Constructor subtyping

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 paper, 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. In the first part of the paper, we show that the calculus is confluent and strongly normalizing. In the second part of the paper, we 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. Finally, in the third part of the paper, we extend our calculus with unbounded recursion and show that confluence is preserved.

1 Introduction

Type systems [3, 8] lie at the core of modern functional programming languages, such as Haskell [28] or ML [26], and proof assistants, such as Coq [4] or PVS [32]. 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. [25, 30, 31], and in proof assistants, see e.g. [7, 24, 32].

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:

```
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 a previous paper [5], the first author 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;
3. we define a set of expansive extensionality rules, including η-expansion, surjective pairing, and a suitable expansion rule for datatypes, so as to obtain a well-behaved theory of canonical inhabitants (i.e. of closed expressions in normal forms). The latter is fundamental for a proper semantical understanding of the calculus and for several applications related to proof assistants, such as unification.

The main technical contribution of this paper is to show that the calculus enjoys several fundamental meta-theoretical properties including confluence, subject reduction, strong normalization and a well-behaved theory of canonical inhabitants. These results lay the foundations for constructor subtyping and open the possibility of using constructor subtyping in programming languages and proof assistants, see Section 7.

Organization of the paper The paper is organized as follows: in Section 2, we provide an informal account of constructor subtyping. In Section 3, we introduce a simply typed λ-calculus with record types and datatypes, and which supports both record subtyping and constructor subtyping. In Section 4, we establish some fundamental meta-theoretical properties of the calculus. In Section 5, we motivate the use of expansive extensionality rules, show that they preserve confluence and strong normalization and lead to a well-behaved theory of canonical inhabitants. In Section 6, we extend our core language with fixpoint operators, and show the resulting calculus to be confluent. Finally, we conclude in Section 7. Because of space constraints, proofs are merely sketched or omitted. We refer the reader to [6] for further details.

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.

```
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.

```
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`:

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

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

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

from the fact that `Odd` $\leq$ `Nat`, `Even` $\leq$ `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
```

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

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. 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 ;
```

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 ;
```

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 [29]. Yet another example of language that can be expressed with constructor semantics is mini-ML [22], 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 ;
```

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.

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

In this section, we introduce the core calculus $\lambda_{\rightarrow,[],\mathsf{data}}$. 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 $\mathcal{X}$.

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

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

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

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

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, $c_1 : \mathsf{nat} \rightarrow d$ and $c_2 : (\mathsf{nat} \rightarrow d) \rightarrow d$ are strictly positive w.r.t. d, whereas $c_3 : (d \rightarrow d) \rightarrow d$ 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 $\alpha \rightarrow d[\alpha]$, while it appears negatively in the domain of $(\alpha \rightarrow \mathsf{nat}) \rightarrow d[\alpha]$.
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 previously defined datatype identifiers $\mathcal{F}$, written $\aleph \vdash_k \sigma$ $\mathsf{pretype}(\mathcal{F})$, is defined by the rules of Figure 1.*

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

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

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

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

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

Fig. 1. Pre-type formation rules

Note that in (predata-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.*

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

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

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

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

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*

– $\mathsf{ar}(d) = \mathsf{ar}(d') = m$;

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

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

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

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

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

Fig. 3. Subtyping rules

– *every declaration* $c : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow d[\alpha_1, \ldots, \alpha_m]$ *in* $\aleph$ *is matched by another declaration* $c : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow d'[\alpha_1, \ldots, \alpha_m]$ *in* $\aleph$.

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

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

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

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

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

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

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

Fig. 4. Positive-Negative rules

Definition 6. $\mathsf{di}(D)$ *denote the set of datatype identifiers of D. It can be defined inductively as follows:*

1. $\mathsf{di}(.) = \emptyset$
2. $\mathsf{di}(c : \rho_1 \rightarrow \ldots \rightarrow \rho_n \rightarrow d[\boldsymbol{\alpha}]) = \{d\}$

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

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

Fig. 5. Strictly positive rules

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

Fig. 6. Constructor type rule

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

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

$$(main1) \qquad \frac{\aleph; c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}] \ \mathsf{ok}(\mathcal{F})}{\mathsf{main}_d(c : \rho_1 \to \ldots \to \rho_n \to d[\boldsymbol{\alpha}])}$$

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

Fig. 7. Main d-declaration rules

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

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

$$\aleph \vdash \tau' \ \mathsf{coty}(d)\mathcal{F} \ \wedge \ \mathsf{main}_d(c : \tau) \quad \Rightarrow \quad \aleph; D; \vdash \ \tau \le \tau'$$

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

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

As you may notice, the rules of Figure 8 introduce a new kind of judgment $\aleph; D \ \mathsf{ok}(\mathcal{F})$ which means that over the datatype context $\aleph$ we are constructing a new datatype D in a legal way. This judgment is very similar to the legal-judgment. The difference is that the ok-judgment works with an "open datatype".

