A Type Checker for a Logical Framework with Union and Intersection Types

Claude Stolze
IRIF, Université de Paris, France
Claude.Stolze@irif.fr

Luigi Liquori
Université Côte d'Azur, Nice, France
Inria Sophia Antipolis - Méditerranée, France
Luigi.Liquori@inria.fr

Abstract

We present the syntax, semantics, typing, subtyping, unification, refinement, and REPL of Bull, a prototype theorem prover based on the Δ-Framework, i.e. a fully-typed Logical Framework à la Edinburgh LF decorated with union and intersection types, as described in previous papers by the authors. Bull also implements a subtyping algorithm for the Type Theory Ξ of Barbanera-Dezani-de'Liguoro. Bull has a command-line interface where the user can declare axioms, terms, and perform computations and some basic terminal-style features like error pretty-printing, subexpressions highlighting, and file loading. Moreover, it can typecheck a proof or normalize it. These terms can be incomplete, therefore the typechecking algorithm uses unification to try to construct the missing subterms. Bull uses the syntax of Berardi's Pure Type Systems to improve the compactness and the modularity of the kernel. Abstract and concrete syntax are mostly aligned and similar to the concrete syntax of Coq. Bull uses a higher-order unification algorithm for terms, while typechecking and partial type inference are done by a bidirectional refinement algorithm, similar to the one found in Matita and Beluga. The refinement can be split into two parts: the essence refinement and the typing refinement. Binders are implemented using commonly-used de Bruijn indices. We have defined a concrete language syntax that will allow user to write Δ-terms. We have defined the reduction rules and an evaluator. We have implemented from scratch a refiner which does partial typechecking and type reconstruction. We have experimented Bull with classical examples of the intersection and union literature, such as the ones formalized by Pfenning with his Refinement Types in LF and by Pierce. We hope that this research vein could be useful to experiment, in a proof theoretical setting, forms of polymorphism alternatives to Girard's parametric one.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Proof theory

Keywords and phrases Intersection types, Union types, Dependent types, Subtyping, Type checker, Refiner, Δ-Framework

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.37

Category System Description

Supplementary Material BULL: `https://github.com/cstolze/Bull`

Funding Work supported by the COST Action CA15123 EUTYPES "The European research network on types for programming and verification"

Acknowledgements This work could not be have be done without the many useful discussions with Furio Honsell, Ivan Scagnetto, Ugo de'Liguoro, Daniel Dougherty, and the Anonymous Reviewers.

1 Introduction

This paper provides a unifying framework for two hitherto unreconciled understandings of types: i.e. types-as-predicates à la Curry and types-as-propositions à la Church. The key to our unification consists in introducing, implementing and experimenting *strong proof-*

functional connectives [40, 2, 3] in a dependent type theory such as the Edinburgh Logical Framework (LF) [21]. Both Logical Frameworks and Proof-Functional Logic consider proofs as first-class citizens, albeit differently.

Strong proof-functional connectives take seriously into account the shape of logical proofs, thus allowing for polymorphic features of proofs to be made explicit in formulæ. Hence they provide a finer semantics than classical/intuitionistic connectives, where the meaning of a compound formula depends only on the *truth value* or the *provability* of its subformulæ. However, existing approaches to strong proof-functional connectives are all quite idiosyncratic in mentioning proofs. Existing Logical Frameworks, on the other hand, provide a uniform approach to proof terms in object logics, but they do not fully capitalize on subtyping.

This situation calls for a natural combination of the two understandings of types, which should benefit both worlds. On the side of Logical Frameworks, the expressive power of the metalanguage would be enhanced thus allowing for shallower encodings of logics, a more principled use of subtypes [36], and new possibilities for formal reasoning in existing interactive theorem provers. On the side of type disciplines for programming languages, a principled framework for proofs would be provided, thus supporting a uniform approach to "proof reuse" practices based on type theory [14, 38, 11, 20, 8].

Therefore, in [25] we extended LF with the connectives of *strong intersection* (corresponding to intersection types [4, 5]) and *strong union* (corresponding to union types [31, 2]) of Proof-Functional Logic [40]. We called this extension the Δ-Framework (LF_{Δ}), since it builds on the Δ-calculus [28]. As such, LF_{Δ} subsumes many expressive type disciplines in the literature [36, 2, 3, 38, 11].

It is not immediate to extend the Curry-Howard isomorphism to logics supporting strong proof-functional connectives, since these connectives need to compare the shapes of derivations and do not just take into account the provability of propositions, i.e. the inhabitation of the corresponding type. In order to capture successfully strong logical connectives such as $\cap$ or $\cup$, we need to be able to express the rules:

$$\frac{\mathcal{D}_1 : A \quad \mathcal{D}_2 : B \quad \mathcal{D}_1 \equiv_{\mathcal{R}} \mathcal{D}_2}{A \cap B}\ (\cap I) \qquad \frac{\mathcal{D}_1 : A \supset C \quad \mathcal{D}_2 : B \supset C \quad A \cup B \quad \mathcal{D}_1 \equiv_{\mathcal{R}} \mathcal{D}_2}{C}\ (\cup E)$$

where $\equiv_{\mathcal{R}}$ is a suitable equivalence between logical proofs. Notice that the above rules suggest immediately intriguing applications in polymorphic constructions, i.e. the *same evidence* can be used as a proof for different statements.

Pottinger [40] was the first to study the strong connective $\cap$. He contrasted it to the intuitionistic connective $\wedge$ as follows: *"The intuitive meaning of $\cap$ can be explained by saying that to assert $A \cap B$ is to assert that one has a reason for asserting A which is also a reason for asserting B [while] to assert $A \wedge B$ is to assert that one has a pair of reasons, the first of which is a reason for asserting A and the second of which is a reason for asserting B".*

A logical theorem involving intuitionistic conjunction which does not hold for strong conjunction is $(A \supset A) \wedge (A \supset B \supset A)$, otherwise there should exist a closed λ-term having simultaneously both one and two abstractions. López-Escobar [29] and Mints [33] investigated extensively logics featuring both strong and intuitionistic connectives especially in the context of *realizability* interpretations.

Dually, it is in the $\cup$-elimination rule that proof equality needs to be checked. Following Pottinger, we could say that *asserting $(A \cup B) \supset C$ is to assert that one has a reason for $(A \cup B) \supset C$, which is also a reason to assert $A \supset C$ and $B \supset C$.* The two connectives differ since the intuitionistic theorem $((A \supset B) \vee B) \supset A \supset B$ is not derivable for $\cup$, otherwise there would exist a term which behaves both as $\mathbf{I}$ and as $\mathbf{K}$.

Strong connectives arise naturally in investigating the propositions-as-types analogy for intersection and union type assignment systems. From a logical point of view, there are many proposals [33, 36, 47, 42, 34, 10, 9, 39, 19, 18] to find a suitable logic to fit intersection and union: we also refer to [15, 25, 43] for a detailed discussion.

The LF_Δ Logical Framework introduced in [25] extends [28] with union types and dependent types. The novelty of LF_Δ in the context of Logical Frameworks, lies in the full-fledged use of strong proof-functional connectives, which to our knowledge has never been explored before. Clearly, all Δ-terms have a computational counterpart.

Useful applications rely on using Proof-Functional Logics (that care about the proof-derivation), instead of Truth-Functional Logics (that care about the validity of the conclusion). In a nutshell: *i*) Intersection-types better catch the class of strongly normalizing terms than $F\omega$ (see, among others, [46]). *ii*) Union and intersection types allow for a little form of abstract interpretation (see Pierce $IsZero$ example [38, 2]) that cannot easily be encoded in $LF\omega$. *iii*) Proof-Functional Logics introduces a notion of "proof-synchronization", as showed in the operational semantics of the Δ-calculus, see Definition 4.3 and 5 of [28]: two proofs are "in-sync" iff their "untyped essences" are equals. This could be possibly exploited in a synchronized tactic in Bull. *iv*) Union types can capture a natural form of parallelism that could be conveniently put to use in formalizing reasoning on concurrent execution, as in proving correctness of concurrency control protocols or in branch-prediction techniques.

This paper presents the implementation of Bull [44, 43], an Interactive Theorem Prover (ITP) based on the Δ-Framework [45, 25]. The first author wrote this theorem prover from scratch for three years. Bull have a command-line interface program where the user can declare axioms, terms, and perform computations. These terms can be incomplete, therefore the typechecking algorithm uses unification to try to construct the missing subterms.

We have designed and implemented a novel subtyping algorithm [27] which extends the well-known algorithm for intersection types, designed by Hindley [23], with union types. Our subtyping algorithm has been mechanically proved correct in Coq and extracted in OCaml, extending the proof of a subtyping algorithm for intersection types of Bessai et al. [7].

We have implemented several features. A *Read-Eval-Print-Loop* (REPL) allows to define axioms and definitions, and performs some basic terminal-style features like error pretty-printing, subexpressions highlighting, and file loading. Moreover, it can typecheck a proof or normalize it. We use the Berardi's syntax of Pure Type Systems [6] to improve the compactness and the modularity of the kernel. Abstract and concrete syntax are mostly aligned: the concrete syntax is similar to the concrete syntax of Coq.

We have designed a *higher-order unification algorithm* for terms, while typechecking and partial type inference are done by our *bidirectional refinement algorithm*, similar to the one found in Matita [1]. The refinement can be split into two parts: the essence refinement and the typing refinement. The bidirectional refinement algorithm aims to have partial type inference, and to give as much information as possible to the unifier. For instance, if we want to find a $?y$ such that $\vdash_\Sigma \langle \lambda x{:}\sigma.x, \lambda x{:}\tau.?y\rangle : (\sigma \to \sigma) \cap (\tau \to \tau)$, then we can infer that $x{:}\tau \vdash ?y : \tau$ and that $\wr\, ?y \,\wr =_\beta x$.

This paper is organized as follows: in Section 2, we introduce the language we have implemented. In Section 3, we define the reduction rules and explain the evaluation process. In Section 4, we present the subtyping algorithm. In Section 5, we present the unifier. In Section 6, we present the refiner which does partial typechecking and type reconstruction. In Section 7, we present the REPL. In Section 8, we present possible enhancements of the type theory and of the ITP. Appendix contains interesting encodings that could be typechecked in Bull and could help the reader to understand the usefulness of adding ad hoc polymorphism and proof-functional operators to LF.

2 Syntax of terms

The syntax for the logical framework we have designed and implemented is as follows:

$$
\begin{array}{rcll}
\Delta, \sigma & ::= & s \ , \ c \ , \ v \ , \ \_ \ , \ ?x[\Delta; ...; \Delta] & \text{Sorts, Const, Vars, Placeholders and Metavars} \\
& | & \textbf{let}\ x{:}\sigma := \Delta\ \textbf{in}\ \Delta & \text{Local definition} \\
& | & \Pi x{:}\sigma.\Delta \ , \ \lambda x{:}\sigma.\Delta \ , \ \Delta\, S & \Pi\text{- and } \lambda\text{-abstraction and application} \\
& | & \sigma \cap \sigma \ , \ \sigma \cup \sigma & \text{Intersection and Union types} \\
& | & \langle \Delta, \Delta \rangle \ , \ \mathsf{pr}_1\, \Delta \ , \ \mathsf{pr}_2\, \Delta & \text{Strong pair and Left/Right projections} \\
& | & \textbf{smatch}\ \Delta\ \textbf{return}\ \sigma\ \textbf{with}\ [x{:}\sigma \Rightarrow \Delta \mid x{:}\sigma \Rightarrow \Delta] & \text{Strong sum} \\
& | & \mathsf{in}_1\, \sigma\, \Delta \ , \ \mathsf{in}_2\, \sigma\, \Delta \ , \ \textbf{coe}\, \sigma\, \Delta & \text{Left/Right injections and Coercions} \\
S & ::= & () \mid (S;\Delta) & \text{Typed Spines}
\end{array}
$$

By using a Pure Type System approach [6], all the terms are read through the same parser. The main differences with the Δ-Framework [25] are the additions of a placeholder and meta-variables, used by the refiner. We also added a **let** operator and changed the syntax of the strong sum **smatch** so it looks more like the concrete syntax used in the implementation. A meta-variable $?x[\Delta_1; ...; \Delta_n]$ has the, so called, *suspended substitutions* $\Delta_1; ...; \Delta_n$, which will be explained clearly in Subsection 2.4. Finally, following the Cervesato-Pfenning jargon [12], applications are in *spine form*, i.e. the arguments of a function are stored together in a list, exposing the head of the term separately. We also implemented a corresponding syntax for the untyped counterpart of the framework, called *essence* [28], where all pure λ-terms M and spines are defined as follows:

$$
\begin{array}{rcll}
M, \varsigma & ::= & s \ , \ c \ , \ x \ , \ \_ \ , \ ?x[M; ...; M] & \text{Sorts, Const, Vars, Placeholders and Metavars} \\
& | & \textbf{let}\ x := M\ \textbf{in}\ M & \text{Local definition} \\
& | & \Pi x{:}\varsigma.\varsigma \ , \ \lambda x.M \ , \ M\, R & \Pi\text{- and } \lambda\text{-abstraction and application} \\
& | & \varsigma \cap \varsigma \ , \ \varsigma \cup \varsigma & \text{Intersection and Union types} \\
R & ::= & () \mid (R;M) & \text{Untyped Spines}
\end{array}
$$

Note that essences of types (ς) belongs to the same syntactical set as essences of terms.

