An Extension of HM(X) with Bounded Existential and Universal Data-Types

Vincent Simonet*
Vincent.Simonet@inria.fr

Abstract

We propose a conservative extension of HM(X), a generic constraint-based type inference framework, with bounded existential (a.k.a. abstract) and universal (a.k.a. polymorphic) data-types. In the first part of the article, which remains abstract of the type and constraint language (i.e. the logic X), we introduce the type system, prove its safety and define a type inference algorithm which computes principal typing judgments. In the second part, we propose a realistic constraint solving algorithm for the case of structural subtyping, which handles the non-standard construct of the constraint language generated by type inference: a form of bounded universal quantification.

1 Introduction

HM(X) is a generic constraint-based type inference system, originally defined for the λ-calculus with *let* by Odersky, Sulzmann and Wehr [OSW99]. It goes on with the tradition of the Hindley–Milner type system by providing the combination of *let*-polymorphism and a complete type reconstruction algorithm. However, the interest of HM(X) lies in its greater generality: indeed, it is fully parametrized by a logic, X, which is used for expressing types and relating them with constraints. Then, instantiating the framework with different possible logics yields a large variety of type systems. For instance, letting X be the standard Herbrand logic retrieves the usual unification-based Hindley–Milner system. Similarly, choosing a logic equipped with a partial order between types yields a type system featuring both subtyping and *let*-polymorphism. Another contribution of HM(X) resides in its treatment of the typing problem as a constraint. This approach allows modular and systematic definitions of type inference systems, by reducing the question of determining whether a program is well-typed to constraint resolution.

The HM(X) framework can naturally be extended to deal with the whole ML programming language, including references [Pot01], exceptions and variant or record data-types. It also has been used as the basis of type systems for some advanced programming constructs [Pot00], or to describe static analyses [PS03]. However, all these extensions are orthogonal to the treatment of polymorphism which remains in the ML tradition.

Besides, Odersky and Läufer [OL92] proposed an extension of the ML language with existential quantification in algebraic data-types, which yields a Hindley/Milner style version of Mitchell and Plotkin's abstract types [MP88]. In this extension, the types provided in definitions of algebraic data-types may comprise locally quantified variables, e.g.:

```
type t = K of Exists 'a. 'a * ('a -> int)
```

(Following Odersky and Läufer [OL92], we adopt a Caml-like [LDG+] syntax for the examples, but we explicitly mention quantifiers in declarations.) The intuition is that the type `t` is isomorphic to the existential type $\exists\alpha.\alpha\times(\alpha\to\mathsf{int})$. Any value of type $\alpha\times(\alpha\to\mathsf{int})$, for some α, can be packed in the type `t` by a simple application of the constructor `K`; then, it can be accessed *via* a regular pattern-matching:

```
match (v : t) with
| K x -> e
```

In short, the expression `e` must be type-checked in an environment where the program variable `x` receives the type $\kappa\times(\kappa\to\mathsf{int})$ where κ is a new type symbol that cannot occur in the type of `e`, in order to preserve abstraction. It is equivalent to say that the function `fun x -> e` must receive the polymorphic type scheme $\forall\alpha.\alpha\times(\alpha\to\mathsf{int})\to\tau$ for some type τ which does not involve α. This style of existential quantification has been implemented in compilers for Hope [Per90], Caml [MP93] and Haskell [Aug94], and has shown its practicality and its value for programming: it provides in particular first class data abstraction and allows heterogeneous data structures aggregating different implementations of the same abstract type. What is more, the principle of combining the introduction and the elimination of existential quantification with the data-type mechanism seems most appealing: this provides the necessary source code annotations for preserving type inference, while remaining reasonably lightweight and natural for the programmer.

In the same spirit, Rémy [Rém94] proposed the introduction of universally quantified variables in data-types, as in the following declaration:

```
type u = L of ForAll 'a. ('a -> 'a)
```

Similarly, `u` is isomorphic to the polymorphic type $\forall\alpha.(\alpha\to\alpha)$. This extension permits to counterbalance in some situations the limitations of *let* polymorphism and allows an encoding of System F. For instance, this is useful for defining structures which contain simultaneously data and functions, as in object-oriented programming, and also allows the introduction of higher-order functions whose arguments have polymorphic types. Such a need is particularly liable to

*INRIA, BP 105, F-78153 Le Chesnay Cedex, France.

appear in the presence of abstract data-types, because—in short—any function which manipulates a value coming from some existential data-type must be polymorphic.

2 Overview

In this paper, we propose an extension, called $\mathrm{HM}_{\exists\forall}(X)$, of the $\mathrm{HM}(X)$ framework with bounded existential and universal data-types. The language we consider is a call-by-value λ-calculus with *let* and references. As for the original $\mathrm{HM}(X)$, our system is parametrized by the logic of types and constraints, which allows different instances.

Let us now detail the contributions of this paper. First and foremost, it provides a general template for equipping $\mathrm{HM}(X)$-based type systems with first class existential and universal types: this naturally includes all $\mathrm{HM}(X)$ instances, but also other systems which follow the same guidelines, e.g. [CP01, PS03]. Obviously, the reasons which lead to introduce such mechanisms are not particular to these programming languages and systems; they correspond to those which are largely discussed in the literature, some of which we have mentioned in the introduction. However, the need for a first class data abstraction mechanism is likely to increase in languages featuring advanced and expressive type systems: roughly speaking, the more precisely types describe expressions, the more one needs to aggregate values of different types. We observed this phenomenon while experimenting with our $\mathrm{HM}(X)$-like type system tracing information flow in ML [PS03] and its prototype implementation for the Caml language, Flow Caml [Sim03a]. In this system, usual ML types are annotated with *security levels* belonging to some arbitrary lattice and representing a hierarchy of principals: for instance, modeling the security policy of a bank, one may have one security level for every client. The information concerning each client can be stored in a record, which may be defined as follows:

```
type 'a client_info =
  { cash: 'a int;
    send_msg: 'a int -> unit;
    ... }
```

This definition is identical to that we would have in Caml, with the difference that the record type `client_info` must carry one parameter which is the security level of the client. This security level also appears in the record components types: the field `cash` stores the current balance of the client's account, hence it has the type of an integer labeled by the client's security level. Similarly, `send_msg` is a function which allows sending a message to the given client. The bank's security policy naturally requires that each client can only receive information about its personal situation, so the argument of `send_msg` must have (at most) the security level `'a`. In Caml, all records storing information about clients would have the same unannotated type, `client_info`, so it would be possible to store the clients file in some data structure, such as a list of type `client_info list`. On the contrary, in Flow Caml, records corresponding to distinct clients have incompatible types—because their annotations differ—and, as a consequence, the clients file forms a heterogeneous set of records which is not representable. A natural solution consists in making the client's security level existentially quantified in the declaration: in section 6.2, we will follow up on this example and show how extending Flow Caml's type system with the framework of the current paper allows manipulating such data structures.

Secondly, instantiating $\mathrm{HM}_{\exists\forall}(X)$ with the standard Herbrand logic, we retrieve a system close to those of Odersky and Läufer [OL92] and Rémy [Rém94]. However, we believe our constraint-based presentation to be more modular and lightweight, because it clearly separates issues related with the logical definition of the type system from those concerning type inference. Indeed, in previous approaches, the typing rule for opening abstract values introduce *Skolem* types for existentially quantified variables, i.e. new incompatible (for unification) type symbols. Although this is an efficient implementation technique for dealing with existentially quantified variables in a unification-based type inference engine, this is not really handy to manipulate when studying the type system itself, e.g. proving type safety results. Our approach stands in sharp contrast: the logical presentation of the type system relies only on the usual type generalization mechanism. For the purpose of type inference, the constraint language is enriched with a construct which provides a form of bounded universal quantification, whose resolution is delegated to an external constraint solver.

As a third contribution, this paper describes a realistic algorithm for solving this extended constraint language for the case of structural subtyping. To the best of our knowledge, this is the first practical algorithm to be proposed which deals with a form of bounded universal quantification in subtyping constraints.

We will now proceed as follows. In section 3, we introduce the languages of types, constraints and programs. Then, in section 4 we define the type system $\mathrm{HM}_{\exists\forall}(X)$, prove it ensures safety of program execution and describe an algorithm which computes principal typing judgments. The presentation and the proofs of these topics are carried out in a general context, with limited assumptions about the model in which types and constraints are interpreted. In section 5, we propose a realistic algorithm for solving constraints generated by type inference in the case of structural subtyping. Section 6 concludes with some discussion about the possible continuations of this work, and about integrating it into our information flow analyzer [PS03, Sim03a]. By lack of space, only the most significant fragments of proofs are given. However, they can be found integrally in the full version of this paper [Sim03b].

3 The core language

We let a *variance* ν be a pair of booleans (ν^+, ν^-). We write $\oplus$ (read: *covariant*), $\ominus$ (*contravariant*) and $\odot$ (*invariant*) for the variances $(1,0)$, $(0,1)$ and $(1,1)$, respectively. A *signature* of arity n is a list of n variances $[\nu_1, \ldots, \nu_n]$.

3.1 Types and constraints

Our framework is parametrized by a first-order logic, X, whose variables, terms and formulas are respectively *type variables*, *types* and *constraints*. Type variables are supposed to be in infinite number and are denoted by the meta-variables α and β. We let $\bar{\alpha}$ stand for a finite list of types variables, and given two sets of variables V_1 and V_2, we let $V_1 \# V_2$ be a shorthand for $V_1 \cap V_2 = \emptyset$.

We follow the model-based approach of Pottier [Pot01] by interpreting the logic in a model $(T, \leq)$, which is a partially ordered set whose elements t are *ground types*. This differs from the original presentation of HM(X) [SOW97] where constraints are viewed as elements of an abstract

cylindric constraint system. Although it is slightly less general, we believe our presentation to be more concise. Furthermore, it allows manipulating *solutions* of constraints, which is of greater help when one describes constraint resolution algorithms. As it is usual, we do not fully specify the first-order logic X nor the model T, but only state a few requirements they must fulfill. Thus, our framework remains abstract from several choices, as the flavor of subtyping at hand (e.g. structural or non-structural subtyping). In section 5, we will describe one of the possible instances.

The logic X must at least comprise the following language of types:

$$\tau ::= \alpha \mid \mathsf{unit} \mid \tau \to \tau \mid \tau \ \mathsf{ref} \mid \varepsilon(\bar{\tau}) \mid \pi(\bar{\tau}) \mid \dots$$

Types include type variables, a base type unit, functions and references types. Existential and universal types are named: we let ε and π range over disjoint sets of *existential* and *universal constructors*, respectively. For the time being, we do not specify how data-types are declared, we will address this question in section 3.2. We just require each constructor to have a signature which specifies in particular its arity.

If ν is a variance, we define $\leq^\nu$ by:

$$t \leq^\nu t' \Leftrightarrow ((\text{if } \nu^+ \text{ then } t \leq t') \text{ and } (\text{if } \nu^- \text{ then } t' \leq t))$$

Let a *symbol* be either an existential or universal constructor, or one of unit, $\to$ and ref. We equip the last three symbols with the usual signatures: $\mathrm{sig}(\mathsf{unit}) = []$, $\mathrm{sig}(\to) = [\ominus, \oplus]$ and $\mathrm{sig}(\mathsf{ref}) = [\odot]$. A symbol φ of signature $[\nu_1, \dots, \nu_n]$ is interpreted in the model by a total mapping $[\![\varphi]\!]$ from T^n to T, such that $[\![\varphi]\!](t_1, \dots, t_n) \leq [\![\varphi]\!](t'_1, \dots, t'_n)$ if and only if, for all i, $t_i \leq^{\nu_i} t'_i$. Moreover, we assume the interpretations of two distinct symbols to produce incomparable ground types. Types are interpreted in the model by *assignments*: an assignment ρ is a total mapping from types to ground types which is a morphism for symbols (i.e. for all φ and $\tau_1, \dots, \tau_n$, $\rho(\varphi(\tau_1, \dots, \tau_n)) = [\![\varphi]\!](\rho(\tau_1), \dots, \rho(\tau_n))$). Constraints are interpreted in the model by a two-place predicate $\cdot \vdash \cdot$, whose arguments are an assignment and a constraint; if $\rho \vdash C$ holds, we say that ρ *satisfies* or is a *solution* of C.

