a boolean value saying if this number is even or not.

$$
\begin{array}{l}
\mathsf{mev} = \mathsf{letrec}_1\,(\,f \;:\; \mathrm{List}[\mathrm{Nat}] \to \mathrm{List}[\,[\mathrm{n} : \mathrm{Nat}, \mathrm{b} : \mathrm{Bool}]\,] \\
\quad =_1 \lambda l{:}\mathrm{List}[\mathrm{Nat}].\ \mathsf{case}^{\mathrm{List}[\,[\mathrm{n:Nat,b:Bool}]\,]}_{\mathrm{List}[\mathrm{Nat}]}\ \mathrm{l}\ \mathsf{of}\ \{\ nil \Rightarrow nil[\,[\mathrm{n} : \mathrm{Nat}, \mathrm{b} : \mathrm{Bool}]\,]\ | \\
\qquad\qquad cons \Rightarrow \lambda h{:}\mathrm{Nat}.\ \lambda \mathrm{t}{:}\mathrm{List}[\mathrm{Nat}]. \\
\qquad (cons[\,[\mathrm{n} : \mathrm{Nat}, \mathrm{b} : \mathrm{Bool}]\,]\,[\mathrm{n} = h, \mathrm{b} = (\mathsf{even}\,h)]\,(f\,t)) \\
\qquad\qquad \} \\
)\ \ :\ \ \mathrm{List}[\mathrm{Nat}] \to \mathrm{List}[\,[\mathrm{n} : \mathrm{Nat}, \mathrm{b} : \mathrm{Bool}]\,]
\end{array}
$$

Proposition 18 (Confluence). *The relation $\to_{total}$ is confluent on typable expressions*

Proof. We already know that $\to_{basic+\delta}$ is confluent. Using the Tait-Martin-Löf technique, one can prove $\to_\mu$ is confluent. To conclude, observe that $\to_{basic+\delta}$ and $\to_\mu$ commute, hence by the Hindley-Rosen Lemma their union is confluent. □

Lemma 49. *If $\Gamma_1, x_1 : \tau_1, \ldots, \Gamma_n, x_n : \tau_n, \Gamma_{n+1} \vdash M : \sigma$ and $\Gamma_1, \ldots, \Gamma_{n+1} \vdash N_i : \tau_i$ for $1 \le i \le n$, then $\Gamma_1, \ldots, \Gamma_{n+1} \vdash M\{x_1 := N_1, \ldots, x_n := N_n\} : \sigma$.*

Proof. By induction on the generation of $\Gamma_1, x_1 : \tau_1, \ldots, \Gamma_n, x_n : \tau_n, \Gamma_{n+1} \vdash M : \sigma$, assuming that $\Gamma_1, \ldots, \Gamma_{n+1} \vdash N_i : \tau_i$ is derivable for $1 \le i \le n$. □

Lemma 50. *If $\Gamma \vdash M\, N_1 \ldots N_n : \sigma$, then there exist types $\rho_1, \ldots, \rho_n$ such that $\Gamma \vdash M : \rho_1 \to \ldots \to \rho_n \to \sigma'$, with $\sigma' \le \sigma$, and $\Gamma \vdash N_i : \rho_i$ for $1 \le i \le n$.*

Proof. By inspection on the derivition of $\Gamma \vdash M\, N_1 \ldots N_n : \sigma$. □

Lemma 51. *If $\Gamma \vdash \mathsf{letrec}_j(f_1 : \tau_1 =_{k_1} a_1, \ldots, f_n : \tau_n =_{k_n} a_n) : \sigma$, then $\Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \vdash a_i : \tau_i$ for $1 \le i \le n$.*

Proof. By inspection on the derivition of $\Gamma \vdash \mathsf{letrec}_j(f_1 : \tau_1 =_{k_1} a_1, \ldots, f_n : \tau_n =_{k_n} a_n) : \sigma$. □

Proposition 19 (Subject reduction). *Typing is closed under $\to_{total}$:*

$$\Gamma \vdash a : \sigma \quad \wedge \quad a \to_{total} b \quad \Rightarrow \quad \Gamma \vdash b : \sigma$$

Proof. By induction on the derivation of $\Gamma \vdash a : \sigma$. This proof is very similar to the proof of Proposition 13. We only have to consider two new cases:

1. When the last rule applied is the (letrec) rule. In this case $a \equiv \mathsf{letrec}_j(f_1 : \tau_1 =_{k_1} a_1, \ldots, f_n : \tau_n =_{k_n} a_n)$ with $1 \le j \le n$, $\sigma \equiv \tau_j$ and $\Gamma \vdash a : \sigma$ is a direct consequence of $\Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \vdash a_j : \tau_j$. Then the only way a can be reduced is $a_i \to_{total} a_i'$, with $1 \le i \le n$. In this case, $\mathsf{letrec}_j(\ldots, f_i : \tau_i =_{k_i} a_i, \ldots) \to_{total} \mathsf{letrec}_j(\ldots, f_i : \tau_i =_{k_i} a_i', \ldots)$. By induction hypothesis $\Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \vdash a_i' : \tau_n$ for $1 \le i \le n$. Hence, $\Gamma \vdash \mathsf{letrec}_j(\ldots, f_i : \tau_i =_{k_i} a_i', \ldots) : \tau_j$ follows using (letrec).