2.1 Concrete syntax

The concrete syntax of the terms has been implemented with OCamllex and OCamlyacc. Its simplified syntax is as follows:

```
term ::= Type                                  # type
| let ID [args] [: term] := term in term      # let
| ID                                 # variables and constants
| forall args, term                  # dependent product
| term -> term                       # non-dependent product
| fun args => term                   # lambda-abstraction
| term term                          # application
| term & term                        # intersection of types
| term | term                        # union of types
| <term,term>                        # strong pair
| proj_l term                        # left  projection of a strong pair
| proj_r term                        # right projection of a strong pair
| smatch term [as ID] [return term] with ID [: term] => term, ID [: term] => term end
| inj_l term term                    # left  injection of a strong sum        # strong sum
| inj_r term term                    # right injection of a strong sum
| coe term term                      # coercion
| _                                  # wildcard
```

Identifiers `ID` refers to any alphanumeric string (possibly with underscores and apostrophes). The non-terminal symbol `args` correspond to a non-empty sequence of arguments, where an argument is an identifier, and can be given with its type. In the latter case, you should parenthesize it, for instance `(x : A)`, and if you want to assign the same type to several consecutive arguments, you can e.g. write `(x y z : A)`. Strong sums have a complicated syntax. For instance, consider this term:

```
smatch foo as x return T with y : T1 ⇒ bar, z : T2 ⇒ baz end
```

The above term in the concrete syntax corresponds to
smatch `foo` **return** λx:_.`T` **with** [y:`T1` ⇒ `bar` | z:`T2` ⇒ `baz`]
in the abstract syntax. The concrete syntax thus guarantees that the returned type is a λ-abstraction, and it allows a simplified behaviour of the type reconstruction algorithm. The behaviour of the concrete syntax is intended to mimic Coq.

2.2 Implementation of the syntax

In the OCaml implementation, Δ-terms and their types along with essences and type essences are represented with a single type called `term`. It allows some functions (such as the normalization function) to be applied both on Δ-terms and on essences.

```
type term =
  | Sort of location * sort
  | Let of location * string * term * term * term (* let s : t1 := t2 in t3 *)
  | Prod of location * string * term * term (* forall s : t1, t2 *)
  | Abs of location * string * term * term (* fun s : t1 => t2 *)
  | App of location * term * term list (* t t1 t2 ... tn *)
  | Inter of location * term * term (* t1 & t2 *)
  | Union of location * term * term (* t1 | t2 *)
  | SPair of location * term * term (* < t1, t2 > *)
  | SPrLeft of location * term (* proj_l t1 *)
  | SPrRight of location * term (* proj_r t1 *)
  | SMatch of location * term * term * string * term * term * string * term * term
                  (* match t1 return t2 with s1 : t3 => t4 , s2 : t5 => t6 end *)
  | SInLeft of location * term * term (* inj_l t1 t2 *)
  | SInRight of location * term * term (* inj_r t1 t2 *)
  | Coercion of location * term * term (* coe t1 t2 *)
  | Var of location * int (* de Bruijn index *)
  | Const of location * string (* variable name *)
  | Underscore of location (* meta-variables before analysis *)
  | Meta of location * int * (term list) (* index and substitution *)
```

The constructors of `term` contain the location information retrieved by the parser that allows the typechecker to give the precise location of a subterm to the user, in case of error.

The `App` constructor takes as parameters the applied function and the list of all the arguments. The list of parameters is used as a stack, hence the rightmost argument is the head of the list, and can easily be removed in the OCaml recursive functions. The variables are referred to as strings in the `Const` constructor, and as de Bruijn indices in `Var` constructors.

The parser does not compute de Bruijn indices, it gives the variables as strings. The function `fix_index` replaces bound variables by de Bruijn indices. We still keep track of the string names of the variables, in case we have to print them back. Its converse function, `fix_id`, replaces the de Bruijn indices with the previous strings, possibly updating the string

names in case of name clashes. For instance, the string printed to the user, showing the normalized form of (`fun (x y : nat) ⇒ x`) `y`, is `fun y0 : nat ⇒ y` : the bound variable `y` has been renamed `y0`. The meta-variables are generated by the typecheckers, and their identifier is an integer. We have defined several helper functions to ease the process of terms.

The generic function `visit_term f g h t` looks at the children of the term `t`, and: *i*) every child `t1` outside of a binder is replaced with `f t1`; *ii*) every child `t1` inside the binding of a variable whose name (a string) is `s` is replaced with `g s t1`, while `s` is replaced with `h s t1`.

The functions `g` and `h` take a string as an argument, for helping the implementation of the `fix_index` and `fix_id` functions.

The function `map_term` is a kind of mapping function: `map_term k f t` finds every variable `Var(l, n)` inside the term `t`, and replaces it by `f (k+offset) l n`, where `offset` is the number of extra bindings.

The `lift` and `map_term` functions allow us to define a substitution in a clean way:

```
(* update all indices greater than k by adding n to them *)
let lift k n = map_term k (fun k l m → if m < k then Var (l, m) else Var (l, m+n))

(* Transform (lambda x. t1) t2 into t1[t2/x] *)
let beta_redex t1 t2 =
  let subst k l m =
    if m < k then Var (l, m) (* bound variable *)
    else if m = k then (* x *)
      lift 0 k t2
    else (* the enclosing lambda goes away *)
      Var (l, m−1)
  in map_term 0 subst t1
```

2.3 Environments

There are four kinds of environments, namely:

1. the *global environment* (noted Σ). The global environment holds constants which are fully typechecked: $\Sigma ::= \cdot \mid \Sigma, c{:}\varsigma@\sigma \mid \Sigma, c := M@\Delta : \varsigma@\sigma$. Intuitively, $c{:}\varsigma@\sigma$ is a declaration of a constant (or axiom), and $c := M@\Delta : \varsigma@\sigma$ corresponds to a global definition.
2. the *local environment* (noted Γ). It is used for the first step of typechecking, and looks like a standard environment: $\Gamma ::= \cdot \mid \Gamma, x{:}\sigma \mid \Gamma, x := \Delta : \sigma$. Intuitively, $x{:}\sigma$ is a variable introduced by a λ-abstraction, and $x := \Delta : \sigma$ is a local definition introduced by a **let**.
3. the *essence environment* (noted Ψ). It is used for the second step of typechecking, and holds the essence of the local variables: $\Psi ::= \cdot \mid \Psi, x \mid \Psi, x := M$. Intuitively, x is a variable introduced by a λ-abstraction, and $x := M$ is a local definition introduced by a **let**. Notice that the variable x in the BNF expression Ψ, x carries almost no information. However, since local variables are referred to by their de Bruijn indices, and these indices are actually their position in the environment, it follows that they have to appear in the environment, even when there is no additional information.
4. the *meta-environment* (noted Φ). It is used for unification, and records meta-variables and their instantiation whenever the unification algorithm has found a solution: $\Phi ::= \cdot \mid \Phi, \mathbf{sort}(?x) \mid \Phi, ?x := s \mid \Phi, (\Gamma \vdash ?x : \sigma) \mid \Phi, (\Gamma \vdash ?x := \Delta : \sigma) \mid \Phi, \Psi \vdash ?x \mid \Phi, \Psi \vdash ?x := M$. Intuitively, since there are some meta-variables for which we know they have to be sorts, it follows that $\mathbf{sort}(?x)$ declares a meta-variable $?x$ which correspond either to Type or Kind, and $?x := s$ is the instantiation of a sort $?x$. Also, $\Gamma \vdash ?x : \sigma$ is the declaration of a meta-variable $?x$ of type σ which appeared in a local environment Γ, and

$\Gamma \vdash ?x := \Delta : \sigma$ is the instantiation of the meta-variable $?x$. Concerning meta-variables inside essences, $\Psi \vdash ?x$ is the declaration of a meta-variable $?x$ in an essence environment Ψ, and $\Psi \vdash ?x := M$ is the instantiation of $?x$.

2.4 Suspended substitution

We shortly introduce suspended substitution, as presented in [1]. Let's consider the following example: if we want to unify $(\lambda x{:}\sigma.?y)\, c_1$ with c_1, we could unify $?y$ with c_1 or with x, the latter being the preferred solution. However, if we normalize $(\lambda x{:}\sigma.?y)\, c_1$, we should record the fact that c_1 can be substituted by any occurrence of x appearing in $?y$, even though the term which will replace $?y$ is currently unknown. That is the purpose of suspended substitution: the term is actually noted $(\lambda x{:}\sigma.?y[x])\, c_1$ and reduces to $?y[c_1]$, noting that c_1 has replaced x.

▶ **Definition 1** (Erase function and suspended substitution).

1. *The vector $x_1;\ldots;x_n$ is created using the erase function $\overline{\cdot}$, defined as $\overline{x_1{:}\sigma_1;\ldots;x_n{:}\sigma_n} \stackrel{def}{=} x_1;\ldots;x_n$ and $\overline{x_1;\ldots;x_n} \stackrel{def}{=} x_1;\ldots;x_n$.*
2. *When we want to create a new meta-variable in a local context $\Gamma = x_1{:}\sigma_1;\ldots;x_n{:}\sigma_n$, we create a meta-variable $?y[\overline{\Gamma}] \equiv ?y[x_1;\ldots;x_n]$. The vector $\Delta_1;\ldots;\Delta_n$ inside $?y[\Delta_1;\ldots;\Delta_n]$ is the suspended substitution of $?y$. Substitutions for meta-variables and their suspended substitution are propagated as follows:*

$$?y[\Delta_1;\ldots;\Delta_n][\Delta/x] \stackrel{def}{=} ?y[\Delta_1[\Delta/x];\ldots;\Delta_n[\Delta/x]]$$
$$?y[M_1;\ldots;M_n][N/x] \stackrel{def}{=} ?y[M_1[N/x];\ldots;M_n[N/x]]$$

3 The evaluator of Bull

The evaluator follows the applicative order strategy, which recursively normalizes all subterms from left to right (with the help of the `visit_term` function, see full code in [44]), then: if the resulting term is a redex, reduces it, then uses the same strategy again; or else, the resulting term is in normal form.

3.1 Reduction rules

The notions of reduction, from which we can define one-step reduction, multistep reduction, and equivalence relation, are defined below.

▶ **Definition 2** (Reductions).

