A Generic Type System for CLP($\mathcal{X}$)

François Fages, Massimo Paltrinieri
LIENS CNRS, Ecole Normale Supérieure,
45 rue d'Ulm, 75005 Paris, France,
{fages,palmas}@dmi.ens.fr

Abstract

We propose a generic static type system for Constraint Logic Programming including subtyping and parametric polymorphism. The first aim of this prescriptive type system is to detect type errors statically in CLP programs. The system introduces a type discipline on the way CLP programs and libraries can be composed, while still maintaining the ability to type meta-programming predicates, thanks to the flexibility of subtyping. We show that subject reduction holds under very general assumptions. We give a polynomial-time algorithm for type checking, and we indicate that type inference for variables and predicates is possible in the universe of infinite regular types, but is an open problem in the universe of finite types.

1 Introduction

The class of Constraint Logic Programming languages, CLP($\mathcal{X}$), as introduced by Jaffar and Lassez [18], is a class of programming languages based on the logic programming paradigm, parametrized by some mathematical structure $\mathcal{X}$, the domain of discourse. CLP is a simple and powerful model of computation that has proven its ability to solve declaratively a wide variety of applications, ranging from combinatorial optimization problems to complex system modeling [19].

Inherited from the Logic Programming tradition, constraint logic programs generally have no static types or have a poor type structure. However, the structure $\mathcal{X}$ of interest is generally a quite complex combination of basic structures that may include integer arithmetic, real arithmetic, booleans, lists, Herbrand terms, infinite terms, feature structures, ..., with implicit coercions as in e.g. Prolog IV [9]. Even the early CLP($\mathcal{R}$) system of [18] 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 [18] 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]]`).

Making the underlying type system explicit is not of purely theoretical in-

terest, as there is also a growing need for using a type system at compile-time for *detecting errors statically* in CLP($\mathcal{X}$) programs, instead of dynamically at run-time as in an untyped CLP system. The need for *a flexible type discipline* becomes even more accurate nowadays where CLP systems grow in size and various libraries and modules can be composed and re-used. It is the purpose of a *prescriptive type system*, such as Mycroft-O'Keefe system [23], to impose type restrictions on programs with respect to type declarations, e.g. as in Gödel [15]. On the other hand the purpose of a descriptive type system is to describe semantic properties of programs, such as approximating its success set via abstract interpretation. The latter approach will not be considered in this paper where we shall consider successes and finite failures as symmetrical and orthogonal to our typing issue.

The type system of Mycroft-O'Keefe is an adaptation to logic programs of the *parametric polymorphism* of ML. A *type inference algorithm* is given in [21]. In this sytem, types are first-order terms, type variables inside types, like α in $list(\alpha)$, express type parameters, so that programs defined over data structures of type $list(\alpha)$ can be used polymorphically over any list of elements of some type α. The flexibility of parametric polymorphism is however by far insufficient to handle properly coercions and complex structures in constraint logic programs.

A type system for CLP($\mathcal{X}$) should also support the meta-programming facilities of logic programming, through the ability to type meta-predicates such as `call(X)`, `setof(X,G,L)` or `arg(N,T,A)`. One way to acquire such a flexibility is to allow *subtypes*. A subtype relation between types $list(\alpha) < term$ allows to type the predicate $arg : nat \times term \times term \rightarrow bool$ so that, for instance, the atom $arg(1, [X|L], X)$ is well-typed although the second argument is a list. Similarly, we can type $call : bool \rightarrow bool$, $freeze : term \times bool \rightarrow bool$, $setof : \alpha \times bool \times list(\alpha) \rightarrow bool$, and the subtyping relation $bool < term$ allows to apply the arg predicate to atoms as well, whereas applying the `call` predicate to a list would raise a type error.

Static types with subtyping relations between type constructors are also compatible with powerful forms of object-oriented programming in CLP over computation domains with inheritance hierarchies, such as ψ-terms [1] [2], objects of POS logic [24] and feature structures [3] [26] [8].

Combining parametric polymorphism with subtyping in a static type system is thus a major issue. The problem has been deeply investigated in the study of object-orientation in higher-order functional languages [5]. In (first-order) logic languages, the picture is quite different as the difficulties due to the contravariance of the arrow constructor type ($\alpha \rightarrow \beta \preceq \alpha' \rightarrow \beta'$ if $\alpha' \preceq \alpha$ and $\beta \preceq \beta'$) disappear in absence of higher-order functional types.

Several type systems combining parametric polymorphism with subtypes have been proposed for logic programs. However a number of problems still appear: except in a restricted case in [24], subtype relations between (parametric) type constructors are not allowed. That prevents subtype relations as $list(\alpha) < term$ (also unallowed in [24]) and thus forbids to type meta-

predicates. Furthermore, except in a restricted setting in [6], type inference algorithms have been shown to be incomplete in general [24] [14] [16] [22].

By abstracting from particular structures as required in the CLP scheme, we study in this paper a *generic* static type system for CLP programs, independently of any specific computation domain $\mathcal{X}$. Section 2 presents a type system that includes parametric polymorphism and subtype relations between constructors of different arities. We show that in this system the subject reduction property (i.e. goal resolution preserves static types) holds independently of the computation domain under the only assumption that the type of predicates is definitional-generic in the sense of [21] [20].

Section 3 shows that type checking reduces in polynomial-time to the solving of subtype inequalities of some simple form.

Section 4 studies the solving of subtype inequalities in presence of subtype relations between constructors of different arities. We give a linear time algorithm for solving the systems of inequalities of the simple form needed for type checking, and we indicate that the system of inequalities involved in type inference can be solved in the universe of infinite regular types, but that it is an open problem in the universe of finite types.

The last section gives examples of type inference for variables and predicates.

2 Typing CLP($\mathcal{X}$) Programs

In this presentation we conform where possible to the notations of the Mycroft O'Keefe system in [21] [23].

2.1 Poterms as Types

Our type system is based on a structure of partially ordered terms, called poterms, that we use for representing types of CLP programs. 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 [1] [25] [2] [3] 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.

Let V_{Type} be a (countably infinite) set of type variables, called *parameters*, denoted by, α, β,... Let $(F_{Type}, \leq_F)$ be a poset of type constructors, noted κ..., given with their arity (constants, also called *basic types*, are type constructors of arity 0). We assume that F_{Type} contains a constant *bool* (used for typing predicates).

