Using Modes to Ensure Subject Reduction for Typed Logic Programs with Subtyping

Jan-Georg Smaus, Francois Fages, Pierre Deransart

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Using Modes to Ensure Subject Reduction for Typed Logic Programs with Subtyping

Jan–Georg Smaus — François Fages — Pierre Deransart

N° 4020

October 2000

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4020--FR+ENG

Using Modes to Ensure Subject Reduction for Typed Logic Programs with Subtyping

Jan–Georg Smaus* , François Fages† , Pierre Deransart‡

Thème 2 — Génie logiciel
et calcul symbolique
Projet Contraintes

Rapport de recherche n° 4020 — October 2000 — 24 pages

Abstract: We consider a general prescriptive type system with parametric polymorphism and subtyping for logic programs. 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". It is well-established that without subtyping, this property is readily obtained for logic programs w.r.t. their standard (untyped) execution model. Here we give syntactic conditions that ensure subject reduction also in the presence of general subtyping relations between type constructors. The idea is to consider logic programs with a fixed dataflow, given by modes.

Key-words: typed logic programs, modes, type systems, subtyping, subject reduction

This paper is the complete version of a paper presented at FST&TCS 2000 [18]. It contains all proofs omitted there for space reasons.

* jan.smaus@cwi.nl, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.
† francois.fages@inria.fr
‡ pierre.deransart@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Utilisation des modes pour garantir la propriété de "subject reduction" pour les programmes logiques typés avec sous-typage

Résumé : Nous considérons un système de types prescriptif avec polymorphisme paramétrique et sous-typage pour les programmes logiques. La propriété de "subject reduction" exprime la cohérence du système de types vis à vis du modèle d'exécution: si un programme est "bien typé", alors toutes les dérivations à partir d'un but "bien typé" sont encore "bien typées". Il est bien établi que sans sous-typage, cette propriété est vérifiée par les programmes logiques munis de leur modèle d'exécution standard (non typé). Dans cet article nous donnons des conditions syntaxiques qui garantissent cette propriété également en présence de relations de sous-typage entre constructeurs de types. L'idée est de considérer les programmes logiques ayant un flot de données fixe, déterminé par des modes.

Mots-clés : programmes logiques typés, modes, systèmes de types, sous-typage, "subject reduction"

1 Introduction

Prescriptive types are used in logic and functional programming to restrict the underlying syntax so that only "meaningful" expressions are allowed. This allows for many programming errors to be detected by the compiler. Gödel [9] and Mercury [19] are two implemented typed logic programming languages.

A natural stability property one desires for a type system is that it is consistent with the execution model: once a program has passed the compiler, it is guaranteed that "well-typed" configurations will only generate "well-typed" configurations at runtime. Adopting the terminology from the theory of the λ-calculus [21], this property of a typed program is called *subject reduction*. For the simply typed λ-calculus, subject reduction states that the type of a λ-term is invariant under reduction. This translates in a well-defined sense to functional and logic programming.

Semantically, a type represents a set of terms/expressions [10, 11]. Now subtyping makes type systems more expressive and flexible in that it allows to express inclusions among these sets. For example, if we have types *int* and *real* defined in the usual way, we would probably want to declare $int \leq real$, i.e., the set of integers is a subset of the set of reals. More generally, subtype relations like for example $list(u) < term$, which expresses the possibility of viewing a list as a term, make it possible to type Prolog meta-programming predicates [6], as shown in Ex. 4 below and Sec. 6.

In functional programming, a type system that includes subtyping would then state that wherever an expression of type σ is expected as an argument, any expression having a type $\sigma' \leq \sigma$ may occur. Put differently, an expression of type σ can be used wherever an expression of type $\sigma' \geq \sigma$ is expected. The following example explains this informally, using an ad hoc syntax.

Example 1 *Suppose we have two functions* $sqrt : real \to real$ *and* $fact : int \to int$ *which compute the square root and factorial, respectively. Then* $sqrt\ (fact\ 3)$ *is a legal expression, since* $fact\ 3$ *is of type int and may therefore be used as an argument to sqrt, because sqrt expects an argument of type real, and* $int < real$.