1. *for Δ-terms:*

$$\begin{array}{rcll}
(\lambda x{:}\sigma.\Delta_1)\,\Delta_2 & \mapsto_\beta & \Delta_1[\Delta_2/x] & \\
\lambda x{:}\sigma.\Delta\, x & \mapsto_\eta & \Delta & \text{if } x \notin \mathsf{Fv}(\Delta) \\
pr_i\,\langle \Delta_1, \Delta_2\rangle & \mapsto_{pr_i} & \Delta_i & \\
\multicolumn{4}{l}{\textbf{smatch } in_i\,\Delta_3 \textbf{ return } \rho \textbf{ with } [x{:}\sigma \Rightarrow \Delta_1 \mid x{:}\tau \Rightarrow \Delta_2]} \\
 & \mapsto_{in_i} & \Delta_i[\Delta_3/x] & \\
\textbf{let } x{:}\sigma := \Delta_1 \textbf{ in } \Delta_2 & \mapsto_\zeta & \Delta_2[\Delta_1/x] & \\
c & \mapsto_{\delta\Sigma} & \Delta & \text{if } (c := M\,@\Delta : \varsigma\,@\sigma) \in \Sigma \\
x & \mapsto_{\delta\Gamma} & \Delta & \text{if } (x := \Delta : \sigma) \in \Gamma \\
?x[\Delta_1;\ldots;\Delta_n] & \mapsto_{\delta\Phi} & \Delta[\overrightarrow{\Delta_i/\Gamma}] & \text{if } (\Gamma \vdash ?x := \Delta : \sigma) \in \Phi \\
?x[\Delta_1;\ldots;\Delta_n] & \mapsto_{\delta\Phi} & s & \text{if } ?x := s \in \Phi
\end{array}$$

2. *for pure λ-terms:*

$$
\begin{array}{rcll}
(\lambda x.M)\,N & \mapsto_{\beta} & M[N/x] & \\
\lambda x.M\,x & \mapsto_{\eta} & M & \textit{if } x \notin \mathsf{Fv}(M) \\
\textbf{let } x := M \textbf{ in } N & \mapsto_{\zeta} & N[M/x] & \\
c & \mapsto_{\delta\Sigma} & M & \textit{if } (c := M @ \Delta : \varsigma @ \sigma) \in \Sigma \\
x & \mapsto_{\delta\Psi} & M & \textit{if } (x := M) \in \Psi \\
?x[M_1; \ldots; M_n] & \mapsto_{\delta\Phi} & N[\overrightarrow{M_i/\Psi}] & \textit{if } (\Psi \vdash ?x := M) \in \Phi \\
?x[M_1; \ldots; M_n] & \mapsto_{\delta\Phi} & s & \textit{if } ?x := s \in \Phi
\end{array}
$$

3.2 Implementation

When the user inputs a term, the refiner creates meta-variables and tries to instantiate them, but this should remain as much as possible invisible to the user. Therefore the term returned by the refiner should be meta-variable free, even though not in normal form. Thus, terms in the global signature Σ are meta-variable free, and the $\delta\Phi$ reductions are only used by the unifier and the refiner.

The function `strongly_normalize` works on both Δ-terms and pure λ-terms, and supposes that the given term is meta-variable free. Note that reductions can create odd spines, for instance if you consider the term $(\lambda x{:}\sigma.x\,S_1)\,(\Delta\,S_2)$, a simple β-redex would give $\Delta\,S_2\,S_1$, therefore we merge S_2 and S_1 in a single spine.

```
let rec strongly_normalize is_essence env ctx t =
  let sn_children = visit_term (strongly_normalize is_essence env ctx)
              (fun _ → strongly_normalize is_essence
                       env (Env.add_var ctx (DefAxiom ("",nothing))))
              (fun id _ → id)
  in let sn = strongly_normalize is_essence env ctx in
  (* Normalize the children *)
  let t = sn_children t in
  match t with
  (* Spine fix *)
  | App(l, App(l',t1,t2), t3) →
     sn (App(l, t1, List.append t2 t3))
  (* Beta-redex *)
  | App (l, Abs (l', _,_, t1), t2 :: [ ]) →
     sn (beta_redex t1 t2)
  | App (l, Abs (l', x,y, t1), t2 :: t3)
    → sn @@ app l (sn (App(l,Abs (l',x,y, t1), t3))) t2
  | Let (l, _, t1, t2, t3) → sn (beta_redex t2 t1)
  (* Delta-redex *)
  | Var (l, n) → let (t1, _) = Env.find_var ctx n in
        (match t1 with
         | Var _ → t1
         | _ → sn t1)
  | Const (l, id) → let o = Env.find_const is_essence env id in
              (match o with
               | None → Const(l, id)
               | Some (Const (_,id') as t1,_) when id = id' → t1
               | Some (t1,_) → sn t1)
  (* Eta-redex *)
  | Abs (l,_, _, App (l', t1, Var (_,0) :: l2))
```

```
    → if is_eta (App (l', t1, l2)) then
        let t1 = lift 0 (−1) t1 in
        match l2 with
        | [ ] → t1
        | _ → App (l', t1, List.map (lift 0 (−1)) l2)
      else t
  (* Pair-redex *)
  | SPrLeft (l, SPair (l', x, _)) → x
  | SPrRight (l, SPair (l', _, x)) → x
  (* inj-reduction *)
  | SMatch (l, SInLeft(l',_,t1), _, id1, _, t2, id2, _, _) →
     sn (beta_redex t2 t1)
  | SMatch (l, SInRight(l',_,t1), _, id1, _, _, id2, _, t2) →
     sn (beta_redex t2 t1)
  | _ → t
```

4 The subtyping algorithm of Bull

The subtyping algorithm implemented in Bull is basically the algorithm $\mathcal{A}$ as described and Coq certified/extracted in [27] by the authors. The main judgment is $\Sigma;\Gamma \vdash \sigma \leqslant \tau$. The only difference is that the types are normalized before applying the algorithm. The auxiliary rewriting functions $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4$, described in [27], rewrite terms in normal forms as follows:

```
let rec anf a =
  let rec distr f a b =
    match (a,b) with
    | (Union(l,a1,a2),_) → Inter(l, distr f a1 b, distr f a2 b)
    | (_, Inter(l,b1,b2)) → Inter(l, distr f a b1, distr f a b2)
    | _ → f a b
  in
  match a with
  | Prod(l,id,a,b) → distr (fun a b → Prod(l,id,a,b)) (danf a) (canf b)
  | _ → a
and canf a =
  let rec distr a b =
    match (a,b) with
    | (Inter(l,a1,a2),_) → Inter(l, distr a1 b, distr a2 b)
    | (_,Inter(l,b1,b2)) → Inter(l, distr a b1, distr a b2)
    | _ → Union(dummy_loc,a,b)
  in
  match a with
  | Inter(l,a,b) → Inter(l, canf a, canf b)
  | Union(l,a,b) → distr (canf a) (canf b)
  | _ → anf a
and danf a =
  let rec distr a b =
    match (a,b) with
    | (Union(l,a1,a2),_) → Union(l, distr a1 b, distr a2 b)
    | (_,Union(l,b1,b2)) → Union(l, distr a b1, distr a b2)
    | _ → Inter(dummy_loc, a,b)
  in
  match a with
```

```
  | Inter(l,a,b) → distr (danf a) (danf b)
  | Union(l,a,b) → Union(l, danf a, danf b)
  | _ → anf a
```

It follows that, our subtyping function is quite simple:

```
let is_subtype env ctx a b =
  let a = danf @@ strongly_normalize false env ctx a in
  let b = canf @@ strongly_normalize false env ctx b in
  let rec foo env ctx a b =
  match (a, b) with
  | (Union(_,a1,a2),_) → foo env ctx a1 b && foo env ctx a2 b
  | (_,Inter(_,b1,b2)) → foo env ctx a b1 && foo env ctx a b2
  | (Inter(_,a1,a2),_) → foo env ctx a1 b || foo env ctx a2 b
  | (_,Union(_,b1,b2)) → foo env ctx a b1 || foo env ctx a b2
  | (Prod(_,_,a1,a2),Prod(_,_,b1,b2))
    → foo env ctx b1 a1 && foo env (Env.add_var ctx (DefAxiom("",nothing))) a2 b2
  | _ → same_term a b
  in foo env ctx a b
```

5 The unification algorithm of Bull

Higher-order unification of two terms Δ_1 and Δ_2 aims at finding a most general substitution for meta-variables such that Δ_1 and Δ_2 becomes convertible. The structural rules are given in Figure 1. Classical references are the work of Huet [26], and Dowek et al. [16].

Our higher-order unification algorithm is inspired by the Reed [41] and Ziliani-Sozeau [48] papers. In [48], conversion of terms is quite involved because of the complexity of Coq. For simplicity, our algorithm supposes the terms to be in normal form.

The unification algorithm takes as input a meta-environment Φ, a global environment Σ, a local environment Γ, the two terms to unify Δ_1 and Δ_2, and either fails or returns the updated meta-environment Φ. The rest of the unification algorithm implements *Higher-Order Pattern Unification* (HOPU) [41]. In a nutshell, HOPU takes as an argument a unification problem $?f\ S \stackrel{?}{=} N$, where all the terms in S are free variables and each variable occurs once. For instance, for the unification problem $?f\ y\ x\ z \stackrel{?}{=} x\ c\ y$, it creates the solution $?f := \lambda y{:}\sigma_2.\lambda x{:}\sigma_1.\lambda z{:}\sigma_3.x\ c\ y$. The expected type of x, y, and z can be found in the local environment, but capturing correctly the free variables x, y, and z is quite tricky because we have to permute their de Bruijn indices. If HOPU fails, we recursively unify every subterm.

6 The refinement algorithm of Bull

The Bull refinement algorithm is inspired by the work on the Matita ITP [1]. It is defined using *bi-directionality*, in the style of Harper-Licata [22]. The bi-directional technique is a mix of typechecking and type reconstruction, in order to trigger the unification algorithm as soon as possible. Moreover, it gives more precise error messages than standard type reconstruction. For instance, if `f : (bool -> nat -> bool) -> bool`, then `f (fun x y ⇒ y)` is ill-typed. With a simple type inference algorithm, we would type `f`, then `fun x y ⇒ y` which would be given some type `?x -> ?y -> ?y`, and finally we would try to unify `bool -> nat -> bool` with `?x -> ?y -> ?y`, which fails. However, the failure is localized on the application, whereas it would better be localized inside the argument. More precisely, we would have the following error message:

$$\frac{s_1 \equiv s_2}{\Phi;\Sigma;\Gamma \vdash s_1 \stackrel{?}{=} s_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi}\ (Sort) \qquad \frac{c_1 \equiv c_2}{\Phi;\Sigma;\Gamma \vdash c_1 \stackrel{?}{=} c_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi}\ (Const) \qquad \frac{x_1 \equiv x_2}{\Phi;\Sigma;\Gamma \vdash x_1 \stackrel{?}{=} x_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi}\ (Var)$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Phi_2;\Sigma;\Gamma, x{:}\sigma_1 \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Gamma \vdash \lambda x{:}\sigma_1.\Delta_1 \stackrel{?}{=} \lambda x{:}\sigma_2.\Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (Abs)$$

$$\frac{\begin{array}{l} x_1{:}\sigma_1,\dots,x_n{:}\sigma_n \vdash ?x : \Pi z_1{:}\tau_1 \dots \Pi z_m{:}\tau_m.\tau \in \Phi_1 \qquad y_1 \dots y_n, z_1 \dots z_m \text{ distinct} \\ \Phi_2 \equiv \Phi_1, (x_1{:}\sigma_1,\dots,x_n{:}\sigma_n \vdash ?x := \lambda z_1{:}\tau_1.\dots.\lambda z_m{:}\tau_m.\Delta\overrightarrow{[x_i/y_i]} : \Pi z_1{:}\tau_1 \dots \Pi z_m{:}\tau_m.\tau) \end{array}}{\Phi_1;\Sigma;\Gamma \vdash ?x[y_1;\dots;y_n]\,(z_1;\dots;z_m) \stackrel{?}{=} \Delta \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2}\ (App_1)$$

$$\frac{\Phi_1 \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2}{\Phi_1;\Sigma;\Gamma \vdash \Delta_1\,() \stackrel{?}{=} \Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2}\ (App_2) \qquad \frac{\Phi_1 \vdash \Delta_1\,S_1 \stackrel{?}{=} \Delta_3\,S_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Phi_2 \vdash \Delta_2 \stackrel{?}{=} \Delta_4 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Gamma \vdash \Delta_1\,(S_1;\Delta_2) \stackrel{?}{=} \Delta_3\,(S_2;\Delta_4) \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (App_3)$$