The *type language* is the set *Types* of poterms, noted τ,..., formed over V_{Type} and $(F_{Type}, \leq_F)$:

$$\tau ::= \alpha | \kappa(\tau_1, .., \tau_n).$$

A ground poterm, i.e. a type without parameters, is called *monomorphic*. A poterm containing parameters is called *polymorphic*.

A *substitution* σ is a mapping from variables in V_{Type} to $Types$, extended to a mapping from $Types$ to $Types$ by morphism. Poterms are ordered by the usual *instantiation pre-order* $\leq_{var}$, defined by $\tau \leq_{var} \tau'$, i.e. τ is *an instance* of τ', iff $\tau = \tau'\sigma$ for some substitution σ.

We assume that the order $\leq_F$ on type constructors is given with a function ι that associates to each pair of type constructors $\kappa \leq_F \kappa'$ of arity m and n respectively, an injection $\iota(\kappa, \kappa') : [1, n] \rightarrow [1, m]$ from the arguments of κ' to those of κ (therefore $m \geq n$ is assumed whenever $\kappa \leq_F \kappa'$) The role of ι is to extend $\leq_F$ to a subtype relation on poterms:

Definition 1 *The* subtyping order *is defined as the least relation* $\leq_{sub}$ *over Types satisfying:*

$$\alpha \leq_{sub} \alpha \qquad \textit{for any parameter } \alpha$$

$$\frac{\tau_{\iota 1} \leq_{sub} \tau'_1 \ \ldots \ \tau_{\iota n} \leq_{sub} \tau'_n}{\kappa(\tau_1, \ldots, \tau_m) \leq_{sub} \kappa'(\tau'_1, \ldots, \tau'_n)} \qquad \textit{for all } \kappa,\ \kappa' \textit{ such that } \kappa \leq_F^{\iota} \kappa'.$$

We say that a type τ *is a* subtype *of a type* τ' *(or* τ' *is a* supertype *of* τ*) iff* $\tau' \leq_{sub} \tau$.

Proposition 2 $(Types, \leq_{sub})$ *is a partial order.*

Note that in this definition, all type constructors are covariant. This allows to extend uniformly subtype relations on basic types, e.g. $int \leq_{sub} real$, to subtype relation on constructor types, e.g. $list(int) \leq_{sub} list(real)$, yet $list(int) \not\leq_{sub} list(\alpha)$. Note also that the definition of $\leq_{sub}$ with ι implies that the arity of a type constructor is less or equal to the one of its subtype constructors, moreover $V(\tau') \subseteq V(\tau)$ whenever $\tau \leq_{sub} \tau'$. Hence subtype relations between type constructors, such as $list(\alpha) \leq_{sub} term$, are possible, but not $emptylist \leq_{sub} list(\alpha)$ for example. This restriction is not really mandatory but simplifies the solving of subtype inequalities for type checking.

2.2 Typed CLP($\mathcal{X}$) Programs

CLP($\mathcal{X}$) programs are defined with another signature containing type declarations for function and predicate symbols. Let V be a (countably infinite) set of term variables. Let F be a set of function symbols, denoted by $f, g, ..$, given with their arity $ar(f) \geq 0$ and a type declaration $f : \tau_1 \times ... \times \tau_{ar(f)} \rightarrow \tau$. Let P be a set of predicate symbols, denoted by $p, ..$, given with their arity and a type declaration $p : \tau_1 \times ... \times \tau_{ar(p)} \rightarrow bool$. *Constraint* predicates are distinguished predicates in P, denoted by $c, ...$ P is assumed to contain constraint constants $true$, $false$ and the equality predicate $= : \alpha \times \alpha \rightarrow bool$.

An atom (or a constraint) is an expression of the form $p(t_1, ..., t_n)$ where p is a predicate symbol of arity n and the t_i's are terms.

Following [21], a *typed CLP program* is a CLP program with type declarations for variables inside clauses. The expressions belong to seven categories defined by the grammar:

$$
\begin{array}{lcll}
\tau & ::= & \alpha \mid \kappa(\tau_1, \cdots, \tau_k) & \text{Type} \\
t & ::= & X \mid f(t_1, \cdots, t_k) & \text{Term} \\
A & ::= & p(t_1, \cdots, t_k) & \text{Atom} \\
\phi & ::= & \epsilon \mid c(t_1, ..., t_k) \mid A \mid \phi_1, \phi_2 & \text{Formula} \\
C & ::= & [\forall X_1 : \tau_1, \cdots, X_n : \tau_n](A \leftarrow \phi) & \text{Clause} \\
G & ::= & [\forall X_1 : \tau_1, \cdots, X_n : \tau_n](\leftarrow \phi) & \text{Goal} \\
\Pi & ::= & C_1 \ \cdots C_l & \text{Program}
\end{array}
$$

where X denotes a variable, ϵ the "empty" formula, ϕ_1, ϕ_2 the conjunction of ϕ_1 and ϕ_2. Considering more general formulas, such as negation $\neg\phi$, or the `ask` operation $(A \rightarrow \Phi)$ of concurrent constraint programming, would not cause any extra difficulty w.r.t. our typing issue.

To be well-typed, well-formed expressions must also satisfy type rules. To express type rules, we use the following forms of assertions:

$t : \tau$	(t is a well-typed term of type τ)
A Atom	(A is a well-typed Atom)
ϕ Formula	(ϕ is a well-typed Formula)
C Clause	(C is a well-typed Clause)
G Goal	(G is a well-typed goal)
P Program	(P is a well-typed Program)

A *type context* Γ is a finite set of unique type assignments for variables, each of the form $X : \tau$.

Definition 3 (Well-Typed Program) *A typed program is* well-typed *if it can be derived by the type system in table 1.*

The rule Sub for subtyping allows us to see any expression of some type τ as an expression of any supertype τ' of τ. The type of a variable is determined by the declaration in the context. The rule for terms (resp. atoms) use $\leq_{var}$-polymorphically the type declared for the function (resp. predicate) symbols. The other rules are merely bureaucracy.