Subject reduction in functional programming crucially relies on the fact that there is a clear notion of dataflow. It is always the *arguments* (the "input") of a function that may be smaller than expected, and the result (the "output") may be greater than expected. This is best illustrated by a counterexample, which is obtained by introducing *reference* types.

Example 2 *Suppose we have a function*

$$f : real\ REF \to real$$
$$let\ f(x) = x := 3.14;\ return\ x$$

So f *takes a* reference *(pointer) to a real as argument, assigns the value* 3.14 *to this real, and also return* 3.14. *Even though* $int < real$, *this function cannot be applied to an int REF, since the value* 3.14 *cannot be assigned to an integer.*

In the example, the variable x is used both for input and output, and hence there is no clear direction of dataflow. While this problem is marginal in functional programming (since reference types play no essential role in the paradigm), it is the main problem for subject reduction in logic programming with subtypes, as we show in the next example.

Subject reduction for logic programming means that resolving a "well-typed" goal with a "well-typed" clause will always result in a "well-typed" goal. It holds for parametric polymorphic type systems without subtyping [11, 13].[1]

Example 3 *In analogy to Ex. 1, suppose* `Sqrt/2` *and* `Fact/2` *are predicates of declared type* (`Real`, `Real`) *and* (`Int`, `Int`)*, respectively. Consider the program*

```
Fact(3,6).
Sqrt(6,2.449).
```

and the derivations

$$\texttt{Fact(3,x)},\ \texttt{Sqrt(x,y)} \leadsto \texttt{Sqrt(6,y)} \leadsto \square$$
$$\texttt{Sqrt(6,x)},\ \texttt{Fact(x,y)} \leadsto \texttt{Fact(2.449,y)}$$

In the first derivation, all arguments always have a type that is less than or equal to the declared type, and so we have subject reduction. In the second derivation, the argument `2.449` *to* `Fact` *has type* `Real`*, which is strictly greater than the declared type. The atom* `Fact(2.449,y)` *is illegal, and so we do not have subject reduction.*

In this paper, we address this problem by giving a fixed direction of dataflow to logic programs. This is done by introducing modes [1] and replacing unification with double matching [2], so that the dataflow is always from the input to the output positions in an atom. We impose a condition on the types of terms in the output positions, or more precisely, on the types of the *variables* occurring in these terms: each variable must have *exactly* the declared (expected) type of the position where it occurs.

In Ex. 3, let the first argument of each predicate be input and the second be output. In both derivations, `x` has type `Int`. For the atom `Fact(3,x)`, this is exactly the declared type, and so the condition is fulfilled for the first derivation. In contrast, for the atom `Sqrt(6,x)`, the declared type is `Real`, and so the condition is violated.

The contribution of this paper is a statement that programs that are typed according to a type system with subtyping, and respect certain conditions concerning the modes, enjoy the subject reduction property, i.e., the type system is consistent w.r.t. the (untyped) execution model. This means that effectively the types can be ignored at runtime, which has traditionally been considered as desirable, although there are also reasons for keeping the types during execution [14]. In Sec. 6, we discuss the conditions on programs.

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 set of terms for

[1] However, it has been pointed out [7, 10] that the first formulation of subject reduction by Mycroft and O'Keefe [13] was incorrect, namely in ignoring the transparency condition, which we will define in Section 2.

which a predicate is true. There are few works considering prescriptive type systems for logic programs with subtyping [4, 5, 6, 8, 10]. Hill and Topor [10] give a result on subject reduction only for systems without subtyping, and study general type systems with subtypes. However their results on the existence of principal typings for logic programs with subtyping turned out to be wrong, as pointed out by Beierle [4]. He shows the existence of principal typings with subtype relations between constant types only, and provides type inference algorithms. Beierle and also Hanus [8] do not claim subject reduction for the systems they propose. Fages and Paltrinieri [6] have shown a weak form of subject reduction for constraint logic programs with general subtyping relations, where equality constraints replace term substitutions in the execution model.