$$\frac{\begin{array}{c}\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2 \\ \Phi_2;\Sigma;\Gamma \vdash \tau_1 \stackrel{?}{=} \tau_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \cap \tau_1 \stackrel{?}{=} \sigma_2 \cap \tau_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (\cap) \qquad \frac{\begin{array}{c}\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2 \\ \Phi_2;\Sigma;\Gamma \vdash \tau_1 \stackrel{?}{=} \tau_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \cup \tau_1 \stackrel{?}{=} \sigma_2 \cup \tau_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (\cup)$$

$$\frac{\begin{array}{c}\Phi_1;\Sigma;\Gamma \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2 \\ \Phi_2;\Sigma;\Gamma \vdash \Delta_3 \stackrel{?}{=} \Delta_4 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \langle \Delta_1, \Delta_3\rangle \stackrel{?}{=} \langle \Delta_2, \Delta_4\rangle \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (Spair) \qquad \frac{\Phi_1;\Sigma;\Gamma \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2}{\Phi_1;\Sigma;\Gamma \vdash \mathsf{pr}_i\,\Delta_1 \stackrel{?}{=} \mathsf{pr}_i\,\Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2}\ (Proj)$$

$$\frac{\begin{array}{c}\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2 \\ \Phi_2;\Sigma;\Gamma \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \mathsf{in}_i\,\sigma_1\Delta_1 \stackrel{?}{=} \mathsf{in}_i\,\sigma_2\Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (Inj) \qquad \frac{\begin{array}{c}\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2 \\ \Phi_2;\Sigma;\Gamma \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \mathbf{coe}\,\sigma_1\,\Delta_1 \stackrel{?}{=} \mathbf{coe}\,\sigma_2\,\Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (Coe)$$

$$\frac{\begin{array}{l}\Phi_1;\Sigma;\Gamma \vdash \Delta \stackrel{?}{=} \Delta' \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2 \qquad \Phi_2;\Sigma;\Gamma \vdash \tau \stackrel{?}{=} \tau' \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3 \qquad \Phi_3;\Sigma;\Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_1' \overset{\mathcal{U}}{\rightsquigarrow} \Phi_4 \\ \Phi_4;\Sigma;\Gamma, x{:}\sigma_1 \vdash \Delta_1 \stackrel{?}{=} \Delta_1' \overset{\mathcal{U}}{\rightsquigarrow} \Phi_5 \quad \Phi_5;\Sigma;\Gamma \vdash \sigma_2 \stackrel{?}{=} \sigma_2' \overset{\mathcal{U}}{\rightsquigarrow} \Phi_6 \quad \Phi_6;\Sigma;\Gamma, x{:}\sigma_2 \vdash \Delta_2 \stackrel{?}{=} \Delta_2' \overset{\mathcal{U}}{\rightsquigarrow} \Phi_7\end{array}}{\begin{array}{l}\Phi_1;\Sigma;\Gamma \vdash \mathbf{smatch}\ \Delta\ \mathbf{return}\ \tau\ \mathbf{with}\,[x{:}\sigma_1 \Rightarrow \Delta_1 \mid x{:}\sigma_2 \Rightarrow \Delta_2] \\ \stackrel{?}{=} \mathbf{smatch}\ \Delta'\ \mathbf{return}\ \tau'\ \mathbf{with}\,[x{:}\sigma_1' \Rightarrow \Delta_1 \mid x{:}\sigma_2' \Rightarrow \Delta_2'] \overset{\mathcal{U}}{\rightsquigarrow} \Phi_7\end{array}}\ (Ssum)$$

$$\frac{\Phi_1;\Sigma;\Gamma, x{:}\sigma \vdash \Delta_1 \stackrel{?}{=} \Delta_2\,x \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2}{\Phi_1;\Sigma;\Gamma \vdash \lambda x{:}\sigma.\Delta_1 \stackrel{?}{=} \Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2}\ (\eta_l) \qquad \frac{\Phi_1;\Sigma;\Gamma, x{:}\sigma \vdash \Delta_1\,x \stackrel{?}{=} \Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2}{\Phi_1;\Sigma;\Gamma \vdash \Delta_1 \stackrel{?}{=} \lambda x{:}\sigma.\Delta_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_2}\ (\eta_r)$$

Figure 1 Structural rules of the unification algorithm.

```
f (fun x y ⇒ y)
          ^

Error: the term "y" has type "nat" while it is expected to have type "bool".
```

Our typechecker is also a *refiner*: intuitively, a refiner takes as input an incomplete term, and possibly an incomplete type, and tries to infer as much information as possible in order to reconstruct a well-typed term. For example, let's assume we have in the global environment the following constants:

```
(eq : nat -> nat -> Type), (eq_refl : forall x : nat, eq x x)
```

Then refining the term `eq_refl _ : eq _ 0` would create the following term:

```
eq_refl 0 : eq 0 0
```

$$\frac{(x{:}\sigma) \in \Gamma \text{ or } (x := \Delta : \sigma) \in \Gamma}{\Phi; \Sigma; \Gamma \vdash x \overset{\Uparrow}{\rightsquigarrow} x : \sigma; \Phi}\ (Var) \qquad \frac{(c{:}\sigma) \in \Sigma \text{ or } (c := \Delta : \sigma) \in \Sigma}{\Phi; \Sigma; \Gamma \vdash c \overset{\Uparrow}{\rightsquigarrow} c : \sigma; \Phi}\ (Const)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : s; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta_1 : \sigma' \overset{\Downarrow}{\rightsquigarrow} \Delta_1'; \Phi_3 \quad \Phi_3; \Sigma; \Gamma, x := \Delta_1' {:} \sigma' \vdash \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \Delta_2' : \tau; \Phi_4}{\Phi_1; \Sigma; \Gamma \vdash \mathbf{let}\ x{:}\sigma := \Delta_1\ \mathbf{in}\ \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \mathbf{let}\ x{:}\sigma' := \Delta_1'\ \mathbf{in}\ \Delta_2' : \tau[\sigma'/x]; \Phi_4}\ (Let)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \overset{\mathcal{F}}{\rightsquigarrow} \sigma_1' : s_1; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \sigma_2 \overset{\mathcal{F}}{\rightsquigarrow} \sigma_2' : s_2; \Phi_3 \quad \Phi_3 \vdash (s_1, s_2) \in \mathbf{LF}; \Phi_4}{\Phi_1; \Sigma; \Gamma \vdash \Pi x{:}\sigma_1.\sigma_2 \overset{\Uparrow}{\rightsquigarrow} \Pi x{:}\sigma_1'.\sigma_2' : s_2; \Phi_4}\ (Prod)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : s; \Phi_2 \qquad \Phi_2; \Sigma; \Gamma, x{:}\sigma' \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \tau; \Phi_3 \quad \Phi_3; \Sigma; \Gamma \vdash \Pi x{:}\sigma'.\tau \overset{\mathcal{F}}{\rightsquigarrow} \rho : s; \Phi_4}{\Phi_1; \Sigma; \Gamma \vdash \lambda x{:}\sigma.\Delta \overset{\Uparrow}{\rightsquigarrow} \lambda x{:}\sigma'.\Delta' : \Pi x{:}\sigma'.\tau; \Phi_4}\ (Abs)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma; \Phi_2}{\Phi_1; \Sigma; \Gamma \vdash \Delta\ () \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma; \Phi_2}\ (App_1) \qquad \frac{}{\Phi; \Sigma; \Gamma \vdash \mathsf{Type} \overset{\Uparrow}{\rightsquigarrow} \mathsf{Type} : \mathsf{Kind}; \Phi}\ (T)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \Delta_1\ S \overset{\Uparrow}{\rightsquigarrow} \Delta_1' : \sigma; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \sigma =_\beta \Pi x{:}\sigma_1.\sigma_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta_2 : \sigma_1 \overset{\Downarrow}{\rightsquigarrow} \Delta_2'; \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \Delta_1\ (S; \Delta_2) \overset{\Uparrow}{\rightsquigarrow} \Delta_1'\ \Delta_2' : \sigma_2[\Delta_2'/x]; \Phi_3}\ (App_2)$$

$$\frac{\begin{array}{l}\Phi_1; \Sigma; \Gamma \vdash \Delta_1\ S \overset{\Uparrow}{\rightsquigarrow} \Delta_1' : \sigma; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \Delta_2' : \sigma_1; \Phi_3 \\ \Phi_3, \mathbf{sort}(?y), (\Gamma, x{:}\sigma_1 \vdash ?x : ?y[]); \Sigma; \Gamma \vdash \sigma \overset{?}{=} \Pi x{:}\sigma_1.?x[\overline{\Gamma}; x] \overset{\mathcal{U}}{\rightsquigarrow} \Phi_4\end{array}}{\Phi_1; \Sigma; \Gamma \vdash \Delta_1\ (S; \Delta_2) \overset{\Uparrow}{\rightsquigarrow} \Delta_1'\ \Delta_2' : ?x[\overline{\Gamma}; x][\Delta_2'/x]; \Phi_4}\ (App_3)$$

$$\frac{\begin{array}{l}\Phi_1; \Sigma; \Gamma \vdash \sigma_1 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_1'; \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_2'; \Phi_3\end{array}}{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \cap \sigma_2 \overset{\Uparrow}{\rightsquigarrow} \sigma_1' \cap \sigma_2' : \mathsf{Type}; \Phi_3}\ (\cap) \qquad \frac{\begin{array}{l}\Phi_1; \Sigma; \Gamma \vdash \sigma_1 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_1'; \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_2'; \Phi_3\end{array}}{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \cup \sigma_2 \overset{\Uparrow}{\rightsquigarrow} \sigma_1' \cup \sigma_2' : \mathsf{Type}; \Phi_3}\ (\cup)$$

Figure 2 Rules for $\overset{\Uparrow}{\rightsquigarrow}$ (1st part).

Refinement also enables untyped abstractions: the refiner may recover the type of bound variables, because untyped abstractions are incomplete terms. The typechecking is done in two steps: firstly the term is typechecked without caring about the essence, then we check the essence. The five typing judgments are defined as follows:

▶ **Definition 3** (Typing judgments). *We have five typing judgments, corresponding to five OCaml functions:*

1. *The function* `reconstruct` *takes as inputs a meta-environment Φ_1, a global environment Σ, a local environment Γ, and a term Δ_1. It either fails or fills the holes in Δ_1, which becomes Δ_2, and returns Δ_2 along with its type σ and the updated meta-environment Φ_2. The corresponding judgment is the following $\Phi_1; \Sigma; \Gamma \vdash \Delta_1 \overset{\Uparrow}{\rightsquigarrow} \Delta_2 : \sigma; \Phi_2$, and the most relevant rules are described in Figures 2 and 3;*
2. *The function* `force_type` *takes as inputs a meta-environment Φ_1, a global environment Σ, a local environment Γ, and a term σ_1. It either fails or fills the holes in σ_1, which becomes σ_2 while ensuring it is a type, i.e. its type is a sort s, and returns σ_2 along with s, and the updated meta-environment Φ_2. The corresponding judgment is the following $\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \overset{\mathcal{F}}{\rightsquigarrow} \sigma_2 : \tau; \Phi_2$, and the rules are described in Figure 4. Intuitively, the function reconstructs the type τ of σ_1, then tries to unify τ with* Type *and* Kind*. If it can only do one unification, it keeps the successful one, if both unifications work, we choose unification with a sort meta-variable, so τ is convertible to a sort;*
3. *The function* `reconstruct_with_type` *takes as inputs a meta-environment Φ_1, a global environment Σ, a local environment Γ, a term Δ_1, and its expected type σ. It either fails or fills the holes in Δ_1, which becomes Δ_2 while ensuring its type is σ, and returns Δ_2 along the updated meta-environment Φ_2. The corresponding judgment is the following*

