Typing Constraint Logic Programs

François Fages and Emmanuel Coquery
Projet Contraintes, INRIA-Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France,
(*e-mail:* {francois.fages,emmanuel.coquery}@inria.fr)

Abstract

We present a prescriptive type system with parametric polymorphism and subtyping for constraint logic programs. The aim of this type system is to detect programming errors statically. It introduces a type discipline for constraint logic programs and modules, while maintaining the capabilities of performing the usual coercions between constraint domains, and of typing meta-programming predicates, thanks to the flexibility of subtyping. The property of *subject reduction* expresses the consistency of a prescriptive type system w.r.t. the execution model: if a program is "well-typed", then all derivations starting from a "well-typed" goal are again "well-typed". That property is proved w.r.t. the abstract execution model of constraint programming which proceeds by accumulation of constraints only, and w.r.t. an enriched execution model with type constraints for substitutions. We describe our implementation of the system for type checking and type inference. We report our experimental results on type checking ISO-Prolog, the (constraint) libraries of Sicstus Prolog and other Prolog programs.

Keywords: Type systems, Prolog, Constraint logic programming, Subtyping, Metaprogramming.

1 Introduction

The class CLP($\mathcal{X}$) of Constraint Logic Programming languages was introduced by Jaffar and Lassez (Jaffar & Lassez, 1987) as a generalization of the innovative features introduced by Colmerauer in Prolog II (Colmerauer, 1984; Colmerauer, 1985): namely computing in Prolog with other structures than the Herbrand terms, with inequality constraints and with co-routining.

Inherited from the Prolog tradition, CLP($\mathcal{X}$) programs are untyped. Usually the structure of interest $\mathcal{X}$ is however a quite complex combination of basic structures that may include integer arithmetic, real arithmetic, booleans, lists, Herbrand terms, infinite terms, etc. with implicit coercions between constraint domains like in Prolog IV (Colmerauer, 1996). Even the early CLP($\mathcal{R}$) system of (Jaffar & Lassez, 1987) already combines Herbrand terms with arithmetic expressions in a non-symmetrical way: any arithmetic expression may appear under a Herbrand function symbol, e.g. in a list, but not the other way around. The framework of *many sorted logic* in (Jaffar & Lassez, 1987) is not adequate for representing the type system underlying such a combination, as it forces Herbrand function symbols to have a unique type (e.g. over reals or Herbrand terms), whereas Herbrand functions can be used polymorphically, e.g. in `f(1)` and `f(f(1))`, or the list constructor in a list of list of numbers `[[3]]`.

The type system of Mycroft-O'Keefe (Mycroft & O'Keefe, 1984; Lakshman & Reddy, 1991; Hill & Topor, 1992) is an adaptation to logic programming of the first type system with *parametric polymorphism*, that was introduced by Damas-Milner for the functional programming language ML. In this system, types are first-order terms, type variables inside types, like α in $list(\alpha)$, express type parameters. Programs defined over a data structure of type $list(\alpha)$ can be used polymorphically over any homogeneous list of elements of some type α. Such a type system for Prolog is implemented in the systems Gödel (Hill & Lloyd, 1994) and Mercury (Somogyi *et al.*, 1996) for example. The flexibility of parametric polymorphism is however by far insufficient to handle properly coercions between constraint domains, such as e.g. booleans as natural numbers, or lists as Herbrand terms, and does not support the meta-programming facilities of logic programming, with meta-predicates such as `functor(X,F,N)`, `call(G)` or `setof(X,G,L)`.

Semantically, a ground type represents a set of expressions. Subtyping makes type systems more expressive and flexible in that it allows to express inclusions among these sets. In this paper we investigate the use of subtyping for expressing coercions between constraint domains, and for typing meta-programming predicates. The idea is that by allowing subtype relations like $list(\alpha) \leq term$, an atom like $functor([X|L], F, N)$ is well-typed with type declaration $functor : term \times atom \times int \rightarrow pred$, although its first argument is a list. Similarly, we can type $call : pred \rightarrow pred$, $freeze : term \times pred \rightarrow pred$, $setof : \alpha \times pred \times list(\alpha) \rightarrow pred$. The absence of subtype relation $list(\alpha) \not\leq pred$, has for effect to raise a type error if the `call` predicate is applied to a list. On the other hand, the subtype relation $pred \leq term$ makes coercions possible from goals to terms.

Most type systems with subtyping for logic programming languages that have been proposed are descriptive type systems, i.e. their purpose is to describe the success set of the program, they require that a type for a predicate upper approximates its denotation. On the other hand, in prescriptive type systems, types are syntactic objects defined by the user to express the intended use of function and predicate symbols in programs. Note that the distinction between descriptive and prescriptive type systems is orthogonal to the distinction between type checking and type inference which are possible in both approaches.

There are only few works considering prescriptive type systems for logic programs with subtyping (Beierle, 1995; Dietrich & Hagl, 1988; Hanus, 1992; Hill & Topor, 1992; Yardeni *et al.*, 1992; Smolka, 1988). In these systems however, subtype relations between parametric type constructors of different arities, like $list(\alpha) \leq term$, are not allowed, thus they cannot be used to type metaprogramming predicates and have not been designed for that purpose. The system Typical (Meyer, 1996) possesses an ad hoc mechanism for typing metapredicates which makes it quite difficult to use. Our objective is to propose a simple type system that allows for a uniform treatment of prescriptive typing issues in constraint logic programs.

In a prescriptive type system, the property of *subject reduction* expresses the consistency of the type system w.r.t. the execution model: if a program is "well-typed", then all derivations starting in a "well-typed" goal are again "well-typed". This is a well-known result of the polymorphic type system without subtyping (Mycroft & O'Keefe, 1984; Lakshman & Reddy, 1991; Hill & Topor, 1992) but when subtypes are added to the picture, the absence of a fixed data flow in logic programs makes the obtention of a similar result problematical. Beierle (Beierle, 1995) shows the existence of principal typings with subtype relations between basic types, and provides type inference algorithms, however Beierle and also Hanus (Hanus, 1992) do not claim subject reduction for the systems they propose. In general types are kept at run-time (Hanus, 1992; Yardeni *et al.*, 1992) or modes are introduced to restrict the data flow (Dietrich & Hagl, 1988; Smaus *et al.*, 2000; Somogyi *et al.*, 1996).

In this paper, by abstracting from particular structures as required in the CLP scheme, we study a prescriptive type system for CLP programs, that is independent from any specific constraint domain $\mathcal{X}$. Section 2 presents the type system that includes parametric polymorphism and subtype relations between type constructors of different arities, in a quite general type structure of poset with suprema. We show two subject reductions results. One is relative to the abstract execution model of constraint programming, which proceeds only by accumulation of constraints. The proof of subject reduction holds independently of the computation domain, under the assumption that the type of predicates satisfies the definitional genericity principle (Lakshman & Reddy, 1991). The second subject reduction result is relative to the more concrete execution model of CLP with substitution steps. We show that for this second form it is necessary to keep at run-time the typing constraints on variables inside well-typed programs and queries.

Section 3 describes the type checking algorithm and shows that the system of subtype inequalities generated by the type checker are left-linear and acyclic. Section 4 presents a linear time algorithm for solving left-linear and acyclic systems of subtype inequalities, and describes the cubic time algorithm of Pottier (Pottier, 2000a) for solving general systems of inequalities, under the additional assumption that the types form a lattice. Section 5 presents type inference algorithms for inferring the types of variables and predicates in program clauses.

Section 6 describes our implementation which is available from (Coquery, 2000). The solving of subtype inequalities is done by an interface to the Wallace constraint-handling library (Pottier, 2000b). In section 7 we report our experimental results on the use of this implementation to type check ISO-Prolog, the libraries of Sicstus Prolog, including constraint programming libraries, and other Prolog programs.

2 Typed Constraint Logic Programs

In this section we describe our type system as a logic for deriving type judgments about CLP programs.

2.1 Types

The type system we consider is based on a structure of partially ordered terms, called poterms, that we use for representing types with both parametric polymorphism and subtype polymorphism. Poterms generalize first-order terms by the definition of a subsumption order based on function symbols, that comes in addition to the instantiation preorder based on variables. Poterms are similar to order-sorted feature terms or ψ-terms (Ait-Kaci & Nasr, 1986; Smolka, 1988; Ait-Kaci *et al.*, 1997) but we find it more convenient here to adopt a term syntax (with matching by position) instead of a record syntax (with matching by name) for denoting static types.

The set of types $\mathcal{T}$ is the set of terms formed over a denumerable set $\mathcal{U}$ of *type variables* (also called *parameters*), denoted by α, β,..., and a finite set of *constructors* $\mathcal{K}$, where with each symbol $K \in \mathcal{K}$ an arity $m \geq 0$ is associated (by writing K/m). *Basic types* are type constructors of arity 0. We assume that $\mathcal{K}$ contains a basic type *pred*. A *flat type* is a type of the form $K(\alpha_1,\ldots,\alpha_m)$, where $K \in \mathcal{K}$ and the α_i are distinct parameters.

The set of type variables in a type τ is denoted by $V(\tau)$. The set of ground types $\mathcal{G}$ is the set of types containing no variable. We write $\tau[\sigma/\alpha]$ to denote the type obtained by replacing all the occurrences of α by σ in τ. We write $\tau[\sigma]$ to denote that the type τ strictly contains the type σ as a subexpression. The size of a type τ, defined as the number of occurrences of constructors and parameters in τ, is denoted by size(τ).