In the following we will see that the type of the equality predicate $=$: $\alpha \times \alpha \rightarrow bool$ plays a central role that marks a fundamental difference with other approaches to the typing of logic programs. In our type system the equality constraint $s : \tau = t : \tau'$ raises a type error if and only if τ and τ' have no common *supertype*, i.e. iff s and t are not comparable as objects of some type in the hierarchy. On the other hand, the constraint $s : \tau = t : \tau'$ is true iff s and t can be made equal to some value of a type τ'' subtype of τ and τ'. Therefore if τ and τ' have no common *subtype*, we know that the constraint will be false, hence this information can be used by the compiler to optimize

$$(Sub) \qquad \frac{\Gamma \vdash t : \tau \quad \tau \leq_{sub} \tau'}{\Gamma \vdash t : \tau'}$$

$$(Var) \qquad \Gamma \vdash X : \Gamma(x)$$

$$(Func) \qquad \frac{\Gamma \vdash t_1 : \tau_1\sigma \ \cdots \ \Gamma \vdash t_k : \tau_k\sigma}{\Gamma \vdash f(t_1, \cdots, t_k) : \tau\sigma} \quad \text{if } f : \tau_1 \times \cdots \times \tau_k \to \tau \text{ for any substitution } \sigma$$

$$(Atom) \qquad \frac{\Gamma \vdash t_1 : \tau_1\sigma \ \cdots \ \Gamma \vdash t_k : \tau_k\sigma}{\Gamma \vdash p(t_1, \cdots, t_k)\ Atom} \quad \text{if } p : \tau_1 \times \cdots \times \tau_k \to bool \text{ for any substitution } \sigma$$

$$(Empty) \qquad \Gamma \vdash \epsilon\ Formula$$

$$(Formula) \qquad \frac{\Gamma \vdash A\ Atom}{\Gamma \vdash A\ Formula}$$

$$(Conj) \qquad \frac{\Gamma \vdash \phi_1\ Formula \quad \Gamma \vdash \phi_2\ Formula}{\Gamma \vdash (\phi_1, \phi_2)\ Formula}$$

$$(Clause) \qquad \frac{\Gamma \vdash A\ Atom \quad \Gamma \vdash \phi\ Formula \quad \Gamma = \{X_1 : \tau_1 ... X_n : \tau_n\}}{\vdash [\forall X_1 : \tau_1, \cdots, X_n : \tau_n](A \leftarrow \phi)\ Clause}$$

$$(Goal) \qquad \frac{\{X_1 : \tau_1, \cdots, X_n : \tau_n\} \vdash \phi\ Formula}{\vdash [\forall X_1 : \tau_1, \cdots, X_n : \tau_n](\leftarrow \phi)\ Goal}$$

$$(Prog) \qquad \frac{\vdash C_1\ Clause \ \cdots \ \vdash C_l\ Clause}{\vdash (C_1 \ \cdots C_l)\ Program}$$

Table 1: The type system for CLP($\mathcal{X}$) programs.

the generated code (similarly to other descriptive type systems for static analyses), but certainly not to raise a static type error, as here successes and failures are considered symmetrical and orthogonal to our prescriptive typing issue. The only possible type error for $=$ in our system is when the arguments are not in the same category, i.e. have no common supertype.

Therefore, in a hierarchy like $int < term$, $bool < term$, with type declarations $integer : int \to bool$, $boolean : bool \to bool$, the goal $integer(X)$, $boolean(X)$ raises a type error, whereas the goal $X = Y$, $integer(X)$, $boolean(Y)$ doesn't raise a type error as $X : int$, $Y : bool$ is a possible typing (if the type $term$ is in the hierarchy). The crux of our approach is to see the equality predicate as a *constraint*, that may perform coercions in presence of subtype relations (e.g. like in Prolog IV [9]), and not as a

shorthand for variable renaming and substitution (e.g. like in [13, 14])[1].

2.3 Subject Reduction

Subject reduction is the property that evaluation rules transform a well-typed expression into another well-typed expression. The evaluation rule for CLP is top-down (left-right) SLD-resolution, so what has to be proven is that resolution transforms a well-typed goal into a well-typed goal. This proves that the execution model of CLP is consistent with the static type system, since a well-typed CLP program remains well typed along execution.

The next theorem proves the subject reduction property of our type system for CLP under the assumption that a predicate must be used $\leq_{var}$-monomorphically inside its definition with (mutually recursive) clauses. This notion of "definitional genericity" was introduced in [21] for escaping from the undecidability results for inferring the type of predicates, similarly to ML, inside mutually recursive definitions [20]. Here we show that it provides also a sufficient condition for subject reduction in our context (see appendix).

Hypothesis 1 (Definitional-genericity) *The type of the arguments of a defining occurrence of a predicate in a clause (an occurrence to the left of "$\leftarrow$") must be a subtype (not an instance) of a variable renaming of the declared type of the predicate.*

For example, the predicate *member* can be typed $\leq_{var}$-polymorphically, i.e. $member : \alpha \times list(\alpha) \rightarrow list(\alpha)$, if its definition doesn't not contain special facts like $member(1, [1])$, which otherwise force its type to be $member : int \times list(int) \rightarrow list(int)$, for satisfying the definitional-genericity condition.

Theorem 4 (Subject Reduction) *Let P be a definitional generic well-typed program. Let $\mathbf{R} \equiv p(t), R$ be a well-typed goal, and $\mathbf{R'} \equiv t = t_0, p_1(t_1), \ldots, p_n(t_n), R$ be a resolvent obtained by one step of SLD resolution with a program clause $p(t_0) \leftarrow p_1(t_1), .., p_n(t_n)$. If Γ is a type context for $\mathbf{R}$, i.e. $\Gamma \vdash \mathbf{R}$, then there exists an extension Γ' of Γ to possible further variables occurring in $\mathbf{R'}$, such that $\Gamma' \vdash \mathbf{R'}$.*

The subject reduction property means that static types can be dropped at execution time. It should be clear however that in presence of subtype relations, the static types usually denote object categories that the constraint solver needs to know in order to check the satisfiability of constraints, hence dynamic types cannot be eliminated in general. The dynamic types may contain more information than the static types, such as for example whether a variable is instanciated or not. The dynamic type Var cannot be a static type in our sense as it is not a constructor type, hence the issue of moding

[1] If needed, these operations of aliasing can be added without difficulty w.r.t. type checking or type inference, simply by adding a special predicate $\equiv$ such that $s : \tau \equiv t : \tau'$ is well-typed iff $\tau = \tau'$.

that is part of the descriptive type approach is clearly beyond the scope of this paper.