Observe that Definitions 2-8 are mutually dependent. Note that Definitions 5 and 8 above are enforced by the side-conditions 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 $\aleph$.

(empty)	$.;\ \mathsf{legal}\emptyset$	
(close)	$\dfrac{\aleph; D\ \mathsf{ok}(\mathcal{F})}{\aleph; D;\ \mathsf{legal}\mathcal{F} \cup \mathsf{di}(D)}$,	$\aleph\ \mathsf{compatible}_{\mathcal{F}}(D)$
(add-cons)	$\dfrac{\aleph; D\ \mathsf{ok}(\mathcal{F}) \qquad \aleph \vdash \tau\ \mathsf{coty}(d)\mathcal{F}}{\aleph; D, c : \tau\ \mathsf{ok}(\mathcal{F})}$	
(add-data)	$\dfrac{\aleph\ \mathsf{legal}\mathcal{F}}{\aleph\ \mathsf{ok}(\mathcal{F})}$	

Fig. 8. Datatype rules

3.2 Expressions and typing

In this subsection, we conclude the definition of $\lambda_{\rightarrow,[],\mathsf{data}}$ 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 let $\mathcal{T}_0$ be the set of legal types in $\aleph$; finally $\sigma, \tau, \ldots$ are assumed to range over $\mathcal{T}_0$.

Definition 9. *The set $\mathcal{E}$ of* expressions *is given by the abstract syntax:*

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

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 10 (Typing).

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 $\tau_i \in \mathcal{T}_0$.*
2. *A* judgment *is a triple of the form $\Gamma \vdash a : \tau$, where Γ is a context, $a \in \mathcal{E}$ and $\tau \in \mathcal{T}_0$.*
3. *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 $\xi_1 \rightarrow \ldots \rightarrow \xi_n \rightarrow \sigma$ whenever $\tau = \xi_1 \rightarrow \ldots \rightarrow \xi_n \rightarrow d[\boldsymbol{\rho}]$.*
4. *An expression $a \in \mathcal{E}$ is* typable *if $\Gamma \vdash a : \sigma$ for some context Γ and type σ.*

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

Definition 11.

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

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

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

$$[l_1 = a_1, \ldots, l_n = a_n].l_i \rightarrow_{\pi} a_i$$

$$
\begin{array}{lcl}
\text{(start)} & \Gamma \vdash x : \tau & \text{if } x : \tau \in \Gamma \\[2ex]
\text{(application)} & \dfrac{\Gamma \vdash e : \tau \to \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \sigma} & \\[2ex]
\text{(abstraction)} & \dfrac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \sigma} & \\[2ex]
\text{(record)} & \dfrac{\Gamma \vdash e_i : \tau_i \quad (1 \leq i \leq n)}{\Gamma \vdash [l_1 = e_1, \ldots, l_n = e_n] : [l_1 : \tau_1, \ldots, l_n : \tau_n]} & \\[2ex]
\text{(select)} & \dfrac{\Gamma \vdash e : [l_1 : \tau_1, \ldots, l_n : \tau_n]}{\Gamma \vdash e.l_i : \tau_i} & \text{if } 1 \leq i \leq n \\[2ex]
\text{(constructor)} & \dfrac{\Gamma \vdash b_i : \rho_i\{\boldsymbol{\alpha} := \boldsymbol{\tau}\} \quad (1 \leq i \leq k)}{\Gamma \vdash c[\boldsymbol{\tau}]\, \boldsymbol{b} : d[\boldsymbol{\tau}]} & \text{if } c : \rho_1 \to \ldots \to \rho_k \to d[\boldsymbol{\alpha}] \in \aleph \\[2ex]
\text{(case)} & \dfrac{\Gamma \vdash a : d[\boldsymbol{\rho}] \quad \Gamma \vdash b_i : (\tau_i\{\boldsymbol{\alpha} := \boldsymbol{\rho}\})^{\sigma} \quad (1 \leq i \leq n)}{\Gamma \vdash \mathsf{case}^{\sigma}_{d[\boldsymbol{\rho}]}\, a \;\mathsf{of}\; \{c_1 \Rightarrow b_1 \mid \ldots \mid c_n \Rightarrow b_n\} : \sigma} & \\[2ex]
\text{(subsumption)} & \dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \sigma} & \text{if } \aleph \vdash \tau \leq \sigma
\end{array}
$$

Fig. 9. Typing rules

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 too weak an equational theory. However, the typing rules will enforce $\boldsymbol{\tau} \leq \boldsymbol{\tau'}$ on legal terms.