On the other hand, the idea of introducing modes to ensure subject reduction for standard logic programs was already proposed by Dietrich and Hagl [5]. However they do not study the decidability of the conditions they impose on the subtyping relation. Furthermore since each result type must be transparent (a condition we will define later), this means effectively that in general, subtype relations between type constructors of different arities are forbidden. We illustrate this with an example.

Example 4 *Assume types* `Int`, `String` *and* `List(u)` *defined as usual, and a type* `Term` *that contains all terms (so all types are subtypes of* `Term`*). Moreover, assume* `Append` *as usual with declared type* $(\texttt{List(u)}, \texttt{List(u)}, \texttt{List(u)})$*, and a predicate* `Functor` *with declared type* $(\texttt{Term}, \texttt{String})$*, which gives the top functor of a term. In our formalism, we could show subject reduction for the query* $\texttt{Append}([1], [], \texttt{x})$, $\texttt{Functor}(\texttt{x}, \texttt{y})$*, whereas this is not possible in [5] because the subtype relation between* `List(Int)` *and* `Term` *cannot be expressed.*

The plan of the paper is as follows. Section 2 mainly introduces the type system. In Sec. 3, we show how expressions can be typed assigning different types to the variables, and we introduce *ordered substitutions*, which are substitutions preserving types, and thus ensuring subject reduction. In Sec. 4, we show under which conditions substitutions obtained by unification are indeed ordered. In Sec. 5, we show how these conditions on unified terms can be translated into conditions on programs and derivations.

2 The Type System

We will use the type system of [6]. First we recall some basic concepts [1]. When we refer to a *clause in a program*, we mean a copy of this clause whose variables are renamed apart from variables occurring in other objects in the context. A query is a sequence of atoms. A query is a sequence of atoms. A query Q' is a **resolvent of** a query Q and a clause $H \leftarrow \mathbf{B}$ if $Q = A_1, \ldots, A_m$, $Q' = (A_1, \ldots, A_{k-1}, \mathbf{B}, A_{k+1}, \ldots, A_m)\theta$, and H and A_k are unifiable with MGU θ. **Resolution steps** and **derivations** are defined in the usual way.

2.1 Type expressions

The set of types $\mathcal{T}$ is given by the term structure based on a finite set of **constructors** $\mathcal{K}$, where with each $K \in \mathcal{K}$ an arity $m \geq 0$ is associated (by writing K/m), and a denumerable set $\mathcal{U}$ of **parameters**. A **flat type** is a type of the form $K(u_1, \ldots, u_m)$, where $K \in \mathcal{K}$ and the u_i are distinct parameters. We write $\tau[\sigma]$ to denote that the type τ strictly contains the type σ as a subexpression. We write $\tau[u/\sigma]$ to denote the type obtained by replacing all the occurrences of u by σ in τ. The size of a type τ, defined as the number of occurrences of constructors and parameters in τ, is denoted by size(τ).

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. The **domain** of a type substitution is denoted by *dom*, the parameters in its **range** by *ran*. The set of parameters in a syntactic object o is denoted by $pars(o)$.

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 injection $\iota_{K,K'} : \{1, \ldots, m'\} \to \{1, \ldots, m\}$ is associated such that $\iota_{K,K''} = \iota_{K,K'} \circ \iota_{K',K''}$ whenever $K \leq K' \leq K''$. This order is extended to the **subtyping order** on types, denoted by $\leq$, as the least relation satisfying the rules in Table 1.

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

Table 1: The subtyping order on types

Proposition 1 *If $\sigma \leq \tau$ then size(σ) $\geq$ size(τ) .*

Proof: By structural induction on τ. □

Proposition 2 *If $\sigma \leq \tau$ then $\sigma\Theta \leq \tau\Theta$ for any type substitution Θ.*

Proof: By structural induction on τ. □

Proposition 3 *For each type τ, the set $\{\sigma \mid \tau \leq \sigma\}$ has a maximum, which is denoted by $Max(\tau)$.*

Proof: By structural induction on τ. □

Proposition 4 *For all types τ and σ, $Max(\tau[u/\sigma]) = Max(\tau)[u/Max(\sigma)]$.*