3 Type Checking

The rule system given above doesn't provide a type checking algorithm because the expressions in the premises of the (*Sub*) rule are not subexpressions of the conclusion, the substitution σ in the (*Func*) and (*Atom*) rules is not determined, and the (*Func*) and (*Sub*) rules are ambiguous.

The (*Sub*), (*Func*) and (*Atom*) rules can be replaced with two new rules:

$$(Func') \quad \frac{\Gamma \vdash t_1 : \tau_1' \ \cdots \ \Gamma \vdash t_k : \tau_k'}{\Gamma \vdash f(t_1, \cdots, t_k) : \tau\sigma} \quad \text{if } f : \tau_1 \times \cdots \times \tau_k \to \tau,\ \tau_i' \leq_{sub} \tau_i\sigma \text{ for some substitution } \sigma,\ 1 \leq i \leq k$$

$$(Atom') \quad \frac{\Gamma \vdash t_1 : \tau_1' \ \cdots \ \Gamma \vdash t_k : \tau_k'}{\Gamma \vdash p(t_1, \cdots, t_k)\ Atom} \quad \text{if } p : \tau_1 \times \cdots \times \tau_k,\ \tau_i' \leq_{sub} \tau_i\sigma \text{ for some substitution } \sigma,\ 1 \leq i \leq k$$

The new system obtained by replacing the (*Sub*), (*Func*) and (*Atom*) rules with the (*Func'*) and (*Atom'*) rules provides a *deterministic* syntax-directed type system suitable for type checking. It is easy to check that the new system is equivalent to the original system in the sense of

Proposition 5 *A program is well typed in the original system iff it is well typed in the new one.*

The construction of a substitution σ in rule (Func') and (Atom'), can be solved as follows. First of all remark that the variables in the types of clause variables are not under the scope of these substitutions, as they act only on the variables of the (renamed apart) type declarations for function and predicate symbols. Therefore in order to avoid unsound instanciations, the variables in the types of the clause variables will be replaced by new constants.

Now let Σ be the collection of inequations $\leq_{sub}$ imposed on types by rules (Func') and (Atom') in a derivation. Let us define the size of a system of inequalities as the number of symbols[2], clearly

Proposition 6 *The size of the system Σ of inequalities associated to a typed program is $O(nvd)$ where v the size of the type declarations for variables in the program, n is the size of the program, and d the size of the type declarations for function and predicate symbols.*

[2]By taking a compact representation with dags for terms, atoms and types, and by representing inequalities by special arcs between type nodes, a linear-size representaton of the system of inequalities is possible.

As the type system is deterministic we have:

Proposition 7 *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. Let us write $\tau[\alpha]$ for a type τ containing a parameter α as subterm (or equal to α).

Definition 8 *A system Σ of inequalities is* left-linear *if any variable has at most one occurrence at the left of $\leq$ in the system. Σ is* acyclic *if it does not contain any cycle, i.e., a set of inequalities of the form $\tau_1[\alpha] \leq \tau'_1[\alpha_1],\ \tau_2[\alpha_1] \leq \tau'_2[\alpha_2],\ ..,\ \tau_n[\alpha_{n-1}] \leq \tau'_n[\alpha]$.*

As the type parameters in the types of clause variables have been renamed into constants, the only type variables *at the left* of $\leq_{sub}$ in the system Σ, are those that come from the result type of a function declaration, e.g. α in $nil : list(\alpha)$. As the variables in a result type are distinct and renamed, the system Σ is thus stratified and acyclic, and the variables in Σ have a single occurrence at the left of $\leq_{sub}$, so Σ is left-linear.

Proposition 9 *The system of inequalities generated by the type checking algorithm is acyclic and left-linear.*

The next section studies subtype inequalities and presents an algorithm for solving acyclic left-linear systems in linear time (prop. 13). This shows with prop. 6 that type checking can be done in cubic time.

4 Subtype Inequalities

The *satisfiability of subtype inequalities (SSI) problem* [28] is the problem of determining whether a system of inequalities $\bigwedge_{i=1}^{n} \tau_i \leq_{sub} \tau'_i$ over types $\tau_1, \tau'_1, ..., \tau_n, \tau'_n$ has a solution, i.e., whether there exists a substitution σ such that $\bigwedge_{i=1}^{n} \tau_i\sigma \leq_{sub} \tau'_i\sigma$. This is not to be confused with the term semi-unification problem defined with the instantiation ordering $\bigwedge_{i=1}^{n} \tau_i\sigma \ \leq_{var} \tau'_i\sigma$, and whose undecidability was shown in [20].

Definition 10 *A* solution *to an inequality $\tau \leq_{sub} \tau'$ is a substitution σ such that $\tau\sigma \leq_{sub} \tau'\sigma$. A* principal solution *is a solution σ such that for any solution σ' there exists a substitution ρ such that $\forall \alpha \in V\ \alpha\sigma' \leq_{sub} \alpha\sigma\rho$.*

Type inference for simply typed lambda calculus with subtyping is polynomial time equivalent to the SSI problem [17]. For this reason, 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 two different axes:

- the structure of $(F, \leq_F)$
- the form of the types.

Special cases of the SSI problem are obtained by restricting the structure of $(F, \leq_F)$ to: disjoint union of lattices, n-crown, disjoint union of posets with suprema, partial orders.

Special cases of the SSI problem are also obtained by restricting the form of the terms:

Flat-SSI	types are just variables or constants
Uniform-SSI	finite terms without variables
	built over $(F, \leq_F)$ and $\rightarrow$
Homogeneous-SSI	relation between constructors of same arity
Non-uniform-SSI	general poterms.