2. $a \equiv \mathsf{letrec}_j(f_1 : \tau_1 =_{k_1} a_1, \ldots, f_n : \tau_n =_{k_n} a_n)\, p_1 \ldots p_{k_j - 1}\, (c\,[\boldsymbol{\sigma}]\, \boldsymbol{b})$. In this case $a \to_{total} b$ with $b \equiv a_j\{f_1 := \mathsf{letrec}_1(\boldsymbol{f} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a}), \ldots, f_n := \mathsf{letrec}_n(\boldsymbol{f} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a})\}\, p_1 \ldots p_{k_j - 1}\, (c\,[\boldsymbol{\sigma}]\, \boldsymbol{b})$. By Lemma 50 we have $\Gamma \vdash \mathsf{letrec}_j(f_1 : \tau_1 =_{k_1} a_1, \ldots, f_n : \tau_n =_{k_n} a_n) : \rho_1 \to \ldots \to \rho_{k_j - 1} \to \rho_{k_j} \to \sigma'$ with $\sigma' \le \sigma$, $\Gamma \vdash p_r : \rho_r$ for $1 \le r \le k_j - 1$, and $\Gamma \vdash (c\,[\boldsymbol{\sigma}]\, \boldsymbol{b}) : \rho_{k_j}$.
   By Lemma 51 we have $\Gamma, f_1 : \tau_1, \ldots, f_n : \tau_n \vdash a_j : \rho_1 \to \ldots \to \rho_{k_j - 1} \to \rho_{k_j} \to \sigma'$. Using Lemma 49, $\Gamma \vdash a_j\{f_1 := \mathsf{letrec}_1(\boldsymbol{f} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a}), \ldots, f_n := \mathsf{letrec}_n(\boldsymbol{f} : \boldsymbol{\tau} =_{\boldsymbol{k}} \boldsymbol{a})\} : \rho_1 \to \ldots \to \rho_{k_j - 1} \to \rho_{k_j} \to \sigma'$. Then, using the (application) rule k_j times we get $\Gamma \vdash b : \sigma'$. Hence, $\Gamma \vdash b : \sigma$ follows by (subsumption).

□

7 Coherent overloading

Definition 57 (Expressions). *We extend the set of expressions of $\mathcal{E}_{\delta,\mu}$ with the clause*

$$\{a_1 : d_1[\boldsymbol{\tau_1}] \to \sigma_1, \ldots, a_n : d_n[\boldsymbol{\tau_n}] \to \sigma_n\}$$

Definition 58 (Coherence). *Let*

Definition 59 (Typing).
A judgment is derivable *if it can be inferred from the set of typing rules that result of adding the following rule to the rules of system $\lambda_{\to,[],\mathsf{data},\mathsf{def},\mathsf{fix}}$.*

$$\text{(c-o)} \qquad \frac{\Gamma \vdash a_i : d_i[\boldsymbol{\tau_i}] \to \sigma_i \qquad (1 \le i \le n)}{\Gamma \vdash \{a_1 : d_1[\boldsymbol{\tau_1}] \to \sigma_1, \ldots, a_n : d_n[\boldsymbol{\tau_n}] \to \sigma_n\} : d_k[\boldsymbol{\tau_k}] \to \sigma_k} \quad , \quad \text{with } 1 \le k \le n, \text{ if } a_1 \sim \ldots \sim a_n$$

????

8 Conclusion and directions for further work

In this paper, we have introduced a simply typed λ-calculus with record types and parametric datatypes. The calculus supports a combination of record subtyping and constructor subtyping and thus provides a flexible type system. We have shown the calculus to be well-behaved.

In the future, we intend to study *definitions* for $\lambda_{\rightarrow,[],\mathsf{data}}$ and its extensions. Our goal is to aggregate a theory of definitions which is flexible enough to support overloaded definitions, such as multiplication $*$:

$$\begin{aligned} * &= *_1 : \mathrm{Nat} \rightarrow \mathrm{Even} \rightarrow \mathrm{Even} \\ &= *_2 : \mathrm{Even} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Even} \\ &= *_3 : \mathrm{Odd} \rightarrow \mathrm{Odd} \rightarrow \mathrm{Odd} \\ &= *_4 : \mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat} \end{aligned}$$

where each $*_i$ is defined using case-expressions and recursion. As suggested by the above example, the idea is to allow identifiers to stand for several functions that have a different type. To do so, several options exist: for example, one may require the definitions to be coherent in a certain sense. Alternately, one may exploit some strategy, see e.g. [10, 20], to disambiguate the definitions. Both approaches deserve further study.