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \Delta_1 \overset{\Uparrow}{\rightsquigarrow} \Delta_1' : \sigma_1;\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \Delta_2' : \sigma_2;\Phi_3 \quad \Phi_3;\Sigma;\Gamma \vdash \sigma_1 \cap \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \Phi_4}{\Phi_1;\Sigma;\Gamma \vdash \langle \Delta_1, \Delta_2\rangle \overset{\Uparrow}{\rightsquigarrow} \langle \Delta_1', \Delta_2'\rangle : \sigma_1 \cap \sigma_2;\Phi_4}\ (Spair)$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma;\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \sigma =_\beta \sigma_1 \cap \sigma_2}{\Phi_1;\Sigma;\Gamma \vdash \mathsf{pr}_i\ \Delta \overset{\Uparrow}{\rightsquigarrow} \mathsf{pr}_i\ \Delta' : \sigma_i;\Phi_2}\ (Proj_1)$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma;\Phi_2 \quad \Phi_2,(\Gamma \vdash ?x_1 : \mathsf{Type}),(\Gamma \vdash ?x_2 : \mathsf{Type});\Sigma;\Gamma \vdash \sigma \overset{?}{=} ?x_1[\overline{\Gamma}] \cap ?x_2[\overline{\Gamma}] \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Gamma \vdash \mathsf{pr}_i\ \Delta \overset{\Uparrow}{\rightsquigarrow} \mathsf{pr}_i\ \Delta' : ?x_i[\overline{\Gamma}];\Phi_3}\ (Proj_2)$$

$$\frac{\begin{array}{l}\Phi_1;\Sigma;\Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma';\Phi_2 \qquad \Phi_2;\Sigma;\Gamma \vdash \lambda x{:}\tau_1.\tau_2 : \Pi x{:}\sigma.\mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \lambda x{:}\tau_1'.\tau_2';\Phi_3\\ \Phi_3;\Sigma;\Gamma \vdash \sigma_1 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_1';\Phi_4 \quad \Phi_4;\Sigma;\Gamma \vdash \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_2';\Phi_5\\ \Phi_5;\Sigma;\Gamma \vdash \sigma' \overset{?}{=} \sigma_1' \cup \sigma_2' \overset{\mathcal{U}}{\rightsquigarrow} \Phi_6 \quad \Phi_6;\Sigma;\Gamma, x{:}\sigma_1' \vdash \Delta_1 : \tau_2'[\mathsf{in}_1\,\sigma_2'\,x/x] \overset{\Downarrow}{\rightsquigarrow} \Delta_1';\Phi_7\\ \Phi_7;\Sigma;\Gamma, x{:}\sigma_2' \vdash \Delta_2 : \tau_2'[\mathsf{in}_2\,\sigma_1'\,x/x] \overset{\Downarrow}{\rightsquigarrow} \Delta_2';\Phi_8\end{array}}{\begin{array}{l}\Phi_1;\Sigma;\Gamma \vdash \mathbf{smatch}\ \Delta\ \mathbf{return}\ \lambda x{:}\tau_1.\tau_2\ \mathbf{with}\ [x{:}\sigma_1 \Rightarrow \Delta_1 \mid x{:}\sigma_2 \Rightarrow \Delta_2] \overset{\Uparrow}{\rightsquigarrow}\\ \quad \mathbf{smatch}\ \Delta'\ \mathbf{return}\ \lambda x{:}\tau_1'.\tau_2'\ \mathbf{with}\ [x{:}\sigma_1' \Rightarrow \Delta_1' \mid x{:}\sigma_2' \Rightarrow \Delta_2'] : \tau_2'[\Delta'/x];\Phi_8\end{array}}\ (Ssum)$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : s;\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \tau;\Phi_3 \quad \Sigma;\Gamma \vdash \tau \leqslant \sigma'}{\Phi_1;\Sigma;\Gamma \vdash \mathbf{coe}\,\sigma\,\Delta \overset{\Uparrow}{\rightsquigarrow} \mathbf{coe}\,\sigma'\,\Delta' : \sigma';\Phi_3}\ (Coe)$$

$$\frac{}{\Phi;\Sigma;\Gamma \vdash \_ \overset{\Uparrow}{\rightsquigarrow} ?x[\overline{\Gamma}] : ?y[\overline{\Gamma}];\Phi, \mathbf{sort}(?z),(\Gamma \vdash ?y : ?z[\,]),(\Gamma \vdash ?x : ?y[\overline{\Gamma}])}\ (Wildcard)$$

$$\frac{\begin{array}{l}(\Gamma' \vdash ?x : \sigma) \in \Phi \text{ or } (\Gamma' \vdash ?x := \Delta : \sigma) \in \Phi\\ \Gamma' = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n \quad \Phi_i;\Sigma;\Gamma \vdash \Delta_i : \sigma_i \overset{\Downarrow}{\rightsquigarrow} \Delta_i';\Phi_{i+1} \quad (i = 1 \ldots n)\end{array}}{\Phi_1;\Sigma;\Gamma \vdash ?x[\Delta_1;\ldots;\Delta_n] \overset{\Uparrow}{\rightsquigarrow} ?x[\Delta_1';\ldots;\Delta_n'] : \sigma[\overrightarrow{\Delta_i'/\Gamma'}];\Phi_{n+1}}\ (Meta{-}Var)$$

Figure 3 Rules for $\overset{\Uparrow}{\rightsquigarrow}$ (2nd part).

$\Phi_1;\Sigma;\Gamma \vdash \Delta_1 : \sigma \overset{\Downarrow}{\rightsquigarrow} \Delta_2;\Phi_2$, *and the rules are described in Figure 5. There is a rule* (*Default*) *which applies only if none of the other rules work. The acute reader could remark two subtle things:*

- **a.** *we chose not to add any inference rule for coercions, because we believe it would make error messages clearer: more precisely, if we want to check that* $\mathbf{coe}\,\sigma\,\Delta$ *has type* τ*, there could be two errors happening concurrently: it is possible that the type of* Δ *is not a subtype of* σ*, and at the same time* σ *is not unifiable with* τ*. We think that the error to be reported should be the first one, and in this case the* (*Default*) *rule is sufficient;*
- **b.** *the management of de Bruijn indices for the* (*Let*) *is tricky: if we want to check that* $\mathbf{let}\ x{:}\sigma := \Delta_1\ \mathbf{in}\ \Delta_2$ *has type* τ *in some local context* Γ*, we recursively check that* Δ_2 *has type* τ *in the local context* $\Gamma, x := \Delta_1' : \sigma'$ *for some* Δ_1'*, but the de Bruijn indices for* τ *correspond to the position of the local variables in the local context, which has been updated. We therefore have to increment all the de Bruijn indices in* τ*, in order to report the fact that there is one extra element in the local context;*

4. *The function* `essence` *takes as inputs a meta-environment* Φ_1*, a global environment* Σ*, an essence environment* Ψ*, and a term* Δ*. It either fails or construct its essence* M*, and returns* M *along with the updated meta-environment* Φ_2*. The corresponding judgment is the following* $\Phi_1;\Sigma;\Psi \vdash \Delta \overset{\mathcal{E}\Uparrow}{\rightsquigarrow} M;\Phi_2$*, and the rules are described in Figure 6;*

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\sigma \overset{\Uparrow}{\rightsquigarrow} \sigma' : \tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\tau \overset{?}{=} \mathsf{Type} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3 \quad \Phi_2;\Sigma;\Gamma\vdash\tau \overset{?}{=} \mathsf{Kind} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3' \quad \Phi_2,\mathbf{sort}(?x);\Sigma\vdash\tau \overset{?}{=} ?x[\,] \overset{\mathcal{U}}{\rightsquigarrow} \Phi_4}{\Phi_1;\Sigma;\Gamma\vdash\sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : \tau;\Phi_4}\ (\mathit{Force}_1)$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\sigma \overset{\Uparrow}{\rightsquigarrow} \sigma' : \tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\tau \overset{?}{=} \mathsf{Type} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3 \quad \Phi_2;\Sigma;\Gamma \not\vdash \tau \overset{?}{=} \mathsf{Kind} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3'}{\Phi_1;\Sigma;\Gamma\vdash\sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : \tau;\Phi_3}\ (\mathit{Force}_2)$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\sigma \overset{\Uparrow}{\rightsquigarrow} \sigma' : \tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma \not\vdash \tau \overset{?}{=} \mathsf{Type} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3 \quad \Phi_2;\Sigma;\Gamma\vdash\tau \overset{?}{=} \mathsf{Kind} \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3'}{\Phi_1;\Sigma;\Gamma\vdash\sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : \tau;\Phi_3'}\ (\mathit{Force}_3)$$

Figure 4 Rules for $\overset{\mathcal{F}}{\rightsquigarrow}$.

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\sigma \overset{?}{=} \tau \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Gamma\vdash\Delta : \tau \overset{\Downarrow}{\rightsquigarrow} \Delta';\Phi_3}\ (\mathit{Default})$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : s;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\Delta_1 : \sigma' \overset{\Downarrow}{\rightsquigarrow} \Delta_1';\Phi_3 \quad \Phi_3;\Sigma;\Gamma, x := \Delta_1' : \sigma'\vdash\Delta_2 : \tau \overset{\Downarrow}{\rightsquigarrow} \Delta_2';\Phi_4}{\Phi_1;\Sigma;\Gamma\vdash \mathbf{let}\ x{:}\sigma := \Delta_1\ \mathbf{in}\ \Delta_2 : \tau \overset{\Downarrow}{\rightsquigarrow} \mathbf{let}\ x{:}\sigma := \Delta_1'\ \mathbf{in}\ \Delta_2';\Phi_4}\ (\mathit{Let})$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\tau =_\beta \Pi x{:}\tau_1.\tau_2 \quad \Phi_1;\Sigma;\Gamma\vdash\sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma';\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\sigma' \overset{?}{=} \tau_1 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3 \quad \Phi_3;\Sigma;\Gamma, x{:}\sigma'\vdash\Delta : \tau_2 \overset{\Downarrow}{\rightsquigarrow} \Delta';\Phi_4}{\Phi_1;\Sigma;\Gamma\vdash\lambda x{:}\sigma.\Delta : \tau \overset{\Downarrow}{\rightsquigarrow} \lambda x{:}\sigma'.\Delta';\Phi_4}\ (\mathit{Abs})$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\sigma =_\beta \sigma_1 \cap \sigma_2 \quad \Phi_1;\Sigma;\Gamma\vdash\Delta_1 : \sigma_1 \overset{\Downarrow}{\rightsquigarrow} \Delta_1';\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\Delta_2 : \sigma_2 \overset{\Downarrow}{\rightsquigarrow} \Delta_2';\Phi_3}{\Phi_1;\Sigma;\Gamma\vdash\langle\Delta_1,\Delta_2\rangle : \sigma \overset{\Downarrow}{\rightsquigarrow} \langle\Delta_1',\Delta_2'\rangle;\Phi_3}\ (\mathit{Spair})$$

$$\frac{\Phi_1,(\Gamma\vdash ?x : \mathsf{Type});\Sigma;\Gamma\vdash\sigma \cap ?x : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\Delta : \sigma\cap ?x \overset{\Downarrow}{\rightsquigarrow} \Delta';\Phi_3}{\Phi_1;\Sigma;\Gamma\vdash \mathsf{pr}_1\,\Delta : \sigma \overset{\Downarrow}{\rightsquigarrow} \mathsf{pr}_1\,\Delta';\Phi_3}\ (\mathit{Proj}_1)$$

$$\frac{\Phi_1,(\Gamma\vdash ?x : \mathsf{Type});\Sigma;\Gamma\vdash ?x \cap \sigma : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\Delta : ?x \cap \sigma \overset{\Downarrow}{\rightsquigarrow} \Delta';\Phi_3}{\Phi_1;\Sigma;\Gamma\vdash \mathsf{pr}_2\,\Delta : \sigma \overset{\Downarrow}{\rightsquigarrow} \mathsf{pr}_2\,\Delta';\Phi_3}\ (\mathit{Proj}_2)$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\tau =_\beta \tau_1 \cup \tau_2 \quad \Phi_1;\Sigma;\Gamma\vdash\sigma : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma';\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\sigma' \overset{?}{=} \tau_i \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Gamma\vdash \mathsf{in}_i\,\sigma\Delta : \tau \overset{\Downarrow}{\rightsquigarrow} \mathsf{in}_i\,\sigma'\Delta';\Phi_3}\ (\mathit{Inj})$$

$$\frac{}{\Phi;\Sigma;\Gamma\vdash\_ : \sigma \overset{\Downarrow}{\rightsquigarrow} ?x[\overline{\Gamma}];\Phi,(\Gamma\vdash ?x : \sigma)}\ (\mathit{Wildcard})$$

Figure 5 Rules for $\overset{\Downarrow}{\rightsquigarrow}$.