Complexity results for Uniform-SSI have been provided over specific structures of the poset $(F, \leq_F)$. When $(F, \leq_F)$ is diecrete, i.e. no different elements are comparable, the problem reduces to unfication since $\leq_F$ reduces in this case to equality. If $(F, \leq_F)$ is a disjoint union of lattices (for which the discrete poset is a special case, namely a disjoint union of one-element lattices), then there is a polynomial-time algorithm that solves the Uniform-SSI problem [28]. When $(F, \leq_F)$ is a n-crown, the Uniform-SSI algorithm is Pspace-hard [28]. In the general case, i.e. $(F, \leq_F)$ is a partial order, Frey [10] recently proved that UNIFORM-SSI is Pspace-complete. These results are summarized in the following table:

structure of $(F, \leq_F)$	**Uniform-SSI**
disjoint union of lattices	P-complete [28]
n-crown	Pspace-hard [28]
partial order	Pspace-complete [10]

Homogeneous SSI have been studied in [25] in a quasi-lattice structure. An algorithm is given and proven correct for declarations without parametric polymorphism. When parametric polymorphism is taken into account, the algorithm may return typings which are not most general or fail even if most general typings exist. The difficulty is due to the fact that the algorithm relies on the computability of the least upper bound between the type declared for a predicate argument and the type of the actual argument. The computability is not guaranteed in presence of parametric polymorphism, due to the existance of infinite ascending chains like $\perp \leq list(\perp) \leq list(list(\perp)) \leq \cdots$.

As already pointed out, in our poterm structure, if $\tau \leq_{sub} \tau'$ then $V(\tau') \subseteq V(\tau)$, hence we can prove that

Proposition 11 *There is no poterm α such that $\alpha \leq_{sub} \tau[\alpha]$. If $\tau[\alpha] \leq_{sub} \alpha$ has a solution then there exists a poterm τ' such that $\tau[\alpha] \leq_{sub} \tau'$ and $\alpha \notin V(\tau')$.*

For instance, $\alpha \leq_{sub} list(\alpha)$ has no (finite) solution in any poterm structure. On the other hand, if $list(\alpha) \leq_{sub} term$ then $list(\alpha) \leq_{sub} \alpha$ has one solution: $\alpha = term$. The following algorithm benefits from this property.

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 principal solutions in posets $(F, \leq_F)$ satisfying the following

Hypothesis 2 *For every function symbol κ in $(F, \leq_F)$ the set $\{\kappa' | \kappa \leq_F \kappa'\}$ of greater elements than κ, admits a maximum element denoted by $\overline{\kappa}$.*

This means that every $\leq_F$-connected component of function symbols admits a maximum element. The definition extends naturally to types as follows:

1. $\overline{\alpha} = \alpha$
2. $\overline{\kappa(\tau_1, .., \tau_n)} = \overline{\kappa}(\overline{\tau_1}, .., \overline{\tau_n})$

The hypothesis entails the existence of a greatest supertype $\overline{\tau}$ for any type τ, as well as the existence of a root in every $\leq_{sub}$-connected component of types. For example a hierarchy 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 hypothesis doesn't assume, nor it is implied by, the existence of a least upper bound to types having a upper bound (sup-quasi-lattice hypothesis in [25]).

In this section, a system Σ is in *solved form* if it contains only equations of the form

$$\{\alpha_i = \tau_i\}_i$$

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

(Dec)	$f(\tau_1, ..., \tau_m) \leq g(\tau'_1, ..., \tau'_n) \wedge \Sigma \longrightarrow \bigwedge_{i=1}^n \tau_{\iota i} \leq \tau'_i \wedge \Sigma$
	if $f \leq_F^\iota g$
(VarLeft)	$\alpha \leq \tau \wedge \Sigma \longrightarrow \alpha = \tau \wedge \Sigma[\tau/\alpha]$
	if $\tau \neq \alpha$, $\alpha \notin V(\tau)$.
(VarRight)	$\tau \leq \alpha \wedge \Sigma \longrightarrow \alpha = \overline{\tau} \wedge \Sigma[\overline{\tau}/\alpha]$
	if $\tau \notin V$, $\alpha \notin V(l)$ for any $l \leq r \in \Sigma$, and $\alpha \notin V(\overline{\tau})$.

By taking as complexity measure of the system the sum of the sizes of the terms in the left-hand side of inequalities (without equalities), each rule strictly decreases the complexity of the system: (Dec) and (VarLeft) by one, (VarRight) by the size of τ.

Lemma 12 *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.*

One can easily check that each rule preserves the left-linearity as well as the acyclicity of the system, and that each rule preserves the satisfiability of the system, as well as its principal solution if one exists, from which follows:

Theorem 13 *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 $\sigma_{\Sigma'}$ is a principal solution of Σ.*

Hence acyclic left-linear subtype inequalities can be solved in linear time.

4.2 The general case

In this section we consider the structure of poterms with the additional hypothesis that the poset of type constructor symbols $(F, \leq_F)$ is an inf-quasi-lattice (i.e. a greatest lower bound exists whenever a lower bound exists).

Hypothesis 3 *$(F, \leq_F)$ is an inf-quasi-lattice.*

The idea is to check the consistency of a general system of inequalities by replacing in the algorithm of the previous section the (VarLeft) rule by a rule computing the g.l.b. of the supertypes of a type:

$$\text{(VarLeft)} \quad \alpha \leq \tau \wedge \alpha \leq \tau' \wedge \Sigma \quad \begin{array}{l} \longrightarrow \alpha \leq glb(\tau, \tau') \wedge \Sigma \\ \longrightarrow \bot \text{ if } \tau \text{ and } \tau' \text{ have no lower bound} \end{array}$$

This is not sufficient for checking the satisfiability of the system in the structure of finite poterms. However it can be proved that this is sufficient for proving the satisfiability of the system in the structure of infinite regular types (i.e. recursive types [5]), which admit solutions to equations of the form $\alpha = list(\alpha)$ (namely the type $list(list(...)))$.

In absence of subtype relation between type constructors of different arities, Amadio & Cardelli's algorithm for checking consistency with recursive types is polynomial in a lattice structure, while the satisfiability in finite types (with a generalization of Fuh & Mishra's algorithm [12]) has been shown by Frey [10] Pspace-complete in an arbitrary poset.

The technique used in [10] for proving consistency in infinite types can be generalized to our case with subtype relations between type constructors of different arities in a poterm structure. The study of the algorithm is beyond the scope of this paper but will be the matter of a forthcoming paper, we refer to [10] for details.

5 Type Inference

Definition 14 *A semi-typed program is a typed CLP program without type declarations for variables. A* variable typing *for a semi-typed program P is a type declaration for variables in each clause of the program. A variable*

typing Γ *is a* principal variable typing *if for any other variable typing* Γ' *for* P *there exists a substitution* σ *such that* $\forall x \in V\ \Gamma'(x) \leq_{sub} \Gamma(x)\sigma$.