4 Meta-theory of the core language

In this section, we summarize some basic properties of the core language.

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.

Proposition 2 (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 structure of the derivations, using some basic properties of subtyping.

As usual, we say that an expression e is *strongly normalizing* with respect to a relation $\rightarrow$ 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 $\rightarrow$.

Proposition 3 (Strong normalization). *$\rightarrow_{basic}$ is strongly normalizing on typable expressions:*

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

Proof. By a standard computability argument.

We now turn to type-checking. One cannot 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

$$\begin{array}{rcl} \sigma \in \mathsf{min}_\Gamma(a) & \Rightarrow & \Gamma \vdash a : \sigma \\ \Gamma \vdash a : \sigma & \Rightarrow & \exists \tau \in \mathsf{min}_\Gamma(a).\ \tau \leq \sigma \end{array}$$

The set $\mathsf{min}_\Gamma(a)$, which is defined in the obvious way, is finite because there are only finitely many declarations for each constructor.

Proposition 4. *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 $\tau \leq \sigma$.

5 Extensionality

5.1 Motivations

Extensionality, as embodied e.g. in η-conversion, is a basic feature of many type systems. Traditionally, extensionality equalities are oriented as contractive rules: e.g. η-conversion is oriented as η-reduction. On the other hand, expansive rules provide an alternative computational interpretation of extensionality equalities: e.g. η-conversion may be oriented as η-expansion. Expansive extensionality rules have numerous applications in categorical rewriting, unification and partial evaluation. In addition to these traditional motivations, which are nicely summarized in [13], subtyping adds some new fundamental reasons to use expansive rules:

1. contractive rules lead to non-confluent calculi, even on well-typed expressions: if we adopt η-reduction for λ-abstractions, then the following critical pair cannot be solved:

$$\begin{tikzcd} & \lambda x{:}\tau.\ (\lambda y{:}\sigma.\ y)\ x \arrow[dl, "\beta"'] \arrow[dr, "\eta"] & \\ \lambda x{:}\tau.\ x & & \lambda y{:}\sigma.\ y \end{tikzcd}$$

   On the other hand, $\lambda x{:}\tau.\ (\lambda y{:}\sigma.\ y)\ x$ is well-typed (of type $\tau \rightarrow \sigma$) whenever $\tau \leq \sigma$ (this observation is due to Mitchell, Hoang and Howard [27]). A similar remark applies to datatypes: if we adopt μ-reduction for lists, as defined by

$$\mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ e\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \lambda a{:}\tau.\ \lambda l{:}\mathsf{list}[\tau].\ \mathsf{cons}[\tau]\mathsf{a}\,\mathsf{l}\} \rightarrow_\mu \mathsf{e}$$

then the following critical pair cannot be solved:

$$\begin{tikzcd} & M \arrow[dl, "\iota"'] \arrow[dr, "\mu"] & \\ \mathsf{nil}[\tau] & & \mathsf{nil}[\sigma] \end{tikzcd}$$

where $M \equiv \mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ (\mathsf{nil}[\sigma])\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \lambda a{:}\tau.\ \lambda l{:}\mathsf{list}[\tau].\ \mathsf{cons}[\tau]\mathsf{a}\,\mathsf{l}\}$.
On the other hand, $\mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ (\mathsf{nil}[\sigma])\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \lambda a : \tau.\ \lambda l : \mathsf{list}[\tau].\ \mathsf{cons}[\tau]\mathsf{a}\,\mathsf{l}\}$ is well-typed (of type $\mathsf{list}[\tau]$) whenever $\sigma \leq \tau$.