5. *The function* `essence_with_hint` *takes as inputs a meta-environment* Φ_1*, a global environment* Σ*, an essence environment* Ψ*, a term* Δ*, and its expected essence* M*. It either fails or succeeds by returning the updated meta-environment* Φ_2*. The corresponding judgment is the following* $\Phi_1;\Sigma;\Psi\vdash M\,@\Delta \overset{\mathcal{E}^\Downarrow}{\rightsquigarrow} \Phi_2$*, and the rules are described in Figure 7. There is a rule* (*Default*) *which applies only if none of the other rules work.*

$$\frac{\Phi_1;\Sigma;\Psi\vdash\Delta_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash M@\Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\langle\Delta_1,\Delta_2\rangle\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_3}\ (Spair)\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}{\Phi_1;\Sigma;\Psi\vdash \mathsf{pr}_i\,\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}\ (Proj)$$

$$\frac{\begin{array}{lll}\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}N;\Phi_2 & \Phi_2;\Sigma;\Psi\vdash\sigma\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma;\Phi_3 & \Phi_3;\Sigma;\Psi\vdash\sigma_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1;\Phi_4\\ \Phi_4;\Sigma;\Psi,x\vdash\Delta_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_5 & \Phi_5;\Sigma;\Psi\vdash\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_2;\Phi_6 & \Phi_6;\Sigma;\Psi,x\vdash M@\Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_7\end{array}}{\Phi_1;\Sigma;\Psi\vdash\mathbf{smatch}\ \Delta\ \mathbf{return}\ \sigma\ \mathbf{with}\,[x{:}\sigma_1\Rightarrow\Delta_1\mid x{:}\sigma_2\Rightarrow\Delta_2]\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}(\lambda x.M)\,N;\Phi_7}\ (Ssum)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}{\Phi_1;\Sigma;\Psi\vdash\mathsf{in}_i\,\sigma\,\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}\ (Inj)\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\sigma\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma;\Phi_2\quad\Phi_2;\Sigma;\Psi,x\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\lambda x{:}\sigma.\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\lambda x.M;\Phi_3}\ (Abs)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\sigma_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1;\Phi_2\quad\Phi_2;\Sigma;\Psi,x\vdash\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_2;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\Pi x{:}\sigma_1.\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\Pi x{:}\varsigma_1.\varsigma_2;\Phi_3}\ (Prod)\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}{\Phi_1;\Sigma;\Psi\vdash\mathbf{coe}\,\sigma\,\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}\ (Coe)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}{\Phi_1;\Sigma;\Psi\vdash\Delta\,()\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2}\ (App_1)\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\Delta_1\,S\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M;\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash\Delta_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}N;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\Delta_1\,(S;\Delta_2)\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M\,N;\Phi_3}\ (App_2)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\sigma_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1;\Phi_2\ \Phi_2;\Sigma;\Psi\vdash\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_2;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\sigma_1\cap\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1\cap\varsigma_2;\Phi_3}\ (\cap)\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\sigma_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1;\Phi_2\ \Phi_2;\Sigma;\Psi\vdash\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_2;\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\sigma_1\cup\sigma_2\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma_1\cup\varsigma_2;\Phi_3}\ (\cup)$$

Figure 6 Rules for $\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}$.

$$\frac{\begin{array}{l}\Phi_1;\Sigma;\Psi\vdash\Delta\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M_2;\Phi_2\\ \Phi_2;\Sigma;\Psi\vdash M_1\overset{?}{=}M_2\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3\end{array}}{\Phi_1;\Sigma;\Psi\vdash M_1@\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\ (Default)\qquad\frac{\begin{array}{l}\Phi_1;\Sigma;\Psi\vdash M@\Delta_1\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2\\ \Phi_2;\Sigma;\Psi\vdash M@\Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3\end{array}}{\Phi_1;\Sigma;\Psi\vdash M@\langle\Delta_1,\Delta_2\rangle\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\ (Spair)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash M@\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2}{\Phi_1;\Sigma;\Psi\vdash M@\mathsf{pr}_i\,\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2}\ (Proj)\qquad\frac{\Phi_1;\Sigma;\Psi\vdash\sigma\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma;\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash M@\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow};\Phi_3}{\Phi_1;\Sigma;\Psi\vdash M@\mathsf{in}_i\,\sigma\,\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\ (Inj)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\sigma\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}\varsigma;\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash\Delta_1\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}M_1;\Phi_3\quad\Phi_3;\Sigma;\Psi,x:=M_1\vdash M@\Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_4}{\Phi_1;\Sigma;\Psi\vdash M@\mathbf{let}\ x{:}\sigma:=\Delta_1\ \mathbf{in}\ \Delta_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_4}\ (Let)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\varsigma=_\beta\Pi x{:}\varsigma_1.\varsigma_2\quad\Phi_1;\Sigma;\Psi\vdash\varsigma_1@\sigma_1\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2\quad\Phi_2;\Sigma;\Psi,x\vdash\varsigma_2@\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\varsigma@\Pi x{:}\sigma_1.\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\ (Prod)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash M_1=_\beta\lambda x.M_2\quad\Phi_1;\Sigma;\Psi,x\vdash M_2@\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2}{\Phi_1;\Sigma;\Psi\vdash M_1@\lambda x{:}\sigma.\Delta\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2}\ (Abs)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\varsigma=_\beta\varsigma_1\cap\varsigma_2\quad\Phi_1;\Sigma;\Psi\vdash\varsigma_1@\sigma_1\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash\varsigma_2@\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\varsigma@\sigma_1\cap\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\ (\cap)$$

$$\frac{\Phi_1;\Sigma;\Psi\vdash\varsigma=_\beta\varsigma_1\cup\varsigma_2\quad\Phi_1;\Sigma;\Psi\vdash\varsigma_1@\sigma_1\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_2\quad\Phi_2;\Sigma;\Psi\vdash\varsigma_2@\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Psi\vdash\varsigma@\sigma_1\cup\sigma_2\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}\Phi_3}\ (\cup)$$

Figure 7 Rules for $\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}$.

7 The Read-Eval-Print-Loop of Bull

The *Read-Eval-Print-Loop* (REPL) reads a command which is given by the parser as a list of atomic commands. For instance, if the user writes:

```
Axiom (a b : Type) (f : a -> b).
```

The parser creates the following list of three atomic commands:

1. the command asking `a` to be an axiom of type `Type`;
2. the command asking `b` to be an axiom of type `Type`;
3. the command asking `f` to be an axiom of type `a -> b`.

The REPL tries to process the whole list. If there is a single failure while processing the list of atomic commands, it backtracks so the whole commands fails without changing the environment. These commands are similar to the vernacular Coq commands and are quite intuitive. Here is the list of the REPL commands, along with their description:

```
Help.                              show this list of commands
Load "file".                       for loading a script file
Axiom term : type.                 define a constant or an axiom
Definition name [: type] := term.  define a term
Print name.                        print the definition of name
Printall.                          print all the signature
                                   (axioms and definitions)
Compute name.                      normalize name and print the result
Quit.                              quit
```

8 Future work

The current version of Bull [44] lacks of the following features that we plan to implement in the next future.

1. *Inductive types* are the most important feature to add, in order to have a usable theorem prover. We plan to take inspiration from the works of Paulin-Mohring [35]. This should be reasonably feasible;
2. *Mixing subtyping and unification* is a difficult problem, especially with intersection and union types. The most extensive research which has been done in this domain is the work of Dudenhefner, Martens, and Rehof [17], where the authors study unification modulo subtyping with intersection types (but no union). It would be challenging to find a unification algorithm modulo subtyping for intersection and union types, but ideally it would allow us to do some implicit coercions. Take for example the famous Pierce code exploiting union and intersection types (full details in the Bull [44] distribution and in Appendix A): it would be interesting for the user to use implicit coercions in this way:

   ```
   Axiom (Neg Zero Pos T F : Type) (Test : Pos | Neg).
   Axiom Is_0 : (Neg -> F) & (Zero -> T) & (Pos -> F).
   Definition Is_0_Test : F := smatch Test with
                                 x ⇒ coe _ Is_0 x
                               , x ⇒ coe _ Is_0 x
                               end.
   ```

   The unification algorithm would then guess that the first wildcard should be replaced with `Pos -> F` and the second one should be replaced with `Neg -> F`, which does not seem feasible if the unification algorithm does not take subtyping into account;

3. *Relevant arrow*, as defined in [25], it could be useful to add more expressivity to our system. Relevant implication allows for a natural introduction of subtyping, in that $A \supset_r B$ morally means $A \leqslant B$. Relevant implication amounts to a notion of "proof-reuse". Combining the remarks in [3, 2], minimal relevant implication, strong intersection and strong union correspond respectively to the implication, conjunction and disjunction operators of Meyer and Routley's Minimal Relevant Logic B^+ [32]. This could lead to some implementation problem, because deciding β-equality for the essences in this extended system would be undecidable;
4. A *Tactic language*, such as the one of Coq, should be useful. Currently Bull has no tactics: conceiving such a language should be feasible in the medium term.

References

1 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1), 2012.

2 Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

3 Franco Barbanera and Simone Martini. Proof-functional connectives and realizability. *Archive for Mathematical Logic*, 33:189–211, 1994.

4 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

5 Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

6 Stefano Berardi. *Type dependence and Constructive Mathematics*. PhD thesis, University of Turin, 1990.

7 Jan Bessai, Jakob Rehof, and Boris Düdder. Fast verified BCD subtyping. In *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, volume 11200 of *Lecture Notes in Computer Science*, pages 356–371. Springer, 2018.

8 Olivier Boite. Proof reuse with extended inductive types. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 50–65, 2004.

9 Viviana Bono, Betti Venneri, and Lorenzo Bettini. A typed lambda calculus with intersection types. *Theoretical Computer Science*, 398(1-3):95–113, 2008.

10 Beatrice Capitani, Michele Loreti, and Betti Venneri. Hyperformulae, Parallel Deductions and Intersection Types. *Electronic Notes in Theoretical Computer Science*, 50(2):180–198, 2001.

11 Joshua E. Caplan and Mehdi T. Harandi. A logical framework for software proof reuse. In *Symposium on Software Reusability (SSR)*, pages 106–113, 1995.

12 Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

13 Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-order abstract syntax in coq. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Typed Lambda Calculi and Applications, TLCA*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 1995.

14 Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995.

15 Daniel J. Dougherty, Ugo de'Liguoro, Luigi Liquori, and Claude Stolze. A realizability interpretation for intersection and union types. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 10017 of *Lecture Notes in Computer Science*, pages 187–205. Springer-Verlag, 2016.

16 Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1-2):183–235, 2000.

17 Andrej Dudenhefner, Moritz Martens, and Jakob Rehof. The intersection type unification problem. In *Formal Structures for Computation and Deduction (FSCD)*, pages 19:1–19:16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

18 Andrej Dudenhefner and Paweł Urzyczyn. Kripke semantics for intersection formulas. Tenth Workshop on Intersection Types and Related Systems, 2020.

19 Thomas Ehrhard. Non-idempotent intersection types in logical form. In *Foundations of Software Science and Computation Structures (FOSSACS/ETAPS)*, volume 12077 of *Lecture Notes in Computer Science*, pages 198–216. Springer, 2020.

20 Amy Felty and Douglas J. Howe. Generalization and reuse of tactic proofs. In *Logic Programming and Automated Reasoning (LPAR)*, pages 1–15, 1994.

21 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

22 Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

23 J. Roger Hindley. The simple semantics for Coppo-Dezani-Sallé types. In *International Symposium on Programming*, pages 212–226, 1982.

24 Furio Honsell, Marina Lenisa, Luigi Liquori, and Ivan Scagnetto. Implementing Cantor's paradise. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 229–250, 2016.

25 Furio Honsell, Luigi Liquori, Claude Stolze, and Ivan Scagnetto. The Delta-framework. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37:1–37:21, 2018.

26 Gérard Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

27 Luigi Liquori and Claude Stolze. A decidable subtyping logic for intersection and union types. In *Topics In Theoretical Computer Science (TTCS)*, volume 10608 of *Lecture Notes in Computer Science*, pages 74–90. Springer-Verlag, 2017.