Proof: By structural induction on τ. □

Note that for Prop. 3, it is crucial that we require that $K/m \leq K'/m'$ implies $m \geq m'$, that is, as we move up in the subtype hierarchy, the arity of the type constructors does not increase. For example, if we allowed for $\texttt{Emptylist/0} \leq \texttt{List/1}$, then by Prop. 2, we would also have $\texttt{Emptylist} \leq \texttt{List}(\tau)$ for all types τ, and so, Prop. 3 would not hold. Note that the possibility of "forgetting" type parameters in subtype relations, as in $\texttt{List/1} \leq \texttt{Anylist/0}$, may provide solutions to inequalities of the form $\texttt{List}(\texttt{u}) \leq \texttt{u}$, e.g. $\texttt{u} = \texttt{Anylist}$. However, we have:

Proposition 5 *An inequality of the form $u \leq \tau[u]$ has no solution. An inequality of the form $\tau[u] \leq u$ has no solution if $u \in pars(Max(\tau))$.*

Proof: For any type σ, we have $\text{size}(\sigma) < \text{size}(\tau[\sigma])$, hence by Prop 1, $\sigma \not\leq \tau[\sigma]$, that is $u \leq \tau[u]$ has no solution.

For the second proposition, we prove its contrapositive. Suppose $\tau[u] \leq u$ has a solution, say $\tau[u/\sigma] \leq \sigma$. By definition of a maximum and Prop. 3, we have $\mathrm{M}ax(\sigma) = \mathrm{M}ax(\tau[u/\sigma])$. Hence by Prop. 4, $\mathrm{M}ax(\sigma) = \mathrm{M}ax(\tau)[u/\mathrm{M}ax(\sigma)]$. By the rules in Table 1, $u \neq \mathrm{M}ax(\tau)$. Therefore $u \notin \mathrm{p}ars(\mathrm{M}ax(\tau))$, since otherwise $\mathrm{M}ax(\sigma) = \mathrm{M}ax(\tau)[u/\mathrm{M}ax(\sigma)]$ would contain $\mathrm{M}ax(\sigma)$ as a strict subexpression which is impossible. □

2.2 Typed programs

We assume a denumerable set $\mathcal{V}$ of **variables**. The set of variables in a syntactic object o is denoted by $\mathrm{v}ars(o)$. We assume a finite set $\mathcal{F}$ (resp. $\mathcal{P}$) of **function** (resp. **predicate**) symbols, each with an arity and a **declared type** associated with it, such that: for each $f \in \mathcal{F}$, the declared type has the form $(\tau_1, \ldots, \tau_n, \tau)$, where n is the arity of f, $(\tau_1, \ldots, \tau_n) \in \mathcal{T}^n$, τ is a flat type and satisfies the *transparency condition* [10]: $\mathrm{p}ars(\tau_1, \ldots, \tau_n) \subseteq \mathrm{p}ars(\tau)$; for each $p \in \mathcal{P}$, the declared type has the form $(\tau_1, \ldots, \tau_n)$, where n is the arity of p and $(\tau_1, \ldots, \tau_n) \in \mathcal{T}^n$. The declared types are indicated by writing $f_{\tau_1 \ldots \tau_n \to \tau}$ and $p_{\tau_1 \ldots \tau_n}$, however it is assumed that the parameters in $\tau_1, \ldots, \tau_n, \tau$ are fresh for each occurrence of f or p. We assume that there is a special predicate symbol $=_{\mathtt{u},\mathtt{u}}$ where $\mathtt{u} \in \mathcal{U}$.

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. In examples we loosely follow Gödel syntax [9].

A **variable typing** (also called *type context* [6]) 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 restriction of a variable typing U to the variables