The constraint language of the logic X must provide the following constructs, whose interpretation is specified by the laws of figure 1:

$$C, D ::= \tau \leq \tau \mid C \wedge C \mid \exists \bar{\alpha}.C \mid \forall \bar{\alpha}.D \Rrightarrow C \mid \dots$$

Constraints include subtyping between types (we let $\tau = \tau'$ be a shorthand for $\tau \leq \tau' \wedge \tau' \leq \tau$), conjunction of constraints and existential quantification. In addition to these standard constructs, we require a custom form of constraint, $\forall \bar{\alpha}.D \Rrightarrow C$, which is generated by the type inference algorithm when it requires an expression to have some bounded polymorphic type. We give a *weak* semantics to universal quantification by requiring the bound D to be satisfiable: an assignment ρ is a solution of $\forall \bar{\alpha}.D \Rrightarrow C$ if it satisfies $\exists \bar{\alpha}.D$ and, for every assignment $\bar{t}$ of the variables $\bar{\alpha}$ such that $\rho[\bar{\alpha} \mapsto \bar{t}]$ satisfies D then $\rho[\bar{\alpha} \mapsto \bar{t}]$ is a solution of C. Doing so preserves the property that any sub-constraint of a satisfiable constraint is also satisfiable, which shall help constraint resolution, see section 5.

We introduce the constant **true** for some arbitrary universally true constraint, e.g. $\exists \alpha.\alpha \leq \alpha$. We assume the standard notions of *free-variables* and *capture-avoiding substitutions* to be defined on constraints ($\mathrm{fv}(C)$ stands for the set of variables free in C and $C[\tau/\alpha]$ is the constraint C where every free occurrence of α is substituted by τ). We write $C_1 \vDash C_2$ if C_1 *implies* C_2, i.e. every solution of C_1 is also a solution of C_2. Two constraints are *equivalent* if they imply each other.

$$\begin{array}{rcl}
\rho \vdash \tau \leq \tau' & \Leftrightarrow & \rho(\tau) \leq \rho(\tau') \\
\rho \vdash C_1 \wedge C_2 & \Leftrightarrow & \rho \vdash C_1 \text{ and } \rho \vdash C_2 \\
\rho \vdash \exists \bar{\alpha}.C & \Leftrightarrow & \exists \bar{t} \ \rho[\bar{\alpha} \mapsto \bar{t}] \vdash C \\
\rho \vdash \forall \bar{\alpha}.D \Rrightarrow C & \Leftrightarrow & \forall \bar{t} \ \rho[\bar{\alpha} \mapsto \bar{t}] \vdash D \Rightarrow \rho[\bar{\alpha} \mapsto \bar{t}] \vdash C \\
 & & \text{and } \exists \bar{t} \ \rho[\bar{\alpha} \mapsto \bar{t}] \vdash D
\end{array}$$

Figure 1: Interpretation of constraints

3.2 Data-type declarations

We now have to provide a way to relate data-type constructors to plain types, i.e. to specify the type of the values each data-type accepts. Our approach superficially differs from that of Odersky and Läufer [OL92]. First, we choose not to make data-types declarations part of the expression language; instead we consider they are defined globally. Second, we simplify the presentation by separating them from variants and records; in other words we only consider one summand variants (or isomorphically one field records). Although combining the introduction of existential or universal quantification with building or matching of data structures is convenient in a real programming language, we believe this to be orthogonal to the formal presentation of the system.

For every existential constructor ε, we assume a declaration of the form:

$$\varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau \qquad \text{with } \mathrm{fv}(\tau, D) \subseteq \bar{\alpha} \cup \bar{\beta}$$

(Such a declaration is considered modulo α-conversion of the variables $\bar{\alpha}$ and $\bar{\beta}$, which are supposed to be distinct.) The intuition behind is that the type $\varepsilon(\bar{\alpha})$ is an abbreviation for the existential type $\exists \bar{\beta}[D].\tau$. The main difference with the case of ML is that existential types supply a constraint D which *bounds* the quantification, i.e. which may relate the quantified variables $\bar{\beta}$ with one another and with the type parameters $\bar{\alpha}$. In words, the above declaration means that an expression of type $\varepsilon(\bar{\alpha})$ can be built from any expression which has the type τ for some instance of $\bar{\beta}$ satisfying D. In the case of a unification-based system where the only expressible relation between types is equality, there was no need for such a constraint, because—roughly speaking—it can always be set to **true** by unification and substitution of type variables. However, this is no longer the case with a more elaborate constraint language, e.g. in the presence of subtyping.

Similarly, a universal constructor π must have a declaration of the form:

$$\pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D].\tau \qquad \text{with } \mathrm{fv}(\tau, D) \subseteq \bar{\alpha} \cup \bar{\beta}$$

which means that a value may be injected in the type $\pi(\bar{\alpha})$ if and only if it has the type τ for every solution $\bar{\beta}$ of D.

Subtyping between data-types has been defined by equipping constructors with signatures. One must ensure that

each of them fits the logical interpretation of the corresponding constructor. Thus, for existential constructors, we require that

$$\begin{array}{c}\text{if } \varepsilon(\bar\alpha_1) \triangleq \exists\bar\beta_1[D_1].\tau_1 \text{ and } \varepsilon(\bar\alpha_2) \triangleq \exists\bar\beta_2[D_2].\tau_2\\ \text{with } \bar\beta_2 \# \mathrm{fv}(\tau_1) \text{ then}\\ D_1 \wedge \varepsilon(\bar\alpha_1) \le \varepsilon(\bar\alpha_2) \vDash \exists\bar\beta_2.(D_2 \wedge \tau_1 \le \tau_2)\end{array}$$

The intuition is that $\varepsilon(\bar\alpha_1) \le \varepsilon(\bar\alpha_2)$ requires every expression of type $\varepsilon(\bar\alpha_1)$ to have the type $\varepsilon(\bar\alpha_2)$: then, for every instance of τ_1 with $\bar\beta_1$ satisfying D_1, there must exist a greater instance of τ_2 for some $\bar\beta_2$ which satisfies D_2. In other words, the variances assigned to the constructor ε must be valid for the underlying existential type, which is necessary for the type system to be safe, see proof of lemma 3. The following is the counterpart for universal constructors:

$$\begin{array}{c}\text{if } \pi(\bar\alpha_1) \triangleq \forall\bar\beta_1[D_1].\tau_1 \text{ and } \pi(\bar\alpha_2) \triangleq \forall\bar\beta_2[D_2].\tau_2\\ \text{with } \bar\beta_1 \# \mathrm{fv}(\tau_2) \text{ then}\\ D_2 \wedge \pi(\bar\alpha_1) \le \pi(\bar\alpha_2) \vDash \exists\bar\beta_1.(D_1 \wedge \tau_1 \le \tau_2)\end{array}$$

The inequality $\pi(\bar\alpha_1) \le \pi(\bar\alpha_2)$ requires that for every instance of $\forall\bar\beta_2[D_2].\tau_2$, there exists a lesser one of $\forall\bar\beta_1[D_1].\tau_1$.

3.3 Expressions

Let x, m range over disjoint denumerable sets of *program variables* and *memory locations*, respectively. Then *operators*, *values* and *expressions* are defined as follows:

$$\begin{array}{rl} op ::= & \mathsf{ref} \mid := \mid\ !\\ v ::= & x \mid m \mid () \mid \lambda x.e \mid op \mid := m \mid \langle v\rangle_\varepsilon \mid \langle v\rangle_\pi\\ e ::= & x \mid m \mid () \mid \lambda x.e \mid op \mid e\,e \mid \mathsf{let}\ x = v\ \mathsf{in}\ e\\ & \mid\ \langle e\rangle_\varepsilon \mid \mathsf{open}_\varepsilon\, e\,\mathsf{with}\, v \mid \langle v\rangle_\pi \mid \mathsf{open}_\pi\, e\end{array}$$

The core of the language is a λ-calculus with references. Expressions include program variables, memory locations, a unit constant, λ-abstractions, three operators for allocating, updating and reading memory cells, and function applications. The let construct has the same meaning as the basic expression $(\lambda x.e)\,v$; however, as usual in ML, it directs the type-checker to generalize x's type. The language is extended with existential and universal introduction and elimination constructs. $\langle\cdot\rangle_\varepsilon$ and $\langle\cdot\rangle_\pi$ are data-type constructors, they tell when to pack values as abstract or polymorphic ones, respectively. Polymorphic values can be read directly with open_π which is nothing but the inverse of the constructor $\langle\cdot\rangle_\pi$. Accessing existential ones is slightly more subtle: for the sake of type soundness, we need the content of an existential value to be used only within a statically known bounded scope: for this purpose, the construct $\mathsf{open}_\varepsilon\, e\,\mathsf{with}\, v$ includes a handling function v which will receive as argument the content of the matched expression e.

The grammar of expressions constrains sub-expressions of some constructs to be values—rather than arbitrary expressions: the binding in let, the handler in $\mathsf{open}_\varepsilon$ and the content of a polymorphic data-type. These sub-expressions correspond to the point where the type system performs some type generalization: following Wright [Wri93], we restrict it to values, to preserve soundness in the presence of references.

A *store* is a partial mapping from memory locations to values. The small-step semantics is defined on *configurations* $e \,/\, \mu$ which consist in a pair of an expression and a *store*. The reduction rules are given in figure 2. As usual in presence of side-effects, we choose a *call-by-value* evaluation strategy; hence *evaluation contexts* $\mathbb{E}$ are defined by the following grammar:

$$\mathbb{E} ::= []\, e \mid v\, [] \mid \langle[]\rangle_\varepsilon \mid \mathsf{open}_\varepsilon\, []\,\mathsf{with}\, v \mid \langle[]\rangle_\pi \mid \mathsf{open}_\pi\, []$$

4 The type system

We now present the type system $\mathrm{HM}_{\exists\forall}(X)$. In section 4.1, it is defined as a logic of deduction rules. Then, in section 4.2, we prove it is safe—i.e. reduction of well-typed expression cannot go wrong—and section 4.3 describes a type inference algorithm, that is, a procedure to derive most general typing judgments.

In this section, we identify constraints modulo logical equivalence. Although the representation of constraints is of main importance for the design of a constraint solver, it does not matter when defining a type system and proving it correct.

4.1 Typing rules

A type scheme σ is a triple of a set of quantifiers $\bar\alpha$, a constraint C and a type τ; we write $\sigma = \forall\bar\alpha[C].\tau$. Type schemes are considered modulo renaming of quantified variables. A *program environment* is a partial mapping from program variables to type schemes. A *memory environment* is a partial mapping from memory locations to types.

The type system $\mathrm{HM}_{\exists\forall}(X)$ is defined by the set of deduction rules of figure 3. Judgments about expressions have the form $C, \Gamma, M \vdash e : \tau$ where τ is the type assigned to the expression, the environments Γ and M give type schemes to e's free program variables and types to memory locations, respectively, and the constraint C carries assumptions about the type variables that are free in Γ, M and τ. By generalization, a type scheme may be assigned to a value, hence a judgment of the form $C, \Gamma, M \vdash v : \sigma$. Two extra forms are employed to reason about stores $C, M \vdash \mu$ and configurations $C, \Gamma \vdash e \,/\, \mu : \tau$. We omit Γ and M in judgments when they are empty.

We now comment on the typing rules of figure 3, starting with the three non-syntax directed ones. Firstly, Generalize generalizes the type of a value to produce a type scheme. All type variables that are not free in the environments and the constraint C can be universally quantified. Moreover, we require the generated scheme to have instances, hence the premise $C \vDash \exists\bar\alpha.C'$. Sub is a standard subsumption rule, allowing an expression which has some type τ' to be used with any greater type τ. Rule Hide makes a type variable local to a sub-derivation, which is of interest before generalization.