We now qualify what kind of subtyping we allow. Intuitively, when a type σ is a subtype of a type τ, this means that each term in σ is also a term in τ. The subtyping relation $\leq$ is designed to have certain nice algebraic properties, stated in propositions below. We assume an order $\leq$ on type constructors such that: $K/m \leq K'/m'$ implies $m \geq m'$, and for each $K \in \mathcal{K}$ the set $\{K' \mid K \leq K'\}$ has a maximum. Moreover, we assume that with each pair $K/m \leq K'/m'$, an injective mapping $\iota_{K,K'} : \{1,\ldots,m'\} \to \{1,\ldots,m\}$ is associated such that $\iota_{K,K''} = \iota_{K,K'} \circ \iota_{K',K''}$ whenever $K \leq K' \leq K''$.

These assumptions mean that as we move up in the hierarchy of type constructors, their arity decreases, and the hierarchy needs not be a lattice but a poset with suprema.

The order on type constructors is extended to a *covariant subtyping order* on types, denoted also by $\leq$, defined as the least relation satisfying the following rules:

(Par)	$\alpha \leq \alpha$	α is a parameter
(Constr)	$\dfrac{\tau_{\iota(1)} \leq \tau'_1 \ \ldots\ \tau_{\iota(m')} \leq \tau'_{m'}}{K(\tau_1,\ldots,\tau_m) \leq K'(\tau'_1,\ldots,\tau'_{m'})}$	$K \leq K'$, $\iota = \iota_{K,K'}$.

Contravariant type constructors could be defined with a subtyping rule similar to rule *Constr* but with the ordering relation reversed for some arguments, like e.g. $\tau_{\iota(i)} \geq \tau'_i$ in the premise of the rule for some argument τ'_i. Such contravariant type constructors are not considered in this paper.

Therefore, if $int \leq float$ then we have $list(int) \leq list(float)$, $list(float) \not\leq$

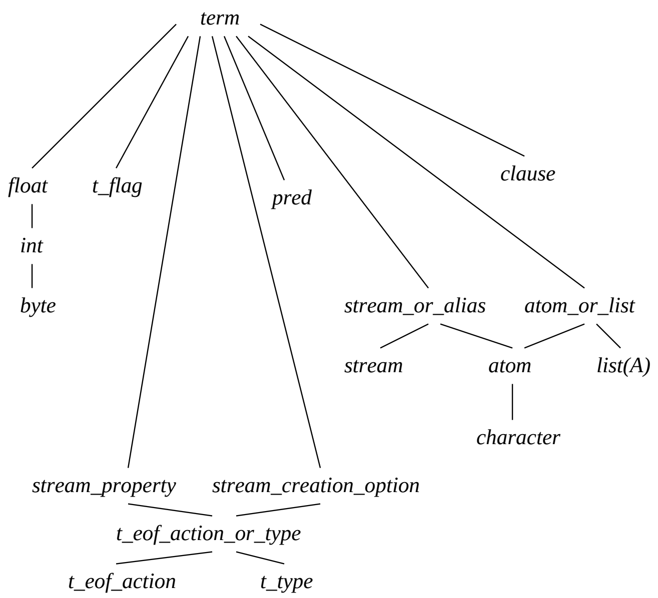

Fig. 1. Part of the type structure for ISO-Prolog.

$list(int)$, and also $list(float) \not\leq list(\alpha)$ as the subtyping order does not include the instantiation pre-order. Intuitively, a ground type represents a set of expressions, and the subtyping order between ground types corresponds to set inclusion. Parametric types do not directly support this interpretation, their parameters denote unkown types.

The type structure given in figure 2.1 represents a part of the types used for type checking ISO-Prolog. The omitted types are the subtypes of *atom* associated to all types, and other types for special values or options. The type $list(\alpha)$ is the only parametric type used for ISO-Prolog. Other parametric types are used for typing Prolog libraries such as $arrays(\alpha)$, $assoc(\alpha, \beta)$, $heaps(\alpha, \beta)$, $ordsets(\alpha)$, $queues(\alpha)$, etc.

A *type substitution* Θ is an idempotent mapping from parameters to types that is the identity almost everywhere. Applications of type substitutions are defined in the obvious way.

Proposition 2.1
If $\sigma \leq \tau$ then $\sigma\Theta \leq \tau\Theta$ for any type substitution Θ.

Proof
By structural induction on τ. $\square$

Proposition 2.2
If $\sigma \leq \tau$ then $\mathrm{size}(\sigma) \geq \mathrm{size}(\tau)$.

Proof

By structural induction on τ. $\square$

Our assumption that for each $K \in \mathcal{K}$, the set $\{K' \mid K \leq K'\}$ has a maximum, together with the arity decreasing assumption, entail the existence of a maximum supertype for any type:

Proposition 2.3
For each type τ, the set $\{\sigma \mid \tau \leq \sigma\}$ has a maximum, which is denoted by $\mathrm{Max}(\tau)$.

Proof
By structural induction on τ. $\square$

This means that every $\leq$-connected component of types has a root. For example, a structure like $a \leq b,\ c \leq b, c \leq d$ violates the hypothesis if b and d have no common supertype serving as a root for the connected component. On the other hand that assumption does not assume, nor it is implied by, the existence of a least upper bound to types having a upper bound (sup-quasi-lattice hypothesis in (Smolka, 1989)).

Proposition 2.4
For all types τ and σ, $\mathrm{Max}(\tau[\sigma/\alpha]) = \mathrm{Max}(\tau)[\mathrm{Max}(\sigma)/\alpha]$.

Proof
By structural induction on τ. $\square$

Note that the possibility of "forgetting" type parameters in subtype relations, as in $list(\alpha) \leq term$, may provide solutions to inequalities of the form $list(\alpha) \leq \alpha$, e.g. $\alpha = term$. However, we have:

Proposition 2.5
An inequality of the form $\alpha \leq \tau[\alpha]$ has no solution. An inequality of the form $\tau[\alpha] \leq \alpha$ has no solution if $\alpha \in V(\mathrm{Max}(\tau))$.

Proof
For any type σ, we have $\mathrm{size}(\sigma) < \mathrm{size}(\tau[\sigma])$, hence by Prop 2.2, $\sigma \not\leq \tau[\sigma]$, that is $\alpha \leq \tau[\alpha]$ has no solution.

For the second proposition, we prove its contrapositive. Suppose $\tau[\alpha] \leq \alpha$ has a solution, say $\tau[\sigma/\alpha] \leq \sigma$. By definition of a maximum and Prop. 2.3, we have $\mathrm{Max}(\sigma) = \mathrm{Max}(\tau[\sigma/\alpha])$. Hence by Prop. 2.4, $\mathrm{Max}(\sigma) = \mathrm{Max}(\tau)[\mathrm{Max}(\sigma)/\alpha]$. By the rules of subtyping we have $\alpha \neq \mathrm{Max}(\tau)$. Therefore $\alpha \notin V(\mathrm{Max}(\tau))$, since otherwise $\mathrm{Max}(\sigma) = \mathrm{Max}(\tau)[\mathrm{Max}(\sigma)/\alpha]$ would contain $\mathrm{Max}(\sigma)$ as a strict subexpression which is impossible. $\square$

2.2 *Well-typed programs*

CLP programs are built over a denumerable set $\mathcal{V}$ of *variables*, a finite set $\mathcal{F}$ of *function* symbols, given with their arity (constants are functions of arity 0), and a finite set $\mathcal{P}$ of *program predicate* and *constraint predicate* symbols given with their arity, containing the equality constraint $=$. A query Q is a finite sequence of constraints and atoms. A program clause is an expression noted $A \leftarrow Q$ where A is an atom formed with a program predicate and Q a query.

A *type scheme* is an expression of the form $\forall\overline{\alpha}\tau_1,\ldots,\tau_n \rightarrow \tau$, where $\overline{\alpha}$ is the set of parameters in types $\tau_1,...,\tau_n,\tau$. We assume that each function symbol $f \in \mathcal{F}$, has a *declared type scheme* of the form $\forall\overline{\alpha}\tau_1,\ldots,\tau_n \rightarrow \tau$, where n is the arity of f, and τ is a flat type. Similarly, we assume that each predicate symbol $p \in \mathcal{P}$ has a declared type scheme of the form $\forall\overline{\alpha}\tau_1,\ldots,\tau_n \rightarrow \mathit{pred}$ where n is the arity of p. The declared type of the equality constraint symbol is $\forall u\ u,u \rightarrow \mathit{pred}$. For notational convenience, the quantifiers in type schemes and the resulting type *pred* of predicates will be omitted in type declarations, the declared type schemes will be indicated by writing $f_{\tau_1\ldots\tau_n\rightarrow\tau}$ and $p_{\tau_1\ldots\tau_n}$, assuming a fresh renaming of the parameters in $\tau_1,\ldots,\tau_n,\tau$ for each occurrence of f or p.

Throughout this paper, we assume that $\mathcal{K}$, $\mathcal{F}$, and $\mathcal{P}$ are fixed by means of declarations in a *typed program*, where the syntactical details are insignificant for our results.

A *variable typing* is a mapping from a finite subset of $\mathcal{V}$ to $\mathcal{T}$, written as $\{x_1 : \tau_1,\ldots,x_n : \tau_n\}$. The type system defines well-typed terms, atoms and clauses relatively to a variable typing U. The typing rules are given in Table 1. The rules basically consist of the rules of Mycroft and O'Keefe plus the subsumption rule. Note that for the sake of simplicity constraints are not distinguished from other atoms in this system.