28 Luigi Liquori and Claude Stolze. The Delta-calculus: Syntax and types. In *Formal Structures for Computation and Deduction (FSCD)*, pages 28:1–28:20. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

29 Edgar G. K. López-Escobar. Proof functional connectives. In *Methods in Mathematical Logic*, volume 1130 of *Lecture Notes in Mathematics*, pages 208–221. Springer-Verlag, 1985.

30 William Lovas and Frank Pfenning. Refinement types for logical frameworks and their interpretation as proof irrelevance. *Logical Methods in Computer Science*, 6(4), 2010.

31 David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

32 Robert K. Meyer and Richard Routley. Algebraic analysis of entailment I. *Logique et Analyse*, 15:407–428, 1972.

33 Grigori Mints. The completeness of provable realizability. *Notre Dame Journal of Formal Logic*, 30(3):420–441, 1989.

34 Alexandre Miquel. The implicit calculus of constructions. In *Typed Lambda Calculi and Applications (TLCA)*, pages 344–359. Springer-Verlag, 2001.

35 Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *Typed Lambda Calculi and Applications (TLCA)*, pages 328–345. Springer, 1993.

36 Frank Pfenning. Refinement types for logical frameworks. In *TYPES*, pages 285–299, 1993.

37 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23(7), pages 199–208. ACM, 1988.

38 Benjamin C. Pierce. *Programming with intersection types, union types, and bounded polymorphism.* PhD thesis, Technical Report CMU-CS-91-205. Carnegie Mellon University, 1991.

39 Elaine Pimentel, Simona Ronchi Della Rocca, and Luca Roversi. Intersection types from a proof-theoretic perspective. *Fundamenta Informaticae*, 121(1-4):253–274, 2012.

40 Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.

41 Jason Reed. Higher-order constraint simplification in dependent type theory. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, pages 49–56. ACM, 2009.

42 Simona Ronchi Della Rocca and Luca Roversi. Intersection logic. In *Computer Science Logic (CSL)*, volume 2142 of *Lecture Notes in Computer Science*, pages 421–428. Springer-Verlag, 2001.

43 Claude Stolze. *Combining union, intersection and dependent types in an explicitly typed lambda-calculus*. PhD thesis, Université Côte d'Azur, Inria, 2019.

44 Claude Stolze. Bull. `https://github.com/cstolze/Bull`, 2020.

45 Claude Stolze, Luigi Liquori, Furio Honsell, and Ivan Scagnetto. Towards a logical framework with intersection and union types. In *Logical Frameworks and Meta-languages: Theory and Practice (LFMTP)*, pages 1–9, 2017.

46 Pawel Urzyczyn. Type reconstruction in fomega. *Mathematical Structures in Computer Science*, 7(4):329–358, 1997.

47 Betti Venneri. Intersection types as logical formulae. *Journal of Logic and Computation*, 4(2):109–124, 1994.

48 Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *ACM SIGPLAN Notices*, volume 50(9), pages 179–191. ACM, 2015.

A Examples

This Appendix presents some examples in LF_Δ along with their code in Bull, showing a uniform approach to the encoding of a plethora of type disciplines and systems which ultimately stem or can capitalize from strong proof-functional connectives and subtyping. The framework LF_Δ presented in [25], and its software incarnation Bull introduced in this paper, are the first to accommodate all the examples and counterexamples that have appeared in the literature. In what follows, we denote by λ^{BDdL} [2] the union and intersection type assignment system à la Curry. Type inference of λ^{BDdL} is, of course, undecidable. We start by showing the expressive power of LF_Δ in encoding classical features of typing disciplines with strong intersection and union. For these examples, we set $\Sigma \stackrel{def}{=} \sigma{:}\mathsf{Type}, \tau{:}\mathsf{Type}$.

Auto application. The judgment $\vdash_{\lambda^{\mathrm{BDdL}}} \lambda x.x\,x : \sigma \cap (\sigma \to \tau) \to \tau$ in λ^{BDdL}, is rendered in LF_Δ by the LF_Δ judgment $\vdash_\Sigma \lambda x{:}\sigma \cap (\sigma \to \tau).(\mathsf{pr}_2\, x)\,(\mathsf{pr}_1\, x) : \sigma \cap (\sigma \to \tau) \to \tau$.

Polymorphic identity. The judgment $\vdash_{\lambda^{\mathrm{BDdL}}} \lambda x.x : (\sigma \to \sigma) \cap (\tau \to \tau)$ in λ^{BDdL}, is rendered in LF_Δ by the judgment $\vdash_\Sigma \langle \lambda x{:}\sigma.x, \lambda x{:}\tau.x\rangle : (\sigma \to \sigma) \cap (\tau \to \tau)$.

Commutativity of union. The judgment $\vdash_\Sigma \lambda x.x : (\sigma \cup \tau) \to (\tau \cup \sigma)$ in λ^{BDdL}, is rendered in LF_Δ by the judgment $\vdash_\Sigma \lambda x{:}\sigma\cup\tau.[\lambda y{:}\sigma.\mathsf{in}_2^\tau\, y, \lambda y{:}\tau.\mathsf{in}_1^\sigma\, y]\, x : (\sigma \cup \tau) \to (\tau \cup \sigma)$.
The Bull code corresponding to these examples is the following:

```
Axiom (s t : Type).
Definition auto_application (x : s & (s -> t)) := (proj_r x) (proj_l x).
Definition poly_id : (s -> s) & (t -> t) := let id1 x := x in
                                    let id2 x := x in < id1, id2 >.
Definition commut_union (x : s | t) := smatch x with
                                          x : s ⇒ inj_r t x
                                        , x : t ⇒ inj_l s x
                                       end.
```

Atomic propositions, non-atomic goals and non-atomic programs: $\alpha, \gamma_0, \pi_0 : \mathsf{Type}$
Goals and programs: $\gamma = \alpha \cup \gamma_0 \qquad \pi = \alpha \cup \pi_0$
Constructors (implication, conjunction, disjunction).
$\mathsf{impl} \quad : \quad (\pi \to \gamma \to \gamma_0) \cap (\gamma \to \pi \to \pi_0)$
$\mathsf{impl}_1 \quad = \quad \lambda x{:}\pi.\lambda y{:}\gamma.\mathsf{in}_2^{\alpha}\,(\mathsf{pr}_1\,\mathsf{impl}\,x\,y) \qquad \mathsf{impl}_2 = \lambda x{:}\gamma.\lambda y{:}\pi.\mathsf{in}_2^{\alpha}\,(\mathsf{pr}_2\,\mathsf{impl}\,x\,y)$
$\mathsf{and} \quad : \quad (\gamma \to \gamma \to \gamma_0) \cap (\pi \to \pi \to \pi_0)$
$\mathsf{and}_1 \quad = \quad \lambda x{:}\gamma.\lambda y{:}\gamma.\mathsf{in}_2^{\alpha}\,(\mathsf{pr}_1\,\mathsf{and}\,x\,y) \qquad \mathsf{and}_2 = \lambda x{:}\pi.\lambda y{:}\pi.\mathsf{in}_2^{\alpha}\,(\mathsf{pr}_2\,\mathsf{and}\,x\,y)$
$\mathsf{or} \quad : \quad (\gamma \to \gamma \to \gamma_0) \qquad \mathsf{or}_1 = \lambda x{:}\gamma.\lambda y{:}\gamma.\mathsf{in}_2^{\alpha}\,(\mathsf{or}\,x\,y)$
$\mathsf{solve}\,p\,g$ indicates that the judgment $p \vdash g$ is valid.
$\mathsf{bchain}\,p\,a\,g$ indicates that, if $p \vdash g$ is valid, then $p \vdash a$ is valid.
$\mathsf{solve} \quad : \quad \pi \to \gamma \to \mathsf{Type} \qquad \mathsf{bchain} : \pi \to \alpha \to \gamma \to \mathsf{Type}$
Rules for solve:
$- \quad : \quad \Pi(p{:}\pi)(g_1, g_2{:}\gamma).\mathsf{solve}\,p\,g_1 \to \mathsf{solve}\,p\,g_2 \to \mathsf{solve}\,p\,(\mathsf{and}_1\,g_1\,g_2)$
$- \quad : \quad \Pi(p{:}\pi)(g_1, g_2{:}\gamma).\mathsf{solve}\,p\,g_1 \to \mathsf{solve}\,p\,(\mathsf{or}_1\,g_1\,g_2)$
$- \quad : \quad \Pi(p{:}\pi)(g_1, g_2{:}\gamma).\mathsf{solve}\,p\,g_2 \to \mathsf{solve}\,p\,(\mathsf{or}_1\,g_1\,g_2)$
$- \quad : \quad \Pi(p_1, p_2{:}\pi)(g{:}\gamma).\mathsf{solve}\,(\mathsf{and}_2\,p_1\,p_2)\,g \to \mathsf{solve}\,p_1\,(\mathsf{impl}_1\,p_2\,g)$
$- \quad : \quad \Pi(p{:}\pi)(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\,p\,a\,g \to \mathsf{solve}\,p\,g \to \mathsf{solve}\,p\,(\mathsf{in}_1^{\gamma_0}\,a)$
Rules for bchain:
$- \quad : \quad \Pi(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\,(\mathsf{impl}_2\,g\,(\mathsf{in}_1^{\pi_0}\,a))\,a\,g$
$- \quad : \quad \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\,p_1\,a\,g \to \mathsf{bchain}\,(\mathsf{and}_2\,p_1\,p_2)\,a\,g$
$- \quad : \quad \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\,p_2\,a\,g \to \mathsf{bchain}\,(\mathsf{and}_2\,p_1\,p_2)\,a\,g$
$- \quad : \quad \Pi(p{:}\pi)(a{:}\alpha)(g, g_1, g_2{:}\gamma).\mathsf{bchain}$
$(\mathsf{impl}_2\,(\mathsf{and}_1\,g_1\,g_2)\,p)\,a\,g \to \mathsf{bchain}\,(\mathsf{impl}_2\,g_1\,(\mathsf{impl}_2\,g_2\,p))\,a\,g$
$- \quad : \quad \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g, g_1{:}\gamma).\mathsf{bchain}\,(\mathsf{impl}_2\,g_1\,p_1)\,a\,g \to \mathsf{bchain}$
$(\mathsf{impl}_2\,g_1\,(\mathsf{and}_2\,p_1\,p_2))\,a\,g$
$- \quad : \quad \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g, g_1{:}\gamma).\mathsf{bchain}$
$(\mathsf{impl}_2\,g_1\,p_2)\,a\,g \to \mathsf{bchain}\,(\mathsf{impl}_2\,g_1\,(\mathsf{and}_2\,p_1\,p_2))\,a\,g$

Figure 8 The LF_Δ encoding of Hereditary Harrop Formulæ.