Rules Var and Loc assign types to program variables and memory locations by looking up the appropriate environment. Note that $\Gamma(x)$ is a type scheme, of which Var makes a fresh instance. Unit deals with the unit constant and Op assigns the standard types to operators: $\mathrm{type}_{\mathsf{ref}}(\tau) = \tau \to \tau\ \mathsf{ref}$, $\mathrm{type}_{:=}(\tau) = \tau\ \mathsf{ref} \to \tau \to \mathsf{unit}$ and $\mathrm{type}_{!}(\tau) = \tau\ \mathsf{ref} \to \tau$. Rules for λ-abstraction Abs, function application App and let-binding Let are standard. In Abs, we slightly abuse notations by mapping x to the simple type τ in the program environment, while it should be $\forall\varnothing[\mathbf{true}].\tau$. When typing $\langle e\rangle_\varepsilon$, rule Exist looks up the declaration of the data-type ε and requires e to have some instance of the type τ which satisfies D (in fact, by Sub, e may

$$\begin{array}{rcll}
\lambda x.e\, v \,/\, \mu & \to & e[v/x] \,/\, \mu & (\beta) \\
\mathsf{let}\ x = v\ \mathsf{in}\ e \,/\, \mu & \to & e[v/x] \,/\, \mu & (\mathrm{let}) \\
\mathsf{open}_\varepsilon \langle v_1 \rangle_\varepsilon\ \mathsf{with}\ v_2 \,/\, \mu & \to & v_2\, v_1 \,/\, \mu & (\varepsilon) \\
\mathsf{open}_\pi \langle v \rangle_\pi \,/\, \mu & \to & v \,/\, \mu & (\pi)
\end{array}$$

$$\begin{array}{rcll}
\mathsf{ref}\, v \,/\, \mu & \to & m \,/\, \mu[m \mapsto v] \quad \text{if } m \notin \mathrm{dom}\,\mu & (\mathrm{ref}) \\
:= m\, v \,/\, \mu & \to & () \,/\, \mu[m \mapsto v] \quad \text{if } m \in \mathrm{dom}\,\mu & (\mathrm{assign}) \\
!\, m \,/\, \mu & \to & \mu(m) \,/\, \mu & (\mathrm{deref}) \\
\mathbb{E}[e] \,/\, \mu & \to & \mathbb{E}[e'] \,/\, \mu' \quad \text{if } e \,/\, \mu \to e' \,/\, \mu' & (\mathrm{context})
\end{array}$$

Figure 2: Semantics

have any type τ' such that $C \vDash \tau' \leq \tau$). In OpenExist, the expression e must have an ε type (for the sake of simplicity, we require the parameters of the data-type to be variables, but this is not restrictive because subsumption allows introducing names for arbitrary sub-terms in types). Then, the handler v must have the type scheme $\forall\bar{\beta}[D].\tau' \to \tau$ so that it is a function able to accept any possible instance of the existential type. Furthermore, in order to preserve type abstraction, the result returned by the handler cannot leak information about the existentially quantified type variables $\bar{\beta}$, hence they must not appear in the type τ. Rule Poly looks up the declaration of the universal data-type π and requires the value v to have the corresponding type scheme. Conversely, in OpenPoly, the expression e must have a π type, and an instance of the scheme found in the declaration is taken.

The last three rules of the system deal with stores and configurations. Store requires each entry of the store to have the type given by the memory environment. Conf allows deriving a judgment about a configuration from those on the expression and the store. ConfHide is required for the type system to enforce subject-reduction: indeed, because of allocation, the memory environment is liable to grow throughout reduction and therefore to involve new type variables. Those can no longer be made local by Hide, but still by ConfHide, because typing judgments on configurations do not mention the memory environment.

For the sake of conciseness, we did not make the two following rules part of our definition of $\mathrm{HM}_{\exists\forall}(X)$, however it can be checked that they are valid, i.e. adding them in the system does not extend the set of derivable judgments:

$$\textsc{Weaken}\ \frac{C, \Gamma, M \vdash e : \varsigma \quad C' \vDash C}{C', \Gamma, M \vdash e : \varsigma} \qquad \textsc{Inst}\ \frac{C, \Gamma, M \vdash v : \forall\bar{\alpha}[C'].\tau \quad C \vDash C'}{C, \Gamma, M \vdash v : \tau}$$

The former states that the constraint in a type judgment about some expression can be weakened (ς stands for either a type τ or a type scheme σ), while the latter allows instantiating type schemes at any place in derivations.

4.2 Type safety

We now state the correctness of the type system, i.e. that evaluation of well-typed expressions cannot go wrong. This could be achieved following a semi-syntactic approach [Pot01], which consists in defining an intermediate *ground* system, $\mathrm{B}_{\exists\forall}(T)$, whose types are those of T. However, we prefer to proceed in a direct way, by proving $\mathrm{HM}_{\exists\forall}(X)$ satisfies subject-reduction, since it shows that each reduction step preserves typings.

Proving type safety is simplified by restricting our attention to *canonical derivations*. We chose a logic, rather than syntax-directed, presentation of the type system, since the former is much more concise than the latter. The restriction to canonical derivations allows to recover the benefit of the latter. Canonical derivations (about expressions or configurations) are those where every instance of Hide is above an instance of Generalize. We write $C, \Gamma, M \vdash_c e : \tau$ if the judgment $C, \Gamma, M \vdash e : \tau$ has a canonical derivation.

Lemma 1 (Canonical derivations) *If $C, \Gamma \vdash e \,/\, \mu : \tau$ holds then there exists a canonical derivation of this judgment.*

Lemma 2 (Subsumption) *Assume $C, \Gamma, M \vdash_c e : \tau$. There exists τ' such that $C \vDash \tau' \leq \tau$ and the derivation of $C, \Gamma, M \vdash_c e : \tau'$ ends by an instance of a syntax-directed rule.*

A memory environment M' extends M if and only if the domain of the latter is a subset of that of the former, and M agrees with M' where both are defined.

We can now state our main lemma.

Lemma 3 (Subject-reduction) *Let $e \,/\, \mu \to e' \,/\, \mu'$. Assume $C, M \vdash_c e : \tau$ and $C, M \vdash \mu$. Then, there exists a memory environment M', which extends M, such that $C, M' \vdash e' : \tau$ and $C, M' \vdash \mu'$.*

Proof. By induction on the derivation of $e \,/\, \mu \to e' \,/\, \mu'$. By lemma 2, we may assume, w.l.o.g., that the derivation of $C, M \vdash_c e : \tau$ ends by an instance of a syntax-directed rule. By lack of space, we only give the case related to existential data-types; all others can be found in the full version of the paper [Sim03b].

∘ *Case* (ε). e is $\mathsf{open}_\varepsilon \langle v_1 \rangle_\varepsilon\ \mathsf{with}\ v_2$ and e' is $v_2\, v_1$ and μ' is μ. The derivation of $C, M \vdash_c e : \tau$ must end with an instance of OpenExist whose premises are $C, M \vdash_c \langle v_1 \rangle_\varepsilon : \varepsilon(\bar{\alpha})$ (1) and $\varepsilon(\bar{\alpha}) \triangleq \exists\bar{\beta}[D].\tau_1$ (2) and $C, M \vdash_c v_2 : \forall\bar{\beta}[D].\tau_1 \to \tau$ (3) and $\bar{\beta} \# \mathrm{fv}(\tau)$ (4).

By lemma 2 and Exist, (1) implies $C, M \vdash v_1 : \tau_1'$ (5) and $\varepsilon(\bar{\alpha}') \triangleq \exists\bar{\beta}'[D'].\tau_1'$ (6) and $C \vDash D'$ (7) and $C \vDash \varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha})$ (8). Invoking (4) and by a renaming of $\bar{\beta}$ in (2) and (3), we may assume $\bar{\beta} \# \mathrm{fv}(\tau_1', C, M)$ (9). By the requirement on ε's signature (section 3.2), (2), (6), (8) and (9) yield $(C \wedge D') \vDash \exists\bar{\beta}.(D \wedge \tau_1' \leq \tau_1)$ and, by (7), $C \vDash \exists\bar{\beta}.(D \wedge \tau_1' \leq \tau_1)$ (10) follows. By (5), Weaken and Sub, we have $(C \wedge D \wedge \tau_1' \leq \tau_1), M \vdash v_1 : \tau_1$, and, by (3) and Inst, $(C \wedge D \wedge \tau_1' \leq \tau_1), M \vdash v_2 : \tau_1 \to \tau$. By App, this yields $(C \wedge D \wedge \tau_1' \leq \tau_1), M \vdash e' : \tau$. Using (4) and (9), by an instance of Hide, we obtain $C \wedge \exists\bar{\beta}.(D \wedge \tau_1' \leq \tau_1), M \vdash e' : \tau$. By Weaken and (10), $C, M \vdash e' : \tau$ follows. □

The previous lemma entails the following, more abstract statement:

Theorem 1 (Subject-reduction) *If $C \vdash e \,/\, \mu : \tau$ and $e \,/\, \mu \to e' \,/\, \mu'$ then $C \vdash e' \,/\, \mu' : \tau$.*

Syntax-directed rules

$$\textsc{Var}\ \frac{\Gamma(x) = \forall\bar{\alpha}[C'].\tau \qquad C \vDash C'}{C, \Gamma, M \vdash x : \tau} \qquad \textsc{Loc}\ C, \Gamma, M \vdash m : M(m)\ \mathsf{ref} \qquad \textsc{Unit}\ C, \Gamma, M \vdash () : \mathsf{unit} \qquad \textsc{Op}\ C, \Gamma, M \vdash op : \mathrm{type}_{op}(\tau)$$

$$\textsc{Abs}\ \frac{C, \Gamma[x \mapsto \tau'], M \vdash e : \tau}{C, \Gamma, M \vdash \lambda x.e : \tau' \to \tau} \qquad \textsc{App}\ \frac{C, \Gamma, M \vdash e_1 : \tau' \to \tau \qquad C, \Gamma, M \vdash e_2 : \tau'}{C, \Gamma, M \vdash e_1\, e_2 : \tau} \qquad \textsc{Let}\ \frac{C, \Gamma, M \vdash v : \sigma \qquad C, \Gamma[x \mapsto \sigma], M \vdash e : \tau}{C, \Gamma, M \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau}$$

$$\textsc{Exist}\ \frac{C, \Gamma, M \vdash e : \tau \qquad \varepsilon(\bar{\alpha}) \triangleq \exists\bar{\beta}[D].\tau \qquad C \vDash D}{C, \Gamma, M \vdash \langle e \rangle_{\varepsilon} : \varepsilon(\bar{\alpha})} \qquad \textsc{OpenExist}\ \frac{C, \Gamma, M \vdash e : \varepsilon(\bar{\alpha}) \qquad \varepsilon(\bar{\alpha}) \triangleq \exists\bar{\beta}[D].\tau' \qquad C, \Gamma, M \vdash v : \forall\bar{\beta}[D].\tau' \to \tau \qquad \bar{\beta} \# \mathrm{fv}(\tau)}{C, \Gamma, M \vdash \mathsf{open}_{\varepsilon}\, e\, \mathsf{with}\, v : \tau}$$

$$\textsc{Poly}\ \frac{C, \Gamma, M \vdash v : \forall\bar{\beta}[D].\tau \qquad \pi(\bar{\alpha}) \triangleq \forall\bar{\beta}[D].\tau}{C, \Gamma, M \vdash \langle v \rangle_{\pi} : \pi(\bar{\alpha})} \qquad \textsc{OpenPoly}\ \frac{C, \Gamma, M \vdash e : \pi(\bar{\alpha}) \qquad \pi(\bar{\alpha}) \triangleq \forall\bar{\beta}[D].\tau \qquad C \vDash D}{C, \Gamma, M \vdash \mathsf{open}_{\pi}\, e : \tau}$$

Non-syntax-directed rules

$$\textsc{Generalize}\ \frac{C \wedge C', \Gamma, M \vdash v : \tau \qquad C \vDash \exists\bar{\alpha}.C' \qquad \bar{\alpha} \# \mathrm{fv}(C, \Gamma, M)}{C, \Gamma, M \vdash v : \forall\bar{\alpha}[C'].\tau} \qquad \textsc{Sub}\ \frac{C, \Gamma, M \vdash e : \tau' \qquad C \vDash \tau' \leq \tau}{C, \Gamma, M \vdash e : \tau} \qquad \textsc{Hide}\ \frac{C, \Gamma, M \vdash e : \tau \qquad \alpha \# \mathrm{fv}(\Gamma, M, \tau)}{\exists\alpha.C, \Gamma, M \vdash e : \tau}$$

Stores and configurations

$$\textsc{Store}\ \frac{\mathrm{dom}\,\mu = \mathrm{dom}\,M \qquad \forall m \in \mathrm{dom}\,M\ \ C, M \vdash \mu(m) : M(m)}{C, M \vdash \mu} \qquad \textsc{Conf}\ \frac{C, \Gamma, M \vdash e : \tau \qquad C, M \vdash \mu}{C, \Gamma \vdash e \,/\, \mu : \tau} \qquad \textsc{ConfHide}\ \frac{C, \Gamma, e \vdash \mu : \tau \qquad \alpha \# \mathrm{fv}(\Gamma, \tau)}{\exists\alpha.C, \Gamma, e \vdash \mu : \tau}$$

Figure 3: The type system $\mathrm{HM}_{\exists\forall}(X)$

A closed configuration $e \,/\, \mu$ is *well-typed* if $C \vdash e \,/\, \mu : \tau$ holds for some type τ and satisfiable constraint C.