(Sub)	$\dfrac{U \vdash t{:}\tau \quad \tau \leq \tau'}{U \vdash t{:}\tau'}$	
(Var)	$\{x : \tau,\ldots\} \vdash x : \tau$	
(Func)	$\dfrac{U \vdash t_1{:}\tau_1\Theta \ \ldots\ U \vdash t_n{:}\tau_n\Theta}{U \vdash f_{\tau_1\ldots\tau_n\rightarrow\tau}(t_1,\ldots,t_n){:}\tau\Theta}$	Θ is a type substitution
(Atom)	$\dfrac{U \vdash t_1{:}\tau_1\Theta \ \ldots\ U \vdash t_n{:}\tau_n\Theta}{U \vdash p_{\tau_1\ldots\tau_n}(t_1,\ldots,t_n)\mathit{Atom}}$	Θ is a type substitution
(Head)	$\dfrac{U \vdash t_1{:}\tau_1\Theta \ \ldots\ U \vdash t_n{:}\tau_n\Theta}{U \vdash p_{\tau_1\ldots\tau_n}(t_1,\ldots,t_n)\mathit{Head}}$	Θ is a renaming substitution
(Query)	$\dfrac{U \vdash A_1\ \mathit{Atom}\ \ldots\ U \vdash A_n\ \mathit{Atom}}{U \vdash A_1,\ldots,A_n\ \mathit{Query}}$	
(Clause)	$\dfrac{U \vdash Q\ \mathit{Query} \quad U \vdash A\ \mathit{Head}}{U \vdash A \leftarrow Q\ \mathit{Clause}}$	

Table 1. *The type system.*

An object, say a term t, is *well-typed* if there exist some variable typing U and some type τ such that $U \vdash t : \tau$. Otherwise the term is *ill-typed* (and likewise for atoms, etc.). A program is well-typed if all its clauses are well-typed.

The distinction between rules *Head* and *Atom* expresses the usual *definitional genericity* principle (Lakshman & Reddy, 1991) which states that the type of a defining occurrence of a predicate (i.e. at the left of "$\leftarrow$" in a clause) must be equivalent up-to renaming to the assigned type of the predicate. The rule *Head* used for deriving the type of the head of the clause is thus not allowed to use substitutions other than variable renamings in the declared type of the predicate. For example, the predicate *member* can be typed polymorphically, i.e. $member : \alpha \times list(\alpha) \to pred$, if its definition does not contain special facts like $member(1, [1])$, that would force its type to be $member : int \times list(int) \to pred$, for satisfying the definitional-genericity condition.

The following proposition shows that if an expression other than a clause or a head is well-typed in a variable typing U, it remains well-typed in any instance $U\Theta$.

Proposition 2.6

For any variable typing U, any type judgement R other than a Head or a Clause, and any type subtitution Θ, if $U \vdash R$ then $U\Theta \vdash R\Theta$.

Proof

By induction on the height of the derivation tree for $U \vdash R$. $\square$

2.3 Subject reduction w.r.t. CSLD resolution

Subject reduction is the property that evaluation rules transform a well-typed expression into another well-typed expression. The evaluation rule for constraint logic programming is CSLD-resolution. To recall this evaluation rule, it is convenient to distinguish in a query Q, the constraint part c (where the sequence denotes the conjunction) from the other sequence of atoms $\overline{A}$. We use the notation $Q = c|\overline{A}$ to make this distinction. Given a constraint domain $\mathcal{X}$ which fixes the interpretation of constraints, a query $c'|\overline{B}$ is a *CSLD-resolvent* of a query $c|\overline{A}$ and a (renamed apart) program clause $p(t_1, ..., t_n) \leftarrow d|\overline{A}$, if

$\overline{A} = A_1, \ldots, A_{k-1}, p(t'_1, \ldots, t'_n), A_{k+1}, \ldots, A_m$,

$\overline{B} = A_1, \ldots, A_{k-1}, \overline{A}, A_{k+1}, \ldots, A_m$,

and the constraint $c' = (c \wedge d \wedge t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n)$ is $\mathcal{X}$-satisfiable.

Theorem 2.1 (*Subject Reduction for CSLD resolution*)

Let P be a well-typed CLP($\mathcal{X}$) program, and Q be a well-typed query, i.e. $U \vdash Q$ *Query* for some variable typing U. If Q' is a CSLD-resolvent of Q, then there exists a variable typing U' such that $U' \vdash Q'$ *Query*.

Proof

Let us assume without loss of generality that $Q = c|p(s), \overline{A}$, and that Q' is a CSLD-resolvent of Q with the program clause $p(t) \leftarrow d|\overline{B}$.

Thus $Q' = c, d, s = t|\overline{A}, \overline{B}$.

As Q is well-typed, we have $U \vdash c|p(s), \overline{A}$ *Query*. And as the program is well typed, there exists a variable typing U'', renamed apart from U, such that $U'' \vdash p(t) \leftarrow d|\overline{B}\ \ Clause$.

Let $p : \tau \rightarrow pred$ be the type declaration of predicate p. Since $U \vdash p(s)\ Atom$, we have $U \vdash s : \tau\Theta$ for some substitution Θ.

Now let $U' = U \cup U''\Theta$. By proposition 2.6, we have $U''\Theta \vdash d|\overline{B},\ \ Query$, thus $U' \vdash c, d|\overline{A}, \overline{B},\ \ Query$, What remains to be shown is $U' \vdash s = t\ Atom$.

Since $U'' \vdash p(t)\ Head$, we have $U'' \vdash t : \tau$. Hence by proposition 2.6, $U''\Theta \vdash t : \tau\Theta$. Therefore we have $U \vdash s : \tau\Theta$ and $U''\Theta \vdash t : \tau\Theta$, from which we conclude $U' \vdash s = t\ Atom$. $\square$

It is worth noting that the previous result would not hold without the definitional genericity condition (expressed in rule *Head*). For example with two constants $a : \tau_a$ and $b : \tau_b$, and one predicate $p : \alpha \rightarrow pred$ defined by the non definitional generic clause $p(a)$, we have that the query $p(b)$ is well typed, but $b = a$ is a resolvent that is ill-typed if τ_a and τ_b have no upper bound.

2.4 Subject reduction w.r.t. substitutions

The CSLD reductions, noted $\longrightarrow_{CSLD}$, are in fact an abstraction of the operational reductions that may perform also substitution steps, noted $\longrightarrow_\sigma$, instead of keeping equality constraints. As in the CLP scheme constraints are handled modulo logical equivalence (Jaffar & Lassez, 1987), it is clear that the diagram of both reductions commutes :

$$
\begin{array}{lllllll}
Q_1 & \longrightarrow_{CSLD} & Q_2 & \longrightarrow_{CSLD} & \cdots & \longrightarrow_{CSLD} & Q_n \\
\downarrow_\sigma & & \downarrow_\sigma & & & & \downarrow_\sigma \\
 & \longrightarrow_{CSLD} & & & \cdots & & \\
 & & \downarrow_\sigma & & & & \downarrow_\sigma \\
 & & & \longrightarrow_{CSLD} & & & \\
 & & & & \cdots & & \cdots \\
 & & & & & & \downarrow_\sigma \\
 & & & & & & Q
\end{array}
$$

However the previous subject reduction result expresses the consistency of types w.r.t. horizontal reduction steps only, that is w.r.t. the abstract execution model which accumulates constraints, but may not hold for more concrete operations of constraint solving and substitutions. For example, with the subtype relations $int \leq term$, $pred \leq term$, the type declarations $=: \alpha \times \alpha \rightarrow pred$, $p : int \rightarrow pred$, and the program $p(X)$, the query $Y = true, p(Y)$ is well typed with $Y : int$, and succeeds with $Y = true$, although the query obtained by substitution, $p(true)$, is ill-typed. In order to establish subject reduction for substitution steps, and be consistent with the semantical equivalence of programs, one needs to consider a typed execution model with type constraints on variables checked at runtime. In the example, the type constraint $Y : int$ with the constraint $Y = true$ is unsatisfiable, the query will be thus rejected at compile-time by checking the satisfiability of its typed constraints.

Definition 2.1
Given a constraint system over some domain $\mathcal{X}$, a *typed constraint system* over $\mathcal{X} \cup 2^{\mathcal{X}}$ is defined by adding type constraints, i.e. expressions of the form $t : \tau$ where t is a term and τ a type. Basic types are interpreted by distinguished subsets of $\mathcal{X}$ and type constructors by mappings between subsets of $\mathcal{X}$ satisfying the subtyping relation $\leq$ and the type declarations for function and predicate symbols. A type constraint $t : \tau$ is satisfiable if there exists a valuation ρ of the variables in t and the free parameters in τ such that $t\rho \in \tau\rho$. A typed constraint system composed of type constraints and constraints over $\mathcal{X}$ is satisfiable if there exists a valuation which satisfies all constraints of the system.

Lemma 2.1
In a typed constraint system, $X : \tau \wedge X = t$ entails $t : \tau$.

Proof
For any valuation ρ, if $X\rho \in \tau\rho$ and $X\rho = t\rho$ then $t\rho \in \tau\rho$. □

Definition 2.2
The TCLP clause (resp. query) associated to a well-typed program (resp. query) in a typed environment U is the clause (resp. query) augmented with the type constraints in U.

Theorem 2.2 (Subject Reduction for substitutions)
Let P be a TCLP program associated to well-typed CLP($\mathcal{X}$) program, and Q be a TCLP query, we have $U \vdash Q\ Query$ for some variable typing U. If Q' is a CSLD-resolvent of Q, then the variable typing U' associated to the type constraints in Q' gives $U' \vdash Q'\ Query$. Furthermore if Q' contains an equality constraint $X = t$ then $U' \vdash Q'[t/X]\ Query$.

Proof
Subject reduction for CSLD resolution follows from theorem 2.1 as TCLP programs are just a special case of well-typed CLP programs. Furthermore one easily checks that the type constraints in Q', that come from the type constraints in Q and from the resolving TCLP clause, give exactly the type environment U' constructed in the proof of the previous theorem, thus $U' \vdash Q' Query$.