(Var)	$\{x:\tau,\ldots\}\vdash x:\tau$	
(Func)	$\frac{U\vdash t_i{:}\sigma_i\ \ \sigma_i\leq\tau_i\Theta\ \ (i\in\{1,\ldots,n\})}{U\vdash f_{\tau_1\ldots\tau_n\to\tau}(t_1,\ldots,t_n){:}\tau\Theta}$	Θ is a type substitution
(Atom)	$\frac{U\vdash t_i{:}\sigma_i\ \ \sigma_i\leq\tau_i\Theta\ \ (i\in\{1,\ldots,n\})}{U\vdash p_{\tau_1\ldots\tau_n}(t_1,\ldots,t_n)\,Atom}$	Θ is a type substitution
(Headatom)	$\frac{U\vdash t_i{:}\sigma_i\ \ \sigma_i\leq\tau_i\ \ (i\in\{1,\ldots,n\})}{U\vdash p_{\tau_1\ldots\tau_n}(t_1,\ldots,t_n)\,Headatom}$	
(Query)	$\frac{U\vdash A_1\ \ Atom\ \ \ldots\ \ U\vdash A_n\ \ Atom}{U\vdash A_1,\ldots,A_n\ \ Query}$	
(Clause)	$\frac{U\vdash Q\ \ Query\quad U\vdash A\ \ Headatom}{U\vdash A\leftarrow Q\ \ Clause}$	

Table 2: The type system.

in a syntactic object o is denoted as $U\restriction_o$. The type system, which defines terms, atoms etc. relative to a variable typing U, consists of the rules shown in Table 2.

If for an object, say a term t, we can deduce for some variable typing U and some type τ that $U\vdash t:\tau$, intuitively this term is *well-typed*. Otherwise the term is *ill-typed* (and likewise for atoms, etc.).

3 The Subtype and Instantiation Hierarchies

3.1 Modifying Variable Typings

Here we present the following result: if we can derive that some object is in the typed language using a variable typing U, then we can always modify U in three ways: extending its domain, instantiating the types, and making the types smaller. First we define:

Definition 1 *Let U, U' be variable typings. We say that U is* **smaller or equal** *U', denoted $U\leq U'$, if $U=\{x_1:\tau_1,\ldots,x_n:\tau_n\}$, $U'=\{x_1:\tau'_1,\ldots,x_n:\tau'_n\}$, and for all $i\in\{1,\ldots,n\}$, we have $\tau_i\leq\tau'_i$.*

The symbols $<$, $\geq$, $>$ are defined in the obvious way.

We use the notation $U'\supseteq\!\leq U$, which means that there exists a variable typing U'' such that $U'\supseteq U''$ and $U''\leq U$.

Lemma 6 *Let U, U' be variable typings and Θ a type substitution such that $U'\supseteq\!\leq U\Theta$. If $U\vdash t:\sigma$, then $U'\vdash t:\sigma'$ where $\sigma'\leq\sigma\Theta$. Moreover, if $U\vdash A\ Atom$ then $U'\vdash A\ Atom$, and if $U\vdash Q\ Query$ then $U'\vdash Q\ Query$.*

Proof: The proof of the first part is by structural induction. For the base case, suppose $t \in \mathcal{V}$. Then by Rule *(Var)*, $t : \sigma \in U$ and hence for some $\sigma' \leq \sigma\Theta$, we have $t : \sigma' \in U'$. Thus again by *(Var)*, $U' \vdash t : \sigma'$.

Now consider the case $t = f_{\tau_1 \ldots \tau_n \to \tau}(t_1, \ldots, t_n)$ where the inductive hypothesis holds for $t_1, \ldots, t_n$. By Rule *(Func)*, there exists a type substitution Θ' such that $\tau\Theta' = \sigma$, and $U \vdash t_i : \sigma_i$ where $\sigma_i \leq \tau_i\Theta'$ for each $i \in \{1, \ldots, n\}$. Thus by Prop. 2, $\sigma_i\Theta \leq \tau_i\Theta'\Theta$. By the inductive hypothesis, for all $i \in \{1, \ldots, n\}$ we have $U' \vdash t_i : \sigma'_i$ where $\sigma'_i \leq \sigma_i\Theta$, therefore by transitivity of $\leq$ we have $\sigma'_i \leq \tau_i\Theta'\Theta$ and hence by Rule *(Func)*, $U' \vdash t : \tau\Theta'\Theta$ (i.e. $U' \vdash t : \sigma\Theta$).