Lemma 4 (Progress) *Consider a closed configuration $e \,/\, \mu$. If it is well-typed and irreducible, then e is a value.*

We omit the proof of this result, which is standard and can be found in [Sim03b].

Let e be a closed expression. $e \,/\, \mu$ is said to *go wrong* if there exists an irreducible configuration $e' \,/\, \mu'$ such that $e \,/\, \mu \rightarrow^* e' \,/\, \mu'$ and e' is not a value. The combination of subject-reduction and progress ensures that well-typed configurations cannot go wrong, which proves the type safety of $\mathrm{HM}_{\exists\forall}(X)$.

Theorem 2 (Safety) *If $e \,/\, \mu$ is closed and well-typed, then it does not go wrong.*

4.3 Type inference

From here on, we restrict our attention to *source language expressions*, i.e. expressions which do not contain memory locations. Indeed, introducing memory locations was only useful to define a small-step semantics and state the type safety theorem.

Figure 4 defines the type inference algorithm as a function, $(\!|\cdot \vdash \cdot : \cdot|\!)$, which takes three arguments: the program environment Γ, the expression to be typed, e, and a type τ. It returns the *minimal* (w.r.t. implication) constraint C under which hypothesis τ is a valid type for e in the environment Γ, i.e. $C, \Gamma \vdash e : \tau$ holds. (In every line of figure 4, we naturally assume the local type variables introduced by the algorithm to be distinct and fresh w.r.t. $\mathrm{fv}(\Gamma, \tau)$.)

The recursive definition of the constraint returned by the type inference algorithm follows the syntax-directed rules of the logical presentation of the type system (figure 3). However, in order to be complete, the algorithm must take in account any possible use in $\mathrm{HM}_{\exists\forall}(X)$ derivations of one of the non-syntax-directed rules. First, Sub is dealt with by generating coercions in constraints at any place where subsumption can occur in $\mathrm{HM}_{\exists\forall}(X)$. In what concerns Hide, the constraints produced by the type inference algorithm systematically expose a minimal number of type variables: the free variables of $(\!|\Gamma \vdash e : \tau|\!)$ are (a subset of) those of Γ and τ. Lastly, generalization (rule Generalize) is handled in the inference process in two different ways. On the one hand, the treatment of polymorphism introduced by let is standard: the type scheme to be assigned to the variable x is not given explicitly in the program, so, to ensure principality, the most general one must be computed and inserted in the environment. For this purpose, a fresh type variable α is introduced for the typing of v, so that α is the only variable which can be generalized in the constraint $(\!|\Gamma \vdash v : \alpha|\!)$ (other free variables are bound in the environment Γ). Hence, $\forall\alpha[(\!|\Gamma \vdash v : \alpha|\!)].\alpha$ is a most general type scheme for v in the environment Γ. On the other hand, in the expressions $\mathsf{open}_\varepsilon\, e\, \mathsf{with}\, v$ and $\langle v \rangle_\pi$ the value v is expected to have a type scheme which is given by the data-type declaration. So, the type inference algorithm computes v's principal typing and must then ensure this to be more general than the expected one. This comparison is simply encoded by a universally quantified constraint: in $\mathsf{open}_\varepsilon\, e\, \mathsf{with}\, v$, the handler v must have the scheme $\forall\bar\beta[D].\tau' \rightarrow \tau$, hence the constraint $\forall\bar\beta.D \Rightarrow (\!|\Gamma \vdash v : \tau' \rightarrow \tau|\!)$. Similarly, in $\langle v \rangle_\pi$, the scheme $\forall\bar\beta[D].\tau$ is expected; this yields the constraint $\forall\bar\beta.D \Rightarrow (\!|\Gamma \vdash v : \tau|\!)$.

The following theorem states that the type inference algorithm is *correct*, i.e. that the constraint it generates is that of a valid typing judgment about the given expression.

Theorem 3 (Correctness) *For all Γ, e and τ, the judgment $(\!|\Gamma \vdash e : \tau|\!), \Gamma \vdash e : \tau$ is derivable.*

Proof. The proof is by induction on the expression e. Again, by lack of space, we only give the cases concerning existential data-types. Those about universal data-types are similar, and others are standard. All can be found in [Sim03b].

∘ *Case* $e = \langle e' \rangle_\varepsilon$. Let $\bar\alpha$ and $\bar\beta$ be distinct variables not in $\mathrm{fv}(\tau, \Gamma)$ (1) and $\varepsilon(\bar\alpha) \triangleq \exists\bar\beta[D].\tau'$ (2). By induction hypothesis, we have $(\!|\Gamma \vdash e' : \tau'|\!), \Gamma \vdash e' : \tau'$. By Weaken, this yields $(\!|\Gamma \vdash e' : \tau'|\!) \wedge D, \Gamma \vdash e' : \tau'$ (3). By an instance of Exist with the premises (3), (2) and $(\!|\Gamma \vdash e' : \tau'|\!) \wedge D \vDash D$, we obtain $(\!|\Gamma \vdash e' : \tau'|\!) \wedge D, \Gamma \vdash e' : \varepsilon(\bar\alpha)$. By Weaken and Sub, $(\!|\Gamma \vdash e' : \tau'|\!) \wedge D \wedge \varepsilon(\bar\alpha) \leq \tau, \Gamma \vdash e' : \tau$ (4) follows. By (1), $\bar\alpha\bar\beta$ appear only in the constraint of the judgment (4), then by Hide, we obtain $\exists\bar\alpha\bar\beta.((\!|\Gamma \vdash e' : \tau'|\!) \wedge D \wedge \varepsilon(\bar\alpha) \leq \tau), \Gamma \vdash e' : \tau$ (5). By (1) and (2), $(\!|\Gamma \vdash e : \tau|\!)$ is $\exists\bar\alpha\bar\beta.((\!|\Gamma \vdash e' : \tau'|\!) \wedge D \wedge \varepsilon(\bar\alpha) \leq \tau)$, hence (5) is the goal.

∘ *Case* $e = \mathsf{open}_\varepsilon\, e'\, \mathsf{with}\, v$. Let $\bar\alpha$ and $\bar\beta$ be distinct variables not in $\mathrm{fv}(\tau, \Gamma)$ (1) and $\varepsilon(\bar\alpha) \triangleq \exists\bar\beta[D].\tau'$ (2). By induction hypothesis, we have $(\!|\Gamma \vdash e' : \varepsilon(\bar\alpha)|\!), \Gamma \vdash e' : \varepsilon(\bar\alpha)$ (3) and $(\!|\Gamma \vdash v : \tau' \rightarrow \tau|\!), \Gamma \vdash v : \tau' \rightarrow \tau$ (4). By Weaken, $D \wedge \forall\bar\beta.D \Rightarrow (\!|\Gamma \vdash v : \tau' \rightarrow \tau|\!), \Gamma \vdash v : \tau' \rightarrow \tau$ (5) follows from (4). Because $\forall\bar\beta.D \Rightarrow (\!|\Gamma \vdash v : \tau' \rightarrow \tau|\!) \vDash \exists\bar\beta.D$, $\bar\beta \# \mathrm{fv}(\forall\bar\beta.D \Rightarrow (\!|\Gamma \vdash v : \tau' \rightarrow \tau|\!))$, by Generalize, (1) and (5) yield $\forall\bar\beta.D \Rightarrow (\!|\Gamma \vdash v : \tau' \rightarrow \tau|\!), \Gamma \vdash v : \forall\bar\beta[D].\tau' \rightarrow \tau$ (6). By Weaken and OpenExist, the judgment $(\!|\Gamma \vdash e' : \varepsilon(\bar\alpha)|\!) \wedge \forall\bar\beta.D \Rightarrow (\!|\Gamma \vdash v : \tau' \rightarrow \tau|\!), \Gamma \vdash e : \tau$ can be derived from (3), (6), (2) and (1). By (1) and Hide, $\exists\bar\alpha.((\!|\Gamma \vdash e' : \varepsilon(\bar\alpha)|\!) \wedge \forall\bar\beta.D \Rightarrow (\!|\Gamma \vdash v : \tau' \rightarrow \tau|\!)), \Gamma \vdash e : \tau$ (7) follows. By (1) and (2), $(\!|\Gamma \vdash e : \tau|\!)$ is $\exists\bar\alpha.((\!|\Gamma \vdash e' : \varepsilon(\bar\alpha)|\!) \wedge \forall\bar\beta.D \Rightarrow (\!|\Gamma \vdash v : \tau' \rightarrow \tau|\!))$, hence (7) is the goal.

We continue by showing the type inference algorithm to be complete, i.e. to produce the *minimal* constraint (w.r.t. implication) under whose hypothesis an expression is typable.

Lemma 5 (Subsumption) $(\!|\Gamma \vdash e : \tau|\!) \wedge \tau \leq \tau' \vDash (\!|\Gamma \vdash e : \tau'|\!)$.

Theorem 4 (Completeness) *If $C, \Gamma \vdash e : \tau$ then $C \vDash (\!|\Gamma \vdash e : \tau|\!)$. If $C, \Gamma \vdash v : \forall\bar\alpha[C'].\tau$ and $\bar\alpha \# \mathrm{fv}(\Gamma)$ then $C \vDash \forall\bar\alpha.C' \Rightarrow (\!|\Gamma \vdash v : \tau|\!)$.*

Proof. By induction on the input derivation. By lack of space, we only give the cases concerning existential data-types. Those about universal data-types are similar, and others are standard.

∘ *Case* Exist. The hypothesis is $C, \Gamma \vdash \langle e \rangle_\varepsilon : \varepsilon(\bar\alpha)$. This derivation ends by an instance of Exist whose premises are $C, \Gamma \vdash e : \tau$ (1), $\varepsilon(\bar\alpha) \triangleq \exists\bar\beta[D].\tau$ (2) and $C \vDash D$ (3). Using the induction hypothesis with (1), we have $C \vDash (\!|\Gamma \vdash e : \tau|\!)$. By (3), we obtain $C \vDash (\!|\Gamma \vdash e : \tau|\!) \wedge D$ (4). Let $\varepsilon(\bar\alpha'\bar\beta') \triangleq \exists\bar\beta[D'].\tau'$ (5) with $\bar\alpha'\bar\beta' \# \mathrm{fv}(\Gamma, \bar\alpha)$ (6). By (2), (5) and lemma 5, (4) implies $C \vDash \exists\bar\alpha'\bar\beta'.((\!|\Gamma \vdash e : \tau'|\!) \wedge D' \wedge \varepsilon(\bar\alpha') \leq \varepsilon(\bar\alpha))$ (7). By (5) and (6), $(\!|\Gamma \vdash \langle e \rangle_\varepsilon : \varepsilon(\bar\alpha)|\!)$ is the right-hand-side of (7), which is hence the goal.