Furthermore, we intend to scale up the results of this paper to more complex type systems.

1. Type systems for programming languages: in line with recent work on the design of higher-order typed (HOT) languages, one may envisage extending $\lambda_{\rightarrow,[],\mathsf{data}}$ with further constructs, including bounded quantification [9], objects [1], bounded operator abstraction [11]. We are also interested in scaling up our results to programming languages with dependent types such as DML [31]. The DML type system is based on constraints, and hence it seems possible to consider constructor subtyping on inductive families, as for example in $\mathsf{X}\ i \leq \mathsf{X}\ j$ if $i \leq j$ where $\mathsf{X}\ i$ is the type $\{0, \ldots, i\}$. Extending constructor subtyping to inductive families is particularly interesting to implement type systems with subtyping.
2. Type systems for proof assistants: the addition of subtyping to proof assistants has been a major motivation for this work. Our next step is to investigate an extension of the Calculus of Inductive/Coinduct, see e.g. [16], with constructor subtyping. As suggested in [5, 12], such a calculus seems particularly appropriate to formalize Kahn's natural semantics [21].

In yet a different direction, it may be interesting to study *destructor subtyping*, a dual to constructor subtyping, in which an inductive type σ is a subtype of another inductive type τ if σ has more destructors than τ. The primary example of destructor subtyping is of course record subtyping, as found in this paper. We leave for future work the study of destructor subtyping and of its interaction with constructor subtyping.

References

1. M. Abadi and L. Cardelli. *A theory of objects.* Springer-Verlag, 1996.
2. H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, revised edition, 1984.
3. H. Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(2):181–215, June 1997.
4. B. Barras *et. al. The Coq Proof Assistant User's Guide. Version 6.2*, May 1998.
5. G. Barthe. Order-sorted inductive types. *Information and Computation*, 1999. To appear.
6. G. Barthe and M.J. Frade. Constructor subtyping. In S.D. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lectures Notes in Computer Science*, pages 109–125. 8th European Symposium on Programming, ESOP'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, Springer-Verlag, March 1999.
7. G. Betarte. *Dependent Record Types and Algebraic Structures in Type Theory.* PhD thesis, Department of Computer Science, Chalmers Tekniska Högskola, 1998.
8. L. Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, March 1996.
9. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
10. G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995.

11. A. Compagnoni and H. Goguen. Typed operational semantics for higher order subtyping. Technical Report ECS-LFCS-97-361, University of Edinburgh, July 1997.
12. T. Coquand. Pattern matching with dependent types. In B. Nordström, editor, *Informal proceedings of Logical Frameworks'92*, pages 66–79, 1992.
13. T. Coquand. Infinite objects type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.
14. Rowan Davies. A practical refinement-type checker for standard ml. In *Sixth International Conference on Algebraic Methodology and Software Technology.*
15. E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, volume B, pages 995–1072. Elsevier Publishing, 1990.
16. E. Giménez. Structural recursive definitions in Type Theory. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1998.
17. Eduardo Giménez. Codifying guarded definitions with recursive schemes. Technical Report 95-07, Ecole Normale Supérieure de Lyon, 1994.
18. J. Goguen and R. Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(3):363–392, September 1994.
19. H. Hosoya, B. Pierce, and D.N. Turner. Subject reduction fails in Java. Message to the TYPES mailing list, 1998.
20. M.P. Jones. Dictionary-free overloading by partial evaluation. In *Proceedings of PEPM'94*, pages 107–117, 1994. University of Melbourne, Australia, Department of Computer Science, Technical Report 94/9.
21. G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.
22. Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 199x. To appear.
23. S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of ICFP'97*, pages 136–149. ACM Press, 1997.
24. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.
25. J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.
26. J. Peterson and K. Hammond (editors). *Haskell 1.4.: A Non-strict, Purely Functional Language*, April 1997.
27. F. Pfenning. Refinement types for logical frameworks. In H. Geuvers, editor, *Informal Proceedings of TYPES'93*, pages 285–299, 1993.
28. B.C. Pierce and D.N. Turner. Local type inference. In *Proceedings of POPL'98*, pages 252–265. ACM Press, 1998.
29. F. Pottier. *Synthèse de types en présence de sous-typage: de la théorie la pratique.* PhD thesis, Université Paris VII, 1998.
30. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
31. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99.* ACM Press, 1999. To appear.