Now suppose $A = p_{\tau_1 \ldots \tau_n}(t_1, \ldots, t_n)$. By Rule *(Pred)*, there exists a type substitution Θ' such that $U \vdash t_i : \sigma_i$ where $\sigma_i \leq \tau_i\Theta'$ for each $i \in \{1, \ldots, n\}$. Thus by Prop. 2, $\sigma_i\Theta \leq \tau_i\Theta'\Theta$. By the first part of the statement, for all $i \in \{1, \ldots, n\}$ we have $U' \vdash t_i : \sigma'_i$ where $\sigma'_i \leq \sigma_i\Theta$, therefore by transitivity of $\leq$ we have $\sigma'_i \leq \tau_i\Theta'\Theta$ and hence by Rule *(Pred)*, $U' \vdash A$ *Atom*.

The final case for a query follows directly from Rule *(Query)*. □

3.2 Typed Substitutions

Typed substitutions are a fundamental concept for typed logic programs. Ignoring subtyping for the moment, a typed substitution replaces each variable with a term of the same type as the variable.

Definition 2 *If $U \vdash x_1 = t_1, \ldots, x_n = t_n$ Query where $x_1, \ldots, x_n$ are distinct variables and for each $i \in \{1, \ldots, n\}$, t_i is a term distinct from x_i, then $(\{x_1/t_1, \ldots, x_n/t_n\}, U)$ is a* **typed (term) substitution**. *The application of a substitution is defined in the usual way.*

To show that applying a typed substitution preserves "well-typedness" for systems with subtyping, we need a further condition. Given a typed substitution (θ, U), the type assigned to a variable x by U must be sufficiently big, so that it is compatible with the type of the term replaced for x by θ.

Example 5 *Consider again Ex. 3. As expected, assume that* `3, 6` *have declared type* `Int`*, and* `2.449` *has declared type* `Real`*, and* `Int` $\leq$ `Real`*. Given the variable typing* $U = \{$`x : Int, y : Int`$\}$*, we have* $U \vdash$ `x : Int`*,* $U \vdash$ `2.449 : Real`*, and hence* $U \vdash$ `x = 2.449` *Atom. So* $(\{$`x/2.449`$\}, U)$ *is a typed substitution. Now we have* $U \vdash$ `Fact(x, y)` *Atom, but* $U \nvdash$ `Fact(2.449, y)` *Atom.*

In the previous example, the type of `x` is too small to accommodate for instantiation to `2.449`. This motivates the following definition.

Definition 3 *A typed (term) substitution $(\{x_1/t_1, \ldots, x_n/t_n\}, U)$ is an* **ordered substitution** *if, for each $i \in \{1, \ldots, n\}$, where $x_i : \tau_i \in U$, there exists σ_i such that $U \vdash t_i : \sigma_i$ and $\sigma_i \leq \tau_i$.*

The following result states that expressions stay "well-typed" when ordered substitutions are applied [10, Lemma 1.4.2]. Moreover, the type of terms may become smaller.

Lemma 7 *Let (θ, U) be an ordered substitution. If $U \vdash t : \sigma$ then $U \vdash t\theta : \sigma'$ for some $\sigma' \leq \sigma$. Moreover, if $U \vdash A$ Atom then $U \vdash A\theta$ Atom, and likewise for queries and clauses.*

Proof: The proof of the first part is by structural induction. For the base case, suppose $t \in \mathcal{V}$. Then by Rule *(Var)*, $t : \sigma \in U$. If $t\theta = t$, there is nothing to show. If $t/s \in \theta$, then by definition of an ordered substitution, $U \vdash s : \sigma'$ and hence $U \vdash t\theta : \sigma'$ where $\sigma' \leq \sigma$.

Now consider the case $t = f_{\tau_1 \ldots \tau_n \to \tau}(t_1, \ldots, t_n)$ where the inductive hypothesis holds for $t_1, \ldots, t_n$. By Rule *(Func)*, there exists a type substitution Θ such that $\tau\Theta = \sigma$, and $U \vdash t_i : \sigma_i$ where $\sigma_i \leq \tau_i\Theta$ for each $i \in \{1, \ldots, n\}$. By the inductive hypothesis, for all $i \in \{1, \ldots, n\}$ we have $U \vdash t_i\theta : \sigma_i'$ where $\sigma_i' \leq \sigma_i$, and hence by transitivity of $\leq$ and Rule *(Func)*, $U \vdash t : \sigma$ (i.e. $\sigma' = \sigma$).