∘ *Case* OpenExist. The hypothesis is $C, \Gamma \vdash \mathsf{open}_\varepsilon\, e\, \mathsf{with}\, v : \tau$. This derivation ends by an instance of OpenExist whose

$$
\begin{array}{rll}
(\!|\Gamma \vdash x : \tau|\!) & = \exists \bar{\alpha}.(C \wedge \tau' \leq \tau) & \Gamma(x) = \forall \bar{\alpha}[C].\tau' \\
(\!|\Gamma \vdash () : \tau|\!) & = \mathsf{unit} \leq \tau & \\
(\!|\Gamma \vdash \lambda x.e : \tau|\!) & = \exists \alpha_1 \alpha_2.((\!|\Gamma[x \mapsto \alpha_1] \vdash e : \alpha_2|\!) \wedge \alpha_1 \to \alpha_2 \leq \tau) & \\
(\!|\Gamma \vdash e_1\, e_2 : \tau|\!) & = \exists \alpha.((\!|\Gamma \vdash e_1 : \alpha \to \tau|\!) \wedge (\!|\Gamma \vdash e_2 : \alpha|\!)) & \\
(\!|\Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau|\!) & = (\!|\Gamma[x \mapsto \forall \alpha[C].\alpha] \vdash e : \tau|\!) \wedge \exists \alpha.C & C = (\!|\Gamma \vdash v : \alpha|\!) \\
(\!|\Gamma \vdash op : \tau|\!) & = \exists \alpha.(\mathrm{type}_{op}(\alpha) \leq \tau) & \\
(\!|\Gamma \vdash \langle e \rangle_{\varepsilon} : \tau|\!) & = \exists \bar{\alpha} \bar{\beta}.((\!|\Gamma \vdash e : \tau'|\!) \wedge D \wedge \varepsilon(\bar{\alpha}) \leq \tau) & \varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau' \\
(\!|\Gamma \vdash \mathsf{open}_{\varepsilon}\, e\, \mathsf{with}\, v : \tau|\!) & = \exists \bar{\alpha}.((\!|\Gamma \vdash e : \varepsilon(\bar{\alpha})|\!) \wedge \forall \bar{\beta}.D \Rightarrow (\!|\Gamma \vdash v : \tau' \to \tau|\!)) & \varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau' \\
(\!|\Gamma \vdash \langle v \rangle_{\pi} : \tau|\!) & = \exists \bar{\alpha}.(\forall \bar{\beta}.D \Rightarrow (\!|\Gamma \vdash v : \tau'|\!) \wedge \pi(\bar{\alpha}) \leq \tau) & \pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D].\tau' \\
(\!|\Gamma \vdash \mathsf{open}_{\pi}\, e : \tau|\!) & = \exists \bar{\alpha} \bar{\beta}.((\!|\Gamma \vdash e : \pi(\bar{\alpha})|\!) \wedge D \wedge \tau' \leq \tau) & \pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D].\tau'
\end{array}
$$

Figure 4: Type inference for $\mathrm{HM}_{\exists\forall}(X)$

premises are $C, \Gamma \vdash e : \varepsilon(\bar{\alpha})$ (1), $\varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau'$ (2), $C, \Gamma \vdash v : \forall \bar{\beta}[D].\tau' \to \tau$ (3) and $\bar{\beta} \# \mathrm{fv}(\tau)$ (4). Thanks to (4), we may assume $\bar{\beta} \# \mathrm{fv}(C, \Gamma)$ (5). Invoking the induction hypothesis, (1) and (3) yield $C \vDash (\!|\Gamma \vdash e : \tau'|\!)$ (6) and $C \vDash \forall \bar{\beta}.D \Rightarrow (\!|\Gamma \vdash v : \tau' \to \tau|\!)$ (7), respectively. Let $\varepsilon(\bar{\alpha}') \triangleq \exists \bar{\beta}[D'].\tau''$ (8) with $\bar{\alpha}' \# \mathrm{fv}(C, \Gamma, \tau)$ (9). By lemma 5, (6) and (7) yield $C \wedge \bar{\alpha} = \bar{\alpha}' \vDash (\!|\Gamma \vdash e : \tau''|\!) \wedge \forall \bar{\beta}.D \Rightarrow (\!|\Gamma \vdash v : \tau'' \to \tau|\!)$. By (9), $C \vDash \exists \bar{\alpha}'.((\!|\Gamma \vdash e : \tau''|\!) \wedge \forall \bar{\beta}.D \Rightarrow (\!|\Gamma \vdash v : \tau'' \to \tau|\!))$ (10) follows. By (5), (8) and (9), $(\!|\Gamma \vdash \mathsf{open}_{\varepsilon}\, e\, \mathsf{with}\, v : \tau|\!)$ is the right-hand-side of (10), which is hence the goal.

5 Solving constraints: the case of structural subtyping

We are done describing the $\mathrm{HM}_{\exists\forall}(X)$ type system and its type inference algorithm, which provides a procedure to compute (the constraint of) the principal typing judgment for a given program. We shall now deal with the *resolution* of constraints: indeed, according to the type safety theorem, a program is well-typed in $\mathrm{HM}_{\exists\forall}(X)$ if and only if the constraint produced by inference is satisfiable in the logic X. However, addressing this question in a general context, without specifying more precisely the properties of the logic and the model is not conceivable: constraints resolution techniques heavily depend on them.

We choose to turn our attention to the case of structural subtyping [Mit91], because our initial motivation resides in the introduction of existential and universal data-types in our information flow inference system for the Caml language [PS03, Sim03a]. In short, with structural subtyping, two comparable ground types must be (finite) trees which have the same shape and only differ by their leaves or *atoms*. This flavor of subtyping naturally arises when one intends to extend a unification-based type system with annotations belonging to a poset, in order to perform some static analysis [AF97, FTA02, PS03].

Resolution of sets of inequalities in the case of structural subtyping has been intensively studied, and efficient algorithms are known for solving and simplifying constraints [FM89, Tiu92, HM95, Sim03c]. However, the type inference algorithm of section 4 produces a non-standard form of constraints, $\forall \bar{\beta}.D \Rightarrow C$, which combines universal quantification with implication. On the one hand, Kuncak and Rinard [KR03] recently showed that the first-order theory of structural subtyping of non-recursive types is decidable; however, this study does not yield a practical algorithm for solving constraints. On the other hand, previous works [HR97, Sim03c] described efficient algorithms for deciding *top-level* entailment of constraints, where all free variables are implicitly universally quantified. However the problem we tackle here is more general. Because of the presence of unquantified free variables, the result of the comparison is no longer a simple boolean: one must determine the minimal hypothesis—if any—about these variables which guarantees the implication to be true. Our approach consists in trying to express this hypothesis itself as a constraint, i.e. to translate every construct $\forall \bar{\beta}.D \Rightarrow C$ into a regular constraint without universal quantification and implication. The interest of this strategy, known as *quantifiers elimination*, is that, because the target constraint language is standard, it benefits from the existing techniques, such as simplification methods.

Designing algorithms for solving constraints of a logic which includes an implication operator is recognized to be a difficult question, because they involve a form of disjunction: the constraint can be solved either by negating the left-hand-side of the implication or by satisfying the right-hand-side, which breaks the usual closure mechanisms. Here, we bypass this problem thanks to two particular properties. Firstly, left-hand-sides of implications are not arbitrary in constraints generated by type inference: they are exactly the constraints which are given in data-type declarations (by convention, all these have been denoted by the meta-variable D in the paper). As a consequence, we impose some restrictions about their form in order to strike a compromise between the expressivity of the system and the efficiency of constraint resolution, as will be explained in subsection 5.3. Secondly, we gave a *weak* semantics to universal quantification: indeed $\forall \bar{\beta}.D \Rightarrow C$ implies $\exists \bar{\beta}.D$ (and, as a consequence, $\exists \bar{\beta}.(D \wedge C)$). This was not necessary for the type inference algorithm (because it could generate $\exists \bar{\beta}.D \wedge \forall \bar{\beta}.D \Rightarrow C$ in place of $\forall \bar{\beta}.D \Rightarrow C$). However, making this explicit in the semantics of constraints serves the constraint solver: in particular, it preserves the property that any sub-constraint of a satisfiable constraint is also satisfiable.

5.1 The ground model

Let us briefly introduce a generic model T_s for structural subtyping, parametrized by a set of *symbols* (which must at least include those of section 3). Every symbol φ must come with a fixed arity $a(\varphi)$ and a signature $\mathrm{sig}(\varphi)$. Then ground

types are finite trees labeled by symbols, defined by:

$$t ::= \varphi(t_1, \ldots, t_{a(\varphi)})$$

Symbols of arity 0 are *ground atoms*; we suppose they are partially ordered by the lattice order $\leq_0$. Other non-constant symbols are *type constructors* and denoted by the meta-variable ϕ. (We do not introduce some kinding system to separate them because this is not required by the forthcoming algorithm.) Subtyping is the smallest partial order between types $\leq$ which includes $\leq_0$ and such that, if $\mathrm{sig}(\phi) = [\nu_1, \ldots, \nu_n]$, then for all $t_1, \ldots, t_n, t'_1, \ldots, t'_n$:

$$\forall i \;\; t_i \leq^{\nu_i} t'_i \Rightarrow \phi(t_1, \ldots, t_n) \leq \phi(t'_1, \ldots, t'_n)$$

We let $\approx$ be the symmetric transitive closure of $\leq$. Every equivalence class of this relation is a set of ground types which have the same shape; on which the subtyping order $\leq$ defines a lattice structure [Sim03c]. It is straightforward to check that $(T_s, \leq)$ satisfies the assumptions made in section 3 about the ground model of types.

5.2 The constraint language

In the forthcoming subsection 5.4, we will describe the solving algorithm as a small step reduction which rewrites constraints. As a consequence, constraints are not only used to denote the problems produced by type inference, but also to describe their internal representation in the solver as well as intermediate steps of computation. Then, in the remainder of this section, we extend the constraint language as follows.

Let a *cset* R be a multiset, interpreted as a conjunction, whose elements are *elementary constraints* of the form $\tau_1 \leq \tau_2$ or $\tau_1 \approx \tau_2$. (This second non-standard form does not improve the expressiveness of the language: indeed, because each of $\approx$'s equivalence classes is a lattice, $\tau_1 \approx \tau_2$ could be encoded by $\exists\alpha.(\alpha \leq \tau_1 \wedge \alpha \leq \tau_2)$. However, the possibility to explicitly remove existentially quantified variables is mandatory in several steps of the algorithm.) Then, constraints are built on top of csets by the following grammar:

$$C ::= R \mid C \wedge \tau \leq \tau \mid \exists\alpha.C \mid \forall\bar{\beta}.D \Rrightarrow C \mid \exists\langle\phi(\bar{\alpha}) \doteq \alpha\rangle.C$$

By scope extrusion, we can restrict the right-hand-side of every conjunction to be a simple inequality—rather than an arbitrary constraint—without loss of expressiveness (see the full version of the paper [Sim03b] for the details). We made this presentation choice because it simplifies the description of the algorithm and avoids some technical issues; however there is no theoretical difficulty in generalizing the framework to an unrestricted conjunction.