Similarly to the type checking algorithm, one can consider the system of inequalities $\Sigma(P)$ obtained by collecting the subtype inequalities along the derivation of a program P, with unknowns for the type of variables. The size of this system is $O(nd)$ where d is the size of the declarations for functions and predicates and n is the size of the program. Clearly

Proposition 15 *A semi-typed CLP program* P *admits a variable typing iff the system of inequalities* $\Sigma(P)$ *is satisfiable.*

Note that the system of inequalities is no longer left-linear, but still acyclic. As indicated in the previous section, under the hypothesis that the types form an inf-quasi-lattice with roots, the solving of these systems of inequalities in infinite regular types can be done with a generalization of Frey's algorithm [10].

Following [21], types for predicates can be inferred from the program, by imposing that a predicate is used with the same type instance (i.e. $\leq_{var}$-monomorphically) in each occurrence within the clauses of its (mutually recursive) definition. The reason for this restriction is, as in ML, to escape from the undecidability of the original semi-unification problem w.r.t. $\leq_{var}$ [20]. Under this restriction, assigning type variables to the argument type patterns of untyped predicates, and using these type patterns without renaming in the rules of the type checker, reduces again the problem of predicate type inference to solving a system of subtype inequalities. With the same argument, one can thus establish the decidability of predicate type inference in the universe of infinite types, and the existence of a principal type to predicates under the restriction of $\leq_{var}$-monomorphic definition.

Principal types for predicates are not very informative however. As the only type constraints are that the type of the arguments are subtypes of the declaration pattern, the components of the principal type of a predicate are always either variables or roots of connected components in the type hierarchy. For example, in the type hierarchy $list(\alpha) \leq_{sub} term$, the type inferred for *append* from its usual definition by two clauses will be $term \times term \times term \rightarrow bool$, which is indeed a principal typing for *append*. On the other hand, computing a *most informative type* for predicates, defined as the principal type w.r.t. $\leq_{var}$ and the most specific w.r.t. $\leq_{sub}$ (i.e. a type t such that for any other type t' there exists a substitution ρ such that $t' \leq_{sub} t\rho$) is not always a good strategy. For example, a predicate testing the equality of a number to 0, defined by the fact $p(0)$, would then be typed $p : zero \rightarrow bool$, instead of $p : num \rightarrow bool$, whenever the hierarchy contains the type $zero < num$.

One pragmatical approach for infering more precise types is to remove subtype relations in the type hierarchy of a module, such as $list(\alpha) \leq_{sub} term$ in a list module, so that the roots of the connected components match the

expected types to be inferred in that module. For example, by masking the subtype relation $list(\alpha) < term$, in the list module, the principal type inferred for *append* is then $list(\alpha) \times list(\alpha) \times list(\alpha) \rightarrow bool$.

6 Conclusion and perspectives

Typing CLP programs for detecting errors statically, while retaining the flexibility required for preserving all the metaprogramming facilities of logic programming, is the challenge that conducted the design of the type system presented in this paper. We have shown that type checking in our system can be done in polynomial time, and that type inference for variables and for predicates is possible in the universe of regular infinite types.

Our next step will be to make experiments with an implementation of our type system on a large constraint logic software, such as Sicstus Prolog [27] for instance. The built-in predicates of Sicstus Prolog are mainly metaprogramming predicates, on top of which a large amount of libraries have been built for ISO Prolog, various data structures, CLP($\mathcal{Q}$), CLP($\mathcal{R}$), CLP($\mathcal{FD}$), objects, etc. We believe that it is possible to completely type Sicstus Prolog and its libraries in our system, while still retaining all the intended use of the predicates.

From a practical point of view, considering regular infinite types is probably not a limitation. From a theoretical point of view however, the decidability of the satisfiability of subtype inequalities in finite types with relations between type constructors of different arities is an interesting open problem, that was already mentionned by Smolka in his thesis [25].

Another aspect of our type system is that it is limited to covariant functors. We do not believe that we can advantageously drop this restriction to include mixed contravariant functors as we would then inherit of the undecidability of system F for type checking. On the other hand, the flexibility of our type system for CLP can be extended in several directions. One is to infer types for function symbols as well. Another issue is to allow the overloading of predicates and function symbols, i.e. the possibility to give the same name to symbols that differ by their type, wherever the type checker can eliminate the ambiguities.

Acknowledgements: We are especially grateful to Alexandre Frey, Michael Hanus and Gert Smolka for fruitful discussions which helped us to clarify some points in the literature on types.

References

[1] H. Ait-Kaci and R. Nasr, LOGIN: A Logical Programming Language with Built-in Inheritance, Journal of Logic Programming, 3, 187-215, 1986.

[2] H. Ait-Kaci, An Introduction to LIFE - Programming with logic, Inheritance, Functions and Equations, in D. Miller (ed.), Proc. of the Internat. Symp. on Logic Programming, Vancouver, BC, pages 52-68, The MIT Press, Cambridge, MA, 1993.

[3] H. Ait-Kaci, A. Podelski and S.C. Goldstein, Order-Sorted Feature Theory Unification, Journal of Logic Programming, pp.100-124, 1997.

[4] A. Aiken and E.L. Wimmers, Type Inclusion Constraints and Type Inference, Conf. Functional Programming Languages and Computer Science, p.31-41, ACM Press, 1993.

[5] R. M. Amadio and L. Cardelli, Subtyping Recursive Types, ACM Transactions on Programming Languages and Systems, vol. 15, No. 4, 575-631, 1993.

[6] C. Beierle, Type Inferencing for Polymorphic Order-Sorted Logic Programs, 12th International Conference on Logic Programming, The MIT Press, 765-779, Tokyo, 1995.

[7] F. Bourdoncle, S. Merz, Type checking higher-order polymorphic multimethods, Proc. POPL'97, pp.302-315, Paris, jan. 1997.

[8] B. Carpenter, Typed feature structures: a generalization of first-order terms, International Symposium on Logic Programming, ILPS91, 202-217, San Diego, 1991.

[9] A. Colmerauer, Specification of Prolog IV, LIM technical report, 1996.