Now consider an atom $A = p_{\tau_1 \ldots \tau_n}(t_1, \ldots, t_n)$. By Rule *(Pred)*, there exists a type substitution Θ such that such that $U \vdash t_i : \sigma_i$ where $\sigma_i \leq \tau_i\Theta$ for each $i \in \{1, \ldots, n\}$. By the inductive hypothesis, for all $i \in \{1, \ldots, n\}$ we have $U \vdash t_i\theta : \sigma_i'$ where $\sigma_i' \leq \sigma_i$, and hence by Rule *(Atom)*, $U \vdash A\theta$ *Atom*. □

4 Conditions for Ensuring Ordered Substitutions

In this section, we show under which conditions it can be guaranteed that the substitutions applied in resolution steps are ordered substitutions.

4.1 Type Inequality Systems

The substitution of a resolution step is obtained by unifying two terms, say t_1 and t_2. In order for the substitution to be typed, it is necessary that we can derive $U \vdash t_1 = t_2$ *Atom* for some variable typing U. We will show that if U is, in a certain sense, maximal, then it is guaranteed that the typed substitution is ordered.

We need to formalise a straightforward concept, namely paths leading to subterms of a term.

Definition 4 *A term t has the subterm t in* **position** ϵ. *If $t = f(t_1, \ldots, t_n)$ and t_i has subterm s in position ζ, then t has subterm s in position $i.\zeta$.*

Example 6 *The term* `F(G(C),H(C))` *has subterm* `C` *in position* 1.1, *but also in position* 2.1. *The position* 2.1.1 *is undefined for this term.*

Let us use the notation $\_ \vdash t :\leq \sigma$ as a shorthand for: there exists a variable typing U and a type σ' such that $U \vdash t : \sigma'$ and $\sigma' \leq \sigma$. To derive $U \vdash t_1 = t_2$ *Atom*, it is clear that

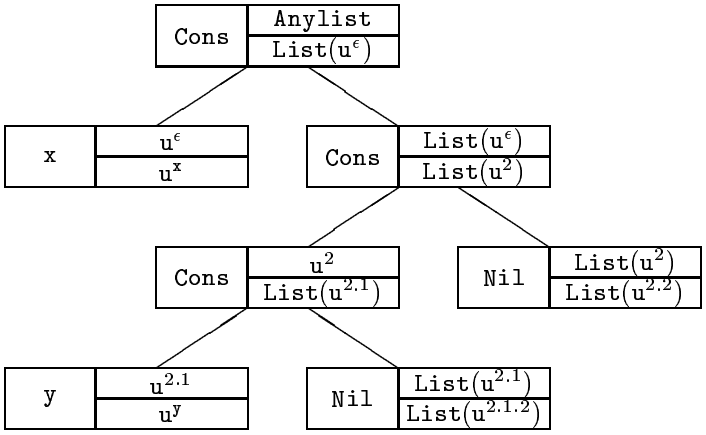

Figure 1: The term $[\mathtt{x},[\mathtt{y}]]$ and associated inequalities

the last step has the form

$$\frac{U \vdash t_1 : \tau_1 \quad U \vdash t_2 : \tau_2 \quad \tau_1 \leq \mathtt{u}\Theta \quad \tau_2 \leq \mathtt{u}\Theta}{U \vdash t_1 =_{\mathtt{u},\mathtt{u}} t_2 \ \mathit{Atom}}$$

That is to say, we use an *instance* $(\mathtt{u},\mathtt{u})\Theta$ of the declared type of the equality predicate, and the types of t_1 and t_2 are both less than or equal to $\mathtt{u}\Theta$. This motivates the following question: Given a term t such that $\_ \vdash t :\leq \sigma$, what are the maximal types of subterm positions (in particular positions filled with variables) of t with respect to σ?