2. contractive rules lead to calculi with too many canonical inhabitants (i.e. closed expressions in normal form): if we adopt μ-reduction for lists then the following expressions are canonical inhabitants of $\mathsf{list}[\tau]$, provided $\sigma \leq \tau$, $a : \sigma$ and $l : \mathsf{list}[\sigma]$:

$$\mathsf{nil}[\sigma] \qquad \mathsf{nil}[\tau] \qquad \mathsf{cons}[\sigma]a\ l \qquad \mathsf{cons}[\tau]a\ l$$

   On the other hand, one would expect canonical inhabitants of $\mathsf{list}[\tau]$ to be of the form

$$\mathsf{nil}[\tau] \qquad \mathsf{cons}[\tau]a\ l$$

   where in the second case l itself is a canonical inhabitant of $\mathsf{list}[\tau]$ and a is a canonical inhabitant of τ. Remarkably we obtain the desired effect if we reverse μ-reduction. With this new reduction rule, which we call μ-expansion and denote by $\rightarrow_{\overline{\mu}}$, we have:

$$\begin{aligned} \mathsf{nil}[\sigma] &\rightarrow_{\overline{\mu}} \mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ \mathsf{nil}[\sigma]\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \mathsf{cons}[\tau]\} \\ &\rightarrow_{\iota} \mathsf{nil}[\tau] \end{aligned}$$

   Similarly, for $a : \sigma$ and $l : \mathsf{list}[\sigma]$, one has:

$$\begin{aligned} \mathsf{cons}[\sigma]a\ l &\rightarrow_{\overline{\mu}} \mathsf{case}^{\mathsf{list}[\tau]}_{\mathsf{list}[\tau]}\ (\mathsf{cons}[\sigma]a\ l)\ \mathsf{of}\ \{\mathsf{nil} \Rightarrow \mathsf{nil}[\tau] \mid \mathsf{cons} \Rightarrow \mathsf{cons}[\tau]\} \\ &\rightarrow_{\iota} \mathsf{cons}[\tau]\ a\ l \end{aligned}$$

   (Strictly speaking, expansive extensionality rules are defined relative to a context and a type and the above reductions are performed at type $\mathsf{list}[\tau]$);

3. expansive rules provide a simple but useful program optimization: if we adopt expansive rules for records, the expression $[n = 3, c = \mathsf{blue}]$ reduces at type $[n : \mathsf{nat}]$ to $[n = 3]$, thus throwing out the irrelevant fields at type $[n : \mathsf{nat}]$.

We therefore embark upon studying an expansive interpretation of extensionality in $\lambda_{\rightarrow,[],\mathsf{data}}$.

5.2 Expansive extensionality rules

The computational behavior of the calculus is now obtained by aggregating the expansive extensionality rules to $\rightarrow_{basic}$. Expansive extensionality rules need to be formulated in a typed framework so we consider judgments of the form

$$\Gamma \vdash a \rightarrow b : \sigma$$

For the sake of uniformity, we first reformulate $\rightarrow_{basic}$ in a typed framework.

Definition 12.

1. Typed *basic*-reduction $\rightarrow_{basic}$ *is defined by the clause*

$$\Gamma \vdash a \rightarrow_{basic} b : \sigma$$

   iff $\Gamma \vdash a : \sigma$ *and* $a \rightarrow_{basic} b$.

2. η-expansion $\rightarrow_{\overline{\eta}}$ *is defined as the quasi-compatible closure (see below) of the rule*

$$\Gamma \vdash a \rightarrow_{\overline{\eta}} \lambda x{:}\tau.\, a\, x : \tau \rightarrow \sigma$$

provided $a \neq \lambda x{:}\tau.\, b$. *The usual rule*

$$\frac{\Gamma \vdash a \rightarrow_{\overline{\eta}} b : \tau \rightarrow \sigma \qquad \Gamma \vdash c : \tau}{\Gamma \vdash a\ c \rightarrow_{\overline{\eta}} b\ c : \sigma}$$

is only allowed under the proviso $b \neq \lambda x{:}\tau.\, a\, x$.