[10] A. Frey, Satisfying Subtype Inequalities in Polynomial Space, Proc. SAS'97, LNCS 1302, 1997.

[11] Y.C. Fuh, P. Mishra, Polymorphic Subtype Inference; Closing the Theory-Practice Gap, Proc. TAPSOFT'89, LNCS 352, pp.167-183, 1989.

[12] Y.C. Fuh, P. Mishra, Type inference with subtypes. Proc. ESOP'88, LNCS 300, pp.94-114, 1988.

[13] M. Hanus, Parametric Order-Sorted Types in Logic Programming, International Joint Conference on Theory and Practice of Software Development, (TAPSOFT 1991), LNCS Vol. 494, 181-200, 1991.

[14] M. Hanus, Logic Programming with Type Specifications, in F. Pfenning Ed., Types in Logic Programming, MIT Press, 1992.

[15] P. Hill, J. Lloyd, The Gödel programming language, MIT Press, 1994.

[16] P. Hill, R. Topor, A semantics for typed logic programs, in F. Pfenning Ed., Types in Logic Programming, MIT Press, 1992.

[17] M. Hoang , J.C. Mitchell, Lower Bounds on Type Inference with Subtypes, ACM Symposium on Principles of Programming Languages, 1995.

[18] J. Jaffar, J.L. Lassez, Constraint Logic Programming, Proc. of POPL'87, Munich, 111-119, 1987.

[19] J. Jaffar, M.J. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming, 19-20, 1994.

[20] A.J. Kfoury, J. Tiruyn, P. Urzyczyn, The Undecidability of the Semi-Unification Problem, Technical Report BUCS-89-010, Boston Univ., Oct. 89.

[21] T.K. Lakshman, U.S. Reddy, Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System, International Symposium on Logic Programming, ILPS91, 202-217, San Diego, 1991.

[22] G. Meyer, Type Checking and Type Inferencing for Logic Programs with Subtypes and Parametric Polymorphism, Informatik Berichte 200, Fern Universitat Hagen, 1996.

[23] A. Mycroft, R.A. O'Keefe, A Polymorphic Type System for Prolog, Artificial Intelligence, 23, 295-307, 1984.

[24] G. Smolka, Logic Programming with Polymprphically Order-Sorted Types, in J. Grabowski, P. Lescanne,, W. Wechler, editors, Algebraic and Logic Programming, LNCS 343, 53-70, 1988.

[25] G. Smolka, Logic Programming over Polymorphically Order-Sorted Types, PhD Thesis, Universitat Kaiserslautern, Germany, 1989.

[26] G. Smolka, Feature Constraint Logics for Unification Grammars, Journal of Logic Programming, 12:51-87, 1992.

[27] SICStus Prolog User's Manual, release 3#5, SICS, october 1996.

[28] J. Tiuryn, Subtype Inequalities, Seventh Symposyum on Logic in Computer Science, 308-315, 1992.

A Appendix: Proofs of some theorems

Proposition 5. A program is well typed in the original system iff it is well typed in the new one.

Proof.
Soundness. If a program is typable in the new system, it is typable in the original one.
In the proof generated by the new system, just replace every occurrence of the (*Func'*) and (*Atom'*) rules respectively with

$$\frac{\dfrac{\Gamma \vdash t_1 : \tau_1 \quad \tau_1 \leq_{ext} \tau_1'}{\Gamma \vdash t_1 : \tau_1'\sigma} \cdots \dfrac{\Gamma \vdash t_k : \tau_k \quad \tau_k \leq_{ext} \tau_k'}{\Gamma \vdash t_k : \tau_k'\sigma}}{\Gamma \vdash f(t_1, \cdots, t_k) : \tau'\sigma} \quad \text{if } f : \tau_1' \times \cdots \times \tau_k' \to \tau'$$

$$\frac{\dfrac{\Gamma \vdash t_1 : \tau_1 \quad \tau_1 \leq_{ext} \tau_1'}{\Gamma \vdash t_1 : \tau_1'\sigma} \cdots \dfrac{\Gamma \vdash t_k : \tau_k \quad \tau_k \leq_{ext} \tau_k'}{\Gamma \vdash t_k : \tau_k'\sigma}}{\Gamma \vdash p(t_1, \cdots, t_k)\ Atom} \quad \text{if } p : \tau_1' \times \cdots \times \tau_k'$$

Completeness. If a program is typable in the original system, it is typable in the new one.
By induction on the structure of the expression to be typed. Let us take the most complex case, i.e., the expression to be type checked is $f(t_1, \cdots, t_k)$, where $f : \tau_1 \times \cdots \times \tau_k \to \tau$. There are two possible cases, namely, the proof terminates by the application of the

1. (*Func*) rule
2. (*Sub*) rule.

Case 1: (Func). According to the (*Func*) rule, $t_1 : \tau_1\sigma \ \cdots \ t_k : \tau_k\sigma$. Then, by the induction hypothesis, the terms $t_1 \ \cdots \ t_k$ are also type checked to $t_1 : \tau_1\sigma \ \cdots \ t_k : \tau_k\sigma$ by the new rule system. By applying the (*Func'*) rule with $\tau' = \tau\sigma$ and $\tau_i' = \tau_i\sigma, i = 1..k$, the expression $f(t_1, \cdots, t_k)$ is also well typed to $\tau\sigma$ by the new rule system.