The last construct of the constraint language is part of the solver's internal representation of constraints. Indeed, the algorithm is based on the possibility offered by structural subtyping to *expand* types and *decompose* inequalities, as illustrated by the following example. Consider the constraint $\alpha \leq \phi(\beta_1, \ldots, \beta_n)$: every solution of this inequality maps α to a type whose root is ϕ; hence we can *expand* this variable by introducing fresh variables $\alpha_1, \ldots, \alpha_n$ and rewriting the constraint as $\exists\alpha_1 \ldots \alpha_n.(\alpha = \phi(\alpha_1, \ldots, \alpha_n) \wedge \phi(\alpha_1, \ldots, \alpha_n) \leq \phi(\beta_1, \ldots, \beta_n))$. Besides, taking advantage of this expansion, it is possible to *decompose* the inequality as a conjunction relating variables instead of type terms: the above constraint is indeed equivalent to $\exists\alpha_1 \ldots \alpha_n.(\alpha = \phi(\alpha_1, \ldots, \alpha_n) \wedge \alpha_1 \leq^{\nu_1} \beta_1 \wedge \cdots \wedge \alpha_n \leq^{\nu_n} \beta_n)$ (where $\mathrm{sig}(\phi) = [\nu_1, \ldots, \nu_n]$). Although it is logically correct, the result does not explicitly record the fact that the variables $\alpha_1, \ldots, \alpha_n$ have been introduced by the expansion, as *names* for α's sub-terms. This is why we provide a dedicated construct for binding variables generated by expansion: $\exists\langle\phi(\bar{\alpha}) \doteq \alpha\rangle.C$ requires α to be a ϕ type and the "fresh" variables $\bar{\alpha}$ are bound to its sub-terms in C. We moreover require α not to appear free in C. This can naturally be encoded as $\exists\bar{\alpha}.(\phi(\bar{\alpha}) = \alpha \wedge C)$. However, the former explicitly relates the introduced variables $\bar{\alpha}$ to the original one α, which guides the constraint solver.

$$\begin{array}{rcl} \rho \vdash R & \Leftrightarrow & \forall(\tau \diamond \tau') \in R \;\; \rho(\tau) \diamond \rho(\tau') \\ \rho \vdash \exists\langle\phi(\bar{\alpha}) \doteq \alpha\rangle.C & \Leftrightarrow & \exists\bar{t} \;\; \rho(\alpha) = \phi(\bar{t}) \text{ and } \rho[\bar{\alpha} \mapsto \bar{t}] \vdash C \end{array}$$

Figure 5: Interpretation of constraints

We extend the interpretation of constraints to the new language in the natural way, see figure 5 ($\diamond$ range over the symbols $\leq$, $\geq$ and $\approx$). We let an *atom* η be either a symbol of arity 0 or a type variable. A cset is *atomic* if and only if it involves only atoms, i.e. all its elements have the form $\eta_1 \diamond \eta_2$.

5.3 Restricting universal quantification

We now explain the restriction our algorithm imposes on the form of constraints allowed in data-type declarations, and, thereby those which appear in the left-hand-side of $\forall\bar{\beta}.D \Rrightarrow C$ constructs. In fact, the restriction concerns not just the constraint D, but the pair of the list of quantified variables $\bar{\beta}$ and the constraint D, which we refer to as a *quantification bound*.

It is worth noting that supposing the constraint D to have no existential quantifier inside does not affect expressiveness: indeed, every variable which is existentially quantified in D can be made universally quantified up front (e.g., with the appropriate capture-avoiding renamings, the constraint $\forall\bar{\beta}.(\exists\alpha.D) \Rrightarrow C$ is equivalent to $\forall\bar{\beta}\alpha.D \Rrightarrow C$). Besides, type constructors can be removed from quantification bounds by the standard expansion and decomposition process. For instance $\forall\beta.(\beta \leq \alpha_1 \to \alpha_2) \Rrightarrow C$ is equivalent to $\forall\beta_1\beta_2.(\alpha_1 \leq \beta_1 \wedge \beta_2 \leq \alpha_2) \Rrightarrow C[\beta_1 \to \beta_2/\beta]$. Similarly, thanks to the weak interpretation of universal quantification, $\forall\beta_1\beta_2.(\beta_1 \to \beta_2 \leq \alpha) \Rrightarrow C$ can be rewritten into $\exists\langle\alpha \doteq \alpha_1 \to \alpha_2\rangle.(\forall\beta_1\beta_2.(\alpha_1 \leq \beta_1 \wedge \beta_2 \leq \alpha_2) \Rrightarrow C[\alpha_1 \to \alpha_2/\alpha])$.

Combining these observations, it is natural to require the constraint of a quantification bound $\forall\bar{\beta}.D \Rrightarrow \cdots$ to be a conjunction of inequalities involving atoms (which, abusing notations, we consider to be a cset). These inequalities allow to relate together the variables of $\bar{\beta}$ and to limit their range by some external bounds (i.e. ground atoms or variables not in $\bar{\beta}$). So, considering a variable β of $\bar{\beta}$, three situations may arise:

1. β has no bound in D, i.e. is only related to variables of $\bar{\beta}$. In this case, no assumption can be made about its structure in the right-hand-side of the implication: for instance, the constraint $\forall\beta.\mathbf{true} \Rrightarrow (\beta \leq \alpha_1 \to \alpha_2)$ is not satisfiable, because β cannot be restricted to range only over arrow types.

2. β has one lower and/or one upper bound(s) in D. Consider for instance the constraint $C_1 = \forall\beta.(\beta \leq$

$\alpha) \Rightarrow (\beta \leq \alpha'_1 \to \alpha'_2)$. Because of the weak semantics of $\forall$, this constraint implies $\exists\beta.(\beta \leq \alpha \wedge \beta \leq \alpha'_1 \to \alpha'_2)$. Hence, any solution of C_1 maps β's bound, α, to an arrow type. This allows expanding α as $\alpha_1 \to \alpha_2$ and it remains to solve $C_2 = \forall\beta.(\beta \leq \alpha_1 \to \alpha_2) \Rightarrow (\beta \leq \alpha'_1 \to \alpha'_2)$: C_1 is indeed equivalent to $\exists\langle\alpha \doteq \alpha_1 \to \alpha_2\rangle.C_2$. As explained above, C_2 is equivalent to $\forall\beta_1\beta_2.(\alpha_1 \leq \beta_1 \wedge \beta_2 \leq \alpha_2) \Rightarrow (\alpha'_1 \leq \beta_1 \wedge \beta_2 \leq \alpha'_2)$. Because α_1 (resp. α_2) is the unique lower (resp. upper) bound of β_1 (resp. β_2), the latter can definitely be assigned to the former in the right-hand-side of the implication. Then C_2 naturally implies $(\alpha'_1 \leq \alpha_1 \wedge \alpha_2 \leq \alpha'_2)$. Next, it is easy to check that these two constraints are in fact equivalent, which completes the resolution.

3. β has several lower or upper bounds in D. In this case, the same decomposition principle applies as in the previous case. However, it may produce constraints whose resolution is difficult to deal with in an efficient manner. To illustrate this point, let us consider the constraint $C_3 = \forall\beta.(\beta \leq \alpha_1 \wedge \beta \leq \alpha_2) \Rightarrow (\beta \leq \alpha)$, which can be rewritten—using the least-upper-bound operator of the lattice as syntactic sugar—into $\forall\beta.(\beta \leq \alpha_1 \sqcap \alpha_2) \Rightarrow (\beta \leq \alpha)$. By the same reasoning as above, C_3 is equivalent to $(\alpha_1 \sqcap \alpha_2 \leq \alpha)$. However, the $\sqcap$ operator now appears in the left-hand-side of an inequality. Designing efficient algorithms for solving such inequalities seems to be a difficult problem, because they encode a form of disjunction.

That is the reason why we will restrict our attention to the first two cases; as our examples will show, we believe they are expressive enough for most usages (see section 6.2).

We now precisely formalize the assumption we make about quantification bounds. For this purpose, we introduce *constraint graphs*: a constraint graph $\mathcal{G}$ of size n is defined as a subset of $\{1,\ldots,n\} \times \{1,\ldots,n\}$. Then, given a list of n types $\tau_1,\ldots,\tau_n$, we let the *graph instance* $\mathcal{G}[\tau_1,\ldots,\tau_n]$ be the conjunction $\wedge\{\tau_i \leq \tau_j \mid (i,j) \in \mathcal{G}\}$. A node i is said to be the lower (resp. upper) bound of $\mathcal{G}$ if and only if for all j, the pair (i,j) (resp. (j,i)) belongs to $\mathcal{G}$. Abusing terminology, we also say in this case that τ_i is the lower (resp. upper) bound of $\mathcal{G}[\tau_1,\ldots,\tau_n]$. Then, we require the constraint D of any quantification bound $\forall\bar{\beta}.D \Mapsto \cdots$ to be writable as a conjunction of graph instances $\mathcal{G}_1[\bar{\tau}_1] \wedge \cdots \wedge \mathcal{G}_n[\bar{\tau}_n]$ such that every variable of $\bar{\beta}$ has at most one occurrence in $\bar{\tau}_1,\ldots,\bar{\tau}_n$, and, for all i, the graph instance $\mathcal{G}_i[\bar{\tau}_i]$ is

- either *unbounded*: every term in $\bar{\tau}_i$ is a variable of $\bar{\beta}$,
- or *bounded*: all variables in $\bar{\tau}_i$ are in $\bar{\beta}$, except those of two terms of $\bar{\tau}_i$ which are respectively lower and upper bounds of the graph.

These correspond to the cases 1 and 2 of the above discussion, respectively. Here are some examples of quantification bounds:

(1) $\forall\beta_1\beta_2\beta_3.(\beta_1 \leq \beta_2 \leq \beta_3) \Mapsto \cdots$
(2) $\forall\beta_1\beta_2.(\alpha_1 \leq \beta_1 \leq \alpha_2 \wedge \alpha_1 \leq \beta_2 \leq \alpha_2) \Mapsto \cdots$
(3) $\forall\beta_1\beta_2.(\varphi_1 \leq \beta_1 \leq \beta_2 \leq \varphi_2) \Mapsto \cdots$

Each of these three examples involve a single graph instance. In (1), the constraint $\beta_1 \leq \beta_2 \leq \beta_3$ is the unbounded instance $\mathcal{G}_1[\beta_1,\beta_2,\beta_3]$ of the graph $\mathcal{G}_1 = \{(1,2);(2,3)\}$. On the contrary, the graph instances in (2) and (3) are bounded: in the former, the bounds are the free variables α_1 and α_2 and the latter is bounded by the ground atoms φ_1 and φ_2.

An atom η is *unbounded* under the quantification $\forall\bar{\beta}.D \Mapsto \cdots$ if and only if it is a variable of $\bar{\beta}$ which appears in a unbounded graph of D. Otherwise, it is *bounded*. Given a graph $\mathcal{G}$, we let $\mathcal{G}^{\oplus}$ be $\mathcal{G}$, $\mathcal{G}^{\ominus}$ be the graph obtained by reversing every edge in $\mathcal{G}$ (i.e. $\{(i,j) \mid (j,i) \in \mathcal{G}\}$) and $\mathcal{G}^{\odot}$ be $\mathcal{G}^{\oplus} \cup \mathcal{G}^{\ominus}$. We let $\mathrm{bnd}_{\bar{\beta}}(D)$ be the set of inequalities $\alpha_1 \leq \alpha_2$ such that α_1 and α_2 are respectively lower and upper bounds of a graph instance of D. We write D^* for the closure of D, that is the smallest cset which contains D, every trivial constraint $\alpha \diamond \alpha$, and such that if $\eta_1 \diamond_1 \eta_2 \in D^*$ and $\eta_2 \diamond_2 \eta_3 \in D^*$ then $\eta_1 \diamond_1\diamond_2 \eta_3 \in D^*$.

5.4 The algorithm

The constraint resolution algorithm consists in the reduction $\rightsquigarrow$ defined in figure 6. Its broad outline is to rewrite the input constraint into a *solved form* S (in the case where the input constraint is not satisfiable, the algorithm can also return the special constant **failure**). Solved forms compose a subset of constraints defined as follows:

$$S ::= R \mid \exists\langle\phi(\bar{\alpha}) \doteq \alpha\rangle.S$$

In short, a solved form is a cset R preceded by a list of binders which record the type structure exhibited by expansion.

The first group of rules in figure 6 governs expansion and decomposition in a cset, with the purpose of making it atomic: rule (A) expands a variable which has some known structure and (B) decomposes an elementary constraint between two types with the same head constructor. If a cset relates two incompatible types, then it is not satisfiable, as reflected by (C).