Further examples

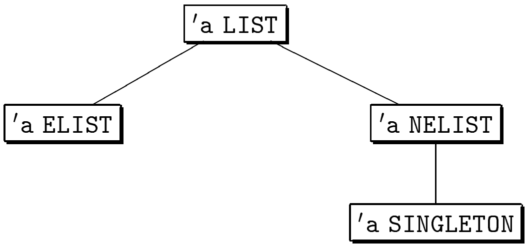
Example 27. The BOPL (Basic Object Programming Language), see [25]: this example describes the syntax of the *Basic Object Programming Language*. We use as datatype identifiers the non-terminal symbols of the grammar. In many situations we have to represent a list of something or an entity that may be a list of something or the emptiness. So we choose to declare initially the datatypes `'a List` and `'a List?`.

```
datatype 'a List  = nil
                  | cons of ('a * 'a List)
and      'a List? = epson
with     'a List ≤ 'a List? ;

datatype LETTER   = a | ... | z | A | ... | Z ;

datatype DIGIT    = 0 | ... | 9 ;

datatype ID       = idl of (ID * LETTER)
                  | idd of (ID * DIGIT)
with     LETTER ≤ ID ;

datatype INT      = int of (DIGIT * INT)
with     DIGIT ≤ INT ;

datatype BINOP    = '+' | '-' | '*' | '=' | and | or | '<' ;

datatype EXP      = op of (EXP * BINOP * EXP)
                  | false | true | nil | self
                  | not, class, () of EXP
                  | ':=' of (ID * EXP)
                  | ';' , while of (EXP * EXP)
                  | new of ID
                  | instof of (EXP * ID List)
                  | sendmes of (EXP * ID * EXP List)
with     INT ≤ EXP ,
         ID ≤ EXP ;

datatype DEC
with     ID List ≤ DEC ;

datatype FORMALS  = args of DEC List? ;

datatype VAR      = var of DEC ;

datatype METHOD   = method of (ID * FORMALS * EXP) ;

datatype CLASS    = class of (ID * VAR List? * METHOD List?)
                  | class_is of (ID * ID) ;

datatype PROGRAM  = prog of (CLASS List * EXP) ;
```

Example 28. CAM - Categorical Abstract Machine [21]

```
datatype VALUE   = i of INT | b of BOOL | null_value ;

datatype COM     = branch of (COMS * COMS)
                 | cur of COMS
                 | push | swap | app | rplac | op | car | cdr | cons
                 | quote of VALUE
and      COMS    = coms of COM List
with     COM ≤ COMS ;

datatype PROGRAM = program of COMS ;
```

Example 29. Here we return to the list datatype, declaring the following hierarchy:




```
datatype 'a ELIST    = Nil ;

datatype 'a LIST     = Nil
                     | Cons of ('a * 'a LIST)
                     | Cons of ('a * 'a ELIST) ;

datatype 'a NELIST   = Cons of ('a * 'a LIST)
                     | Cons of ('a * 'a ELIST) ;

datatype 'a SINGLETON= Cons of ('a * 'a ELIST) ;
```

Note that we declare the empty list type as a parametric datatype `'a ELIST` in order to have the subtyping relation `'a ELIST` $\leq$ `'a LIST`. We can conclude that `'a ELIST` $\leq$ `'a LIST`, `'a NELIST` $\leq$ `'a LIST`, `'a SINGLETON` $\leq$ `'a LIST` and `'a SINGLETON` $\leq$ `'a NELIST`. This example can be written more compactly as follows:

```
datatype 'a ELIST    = Nil
and      'a SINGLETON= Cons of ('a * 'a ELIST)
and      'a LIST     =
and      'a NELIST   = Cons of ('a * 'a LIST)
with     'a ELIST ≤ 'a LIST ,
         'a NELIST ≤ 'a LIST ,
         'a SINGLETON ≤ 'a NELIST ;
```

Note that, by transitivity, we have `'a SINGLETON` $\leq$ `'a LIST`.

Example 30. The definition of Harrop formulae. This example is adapted from [27].

```
datatype 'a Form   = conj of ('a Form * 'a Form)
                   | disj of ('a Form * 'a Form)
                   | imp of ('a Form * 'a Form)
and      'a Goal   = emb of 'a
                   | conj of ('a Goal * 'a Goal)
                   | disj of ('a Goal * 'a Goal)
                   | imp of ('a Prog * 'a Goal)
and      'a Prog   = emb of 'a
                   | conj of ('a Prog * 'a Prog)
                   | imp of ('a Goal * 'a Prog)
with     'a Goal ≤ 'a Form ;
```