Now let $X = t$ be a constraint in a resolvent Q'. Let $X : \tau \in U'$. We have $X : \tau$ in the constraint part of Q' which together with $X = t$ entails $t : \tau$ by lemma 2.1. Therefore it is immediate from the typing rules that by replacing X by t in the derivation of $U' \vdash Q'\ Query$, and by completing the derivation with the derivation of $t : \tau$ instead of $X : \tau$, we get a derivation of $U' \vdash Q'[t/X]\ Query$. □

The effect of type constraints in TCLP programs is to prevent the derivation of ill-typed queries by substitution steps. In addition, queries such as $X : int, X = true, p(X)$ can be rejected at compile-time because of the unsatisfiability of their constraints. Similarly TCLP program clauses having unsatisfiable typed constraints can be rejected at compile-time.

Note that in (Smaus *et al.*, 2000) another result of subject reduction for substitutions is shown without the addition of type constraints but in a very restricted context of moded logic programs.

3 Type checking

The system described by the rules of Table 1 is non-deterministic, since the rule (*Sub*) can be used anywhere in a typing derivation. One can obtain a deterministic type checker, directed by the syntax of the typed program, simply by replacing the rule (*Sub*) by variants of the rules (*Func*), (*Atom*) and (*Head*) with the subtype relation in their premises. This leads to the following type system in table 2.

(Var)	$\{x:\tau,\ldots\} \vdash x:\tau$	
(Func')	$\dfrac{U\vdash t_1{:}\sigma_1 \;\; \sigma_1\leq\tau_1\Theta \;\ldots\; U\vdash t_n{:}\sigma_n \;\; \sigma_n\leq\tau_n\Theta}{U\vdash f_{\tau_1\ldots\tau_n\rightarrow\tau}(t_1,\ldots,t_n){:}\tau\Theta}$	Θ is a type substitution
(Atom')	$\dfrac{U\vdash t_1{:}\sigma_1 \;\; \sigma_1\leq\tau_1\Theta \;\ldots\; U\vdash t_n{:}\sigma_n \;\; \sigma_n\leq\tau_n\Theta}{U\vdash p_{\tau_1\ldots\tau_n}(t_1,\ldots,t_n)\,Atom}$	Θ is a type substitution
(Head')	$\dfrac{U\vdash t_1{:}\sigma_1 \;\; \sigma_1\leq\tau_1\Theta \;\ldots\; U\vdash t_n{:}\sigma_n \;\; \sigma_n\leq\tau_n\Theta}{U\vdash p_{\tau_1\ldots\tau_n}(t_1,\ldots,t_n)\,Head}$	Θ is a renaming substitution
(Query)	$\dfrac{U\vdash A_1 \;\; Atom \;\ldots\; U\vdash A_n \;\; Atom}{U\vdash A_1,\ldots,A_n \;\; Query}$	
(Clause)	$\dfrac{U\vdash Q \;\; Query \quad U\vdash A \;\; Head}{U\vdash A\leftarrow Q \;\; Clause}$	

Table 2. *The type system in second form.*

Proposition 3.1

A program is well typed in the original system if and only if it is well typed in the new one.

Proof

Clearly, if a program is typable in the new system, it is typable in the original one: one has just to replace every occurrence of the (*Func'*) and (*Atom'*) rules respectively with the following derivations:

$$(Func)\frac{(Sub)\dfrac{U\vdash t_1:\tau_1 \quad \tau_1\le\tau_1'\Theta}{U\vdash t_1:\tau_1'\Theta}\ \cdots\ (Sub)\dfrac{U\vdash t_n:\tau_n \quad \tau_n\le\tau_n'\Theta}{U\vdash t_n:\tau_n'\Theta}}{U\vdash f_{\tau_1'\cdot\cdot\times\tau_n'\to\tau'}(t_1,\cdots,t_n):\tau'\Theta}$$

$$(Atom)\frac{(Sub)\dfrac{U\vdash t_1:\tau_1 \quad \tau_1\le\tau_1'\Theta}{U\vdash t_1:\tau_1'\Theta}\ \cdots\ (Sub)\dfrac{U\vdash t_n:\tau_n \quad \tau_n\le\tau_n'\Theta}{U\vdash t_n:\tau_n'\Theta}}{U\vdash p_{\tau_1'\times\cdot\cdot\times\tau_n'}(t_1,\cdots,t_n)\ Atom}$$

$$(Head)\frac{(Sub)\dfrac{U\vdash t_1:\tau_1 \quad \tau_1\le\tau_1'\Theta}{U\vdash t_1:\tau_1'\Theta}\ \cdots\ (Sub)\dfrac{U\vdash t_n:\tau_n \quad \tau_n\le\tau_n'\Theta}{U\vdash t_n:\tau_n'\Theta}}{U\vdash p_{\tau_1'\times\cdot\cdot\times\tau_n'}(t_1,\cdots,t_n)\ Atom}$$

Conversely, if a program is typable in the original system, it is typable in the second one, noted here $\vdash_2$. The proof is by induction on the typing derivation in the original system. The rules *(Var)*, *(Query)* and *(Clause)* remain the same. The rule *(Atom)* and *(Head)* are similar to rule (*Func*). We thus show the property for any term t : if $U\vdash t:\tau$ in the first system, then $U\vdash_2 t:\tau'$ in the second system with $\tau'\le\tau$.

Let us consider the three possible cases, either the proof terminates by the application of the (*Var*) rule, by the application of the (*Func*) rule or by application of the (*Sub*) rule.

The first case is trivial as the rule (*Var*) is the same in both systems.

In the second case, according to the (*Func*) rule, $U\vdash t_1:\tau_1\Theta\ \cdots U\vdash t_n:\tau_n\Theta$. Then, by the induction hypothesis, the terms $t_1\ \cdots\ t_n$ are also type checked to $U\vdash_2 t_1:\tau_1'\ \cdots U\vdash_2 t_n:\tau_n'$ by the second system, with $\tau_i'\le\tau_i\Theta, i=1..n$. By applying the (*Func'*) rule with $\tau_i'=\sigma_i, i=1..n$, we get $U\vdash_2 f(t_1,\cdots,t_n):\tau\Theta$.

In the third case, according to the *(Sub)* rule, $U\vdash t:\tau$ and $\tau\le\tau'$ allows us to deduce $U\vdash t:\tau'$. By induction hypothesis, t is type checked to $U\vdash_2 t:\sigma$ in the second system, where $\sigma\le\tau$. Since $\tau\le\tau'$, we have $\sigma\le\tau'$. So t is type checked to $U\vdash_2 t:\sigma$, where $\sigma\le\tau'$. □

The construction of the substitution Θ needed in rules *(Func'), (Atom')* and *(Head')* for type checking, can be done by solving the system of subtype inequalities collected along the derivation of a type judgement. The parameters in the type environment (i.e. the parameters in the types of variables) are however not under the scope of these substitutions, as they act only on the parameters of the (renamed apart) type declarations for function and predicate symbols. We are thus looking for type substitutions with a restricted domain. For the sake of simplicity however, instead of dealing formally with the domain of type substitutions, we shall simply assume that the parameters in the type of variables are replaced by new constants for checking the satisfiability of subtype inequalities, and avoid unsound instantiations.

Now let Σ be the collection of subtype inequalities $\le$ imposed on types by rules *(Func') (Atom')* and *(Head')* in a derivation. Let us define the size of a system of inequalities as the number of symbols. The size of the system Σ of inequalities associated to a typed program is $O(nvd)$ where v is the size of the type declarations for variables in the program, n is the size of the program, and d is the size of the type declarations for function and predicate symbols.

As the type system is deterministic we have:

Proposition 3.2
A well-formed program is typable if and only the system of inequalities collected along its derivation is satisfiable.

It is worth noting that the system of inequalities Σ collected in this way for type checking have in fact a very particular form.

Definition 3.1
A system Σ of inequalities is *left-linear* if any type variable has at most one occurrence at the left of $\leq$ in the system. Σ is *acyclic* if there exists a ranking function on type variables $r : \mathcal{U} \rightarrow \mathcal{N}$ such that if $\sigma \leq \tau \in \Sigma$, $\alpha \in V(\sigma)$ and $\beta \in V(\tau)$ then $r(\alpha) < r(\beta)$.

Proposition 3.3
The system of inequalities generated by the type checking algorithm is acyclic and left-linear.

Proof
As the type variables in the types of CLP variables have been renamed into constants, the only type variables occurring in Σ are introduced by rules (Func') (Atom') and (Head'), and come from (renamed apart) type declarations of function and predicate symbols. We can thus associate to each type variable α a rank $h(\alpha)$ defined as the height of its introduction node in the derivation tree (i.e. the maximal distance from the node to its leaves). Now a rule (Func'), (Atom') or (Head') at height h posts inequalities of the form $\sigma \leq \tau$, where the rank of the variables in τ is h, and the rank of the variables in σ is $h-1$. The system is thus acyclic.

The type variables at the left of $\leq$ are those parameters that come from the result type of a function declaration, e.g. α in $nil : list(\alpha)$. As the result type is a flat type, the variables in a result type are distinct and renamed apart, hence the variables occurring in a type at the left of $\leq$ have a unique occurrence in the system. The system is thus trivially left-linear. $\square$

Note that if we allowed contravariant type constructors, the previous proposition would not hold.

A linear time algorithm for solving acyclic left-linear systems is given in the next section.