The three next groups of rules deal with the constructs $[] \wedge \tau_1 \leq \tau_2$, $\exists\alpha.[]$ and $\forall\bar{\beta}.D \Rightarrow []$, respectively: because they are not allowed in solved forms, they must be eliminated. Roughly speaking, these rules act by pushing down the construct at hand until it reaches the cset of the constraint; and then it can be eliminated by operating on the cset. Sketching a parallel with an implementation, one could say that solved forms correspond to the memory representation of constraints; the three constructs of the constraint language stand for the *functions* provided to the client of the implementation to incrementally build constraints; and the three corresponding sets of rules can be read as the (recursive) definition of these functions.

Rules (D) and (E) deal with $[] \wedge \tau_1 \leq \tau_2$: the former permutes a conjunction $\wedge$ with a binder $\exists$ (because the variable α is not allowed to appear in the context $\exists\langle\phi(\bar{\alpha}) \doteq \alpha\rangle.[]$, it must be substituted in the types τ_1 and τ_2) while the latter removes an inner-most occurrence of $\wedge$ by inserting the inequality in the cset of the constraint.

Rules of the next group deal with existential quantification: (F) and (G) intend to commute an existential quantification $\exists\beta.[]$ with a binder $\exists\langle\phi(\bar{\alpha}) \doteq \alpha\rangle.[]$. If β is α then (F) applies and the quantification on α disappears. Otherwise, the two quantifiers can be safely commuted by (G). When an existential quantification reaches an atomic cset then it is eliminated by (H) which performs a local transitive closure of the graph described by R. $\diamond_1\diamond_2$ stands for the concatenation of $\diamond_1$ and $\diamond_2$, which is $\leq$ (resp. $\geq$) if $\diamond_1$ and $\diamond_2$ are both $\leq$ (resp. $\geq$) and $\approx$ otherwise; $(R\backslash\alpha)$ stands for the subset of R's constraints which do not involve α, i.e. $\{\eta_1 \diamond \eta_2 \mid \eta_1 \diamond \eta_2 \in R \text{ and } \eta_1, \eta_2 \neq \alpha\}$.

Expansion and decomposition

(A)	R	$\rightsquigarrow \exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.(R[\phi(\bar\alpha)/\alpha])$	$\alpha \diamond \phi(\bar\tau) \in R, \bar\alpha \# \mathrm{fv}(R)$
(B)	$\{\phi(\bar\tau) \diamond \phi(\bar\tau')\} \cup R$	$\rightsquigarrow \{\tau_1 \diamond^{\nu_1} \tau_1', \ldots, \tau_n \diamond^{\nu_n} \tau_n'\} \cup R$	$\mathrm{sig}(\phi) = [\nu_1, \ldots, \nu_n]$
(C)	$\{\phi(\bar\tau) \diamond \varphi(\bar\tau')\} \cup R$	$\rightsquigarrow$ **failure**	$\phi \neq \varphi$

Insertion of an inequality

(D)	$(\exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.S) \wedge \tau_1 \leq \tau_2$	$\rightsquigarrow \exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.(S \wedge \tau_1[\phi(\bar\alpha)/\alpha] \leq \tau_2[\phi(\bar\alpha)/\alpha])$	
(E)	$R \wedge \tau_1 \leq \tau_2$	$\rightsquigarrow R \cup \{\tau_1 \leq \tau_2\}$	

Existential quantification

(F)	$\exists\alpha.(\exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.S)$	$\rightsquigarrow \exists\bar\alpha.S$	
(G)	$\exists\beta.(\exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.S)$	$\rightsquigarrow \exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.(\exists\beta.S)$	$\beta \notin \bar\alpha$
(H)	$\exists\alpha.R$	$\rightsquigarrow \{\eta_1 \diamond_1 \diamond_2 \eta_2 \mid \eta_1 \diamond_1 \alpha, \alpha \diamond_2 \eta_2 \in R; \eta_1, \eta_2 \neq \alpha\} \cup (R \backslash \alpha)$	R atomic

Universal quantification

Scope-extrusion

(I)	$\forall\bar\beta.D \Rightarrow (\exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.S)$	$\rightsquigarrow \exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.(\forall\bar\beta.D[\phi(\bar\alpha)/\alpha] \Rightarrow S)$	$\alpha \notin \bar\beta, \bar\alpha \# (\bar\beta \cup \mathrm{fv}(D))$
(J)	$\forall\alpha\bar\beta.D \Rightarrow (\exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.S)$	$\rightsquigarrow \forall\bar\alpha\bar\beta.D[\phi(\bar\alpha)/\alpha] \Rightarrow S$	α bounded in D and $\bar\alpha \# (\bar\beta \cup \mathrm{fv}(D))$
(K)	$\forall\bar\beta\alpha.D \Rightarrow (\exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.S)$	$\rightsquigarrow$ **failure**	α unbounded in D

Expansion and decomposition

(L)	$\forall\bar\beta\alpha.D \Rightarrow R$	$\rightsquigarrow \forall\bar\beta\bar\alpha.(D[\phi(\bar\alpha)/\alpha]) \Rightarrow (R[\phi(\bar\alpha)/\alpha])$	$\alpha \diamond \phi(\bar\tau) \in D, \bar\alpha \# \mathrm{fv}(D, R, \bar\beta)$
(M)	$\forall\bar\beta.D \Rightarrow R$	$\rightsquigarrow \exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.(\forall\bar\beta.(D[\phi(\bar\alpha)/\alpha]) \Rightarrow (R[\phi(\bar\alpha)/\alpha]))$	$\alpha \diamond \phi(\bar\tau) \in D, \bar\alpha \# \mathrm{fv}(D, R, \bar\beta), \alpha \notin \bar\beta$
(N)	$\forall\bar\beta.(D \cup \mathcal{G}[\phi(\bar\tau_1), \ldots, \phi(\bar\tau_k)]) \Rightarrow R$	$\rightsquigarrow \forall\bar\beta.(D \cup_i \mathcal{G}^{\nu_i}[\tau_{1,i}, \ldots, \tau_{k,i}]) \Rightarrow R$	$\mathrm{sig}(\phi) = [\nu_1, \ldots, \nu_n]$
(O)	$\forall\bar\beta.(D \cup \{\phi(\bar\tau) \leq \phi'(\bar\tau')\}) \Rightarrow R$	$\rightsquigarrow$ **failure**	$\phi \neq \phi'$

Elimination

(P)	$\forall\bar\beta.D \Rightarrow R$	$\rightsquigarrow \mathrm{bnd}_{\bar\beta}(D) \cup \{\mathrm{ub}_{\bar\beta.D}(\eta_1) \leq \mathrm{lb}_{\bar\beta.D}(\eta_2) \mid \eta_1 \leq \eta_2 \in R \backslash D^*\}$ $\cup \{\mathrm{sk}_{\bar\beta.D}(\eta_1) \approx \mathrm{sk}_{\bar\beta.D}(\eta_2) \mid \eta_1 \approx \eta_2 \in R \backslash D^*\}$	D and R atomic
(Q)	$\forall\bar\beta.D \Rightarrow R$	$\rightsquigarrow$ **failure**	D and R atomic and $\beta \diamond \eta \in (R \backslash D^*)$ for some $\beta \in \bar\beta$ unbounded in D

Contexts

(R)	$\mathbb{C}[C]$	$\rightsquigarrow \mathbb{C}[C']$	if $C \rightsquigarrow C'$
(S)	$\mathbb{C}[\mathbf{failure}]$	$\rightsquigarrow$ **failure**	

$$\mathbb{C} ::= [] \wedge \tau \leq \tau \mid \exists\alpha.[] \mid \forall\bar\beta.D \Rightarrow [] \mid \exists\langle\phi(\bar\alpha) \doteq \alpha\rangle.[]$$

Figure 6: Solving constraints in structural subtyping

The fourth group of rules handles universal quantifiers. Rules (I), (J) and (K) deal with binders $\exists$ which appear in the right-hand-side of the implication. If the bound variable, α, is not captured by the universal quantification, then the two quantifiers may be commuted as reflected by (I). It is worth noting that, in general, existential and universal quantifiers cannot be swapped; however this is possible with the restricted form of existential quantification provided by binders $\exists$, because the introduced variables $\bar{\alpha}$ are indeed fully determined by α. The cases where the variable α is universally bound are handled by (J) and (K): if α appears in an unbounded graph of D then the latter applies: the constraint is not satisfiable because α cannot be constrained to range only over ϕ types. On the contrary, if α has bounds in D, it is possible to require them to be ϕ types, hence (J). Rules (L) to (O) perform expansion and decomposition of the constraint D which delimits the range of universal quantification. In conjunction with (A) and (B) they aim at making R and D both atomic. Lastly, rule (P) allows the elimination of universal quantifiers by rewriting a constraint of the form $\forall\bar{\beta}.D \Rightarrow R$ where D and R are atomic into a simple cset. The built cset first contains $\mathrm{bnd}_{\bar{\beta}}(D)$, which is equivalent to $\exists\bar{\beta}.D$. Second, the algorithm considers every constraint $\eta_1 \diamond \eta_2$ of the cset R. Two cases may arise: if it appears in D^*, then it is satisfied by any solution of D so that it can be forgotten. Otherwise, another elementary constraint, which is necessary and sufficient for every solution of D to satisfy $\eta_1 \diamond \eta_2$ (i.e. equivalent to $\forall\bar{\beta}.D \Rightarrow \eta_1 \diamond \eta_2$), is generated:

- In the case where $\eta_1 \diamond \eta_2$ is $\eta_1 \leq \eta_2$, we consider the greatest assignment $\mathrm{ub}_{\bar{\beta}.D}(\eta_1)$ the atom η_1 may receive in the quantification bound $\forall\bar{\beta}.D \Mapsto \cdots$: if η_1 is not one of the quantified variables (i.e. $\eta_1 \notin \bar{\beta}$), $\mathrm{ub}_{\bar{\beta}.D}(\eta_1)$ is naturally η_1 itself; if $\eta_1 \in \bar{\beta}$ and η_1 is bounded in D then $\mathrm{ub}_{\bar{\beta}.D}(\eta_1)$ is the unique lower bound of η_1 in D. We symmetrically define $\mathrm{lb}_{\bar{\beta}.D}(\eta_2)$ and generate the constraint $\mathrm{ub}_{\bar{\beta}.D}(\eta_1) \leq \mathrm{lb}_{\bar{\beta}.D}(\eta_2)$.
- The case where $\eta_1 \diamond \eta_2$ is $\eta_1 \approx \eta_2$ is dealt with similarly. We let $\mathrm{sk}_{\bar{\beta}.D}(\eta)$ be a representative of η's $\approx$ equivalence class under the bound $\forall\bar{\beta}.D \Mapsto \cdots$, e.g. $\mathrm{lb}_{\bar{\beta}.D}(\eta)$ or $\mathrm{ub}_{\bar{\beta}.D}(\eta)$. Then the constraint $\mathrm{sk}_{\bar{\beta}.D}(\eta_1) \approx \mathrm{sk}_{\bar{\beta}.D}(\eta_2)$ is generated.

If one of η_1 and η_2, say η_1, is unbounded in $\forall\bar{\beta}.D \Mapsto \cdots$, $\mathrm{lb}_{\bar{\beta}.D}(\eta_1)$, $\mathrm{ub}_{\bar{\beta}.D}(\eta_1)$ and $\mathrm{sk}_{\bar{\beta}.D}(\eta_1)$ are not defined. Then, (P) cannot be applied, but (Q) yields a failure.

Because recursive types are not valid solutions for constraints in the ground model, one must ensure that the elementary constraints of a solved form do not require cyclic type structures. This verification is commonly referred to as the *occur-check*; because it is standard, we omit its formal description which can be found in previous works [Sim03c]. What is more, this check is necessary to guarantee the termination of expansion and decomposition: as a consequence, it must be performed when expanding a cset or the quantification bound of the inner-most universal quantification, if any.