Pierce's code [38]. It shows the great expressivity of union and intersection types:

$$\begin{array}{rcl} \mathrm{Test} & \stackrel{def}{=} & \text{if } b \text{ then } 1 \text{ else } -1 : Pos \cup Neg \\ \mathrm{Is\_0} & : & (Neg \to F) \cap (Zero \to T) \cap (Pos \to F) \\ (\mathrm{Is\_0}\ \mathrm{Test}) & : & F \end{array}$$

The expressive power of union types highlighted by Pierce is rendered in LF_Δ by:

$$\begin{array}{rcl} Neg & : & \mathsf{Type} \quad Zero : \mathsf{Type} \quad Pos : \mathsf{Type} \quad T : \mathsf{Type} \quad F : \mathsf{Type} \quad \mathrm{Test} : Pos \cup Neg \\ \mathrm{Is\_0} & : & (Neg \to F) \cap ((Zero \to T) \cap (Pos \to F)) \\ \mathrm{Is\_0\_Test} & \stackrel{def}{=} & [\lambda x{:}Pos.(\mathsf{pr}_2\,\mathsf{pr}_2\,\mathrm{Is\_0})\,x, \lambda x{:}Neg.(\mathsf{pr}_1\,\mathrm{Is\_0})\,x]\,\mathrm{Test} \end{array}$$

The Bull code corresponding to this example is the following:

```
Axiom (Neg Zero Pos T F : Type) (Test : Pos | Neg).
Axiom Is_0 : (Neg -> F) & (Zero -> T) & (Pos -> F).
Definition Is_0_Test := smatch Test with
                                  x ⇒ coe (Pos -> F) Is_0 x
                                , x ⇒ coe (Neg -> F) Is_0 x
                                end.
```

Hereditary Harrop formulæ. The encoding of Hereditary Harrop's Formulæ is one of the motivating examples given by Pfenning for introducing Refinement Types in LF [36, 30]. In LF_Δ it can be expressed as in Figure 8 and type checked in Bull, without any reference to intersection types, by a subtle use of union types. We add also rules for solving and backchaining. Hereditary Harrop formulæ can be recursively defined using two mutually recursive syntactical objects called programs (π) and goals (γ):

$$\gamma \quad := \quad \alpha \mid \gamma \wedge \gamma \mid \pi \Rightarrow \gamma \mid \gamma \vee \gamma \qquad\qquad \pi \quad := \quad \alpha \mid \pi \wedge \pi \mid \gamma \Rightarrow \pi$$

The Bull code is the following:

```
(* three base types: atomic propositions, non-atomic goals and non-atomic programs*)
Axiom atom : Type.
Axiom non_atomic_goal : Type.
Axiom non_atomic_prog : Type.

(* goals and programs are defined from the base types *)
Definition goal := atom | non_atomic_goal.
Definition prog := atom | non_atomic_prog.

(* constructors (implication, conjunction, disjunction) *)
Axiom impl : (prog -> goal -> non_atomic_goal) & (goal -> prog -> non_atomic_prog).
Definition impl_1 p g := inj_r atom (proj_l impl p g).
Definition impl_2 g p := inj_r atom (proj_r impl g p).
Axiom and : (goal -> goal -> non_atomic_goal) & (prog -> prog -> non_atomic_prog).
Definition and_1 g1 g2 := inj_r atom (proj_l and g1 g2).
Definition and_2 p1 p2 := inj_r atom (proj_r and p1 p2).
Axiom  or :  (goal -> goal -> non_atomic_goal).
Definition or_1 g1 g2 := inj_r atom (or g1 g2).

(* solve p g means: the judgment p |- g is valid *)
Axiom solve : prog -> goal -> Type.

(* backchain p a g means: if p |- g is valid, then p |- a is valid *)
Axiom backchain : prog -> atom -> goal -> Type.

(* rules for solve *)
Axiom solve_and : forall p g1 g2, solve p g1 -> solve p g2 -> solve p (and_1 g1 g2).
Axiom solve_or1 : forall p g1 g2, solve p g1 -> solve p (or_1 g1 g2).
Axiom solve_or2 : forall p g1 g2, solve p g2 -> solve p (or_1 g1 g2).
Axiom solve_impl : forall p1 p2 g, solve (and_2 p1 p2) g -> solve p1 (impl_1 p2 g).
Axiom solve_atom : forall p a g, backchain p a g -> solve p g ->
                                 solve p (inj_l non_atomic_goal a).

(* rules for backchain *)
Axiom backchain_and1 :
 forall p1 p2 a g, backchain p1 a g -> backchain (and_2 p1 p2) a g.
Axiom backchain_and2 :
 forall p1 p2 a g, backchain p1 a g -> backchain (and_2 p1 p2) a g.
Axiom backchain_impl_atom :
 forall a g, backchain (impl_2 g (inj_l non_atomic_prog a)) a g.
Axiom backchain_impl_impl :
 forall p a g g1 g2,  backchain (impl_2 (and_1 g1 g2) p) a g ->
                      backchain (impl_2 g1 (impl_2 g2 p)) a g.
Axiom backchain_impl_and1 :
 forall p1 p2 a g g1, backchain (impl_2 g1 p1) a g ->
```

```
                    backchain (impl_2 g1 (and_2 p1 p2)) a g.
Axiom backchain_impl_and2 :
 forall p1 p2 a g g1, backchain (impl_2 g1 p2) a g ->
                    backchain (impl_2 g1 (and_2 p1 p2)) a g.
```

Natural deductions in normal form. The second motivating example for intersection types given in [36, 30] is *natural deductions in normal form*. A natural deduction is in normal form if there are no applications of elimination rules of a logical connective immediately following their corresponding introduction, in the main branch of a subderivation.

$$\supset_I : \Pi A, B{:}o.(Elim(A) \to \mathrm{Nf}(B)) \to \mathrm{Nf}^0(A \supset B) \quad o : \mathsf{Type} \quad Elim, \mathrm{Nf}^0 : o \to \mathsf{Type} \quad \supset : o \to o \to o$$
$$\supset_E : \Pi A, B{:}o.Elim(A \supset B) \to \mathrm{Nf}^0(A) \to Elim(B) \qquad \mathrm{Nf} \equiv \lambda A{:}o.\mathrm{Nf}^0(A) \cup Elim(A)$$

The corresponding Bull code is the following:

```
Axiom (o : Type) (impl : o -> o -> o) (Elim Nf0 : o -> Type).
Definition Nf A := Nf0 A | Elim A.
Axiom impl_I : forall A B, (Elim A -> Nf B) -> (Nf0 (impl A B)).
Axiom impl_E : forall A B,  Elim (impl A B) -> Nf0 A -> Elim B.
```

The encoding we give in LF_Δ is a slightly improved version of the one in [36, 30]: as Pfenning, we restrict to the purely implicational fragment. As in the previous example, we use union types to define normal forms $\mathrm{Nf}(A)$ either as pure elimination-deductions from hypotheses $Elim(A)$ or normal form-deductions $\mathrm{Nf}^0(A)$. This example is interesting in itself, being the prototype of the encoding of type systems using canonical and atomic syntactic categories [22] and also of Fitch Set Theory [24].

Encoding of Edinburgh LF. A shallow encoding of LF [21] in LF_Δ making essential use of intersection types can be also type checked. Here we consider LF as defined with several syntactical categories :

M	$::=$	$c \mid x \mid \lambda x{:}\sigma.M \mid M\ M$	Objects	K	$::=$	$\star \mid \Pi x{:}\sigma.K$	Kinds
σ	$::=$	$a \mid \Pi x{:}\sigma.\sigma \mid \lambda x{:}\sigma.\sigma \mid \sigma\ M$	Families	S	$::=$	$\Box$	Superkind

We encode LF using *Higher-Order Abstract Syntax* (HOAS) [37, 13]. Moreover, using intersection types, we can use the same axiom in order to encode both λ-abstractions on objects and λ-abstractions on families, as well as a single axiom to encode both application on objects and application on families, and a single axiom to encode both dependent products on families and dependent products on kinds.

The typing rules, defined as axioms, have similar essence, it could be interesting to investigate how to profit from these similarities for better encodings. We have decided to explore two different approaches:

- for the typing rules, we chose to define distinct axioms `of_1`, `of_2`, and `of_3` of different precise types. We have not found a way for these axioms to share the same essence, so we have to write different (but very similar) rules for each of these different typing judgment;
- for equality, we chose to define a single axiom `eq : (obj | fam) -> (obj | fam) -> Type`. The type of this axiom is not very precise (it implies we could compare objects and families), but we can factorize equality rules with the same shape, e.g. the rule `beta_eq` define equalities between a β-redex and its contractum, both on objects and on families.

The corresponding Bull code is the following:

```
Axiom obj' : Type.
Axiom fam' : Type.
Axiom knd' : Type.
Axiom sup' : Type.
Axiom same : (obj' & fam' & knd' & sup').
Axiom term : (obj' | fam' | knd' | sup') -> Type.

(* obj, fam, knd, and sup are different terms *)
(* but their essence is always (term same) *)
Definition obj := term (coe (obj' | fam' | knd' | sup') (coe obj' same)).
Definition fam := term (coe (obj' | fam' | knd' | sup') (coe fam' same)).
Definition knd := term (coe (obj' | fam' | knd' | sup') (coe knd' same)).
Definition sup := term (coe (obj' | fam' | knd' | sup') (coe sup' same)).

(* Syntax *)
Axiom star : knd.
Axiom sqre : sup.
Axiom lam : (fam -> (obj -> obj) -> obj) & (fam -> (obj -> fam) -> fam).
Definition lam_obj := coe (fam -> (obj -> obj) -> obj) lam.
Definition lam_fam := coe (fam -> (obj -> fam) -> fam) lam.
Axiom pi : (fam -> (obj -> fam) -> fam) & (fam -> (obj -> knd) -> knd).
Definition pi_fam := coe (fam -> (obj -> fam) -> fam) pi.
Definition pi_knd := coe (fam -> (obj -> knd) -> knd) pi.
Axiom app : (obj -> obj -> obj) & (fam -> obj -> fam).
Definition app_obj := coe (obj -> obj -> obj) app.
Definition app_fam := coe (fam -> obj -> fam) app.

(* Typing rules *)
Axiom of_1 : obj -> fam -> Type.
Axiom of_2 : fam -> knd -> Type.
Axiom of_3 : knd -> sup -> Type.
Axiom of_ax : of_3 star sqre.

(* Rules for lambda-abstraction are "essentially" the same *)
Definition of_lam_obj := forall t1 t2 t3, of_2 t1 star ->
     (forall x, of_1 x t1 -> of_1 (t2 x) (t3 x)) -> of_1 (lam_obj t1 t2) (pi_fam t1 t3).
Definition of_lam_fam := forall t1 t2 t3, of_2 t1 star ->
     (forall x, of_1 x t1 -> of_2 (t2 x) (t3 x)) -> of_2 (lam_fam t1 t2) (pi_knd t1 t3).
(* Rules for product are ''essentially'' the same *)
Definition of_pi_fam := forall t1 t2, of_2 t1 star ->
 (forall x, of_1 x t1 -> of_2 (t2 x) star) -> of_2 (pi_fam t1 t2) star.
Definition of_pi_knd := forall t1 t2, of_2 t1 star ->
 (forall x, of_1 x t1 -> of_3 (t2 x) sqre) -> of_3 (pi_knd t1 t2) sqre.
(* Rules for application are ''essentially'' the same *)
Definition of_app_obj := forall t1 t2 t3 t4, of_1 t1 (pi_fam t3 t4) ->
 of_1 t2 t3 -> of_1 (app_obj t1 t2) (t4 t2).
Definition of_app_fam := forall t1 t2 t3 t4, of_2 t1 (pi_knd t3 t4) ->
 of_1 t2 t3 -> of_2 (app_fam t1 t2) (t4 t2).

(* equality *)
Axiom eq : (obj | fam) -> (obj | fam) -> Type.
Definition c_obj (x : obj) := coe (obj | fam) x.
Definition c_fam (x : fam) := coe (obj | fam) x.
Axiom beta_eq : forall t1 f g,
```

```
  smatch f with
  (* object *)
    f ⇒ eq (c_obj (app_obj (lam_obj t1 f) g)) (c_obj (f g))
  (* family *)
   , f ⇒ eq (c_fam (app_fam (lam_fam t1 f) g)) (c_fam (f g))
   end.
Axiom lam_eq : forall t1 t2 f,
  eq (c_fam t1) (c_fam t2) ->
  smatch f with
  (* object *)
    f ⇒ forall g, (forall x, eq (c_obj (f x)) (c_obj (g x))) ->
          eq (c_obj (lam_obj t1 f)) (c_obj (lam_obj t2 f))
  (* family *)
   , f ⇒ forall g, (forall x, eq (c_fam (f x)) (c_fam (g x))) ->
          eq (c_fam (lam_fam t1 f)) (c_fam (lam_fam t2 f))
   end.
Axiom app_eq : forall n1 n2 m1,
  eq (c_obj n1) (c_obj n2) ->
  smatch m1 with
  (* object *)
    m1 ⇒ forall m2, eq (c_obj m1) (c_obj m2) ->
         eq (c_obj (app_obj m1 n1)) (c_obj (app_obj m2 n2))
  (* family *)
  , m1 ⇒ forall m2, eq (c_fam m1) (c_fam m2) ->
         eq (c_fam (app_fam m1 n1)) (c_fam (app_fam m2 n2))
  end.
Axiom pi_eq : forall m1 m2 n1 n2,
  eq (c_fam m1) (c_fam m2) ->
  (forall x, eq (c_fam (n1 x)) (c_fam (n2 x))) ->
  eq (c_fam (pi_fam m1 n1)) (c_fam (pi_fam m2 n2)).
```