4 Subtype inequalities

The *satisfiability of subtype inequalities (SSI) problem* is the problem of determining whether a system of subtype relations[1] $\bigwedge_{i=1}^{n} \tau_i \leq \tau'_i$ over types $\tau_1, \tau'_1, ..., \tau_n, \tau'_n$ has a

[1] The SSI problem should not be confused with the semi-unification problem which is defined with the instantiation pre-ordering, intead of the subtype ordering: $\exists\Theta \bigwedge_{i=1}^{n} \exists\Theta_i\ \tau_i\Theta\Theta_i = \tau'_i\Theta$. The undecidability of semi-unification is shown in (Kfoury *et al.*, 1989).

solution, i.e., whether there exists a substitution Θ such that $\tau_i\Theta\leq\tau_i'\Theta$ holds for all $1 \leq i \leq n$.

Definition 4.1

A *solution* to an inequality $\tau\leq\tau'$ is a substitution Θ such that $\tau\Theta\leq\tau'\Theta$. A *maximal solution* is a solution Θ such that for any solution Θ' there exists a substitution ρ such that $\forall\alpha \in V\ \alpha\Theta'\leq\alpha\Theta\rho$.

The SSI problem has been deeply studied in the functional programming community. Due to the lack of results for the general case, special instances of the SSI problem have been identified along several axes:

- the form of the types: basic types, constructor types, covariant (our case in this paper) or contravariant;
- the structure of the types: (disjoint union of) lattices (Tiuryn, 1992), quasi-lattices (Smolka, 1989), n-crown (Tiuryn, 1992), posets with suprema (our case), partial orders (Frey, 1997);
- the form of the type constraints.

In this section we show that the type constraints generated by the type checking algorithms can be solved in linear time in our quite general structure of types, and that the type constraints generated by the type inference algorithms can be solved in cubic time, under the additional assumption that the types form a lattice.

4.1 The acyclic left-linear case

We show that the satisfiability of acyclic left-linear subtype inequalities can be decided in linear time, and admit maximal solutions in our general type structure $(\mathcal{T}, \leq)$ of posets with suprema.

In this section, we present an algorithm which proceeds by simplification of the subtype inequalities and introduces equations between a parameter and a type. We say that a system Σ is in *solved form* if it contains only equations of the form

$$\{\alpha_1 = \tau_1, \ldots, \alpha_n = \tau_n\}$$

where the α_i's are all different and have no other occurrence in Σ. The substitution $\Theta_\Sigma = \{\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n\}$ associated to a system in solved form Σ is trivially a maximal solution. We show that the following simplification rules compute solved forms for satisfiable acyclic left-linear systems:

(Decomp) $\Sigma,\ K(\tau_1,...,\tau_m) \leq K'(\tau'_1,...,\tau'_n),\ \longrightarrow \Sigma,\ \bigwedge_{i=1}^{n} \tau_{\iota(i)} \leq \tau'_i$
if $K \leq K'$ and $\iota = \iota_{K,K'}$.

(Triv) $\Sigma,\ \alpha \leq \alpha,\ \longrightarrow \Sigma$

(VarLeft) $\Sigma,\ \alpha \leq \tau,\ \longrightarrow \alpha = \tau,\ \Sigma[\tau/\alpha]$
if $\tau \neq \alpha$, $\alpha \notin V(\tau)$.

(VarRight) $\Sigma,\ \tau \leq \alpha,\ \longrightarrow \alpha = \mathrm{Max}(\tau),\ \Sigma[\mathrm{Max}(\tau)/\alpha]$
if $\tau \notin V$, $\alpha \notin V(l)$ for any $l \leq r \in \Sigma$, and $\alpha \notin V(\mathrm{Max}(\tau))$.

Lemma 4.1
The rules terminate in $O(n)$ steps, where n is the sum of the sizes of the terms in the left-hand side of inequalities.

Proof
It suffices to remark that each rule strictly decreases the sum of the size of the terms in the left-hand sides of the inequalities: (Triv) and (VarLeft) by one, (Decomp) by at least one, and (VarRight) by the size of τ. □

One can easily check that each rule preserves the left-linearity as well as the acyclicity of the system, moreover:

Lemma 4.2
Each rule preserves the satisfiability of the system, as well as its maximal solution if one exists.

Proof
Rules (Decomp) and (Triv) preserve all solutions, by definition of the subtyping order. Rule (VarLeft) replaces a parameter α by its upper bound τ. As the system is left-linear this computes the maximal solution for α, and thus preserves the maximal solution of the system if one exists. Rule (VarRight) replaces a parameter α having no occurrence in the left-hand side of an inequality, hence having no upper bound, by the maximum type of its lower bound τ; this computes the maximal solution for α, and thus preserves also the maximal solution of the system if one exists. □

Theorem 4.1
Let Σ be an acyclic left-linear system. Let Σ' be a normal form of Σ. Then Σ is satisfiable iff Σ' is in solved form, in which case $\Theta_{\Sigma'}$ is a maximal solution of Σ.

Proof
Consider a normal form Σ' for Σ. If Σ' contains a non variable pair $\tau \leq \tau'$, as this inequality is irreducible by (Decomp) Σ' has no solution, hence Σ is unsatisfiable by lemma 4.2. Similarly Σ' has no solution if it contains an inequality $\alpha \leq \tau$ with $\alpha \in V(\tau)$ and $\tau \neq \alpha$ (prop. 2.5) or an inequality $\tau \leq \alpha$ with $\alpha \in V(\mathrm{Max}(\tau))$ and $\tau \neq \alpha$ (Prop. 2.5). In the other cases, by irreducibility and by acyclicity, Σ' contains no inequality, hence Σ' contains only equalities that are in solved form, and the substitution associated to Σ' is a maximal solution for Σ. □

4.2 The general case

In absence of subtype relations between type constructors of different arities, checking the consistency of general subtype inequalities in finite types has been shown by Frey (Frey, 1997) Pspace-complete in an arbitrary poset, with a generalization of Fuh & Mishra's algorithm (Fuh & Mishra, 1988).

It is an open problem whether the technique used by Frey for proving consistency in arbitrary posets can be generalized to our case with subtype relations between type constructors of different arities.

If we assume however that the subtyping relation is a lattice, it has been shown by Pottier (Pottier, 2000a) that the satisfiability of subtype inequalities can be checked in cubic time in the structure of infinite regular trees, i.e. recursive types (Amadio & Cardelli, 1993). Note that recursive types admit solutions to equations of the form $\alpha = list(\alpha)$, namely the type $list(list(...))$. Below we present Pottier's algorithm by a set of simplification rules, and show that in acyclic systems the solving of (covariant) subtype constraints on infinite types is equivalent to the solving on finite types.

We assume that the structure of type constructors $(\mathcal{K}, \leq)$ is a lattice with $\bot$ and $\top$ types. We maintain our previous assumption on decreasing arities, except on $\bot$ which is below all (n-ary) type constructors. We also assume that if $K''/n = glb(K, K')$ then $\text{range}(\iota_{K'',K}) \cup \text{range}(\iota_{K'',K'}) = [1, n]$, that is greatest lower bounds do not introduce new parameters. Similarly, if $K''/n = lub(K, K')$ then $\text{range}(\iota_{K,K''}) \cup \text{range}(\iota_{K',K''}) = [1, n]$, Note that there is no loss of generality with this assumption as the lattice of type constructors can always be completed by introducing glb and lub constructors with the right number of parameters.

We consider systems of subtype inequalities between parameters of flat types, that is of the form $\alpha \leq \beta$, $K(\alpha_1, ...\alpha_n) \leq \alpha$ or $\alpha \leq K(\alpha_1, ...\alpha_n)$. Non flat types can be represented in this form by introducing new parameters and inequalities between these parameters and the type they represent.

For the sake of presentation, we assume that the initial system to be solved, Σ_0 over variables V_0, is first completed by introducing new variables γ_S and δ_S for each non empty subset S of V_0, and by adding the inequalities $\gamma_S \leq \alpha$ and $\alpha \leq \delta_S$ for all variables $\alpha \in S$. We also assume that the system is completed by adding the inequality $\alpha \leq \alpha$ for each variable α. Given a system Σ and a set of variables S we define the variable $\gamma(S, \Sigma) = \gamma_{\{\alpha \in V_0 \mid \exists \beta \in S\ \beta \leq \alpha \in \Sigma\}}$ and similarly $\delta(S, \Sigma) = \delta_{\{\alpha \in V_0 \mid \exists \beta \in S\ \alpha \leq \beta \in \Sigma\}}$. The simplification rules are the following:

(Trans) $\Sigma,\ \alpha \leq \beta,\ \beta \leq \gamma \longrightarrow \Sigma,\ \alpha \leq \beta,\ \beta \leq \gamma,\ \alpha \leq \gamma$
if $\alpha \leq \gamma \notin \Sigma$ and $\alpha \neq \gamma$.

(Clash) $\Sigma,\ K(\alpha_1, ..., \alpha_m) \leq \alpha,\ \alpha \leq \beta,\ \alpha \leq K'(\alpha'_1, ..., \alpha'_n) \longrightarrow$ false
if $K \not\leq K'$.

(Dec) $\Sigma,\ K(\alpha_1, ..., \alpha_m) \leq \alpha,\ \alpha \leq \beta,\ \alpha \leq K'(\alpha'_1, ..., \alpha'_n) \longrightarrow$
$\Sigma,\ K(\alpha_1, ..., \alpha_m) \leq \alpha,\ \alpha \leq \beta,\ \alpha \leq K'(\alpha'_1, ..., \alpha'_n),\ \{\alpha_{\iota(j)} \leq \alpha'_j\}_{j\in[1,n]}$

if $K \leq K'$, $\iota = \iota_{K,K'}$ and $\{\alpha_{\iota(j)} \leq \alpha'_j\}_{j\in[1,n]} \not\subset \Sigma \cup \{\alpha \leq \beta\}$.