In the full version of this paper [Sim03b], we give a formal proof that the reduction $\rightsquigarrow$ defines an algorithm rewriting constraints that satisfy the occur-check into an atomic solved form or fails. This addresses the question of constraint resolution. Indeed, solving atomic solved forms is a well-known issue [FM89]: it only requires to check that every path between ground atoms in the graph defined by the inequalities of the cset is valid w.r.t. the partial order $\leq_0$.

6 Discussion and examples

6.1 On possible extensions

The programming language studied in this paper is reduced to its core. However, it can naturally be extended by adding constants in the language, appropriate δ-rules in the semantics, new type constructors and the corresponding typing rules, which should not raise any particular issue. Furthermore, the solving algorithm of section 5 is already equipped for such extensions, because it makes no particular assumption about the set of type constructors.

We did not introduce in the description of the solving procedure any procedure for simplifying constraints; nevertheless, such techniques are mandatory to obtain efficient and scalable algorithms. In [Sim03c], we formalized a complete resolution framework (for standard structural subtyping constraints) that includes a series of simplification techniques, many of which rely on *polarities* assigned to type variables. These techniques can be generalized in the presence of universal quantifiers, considering universally quantified variables as bipolar. What is more, because the mechanism described in section 5 permits to incrementally eliminate universal quantifications from constraints, extending an existing constraint solver for structural subtyping can be done with a limited effort. Indeed, a reasonable strategy consists in, every time a constraint $\forall\bar{\beta}.D \Rightarrow S$ is built, simplifying the constraint S, and then eliminating the universal quantification. Such an extension would preserve most of the existing implementation, in particular the internal representation of constraints.

It would be an interesting question to design algorithms for solving constraints generated by $\mathrm{HM}_{\exists\forall}(X)$ which handle other forms of subtyping. We considered the case of non-structural subtyping where the set of types forms a lattice, as in [Pot00]. Limiting quantification bounds to be conjunctions of inequalities between universally quantified variables, we believe it is possible to solve the constraints generated by $\mathrm{HM}_{\exists\forall}(X)$'s inference algorithm by considering these variables as new symbols in the types lattice—as Skolem types—and to translate the constraint bounding quantification as axioms about the order between symbols. However, further studies are necessary to precisely formalize and prove this possibility.

Lastly, several researchers have recently proposed conservative extensions of ML which offer higher-order polymorphism in a more flexible way. Odersky and Läufer [OL96] studied a type system that allows explicit type scheme annotations for function arguments. Garrigue and Rémy designed PolyML [GR99], where the elimination of polymorphic types is said *semi-explicit*: the source code must mention only the points where polymorphic values are used, their types can be inferred. Subsuming PolyML, ML^{F} [LBR03] made the opening of polymorphic values fully implicit. Actually, these works rely on unification mechanisms and do not consider subtyping; extending them in this direction remains to be explored. Other proposals try to combine subtyping in combination with higher-order polymorphism and type inference [Car93, PT00, OZZ01]; however, they fail to type all ML programs.

6.2 On information flow analysis

We plan to integrate the mechanism of abstract and polymorphic data-types described in the current paper in the

Flow Caml system, our prototype information flow analyzer for the Caml language. We here give a taste of that by considering some examples, which illustrate the expressiveness of $\mathrm{HM}_{\exists\forall}(X)$ in a concrete setting. For the sake of simplicity, we omit some of the security annotations in Flow Caml types (in particular those on top of $\rightarrow$), because they are out of the topic of the current paper.

We follow up on the example of a bank's clients file introduced in section 2. As we have explained, the type of every value giving some information about a client is labeled by its security level: for instance `!alice int` is the type of an integer carrying information about the client `!alice`. The set of security levels is supposed to be equipped with a lattice structure, so we have in particular a security level `!clients` which is the least upper bound of all clients security levels. The framework presented in the current paper allows to get round the problem we pointed out in section 2 by defining the type `client_info` as follows:

```
type client_info = Exists 'a with 'a < !clients .
  { cash : 'a int;
    send_msg : 'a int -> unit;
    ... }
```

This declaration mentions one existentially quantified variable, `'a`, which is the security level of the client, hence it must be less than `!clients`, as reflected by the constraint `'a < !clients` which bounds the quantification (the lower bound of `'a` is implicitly the bottom element, $\bot$, of the security level lattice). Because `'a` is local to the declaration, it does not appear as a parameter of the type. As a result, all client records have the same type, `client_info`, and can be stored in a list of type `client_info list`.

Then, one may define a function which iterates on a list of clients and sends to each of them a message containing its current balance:

```
let rec send_balances = function
  [] -> []
| { cash = x; send_msg = send } :: tl ->
    send x; send_balances tl
```

We combine the elimination of the existential quantification with the matching of the record structure, which allows an intuitive and lightweight syntax. De-sugaring this example in the syntax of the current paper, we realize that the function which corresponds to the second case of the pattern matching

$$\lambda x.send.tl.(send\, x; send\_balances\, tl)$$

must, to preserve type abstraction, have a type scheme where the security level of the integer `x` and `f`'s argument can range over all levels less than `!clients`, i.e.

$$\forall\alpha[\alpha \leq \texttt{!clients}].\\ \alpha\ \mathsf{int} \rightarrow (\alpha\ \mathsf{int} \rightarrow \mathsf{unit}) \rightarrow \mathsf{client\_info\ list} \rightarrow \mathsf{unit}$$

which is naturally the case in our example. On the contrary, consider the following ill-typed piece of code:

```
let illegal_flow = function
  { cash = x1 } :: { send_msg = f2 } :: _ -> f2 x1
| _ -> ()
```

In the case where the clients list comprises at least two entries, this function sends the balance of the first client to the second one. This is not allowed by the bank's security policy and rejected by the type system: in the first clause of the pattern-matching, the pattern introduces two variables, β_1 and β_2, which are the respective existentially quantified variables of the two opened records, ranging independently on security levels less than `!clients`. The clause's body requires β_1 to be less than or equal to β_2, this yields the constraint $\forall\beta_1\beta_2.(\beta_1 \sqcup \beta_2 \leq \texttt{!clients}) \Rrightarrow (\beta_1 \leq \beta_2)$ which is not satisfiable, hence the typing failure.

Our next example is a function which computes the total balance of the bank from a clients file:

```
let rec total = function
  [] -> 0
| { cash = x } :: tl -> x + total tl
```

The result of `total` is an integer which is likely to carry information about any opened record, hence it must be labeled by the least upper bounds of their security levels, which is `!clients`. Hence, this function receives the type client_info list $\rightarrow$ `!clients` int.

As pointed out in the introduction, high order functions operating on existential types generally require second order polymorphic types. For instance, let us try to write a function `check` which takes as argument a record and a function which performs some computation on the client's balance:

```
let check f { cash = x; send_msg = send } =
  send (f x)
```

To preserve abstraction, this definition requires the function `f` to have the type `!clients` int $\rightarrow \bot$ int, i.e. to accept any argument whose level is less than or equal to `!clients` and to produce a result of level $\bot$. Such a type constrains `f`'s result not to depend on its argument, which is naturally not acceptable. A solution consists in packing the function `f` in a polymorphic data-type:

```
type t = ForAll 'a . { op: 'a int -> 'a int }
```

This allows to write:

```
let check { op = f }
          { cash = x; send_msg = send } =
  send (f x)
```

which has the type `t -> client_info -> unit`. In fact, the data-type `t` provides an encoding of second order polymorphic types; some syntactic sugar avoiding the prior declaration of the type `t` would be convenient in such a situation.

Our main direction for future work is to study the possibility to make security levels also *values* of the Flow Caml language. This would permit some dynamic tests—whose correctness must be verified statically—on existentially quantified variables when opening data structures. Continuing with our example, this would allow—for instance—computing the total balance of a subset of clients.

References

[AF97] Alexander S. Aiken and Manuel Fähndrich. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, Lecture Notes in Computer Science LNCS, pages 114–126, Paris, France, September 1997. Springer Verlag. URL: `http://www.cs.berkeley.edu/~aiken/publications/papers/sas97.ps`.

[Aug94] L. Augustsson. Haskell B. user's manual, October 1994.

[Car93] Luca Cardelli. An implementation of FSub. Technical Report 97, Digital Equipment Corporation Systems Research Center, 1993.

[CP01] Sylvain Conchon and François Pottier. JOIN(X): Constraint-based type inference for the join-calculus. In *Proceedings of the 10th European Symposium on Programming*, Lecture Notes in Computer Science LNCS, pages 221–236. Springer Verlag, April 2001. URL: http://pauillac.inria.fr/~fpottier/publis/conchon-fpottier-esop01.ps.gz.

[FM89] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proceedings of the European Joint Conference on Theory and Practice of Software Development*, Lecture Notes in Computer Science LNCS, pages 167–183, Berlin, March 1989. Springer Verlag.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2002. URL: http://http.cs.berkeley.edu/~jfoster/papers/pldi02.ps.gz.

[GR99] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. *Journal of Functional Programming*, 155(1/2):134–168, 1999.

[HM95] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 176–185, New York, NY, USA, January 1995. ACM Press.

[HR97] Fritz Henglein and Jakob Rehof. The complexity of subtype entailment for simple types. In *Proceedings of the 12th IEEE Symposium on Logic in Computer Science*, pages 352–361, June 1997. URL: http://research.microsoft.com/~rehof/lics97.ps.

[KR03] Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*, June 2003.

[LBR03] Didier Le Botlan and Didier Rémy. ML$^{\mathrm{F}}$: Raising ML to the power of system F. In *Proceedings of the 8th ACM International Conference on Functional Programming*, Uppsala, Sweden, August 2003. URL: http://pauillac.inria.fr/~remy/work/mlf/icfp.pdf.

[LDG+] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system. URL: http://caml.inria.fr/.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, 1991.

[MP88] J. Mitchell and G. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[MP93] Michel Mauny and François Pottier. An implementation of Caml Light with existential types. Research Report 2183, Institut de Recherche en Informatique et en Automatique , 1993. URL: http://pauillac.inria.fr/~fpottier/publis/rapport-maitrise.ps.gz.

[OL92] Martin Odersky and Konstantin Läufer. An extension of ML with first-class abstract types. In *Proceedings of the ACM Workshop on ML and its Applications*, pages 78–91, June 1992. URL: ftp://ftp.math.luc.edu/pub/laufer/papers/ml+extypes.ps.gz.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 54–67. ACM Press, January 1996.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. URL: http://www.comp.nus.edu.sg/~sulzmann/publications/tapos.ps.

[OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, March 2001.

[Per90] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, University of London, 1990.

[Pot00] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000. URL: http://pauillac.inria.fr/~fpottier/publis/fpottier-njc-2000.ps.gz.

[Pot01] François Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, Institut de Recherche en Informatique et en Automatique , March 2001. URL: ftp://ftp.inria.fr/INRIA/publication/RR/RR-4150.ps.gz.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003. URL: http://cristal.inria.fr/~simonet/publis/fpottier-simonet-toplas.ps.gz.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

[Rém94] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Science*, pages 321–346, Sendai, Japan, April 1994. Springer Verlag. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/tacs94.ps.gz.

[Sim03a] Vincent Simonet. *Flow Caml*, information flow inference in Objective Caml. URL: http://cristal.inria.fr/~simonet/soft/flowcaml/, June 2003.

[Sim03b] Vincent Simonet. An extension of HM(X) with bounded existential and universal data-types. Full version. URL: http://cristal.inria.fr/~simonet/publis/simonet-ifcp03-full.ps.gz, August 2003.

[Sim03c] Vincent Simonet. Type inference with structural subtyping: The faithful formalization of an efficient constraint solver. Submitted for publication. URL: http://cristal.inria.fr/~simonet/publis/simonet-structural-subtyping.ps.gz, March 2003.

[SOW97] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *Proceedings of the 4th International Workshop on Foundations of Object-Oriented Languages*, January 1997. URL: http://www.cis.upenn.edu/~bcpierce/fool/sulzmann.ps.gz.

[Tiu92] Jerzy Tiuryn. Subtype inequalities. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*, pages 308–317, Santa Cruz, CA, June 1992. IEEE Computer Society Press.

[Wri93] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93-200, Rice University, February 1993.