3. Surjective pairing $\rightarrow_{sp}$ *is defined as the quasi-compatible closure (see below) of the rule*

$$\Gamma \vdash a \rightarrow_{sp} [l_1 = a.l_1,\ \ldots\ , l_n = a.l_n] : [l_1 : \tau_1,\ \ldots\ , l_n : \tau_n]$$

provided $a \neq [l_1 = a_1,\ \ldots\ , l_n = a_n]$. *The usual rule*

$$\frac{\Gamma \vdash a \rightarrow_{sp} b : [l_1 : \tau_1,\ \ldots\ , l_n : \tau_n]}{\Gamma \vdash a.l_i \rightarrow_{sp} b.l_i : \tau_i}$$

is only allowed under the proviso $b \neq [l_1 = a.l_1,\ \ldots\ , l_n = a.l_n]$.
4. μ-expansion $\rightarrow_{\overline{\mu}}$ *is defined as the quasi-compatible closure (see below) of the rule*

$$\Gamma \vdash a \rightarrow_{\overline{\mu}} \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} : d[\boldsymbol{\tau}]$$

provided $a \neq c_i[\boldsymbol{\tau}]\boldsymbol{b}$ *and* $a \neq \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a'\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\}$. *The usual rule*

$$\frac{\Gamma \vdash a \rightarrow_{\overline{\mu}} a' : d[\boldsymbol{\tau}]}{\Gamma \vdash \mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{b}\} \rightarrow_{\overline{\mu}} \mathsf{case}^{\sigma}_{d[\boldsymbol{\tau}]}\ a'\ \mathsf{of}\ \{\boldsymbol{c} \Rightarrow \boldsymbol{b}\} : \sigma}$$

is only allowed under the proviso $a' \neq \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\}$.
5. Typed $full$-reduction $\rightarrow_{full}$ *is defined as the union of* $basic, \overline{\eta}, sp, \overline{\mu}$*-reduction, i.e.*

$$\Gamma \vdash a \rightarrow_{full} b : \sigma \quad \Leftrightarrow \quad \Gamma \vdash a \rightarrow_{basic,\overline{\eta},sp,\overline{\mu}} b : \sigma$$

6. $\twoheadrightarrow_{full}$ *and* $=_{full}$ *are respectively defined as the reflexive-transitive and the reflexive-symmetric-transitive closures of* $\rightarrow_{full}$.

Several points deserve attention:

1. the various restrictions on $\rightarrow_{\overline{\eta}}$, $\rightarrow_{sp}$ and $\rightarrow_{\overline{\mu}}$ are required to enforce strong normalization. Without those restrictions, one would have loops or infinite reductions, see the appendix.
2. unlike the traditional formulations of η-expansion, we do allow η-expansions on λ-abstractions at type $\tau \rightarrow \sigma$ if the type of the variable is not τ. Such a possibility is indeed crucial for expressions of type $\sigma \rightarrow \tau$ to reduce to an expression of the form $\lambda x{:}\sigma.\, e$ at that type. On the other hand, note that η-expansion as defined here does not preserve $\rightarrow_{basic}$-normal forms. For example, for $\tau \leq \sigma$,

$$\vdash \lambda x{:}\sigma.\, x : \tau \rightarrow \sigma$$

is in $\rightarrow_{basic}$-normal form but

$$\begin{aligned}\vdash \lambda x{:}\sigma.\, x &\rightarrow_{\overline{\eta}} \lambda z{:}\tau.\, (\lambda x{:}\sigma.\, x)\ z\ : \tau \rightarrow \sigma \\ &\rightarrow_{\beta} \lambda z{:}\tau.\, z\end{aligned}$$

A similar remark applies to records and case-expressions.

3. $\rightarrow_{\overline{\mu}}$-like rules for datatypes seem to have received very little attention in the literature. As far as we know, only Ghani [16] proposes a possible such rule (his rule is motivated by categorical considerations) but does not study it in detail. Our expansion rule for datatypes is weaker than the one suggested by Ghani [16] and thus is inadequate to capture the categorical view of datatypes as initial algebras in a suitable category. It nevertheless serves its purpose, see Proposition 7.
4. reduction is not preserved under subsumption: that is, one may have

$$\Gamma \vdash a \rightarrow_{full} b : \sigma \quad \wedge \quad \Gamma \not\vdash a \rightarrow_{full} b : \tau$$

for $\sigma \leq \tau$. On the other hand,

$$\Gamma \vdash a \rightarrow_{full} b : \sigma \quad \Rightarrow \quad \Gamma \vdash a =_{full} b : \tau$$

for $\sigma \leq \tau$.

5.3 Preservation of confluence and strong normalization

Expansive extensionality rules preserve the fundamental properties of $\lambda_{\rightarrow,[],\mathsf{data}}$.