(Glb) $\Sigma,\ \alpha \leq K(\alpha_1, ..., \alpha_m),\ \alpha \leq \beta,\ \beta \leq K'(\alpha'_1, ..., \alpha'_n) \longrightarrow$
$\Sigma,\ \alpha \leq K''(\alpha''_1, ..., \alpha''_l),\ \alpha \leq \beta,\ \beta \leq K'(\alpha'_1, ..., \alpha'_n), \Sigma'$

if $K'' \neq K$ or $\Sigma' \not\subset \Sigma \cup \{\alpha \leq \beta\}$
where $K'' = \mathrm{glb}(K, K')$, $\iota = \iota_{K'',K}$, $\iota' = \iota_{K'',K'}$,
$\alpha''_k = \gamma(\{\alpha_{\iota^{-1}(k)}, \alpha'_{\iota'^{-1}(k)}\}, \Sigma \cup \{\alpha \leq \beta\})$, for all $1 \leq k \leq l$,
$\Sigma' = \{\alpha''_{\iota(i)} \leq \alpha_i\}_{i\in[1,m]} \cup \{\alpha''_{\iota'(j)} \leq \alpha'_j\}_{j\in[1,n]}$

(Lub) $\Sigma,\ K(\alpha_1, ..., \alpha_m) \leq \alpha,\ \alpha \leq \beta,\ K'(\alpha'_1, ..., \alpha'_n) \leq \beta \longrightarrow$
$\Sigma,\ K(\alpha_1, ..., \alpha_m) \leq \alpha,\ \alpha \leq \beta,\ K''(\alpha''_1, ..., \alpha''_l) \leq \beta, \Sigma'$

if $K'' \neq K'$ or $\Sigma' \not\subset \Sigma \cup \{\alpha \leq \beta\}$
where $K'' = \mathrm{lub}(K, K')$, $\iota = \iota_{K,K''}$, $\iota' = \iota_{K',K''}$,
$\alpha''_k = \delta(\{\alpha_{\iota(k)}, \alpha'_{\iota'(k)}\}, \Sigma \cup \{\alpha \leq \beta\})$, for all $1 \leq k \leq l$,
$\Sigma' = \{\alpha_{\iota(k)} \leq \alpha''_k,\ \alpha'_{\iota'(k)} \leq \alpha''_k\}_{k\in[1,l]}$

Rule (Trans) computes the transitive closure of inequalities between parameters and is mainly responsible for the cubic time complexity. Rule (Clash) checks the consistency of the lower and upper bounds of parameters. Rule (Dec) decomposes flat types. These three simple rules are in fact sufficient for checking the satisfiability of the system (Pottier, 2000a).

Rules (Glb) and (Lub) make explicit the greatest and the least solution of the system by computing the greatest lower bound of upper bounds of parameters and the least upper bound of their lower bounds. We remark that if the algorithm is applied to an initial system Σ containing a unique inequality of the form $\tau \leq \alpha$ and $\alpha \leq \tau'$ for each parameter α, the algorithm maintains unique upper and lower bounds for each parameter. We note $\mathrm{lb}(\alpha)$ (resp. $\mathrm{ub}(\alpha)$) the lower (resp. upper) bound of α in the system in irreducible form.

Proposition 4.1
The rules terminate.

Proof
Termination with rule (Clash) is trivial. For the other rules, let us consider, as complexity measure of the system, the couple of integers (t, e) ordered by lexicographic ordering, where e, the "entropy" of the system, is the number $v^2 - n$, where

v is the number of parameters in the system, n is the number of inequalities between parameters, and where t, the "temperature" of the system, is the sum of the height of constructors at the right of $\leq$, and of the depth of constructors at the left of $\leq$. The height (resp. depth) of a constructor is the length of the longest path of the constructor to $\perp$ (resp. $\top$) in $(\mathcal{K}, \leq)$. We show that no rule increases the temperature of the system, and each rule either decreases t or e.

Rule (Trans) does not change t and decreases e by 1, Rule (Dec) does not change t and decreases e by at least 1, Rules (Glb) either decreases t if $K'' \neq K'$ or decreases e otherwise, and similarly for rule (Lub). Hence the algorithm terminates. □

Theorem 4.2
(Pottier, 2000a) A system of inequalities is satisfiable over infinite regular trees if and only if the simplification rules do not generate false, in which case the identification of all parameters to their upper bound $\mathrm{ub}(\alpha)$ (resp. their lower bound $\mathrm{lb}(\alpha)$) provides a maximum (resp. minimum) solution.

Furthermore, one can show that in our setting of acyclic systems and covariant constructor types, the solving of subtype constraints on infinite types is equivalent to the solving on finite types.

Theorem 4.3
An acyclic system of inequalities is satisfiable over finite types if and only if the simplification rules do not generate false, in which case the identification of all parameters to their upper bound (resp. lower bound) provides a maximum (resp. minimum) finite solution.

Proof
It is sufficient to remark that the simplification rules preserve the acyclicity of the system, and that in an acyclic system, the identification of the parameters to their bounds creates finite solutions. □

Corollary 4.1
In a lattice structure without $\perp$, an acyclic system of inequalities is satisfiable over finite types if and only if the simplification rules do not generate false and $\mathrm{ub}(\alpha) \neq \perp$ for all parameters α.

5 Type inference

As usual with a prescriptive type system, type reconstruction algorithms can be used to omit type declarations in programs, and still check the typability of the program by the possibility or not to infer the omitted types (Lakshman & Reddy, 1991) . Below we describe algorithms for inferring the type of variables and predicates, assuming type declarations for function symbols.

5.1 Type inference for variables

Types for variables in CLP clauses and queries can be inferred by introducing unknowns for their type in the variable typing, and by collecting the subtype inequalities along the derivation of the type judgement just like in the type checking algorithm.

It is easy to check that the system of subtype inequalities thus collected is still acyclic, as the unknown types for CLP variables appear only in left positions. The system is however not left-linear if a CLP variable has more than one occurrence in a clause or a query.

The second algorithm of the previous section can thus be used to infer the type of variables in CLP clauses and queries.

5.2 Type inference for predicates

Types for predicates can be inferred as well under the assumption that predicates are used monomorphically inside their (mutually recursive) definition (Lakshman & Reddy, 1991). This means that inside a group of mutually recursive clauses, each occurrence (even in the body of a clause) of a predicate defined in these clauses must be typed with rule *Head* instead of rule *Atom*. The reason for this restriction, similar to the one done for inferring the type of mutually recursive functions in ML, is to avoid having to solve a semi-unification problem: i.e. given a system of types $\tau_i,\ \tau_i'$ for $i,\ 1 \leq i \leq n$, finding a substitution Θ such that for all i there exists a substitution Θ_i s.t. $\tau_i\Theta\Theta_i = \tau_i'\Theta$, that is proved undecidable in (Kfoury *et al.*, 1989).

Note that the SSI obtained by collecting the subtype inequalities in the derivation of typing judgements is still acyclic, as the unknown types for predicates appear only in the right-hand sides of the inequalities. The second algorithm of the previous section can thus be used also to infer the type of predicates in CLP programs under the assumption that the structure of types is a lattice without $\perp$.

One consequence of the acyclicity of the system however, is that the maximum type of a predicate is always $\top$. Indeed in our type system a predicate can always be typed as maximally permissive. In the more general structure of posets with suprema, unless the unknown types for predicates are compared with types belonging to different $\leq$-connected components (in which case the predicate is not typable), the substitution of an unknown type by the root of its $\leq$-connected component is always a solution. But in all cases, this is obviously not a very informative type to infer.

Our strategy is to infer two types for predicates: the minimum type of the predicate and a heuristic type. The type inference algorithm proceeds as follows:

- Firstly, the minimum type of the predicate is obtained by computing the minimum solution of the SSI associated to the typing of the complete definition of the predicate. The minimum type of the ith argument of the predicate is the type $\underline{\tau}_i = \mathrm{lb}(\alpha_i)$ where α_i is the unknown type associated to the ith argument of the predicate in the SSI. This minimum type is a lower bound of all possible typings of the predicate.
- Secondly, the heuristic type is computed. This type can be parametric. It is computed in two steps:

- First a heuristic upper type is computed for the predicate. The heuristic upper type $\overline{\tau}_i$ of the ith argument of the predicate is obtained by collecting the upper types $\{\mathrm{ub}(\tau_{X_1}), ..., \mathrm{ub}(\tau_{X_n})\}$ of all the variables $\{X_1, ..., X_n\}$ which occur in the ith position of the predicate in its definining clauses. Let $\tau = \mathrm{glb}\{\mathrm{ub}(\tau_{X_1}), ..., \mathrm{ub}(\tau_{X_n})$ be the greatest lower bound of the types of the variable arguments. We set
$$\begin{aligned}\overline{\tau}_i \quad &= \top \text{ if } \tau = \top \text{ and } \underline{\tau}_i = \bot,\\ &= \underline{\tau}_i \text{ if } \tau = \top \text{ and } \underline{\tau}_i \neq \bot,\\ &= \top \text{ if } \underline{\tau}_i \not\leq \tau,\\ &= \top \text{ if the identification } \tau_i = \tau \text{ creates a cycle},\\ &= \tau \text{ otherwise.}\end{aligned}$$
- Then the heuristic type is computed by inferring a possibly parametric type from the SSI associated to the heuristic upper type. The candidates for parametric types are the parameters bounded by $\bot$ and $\top$ in the SSI associated to the heuristic upper type. Each candidate is checked iteratively by replacing it with a new constant and by identifying all parameters which have the new constant in one of their bounds.

Although tedious, one can easily check that the conditions imposed in the definition of the heuristic type create sound typings. The heuristic types thus provide correct type declarations for type checking the program.