Case 2: (Sub). With no loss of genericity we can assume that the (*Sub*) rule is never used twice consecutively. Then the proof terminates with

$$\frac{\dfrac{\Gamma \vdash t_1 : \tau_1\sigma \ \cdots \ \Gamma \vdash t_k : \tau_k\sigma}{\Gamma \vdash f(t_1, \cdots, t_k) : \tau\sigma \quad \tau\sigma \leq_{ext} \tau'}}{\Gamma \vdash f(t_1, \cdots, t_k) : \tau'} \quad \text{if } f : \tau_1 \times \cdots \times \tau_k \to \tau$$

By the induction hypothesis, the terms $t_1 \ \cdots \ t_k$ are also type checked to $t_1 : \tau_1\sigma \ \cdots \ t_k : \tau_k\sigma$ by the new rule system. By applying the (*Func'*) rule with $\tau' = \tau\sigma$ and $\tau_i' = \tau_i\sigma, i = 1..k$, the expression $f(t_1, \cdots, t_k)$ is also well typed to τ' by the new rule system. □

Subject Reduction Theorem 4. Let $\mathbf{R} \equiv R, p(t)$ be a goal, **R'** $\equiv R, p_1(t_1), .., p_n(t_n), (t = t_0)$ a resolvent obtained by the application of one step of SLD resolution by nondeterministically selecting the program clause $p(t_0) \leftarrow p_1(t_1), .., p_n(t_n)$ satisfying definitional genericity. Let Γ be a type context assigning a type to each variable occurring in $\mathbf{R}$, then there exists an extension Γ' of Γ to possible further variables occurring in **R'**, such that

$$\text{if} \quad \Gamma \vdash \mathbf{R} \quad \text{then} \quad \Gamma' \vdash \mathbf{R'}.$$

Proof. First the construction of Γ': let $p : \tau$ be the static declaration of predicate p, σ a substitution such that $\tau_t \leq_{sub} \tau\sigma$ and Γ'' a type context such that $\Gamma'' \vdash p(t_0) \leftarrow p_1(t_1), .., p_n(t_n)$; the type context Γ' is then defined as $\Gamma' = \Gamma \cup \Gamma''\sigma$. Now, what has to be proven is that $\Gamma' \vdash R, p_1(t_1), .., p_n(t_n), (t = t_0)$, given that $\Gamma \vdash R, p(t)$. By the $(Conj)$ typing rule, this is equivalent to prove that

1. $\Gamma' \vdash R$
2. $\Gamma' \vdash p_1(t_1), .., p_n(t_n)$
3. $\Gamma' \vdash (t = t_0)$

given that $\Gamma \vdash R$ and $\Gamma \vdash p(t)$. These three conditions are proven below.

1. Since Γ' is an extension of Γ, $\Gamma \vdash R$ implies $\Gamma' \vdash R$.
2. Since the program is well-typed, there is a type context Γ'' such that $\Gamma'' \vdash p(t_0) \leftarrow p_1(t_1), .., p_n(t_n)$. Since $\Gamma' = \Gamma \cup \Gamma''\sigma$, then $\Gamma' \vdash p_1(t_1), .., p_n(t_n)$.
3. For $\Gamma' \vdash (t = t_0)$ to hold, since the typing of $=$ is $=: \alpha \times \alpha$, it must be

   $\tau_t \preceq \alpha$

   $\tau_{t_0} \preceq \alpha$

   or, equivalently

   $\tau_t \leq_{sub} \alpha\rho$

   $\tau_{t_0} \leq_{sub} \alpha\rho$

   for some substitution ρ. Let $p : \tau$ be the static declaration of predicate p. Since $\Gamma \vdash p(t)$, then $\tau_t \leq_{sub} \tau\sigma$, for some substitution σ. For the definitional genericity of p, we have that $\tau_{t_0} \leq_{sub} \tau$ and then $\tau_{t_0} \leq_{sub} \tau\sigma$ (up to variable renaming).

   Let us take $\alpha\rho \equiv \tau\sigma$, then we have that $\tau_t \leq_{sub} \alpha\rho$ and $\tau_{t_0} \leq_{sub} \alpha\rho$.

$\square$

Proposition 9. The system of inequalities generated by the type rules is acyclyc and left-linear.

Proof. To see why the system is acyclic, recall that rules generating inequalities are $(Func')$ and $(Atom')$ and they can only be applied in sequences $(Atom')$-$(Func')$ or $(Func')$-$(Func')$.

$(Atom')$-$(Func')$. The application of $(Atom')$ produces

$$\tau_i'[x_1] \leq \tau_i\sigma_1[x_2]$$

where x_1 and x_2 are different since x_1 can be opportunely renamed, while the application of $(Func')$ produces

$$\tau_j'[x_3] \leq \tau_j\sigma_2[x_4]$$

in general and it can produce

$$\tau_j'[x_3] \leq \tau_j\sigma_2[x_1]$$

if $\sigma_2 = \{x_4 \leftarrow x_1\}$. Now, since x_3 can be opportunely renamed, it can be chosen different from x_2 to guarantee that

$$\tau_i'[x_1] \leq \tau_i\sigma_1[x_2], \tau_j'[x_3] \leq \tau_j\sigma_2[x_1]$$

is not a cycle.

Tha case $(Atom')$-$(Func')$ is similar.

The system is left-linear since variables occurring in the left-hand side of inequalities occur in type terms τ_i' introduced by rules $(Func')$ and $(Atom')$ and can be renamed without any loss of generality. □

Proposition 11a There is no poterm x such that $x \leq_{sub} s[x]$.

Proof. For $x \leq_{sub} s[x]$ to hold, there must be a substitution σ such that $x\sigma \leq_{sub} s[x]\sigma$. Let $s[x]$ be the generic term $g(t_1, .., t_{h-1}, x, t_{h+1}, .., t_n)$. For $x\sigma \leq_{sub} g(t_1, .., t_{h-1}, x, t_{h+1}, .., t_n)\sigma$ to hold, x must be of the form

$$f(s_1, .., s_{k-1}, x, s_{k+1}, .., s_m)$$

with $f \leq_F^{\iota} g$, $n \leq m$, $\iota : [1, n] \rightarrow [1, m]$, $\iota(h) = k$. But this would give $x\sigma = f(s_1, .., s_{k-1}, x, s_{k+1}, .., s_m)\sigma$ which is false over finite terms. □

Proposition 11b If $s[x] \leq_{sub} x$ has a solution then $s[x] \leq_{sub} t$ with $x \notin V(t)$, for some t.

Proof. We consider only the case where x occurs at the first level, the other

cases can be proven by induction on the structure of the terms. Let $s[x] \equiv f(s_1, .., s_{k-1}, x, s_{k+1}, .., s_m)$ and $t \equiv g(t_1, .., t_n)$, with $f \leq_F^{\iota} g$, $n \leq m$. If x occurred in t, say $t \equiv g(t_1, .., t_{h-1}, x, t_{h+1}, .., t_n)$, then $x = g(t_1, .., t_{h-1}, x, t_{h+1}, .., t_n)$ in the solution σ, which is false over finite terms. So $x \notin V(t)$ and x is not in the image of ι. $\square$