Proposition 5 (Strong normalization). *The relation $\rightarrow_{full}$ is strongly normalizing on typable expressions.*

Proof. By modifying, along the lines of e.g. [20], the computability argument of Theorem 3.

Proposition 6 (Confluence). *The relation $\rightarrow_{full}$ is confluent on typable expressions.*

Proof. Using Newman's Lemma, strong normalization and weak confluence, which is proved by a case analysis on the possible critical pairs.

5.4 Theory of canonical inhabitants

Below we write $\Gamma \vdash^{nf} a : \tau$ if $\Gamma \vdash a : \tau$ and there is no $b \in \mathcal{E}$ such that $\Gamma \vdash a \rightarrow_{full} b : \tau$. The following result shows that the theory of canonical inhabitants is well-behaved, i.e. that typable closed expressions in normal form have the expected shape.

Proposition 7. *Assume that $\Gamma \vdash^{nf} a : \tau$.*

1. *If $\tau = \sigma \rightarrow \rho$, then $a = \lambda x{:}\sigma.\ b$;*
2. *If $\tau = [l_1 : \sigma_1, \ \ldots \ , l_n : \sigma_n]$, then $a = [l_1 = b_1, \ \ldots \ , l_n = b_n]$.*
3. *If $\tau = d[\boldsymbol{\sigma}]$, then $a = c[\boldsymbol{\sigma}]\boldsymbol{b}$.*

Proof. By a case analysis on the possible normal forms.

The above result may be seen as evidence that the $\overline{\eta}, sp, \overline{\mu}$-rules restore a semantical justification of the system, and in particular of the case-expressions: as every canonical inhabitant of $d[\boldsymbol{\tau}]$ is of the form $c[\boldsymbol{\tau}]\boldsymbol{b}$, it is justified to do pattern-matching on c.

6 Adding fixpoints

$\lambda_{\rightarrow,[],\mathsf{data}}$ has a very restricted computational power. In particular, it does not support recursion. In this section, we study an extension of $\lambda_{\rightarrow,[],\mathsf{data}}$ with fixpoints, and show the resulting calculus to be confluent.

Definition 13.

1. *The set of expressions $\mathcal{E}$ is extended with the clause* $\mathsf{fix}\ x{:}\tau.\,a$.
2. *Fixpoint reduction $\rightarrow_{rec}$ is defined as the compatible closure of the rule*

$$\mathsf{fix}\ x{:}\tau.\,a \rightarrow_{rec} a\{x := \mathsf{fix}\ x{:}\tau.\,a\}$$

3. *The typing system is extended with the rule:*

$$\frac{\Gamma, x : \tau \ \vdash\ a : \tau}{\Gamma\ \vdash\ \mathsf{fix}\ x{:}\tau.\,a : \tau}$$

4. *We let $\rightarrow_{full+rec}$ denote $\rightarrow_{full} \cup \rightarrow_{rec}$.*

We have:

Proposition 8. *The relation $\rightarrow_{full+rec}$ is confluent on typable expressions.*

Proof. Using a standard technique due to Lévy [23], and exploited e.g. in [14]. The idea is to introduce bounded fixpoints, show that the calculus remains strongly normalizing and confluent, and then use some elementary reasoning on abstract reduction systems to conclude that $\rightarrow_{full+rec}$ is confluent.

Obviously, $\rightarrow_{full+rec}$ is not strongly normalizing. In order to preserve strong normalization, one must restrict oneself to *guarded* fix-expressions. Technically, it is achieved by defining the notion of an expression e being guarded, and by adding the side-condition a is guarded in the typing rule for fixpoints. A precise description of the guard mechanism may be found for example in [17].

7 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 particular with respect to canonical inhabitants.

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 $*$:

$$\begin{aligned} * &= *_1 : \mathbb{N} \rightarrow \mathbb{E} \rightarrow \mathbb{E} \\ &= *_2 : \mathbb{E} \rightarrow \mathbb{N} \rightarrow \mathbb{E} \\ &= *_3 : \mathbb{O} \rightarrow \mathbb{O} \rightarrow \mathbb{O} \\ &= *_4 : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \end{aligned}$$

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, 21], 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 [33]. 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/Coinductive Constructions, see e.g. [17], with constructor subtyping. As suggested in [5, 12], such a calculus seems particularly appropriate to formalize Kahn's natural semantics [22].

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*, 199x. To appear.