6 Implementation of the type system

6.1 The Wallace library for solving subtype inequalities

Our current implementation uses the Wallace library by F.Pottier (Pottier, 2000b) for solving the subtype inequalities for type inference and type checking. In both cases, the set of type constructors $(\mathcal{K}, \leq)$ has thus to be a lattice as described in section 4. Note that the type system did not require that condition: $\leq$ could be any arity decreasing order relation on $\mathcal{K}$.

As required in the type inference algorithm, the $\top$ element is distinguished from the type *term* which stands for all Prolog terms. The type $\bot$ is not considered as a valid typing as it is an empty type.

Note that the Wallace library authorizes constrained type schemes, like for example $+ : \forall\alpha \leq \mathit{float}\ \alpha \times \alpha \to \alpha$, which expresses the resulting type of $+$ as a function of the type of its arguments. For the sake of simplicity, we do not consider constrained type schemes in this paper.

6.2 The type checker

The type checker first reads the Prolog files and deduces the files containing type information to load. There is one file for each Prolog file source plus one file for each module used (as `:- use_module(somemodule)` in Sicstus Prolog). The system then loads the type files and builds the structure of type constructors.

The type checker does not impose to give the type of CLP variables in clauses and queries. Instead the type of variables is inferred as described in section 5. The environment U is built with type unknowns for variables. The subtype inequality system is collected by applying the rules of the type system and at each step, Wallace is used to solve the type constraints.

One difficulty appears for checking the definitional genericity condition. A type error must be raised when the definition of a predicate uses, as argument of the head of the clause, a term whose type τ is a subtype of an *instance* of the declared type τ' for this argument, and not just of a *renaming*. But Wallace is not able to make the difference between being a subtype of an instance or of a renaming of a type τ'. The following consideration allows us to work around this difficulty. If τ is a subtype of a renaming of τ', for all instances $\tau'\Theta'$ of τ' there must exist an instance $\tau\Theta$ of τ such that $\tau\Theta \leq \tau'\Theta'$. For checking definitional genericity, we thus replace each parameter α appearing in the declared type of the head predicate by a constructor κ_α, that does not appear in the program, and such that :

- $\kappa_\alpha \leq term$, and $\kappa_\alpha \not\leq \mu$ for all constructor μ, $\mu \neq \kappa_\alpha$, $\mu \neq term$
- $\mu \not\leq \kappa_\alpha$ for all $\mu \neq \kappa_\alpha$.

If the rule *(Atom)* can be applied, using the transformed type, then the rule *(Head)* can be applied as well with the original type.

6.3 Type inference for predicates

As described in section 5, two types are infered for predicates: a minimum type which is a lower bound of all possible typings of the predicate, and a heuristic type which may be parametric.

If type inference is just displayed for user information, we print both types. If it is used for typing automatically the program in a non-interactive manner, then we choose the heuristic bound, since it is the most permissive type.

7 Experimental results

7.1 Detection of programming errors

Here we show a small catalog of the kind of programming errors detected by the type checker.

7.1.1 Inversion of arguments in a predicate or a function

This error can be detected when, for example, a variable occurs in two positions that have incompatible types.

Example 7.1
Consider the following clause where the arguments of the length predicate have been reversed.

```
p(L1,L2,N) :- append(L1,L2,L3),length(N,L3).
```

with the usual declarations :

`append` : $list(\alpha) \times list(\alpha) \times list(\alpha) \rightarrow pred$
`length` : $list(\alpha) \times int \rightarrow pred$

By the rule *(Atom')*, the variable `L3` must be of both types $list(\alpha)$ and int. In the type hierarchy we use, there is no type smaller than $list$ and int. The subtype inequalities in the premise of rule *(Atom')* are thus unsatisfiable and a type error is raised.

Note that this example motivates the discard of type $\perp$: otherwise, no error would be detected on variables, since the empty type $\perp$ could always be inferred for the type of any variable.

7.1.2 Misuse of a predicate or a function

This error is detected when a term of a type τ appears as an argument of a predicate, or of a functor that expects an argument of type τ', but $\tau \not\leq \tau'\Theta$ for any substitution Θ.

Example 7.2
Consider the following clause :

```
p(X,Y) :- Y is (3.5 // X).
```

With type declarations:

`'//'` : $int \times int \rightarrow int$ for integer division,
`is` : $float \times float \rightarrow pred$.

We try to use a *float* (`3.5`) where an *int* is expected. The rule *(Atom')* does not apply.

This kind of error can be detected also inside call to foreign predicates, through the Prolog interface with the C programming language for example.

Example 7.3
Consider the declaration of a predicate `p` defined in C using the Sicstus - C interface :

```
foreign(p, p(+integer)).
```

Such a declaration is interpreted as a type declaration for `p` :

`p` : $int \rightarrow pred$

Then a call in a program such as :

```
:- p(3.14).
```

raises a type error since the argument is a $float$ and the predicate expects an int.

7.1.3 Wrong predicate definition w.r.t. the declared type

This error is detected by two ways, corresponding to the two preceding kinds of errors. In the two following examples, the predicate `p` has been declared with type $int \rightarrow pred$.

Example 7.4
Let `p` be defined by `p([]).`
Here the term `[]` is used as an argument of `p`, which requires that `p` accepts arguments of type *atomic_list*. But $atomic\_list \not\leq int$ and the rule *(Atom')* does not apply.

Example 7.5
Let `p` be defined by :

```
p(X) :- length(X,2).
```
with `length` : $list(\alpha) \times int \rightarrow pred$

In this case, we will infer a type for `X` that must be smaller than $list(\alpha)$ (using rule *(Atom')*, because `X` is used by `length`) and smaller than *int* (using the rule *(Head')*). As before, these types have no common subtypes, and an error is raised.

7.1.4 Violation of the definitional genericity condition

Example 7.6
Let :

```
p([1]).
```
with `p` : $list(\alpha) \rightarrow pred$

Although the argument of `p` is a list, but its type is $list(int)$, an *instance* and not a *renaming* of $list(\alpha)$ (because $int \not\leq \kappa_\alpha$).

This error can also be detected when a variable is in the head of the clause :

Example 7.7
Let :

```
p([X]) :- X < 1.
```
with :

`p` : $list(\alpha) \rightarrow pred$
`<` : $float \times float \rightarrow float$

The variable `X` must be of type *float* and κ_α. The only common subtype is $\perp$ and an error is raised.

7.2 Type checked programs

To test our system, we first tried it on 20 libraries of Sicstus Prolog, that is around 600 predicates. Then we type checked an implementation of CLP(FD) written completely in Prolog, using a lot of meta-predicates, that contains around 170 predicates. These tests where done using type declarations for around 100 built-in ISO Prolog predicates and for some more built-in Sicstus predicates.

Some type errors obtained in the libraries came from the overloading of some function symbols. For example, the function `'-'/2` is used for coding pairs as well as for coding the arithmetic operation over numbers. Another example of overloading comes from options : it happens that some terms are common to two sets of options, of types τ_1, τ_2. In this case, it is enough to create a subtype τ of both τ_1 and τ_2, and tell that the common terms are of type τ.

We also skip the type checking of some particular declarations, such as mode declarations (which are not used by our type system) :

Example 7.8

```
:- mode p(+,-,+) , q(-,?).
```

These declarations can be typed in another type structure for mode declarations, but not in the same type structure as the one for predicates, since the predicate symbols `p`, `q`, `+`, `-` are clearly overloaded in such declarations.

7.3 Type inference for predicates

As said in section (6.3), we infer an interval of types for predicates. Both bounds of the interval may offer interesting information.

Example 7.9

```
append([Head| Tail], List, [Head| Rest]) :-
    append(Tail, List, Rest).
append([], List, List).

Minimum type: list(bottom), list(bottom), list(bottom) -> pred
Heuristic infered type:  list(A), list(A), list(A) -> pred
```

Example 7.10

```
sum_list([], Sum, Sum).
sum_list([Head| Tail], Sum0, Sum) :-
    Sum1 is Head+Sum0, sum_list(Tail, Sum1, Sum).

Minimum type: list(bottom), bottom, bottom -> pred
Heuristic infered type: list(float), float, float -> pred
```

Sometimes, the heuristic infers a too permissive type. This is in particular the case with overloaded arithmetic predicates expressions, that are always typed as `float`, not `int`.

Example 7.11

```
length([],0).
length([_|Tail],R) :- length(Tail, L), R is L+1.

Minimum type:  list(bottom), int -> pred
Heuristic infered type:  list(A), float -> pred
```

The heuristic may also infer a type which is too restrictive.

Example 7.12

```
is_list(X) :- var(X), !, fail.
is_list([]).
is_list([_|Tail]) :- is_list(Tail).

Minimum type:  list(bottom) -> pred
Heuristic infered type:  list(A) -> pred
```

This is a typical example where the maximum type, here

```
is_list: term -> pred
```

is in fact the intended type.

These examples should clearly justify the heuristic approach to type inference for predicates in a prescriptive type system.

Finally, the interesting *flatten* predicate illustrates the remarkable flexibility of the type system.

Example 7.13

```
flatten([],[]) :- !.
flatten([X|L],R) :- !, flatten(X,FX), flatten(L,FL), append(FX,FL,R).
flatten(X,R) :- R=[X].

Minimum type : list(bottom), list(bottom) -> pred
Heuristic infered type : term, list(term) -> pred
```

7.4 Benchmarks

The following table sums up our evaluation results. The first column indicates the type checked Prolog program files. The second column indicates the number of predicates defined in each file first, and then the maximum number of atoms by clause and by complete connected component. The third column indicates the CPU time in seconds for type checking the program with the type declarations for function and predicate symbols. The fourth column indicates the CPU for inferring the types of predicates with the type declarations for function symbols only. The last column indicates the percentage of predicates for which the infered type is exactly the intended type.