6. G. Barthe and M.J. Frade. Constructor subtyping. Technical Report UMDITR9807, Department of Computer Science, University of Minho, 1998.
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. R. Di Cosmo. A brief history of rewriting with extensionality. Presented at the International Summer School on Type Theory and Term Rewriting, Glasgow, September 1996.
14. R. Di Cosmo and D. Kesner. Simulating expansions without expansions. *Mathematical Structures in Computer Science*, 4(3):315–362, September 1994.
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. N. Ghani. *Adjoint rewriting.* PhD thesis, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1995.

17. 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.
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. C.B. Jay and N. Ghani. The virtues of eta-expansion. *Journal of Functional Programming*, 5(2):135–154, April 1995.
21. 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.
22. 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.
23. J.-J. Lévy. An algebraic interpretation of the $\lambda\beta\kappa$-calculus and a labelled λ-calculus. *Theoretical Computer Science*, 2:97–114, 1976.
24. Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 199x. To appear.
25. S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of ICFP'97*, pages 136–149. ACM Press, 1997.
26. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
27. J. C. Mitchell, M. Hoang, and B. T. Howard. Labelling techniques and typed fixed-point operators. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 137–174. Cambridge University Press, 1998.
28. J. Peterson and K. Hammond (editors). *Haskell 1.4.: A Non-strict, Purely Functional Language*, April 1997.
29. F. Pfenning. Refinement types for logical frameworks. In H. Geuvers, editor, *Informal Proceedings of TYPES'93*, pages 285–299, 1993.
30. B.C. Pierce and D.N. Turner. Local type inference. In *Proceedings of POPL'98*, pages 252–265. ACM Press, 1998.
31. F. Pottier. *Synthèse de types en présence de sous-typage: de la théorie la pratique.* PhD thesis, Université Paris VII, 1998.
32. 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.
33. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of POPL'99*. ACM Press, 1999. To appear.

Loops and infinite reductions for unrestricted extensionality rules

For η-expansion:

$$\begin{aligned} \Gamma \vdash\ a\ c &\to_{\overline{\eta}} (\lambda x{:}\tau.\ a\ x)\ c\ : \tau \to \sigma \\ &\to_{\beta} a\ c \end{aligned}$$

For surjective pairing (we treat the case where $a : [l : \tau, l' : \sigma]$ but a similar remark applies to arbitrary records):

$$\begin{aligned} \Gamma \vdash\ a.l &\to_{sp} [l = a.l, l' = a.l'].l\ : \tau \\ &\to_{\pi}\ a.l \end{aligned}$$

For μ-expansion (if we allow constructors to be expanded):

$$\begin{aligned} \Gamma \vdash\ (c_i[\boldsymbol{\tau}]\ \boldsymbol{b}) &\to_{\overline{\mu}} \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ (c_i[\boldsymbol{\tau}]\ \boldsymbol{b})\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \ldots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\}\ : d[\boldsymbol{\tau}] \\ &\to_{\iota}\ c_i[\boldsymbol{\tau}]\ \boldsymbol{b} \end{aligned}$$

and (if we allow case-expressions to be expanded):

$$\begin{aligned}
\Gamma \vdash\ & \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \dots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} && : d[\boldsymbol{\tau}] \\
& \quad \to_{\overline{\mu}}\ \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a_1\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \dots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} \\
& \quad \to_{\overline{\mu}}\ \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a_2\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \dots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} \\
& \quad \to_{\overline{\mu}}\ \dots
\end{aligned}$$

where $a_0 = a$ and

$$a_{i+1} = \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a_i\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \dots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\}$$

and (if we take the compatible closure of $\overline{\mu}$):

$$\begin{aligned}
\Gamma \vdash\ a & \to_{\overline{\mu}} \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \dots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} \quad : d[\boldsymbol{\tau}] \\
& \to_{\overline{\mu}} \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a_1\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \dots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} \\
& \to_{\overline{\mu}} \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a_2\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \dots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\} \\
& \to_{\overline{\mu}} \dots
\end{aligned}$$

where $a_0 = a$ and

$$a_{i+1} = \mathsf{case}^{d[\boldsymbol{\tau}]}_{d[\boldsymbol{\tau}]}\ a_i\ \mathsf{of}\ \{c_1 \Rightarrow c_1[\boldsymbol{\tau}] \mid\ \dots\ \mid c_n \Rightarrow c_n[\boldsymbol{\tau}]\}$$