The last test file is another implementation of CLP(FD) on top of prolog which uses a lot of metaprogramming predicates.

File	# predicates	max # atoms	Type Checking	Type Inference	% exact types
arrays.pl	13	9/16	2.18 s	11.91 s	23 %
assoc.pl	31	11/24	5.29 s	40.13 s	68 %
atts.pl	14	20/119	7.43 s	77.47 s	64 %
bdb.pl	101	27/27	23.56 s	41.10 s	64 %
charsio.pl	15	7/7	1.27 s	2.21 s	33 %
clpb.pl	59	20/77	24.35 s	1827.32 s	n/a
clpq	396	39/160	355.12 s	4034.37 s	n/a
clpr	439	39/160	304.45 s	3958.41 s	n/a
fastrw.pl	4	5/7	0.44 s	0.76 s	100 %
heaps.pl	21	8/18	3.49 s	43.33 s	71 %
jasper.pl	32	11/11	7.43 s	11.97 s	84 %
lists.pl	39	6/9	2.23 s	16.17 s	97 %
ordsets.pl	35	7/18	7.43 s	199.38 s	97 %
queues.pl	12	11/18	1.37 s	4.12 s	75 %
random.pl	11	18/18	2.43 s	4.12 s	55 %
sockets.pl	24	15/27	6.79 s	15.43 s	68 %
terms.pl	13	18/27	6.96 s	308.69 s	77 %
trees.pl	13	6/15	3.07 s	12.64 s	31 %
ugraphs.pl	87	12/24	48.21 s	274.22 s	67 %
clp-fd.pl	163	20/71	24.35 s	59.65 s	n/a

Table 3. *Benchmarks.*

The same algorithm is used for solving the systems of subtype inequalities for type checking and type inference. The difference between computation times comes from the handling of complete connected components of definitions for type inference, whereas for type checking, clauses are type checked one by one. In particular CLP(R) and CLP(Q) have very large mutually recursive clauses.

In the library for arrays, the low percentage of exact matches between the infered type and the intended type is simply due to the typing of indices by *float* instead of *int*. The errors in the other libraries are also due to the typing of arithmetic expressions by *float*, and sometimes to the use of the equality predicate $=_{\alpha,\alpha}$ which creates a typing by *term* for some arguments instead of a more restrictive typing.

In the library CLP(FD), finite domain variables are typed with type *int*. Similarly in the library CLP(R), variables over the reals are typed with type *float*. One consequence is that the type checker then allows coercions from finite domain variables to real constraint variables. To make these coercions work in practice one modification in the CLP(R) library was necessary.

8 Conclusion

Typing constraint logic programs for checking programming errors statically while retaining the flexibility required for preserving all the metaprogramming facilities of logic programming and the usual coercions of constraint programming, is the challenge that conducted the design of the type system presented in this paper. Our experiments with the libraries of Sicstus Prolog have shown that the type system is simple and flexible enough to accept a large variety of constraint logic programs. The main difficulties are located to conflicts of overloading for some predicates or functions. Such ad hoc polymorphism could be resolved by considering disjunctive formula over types (Demoen *et al.*, 1999). Examples have been given also to show that the type system is useful enough for detecting programming errors such as the inversion of arguments in a predicate, or the unintended use of a predicate.

The price to pay for this flexibility is that our type system may be regarded as too permissive. Some intuitively ill-typed queries may be not rejected by the type system. We have analyzed these defects in terms of the subject reduction properties of the type system. In particular we have shown that the addition of the typing constraints on variables to well-typed programs and queries suffices to state subject reduction w.r.t. both CSLD resolution and substitution steps, and has for effect to reject a larger set of clauses and queries by checking the satisfiability of their constraints with the type constraints at compile-time.

The lattice assumption for the type structure, due to the implementation in Wallace of subtype constraints, may be regarded also as too demanding in some cases. We have already relaxed that assumption by rejecting the bottom element from the structure of types. Nevertheless the decidability of subtype constraints under more general assumptions is an interesting open problem. In particular, whether the method of Frey (Frey, 1997) can be extended to cover subtype relations between type constructors of different arities, as required in our approach, is an open question.

Finally, it is worth noting that the results presented here are not limited to logic programming languages. They should be relevant to various constraint programming languages, where the main difficulty is to type check constraint variables, that express the communication between different constraint domains.

Acknowledgment. We would like to acknowledge fruitful discussions with François Pottier, Didier Rémy, Jan Smaus and Alexandre Frey on this work. We are also grateful to the referees for their peer reviews.

References

Ait-Kaci, H., & Nasr, R. (1986). Login: A logical programming language with built-in inheritance. *Journal of logic programming*, **3**, 187–215.

Ait-Kaci, H., Podelski, A., & S.C. Goldstein, K. (1997). Order-sorted feature theory unification. *Journal of logic programming*, **30**(2), 100–124.

Amadio, R.M., & Cardelli, L. (1993). Subtyping recursive types. *Acm transactions on programming languages and systems*, **15**(4), 575–631.

Beierle, C. (1995). Type inferencing for polymorphic order-sorted logic programs. *Pages 765–779 of: 12th international conference on logic programming ICLP'95.* The MIT Press.

Colmerauer, A. (1984). Equations and inequalitions on finite and infinite trees. *Pages*

85–99 of: ICOT (ed), *Proc. of the international conference on fifth generation computer systems FGCS'84*.

Colmerauer, A. (1985). Prolog in 10 figures. *Communications of the acm*, **28**(12), 1296–1310.

Colmerauer, A. (1996). *Specification of Prolog IV.* Tech. rept. LIM Technical Report.

Coquery, E. 2000 (Oct.). *`Tclp`: a generic type checker for constraint logic programs.* `http://pauillac.inria.fr/~coquery/tclp/`.

Demoen, B., de la Banda, M. Garcia, & Stuckey, P.J. 1999 (january). Type constraint solving for parametric and ad-hoc polymorphism. *Pages 217–228 of: Proceedings of the 22nd Australian computer science conference*.

Dietrich, R., & Hagl, F. (1988). A polymorphic type system with subtypes for Prolog. *Pages 79–93 of:* Ganzinger, H. (ed), *Proceedings of the european symposium on programming ESOP'88*. LNCS. Springer-Verlag.

Fages, F., & Paltrinieri, M. 1997 (December). *A generic type system for CLP($\mathcal{X}$).* Tech. rept. Ecole Normale Supérieure LIENS 97-16.

Frey, A. (1997). Satisfying subtype inequalities in polynomial space. *Proceedings of the 1997 international static analysis symposium SAS'97*. LNCS, no. 1302.

Fuh, Y.C., & Mishra, P. (1988). Type inference with subtypes. *Pages 94–114 of: Proc. ESOP'88*. LNCS, no. 300.

Hanus, M. (1992). *Logic programming with type specifications.* MIT Press. In (Pfenning, 1992). Chap. 3, pages 91–140.

Hill, P., & Lloyd, J. (1994). *The Gödel programming language.* MIT Press.

Hill, P. M., & Topor, R. W. (1992). *A semantics for typed logic programs.* MIT Press. In (Pfenning, 1992). Chap. 1, pages 1–61.

Jaffar, J., & Lassez, J.L. (1987). Constraint logic programming. *Pages 111–119 of: Proceedings of the 1987 symposium on principles of programming languages POPL'87*.

Kfoury, A.J., Tiruyn, J., & Urzyczyn, P. 1989 (October). *The undecidability of the semi-unification problem.* Tech. rept. BUCS-89-010, Boston Univ.

Lakshman, T.K., & Reddy, U.S. (1991). Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. *Pages 202–217 of:* Saraswat, V., & Ueda, K. (eds), *Proceedings of the 1991 international symposium on logic programming*. MIT Press.

Meyer, G. (1996). *Type checking and type inferencing for logic programs with subtypes and parametric polymorphism.* Tech. rept. Informatik Berichte 200, Fern Universitat Hagen.

Mycroft, A., & O'Keefe, R.A. (1984). A polymorphic type system for prolog. *Artificial intelligence*, **23**, 295–307.

Pfenning, F. (ed). (1992). *Types in logic programming.* MIT Press.

Pottier, F. (2000a). Simplifying subtyping constraints: a theory. *To appear in information and computation*.

Pottier, F. 2000b (Feb.). *`Wallace`: an efficient implementation of type inference with subtyping.* `http://pauillac.inria.fr/~fpottier/wallace/`.

Smaus, J.G., Fages, F., & Deransart, P. (2000). Using modes to ensure subject reduction for typed logic programs with subtyping. *Pages 214–226 of: Proceedings of foundations of software technology and theoretical computer science FSTTCS'2000*. LNCS, no. 1974. Springer-Verlag.

Smolka, G. (1988). Logic programming with polymorphically order-sorted types. *Pages 53–70 of: Algebraic and logic programming ALP'88*. LNCS, no. 343. J. Grabowski, P. Lescanne, W. Wechler.

Smolka, G. (1989). *Logic programming over polymorphically order-sorted types.* Ph.D. thesis, Universitat Kaiserslautern.

Somogyi, Z., Henderson, F., & Conway, T. (1996). The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of logic programming*, **29**(1–3), 17–64.

Tiuryn, J. (1992). Subtype inequalities. *Pages 308–315 of: Proc. 7th IEEE symposium on logic in computer science LICS'92.*

Yardeni, E., Frühwirth, T., & Shapiro, E. (1992). *Polymorphically typed logic programs.* MIT Press. In (Pfenning, 1992). Chap. 2, pages 63–90.