Example 7 *Let* $\mathtt{List}/1$ *and* $\mathtt{Anylist}/0$ *be type constructors, where* $\mathtt{List}(\tau) \leq \mathtt{Anylist}$ *for all* τ*, and* List *is the usual list type, containing functions* $\mathtt{Nil}_{\rightarrow \mathtt{List(u)}}$ *and* $\mathtt{Cons}_{\mathtt{u},\mathtt{List(u)}\rightarrow\mathtt{List(u)}}$*. Consider the term* $[\mathtt{x},[\mathtt{y}]]$ *(in usual list notation) depicted in Figure 1, and let* $\sigma = \mathtt{Anylist}$*. Each functor in this term is introduced by an application of Rule* (Func)*. Consider for example the term* Nil *in position* 2.1.2*. Any type of it is necessarily an instance of* $\mathtt{List}(\mathtt{u}^{2.1.2})$*, its declared type.*[2] *In order to derive that* $\mathtt{Cons}(\mathtt{y},\mathtt{Nil})$ *is a typed term, this instance must be smaller (by the subtype order) than some instance of the second declared argument type of* Cons *in position* 2.1*, that is,* $\mathtt{List}(\mathtt{u}^{2.1})$*.*

For the term in position 2.1.1*, the variable* y*, a slightly different consideration applies. Its type is given by a variable typing. It is convenient to introduce a parameter* $\mathtt{u}^{\mathtt{y}}$ *for this variable and consider the type assigned to* y *by the variable typing as an instance of* $\mathtt{u}^{\mathtt{y}}$*.*

Analogous arguments can be applied to the other subterms, and so in order to derive that $[\mathtt{x},[\mathtt{y}]]$ *is a term of a type smaller than* Anylist*, we are looking for an instantiation of the*

[2] We use the positions as superscripts to parameters in order to obtain fresh copies of those for every application of a rule.

parameters such that for each box corresponding to a position, the type in the lower *subbox is smaller than the type of the* upper *subbox. That is, we are looking for type substitutions such that*

$$\begin{aligned}
\mathtt{u}^{\mathtt{y}}\Theta^{\mathtt{y}} &\leq \mathtt{u}^{2.1}\Theta^{2.1} \\
\mathtt{List}(\mathtt{u}^{2.1.2})\Theta^{2.1.2} &\leq \mathtt{List}(\mathtt{u}^{2.1})\Theta^{2.1} \\
\mathtt{List}(\mathtt{u}^{2.1})\Theta^{2.1} &\leq \mathtt{u}^{2}\Theta^{2} \\
\mathtt{List}(\mathtt{u}^{2.2})\Theta^{2.2} &\leq \mathtt{List}(\mathtt{u}^{2})\Theta^{2} \\
\mathtt{u}^{\mathtt{x}}\Theta^{\mathtt{x}} &\leq \mathtt{u}^{\epsilon}\Theta^{\epsilon} \\
\mathtt{List}(\mathtt{u}^{2})\Theta^{2} &\leq \mathtt{List}(\mathtt{u}^{\epsilon})\Theta^{\epsilon} \\
\mathtt{List}(\mathtt{u}^{\epsilon})\Theta^{\epsilon} &\leq \mathtt{Anylist}
\end{aligned}$$

For each position ζ, the type substitution Θ^{ζ} corresponds to the application of Rule (Func) *that introduces the functor in this position. For each variable x, the type substitution Θ^{x} defines a variable typing for x. Note however that since the parameters in each application are renamed, we can simply consider a single type substitution Θ which is the union of all Θ^{ζ}.*

We see that in order for $\_ \vdash t :\leq \sigma$ to hold, a solution to a certain *type inequality system* (set of inequalities between types) must exist.

Definition 5 *Let t be a term and σ a type such that $\_ \vdash t :\leq \sigma$. For each position ζ where t has a non-variable subterm, we denote the function in this position by $f^{\zeta}_{\tau_1^{\zeta},\ldots,\tau_{n^{\zeta}}^{\zeta}\to\tau^{\zeta}}$ (assuming that the parameters in $\tau_1^{\zeta},\ldots,\tau_{n^{\zeta}}^{\zeta},\tau^{\zeta}$ are fresh, say by indexing them with ζ). For each variable x occurring in t, we introduce a parameter u^x (so $u^x \notin pars(\sigma)$). The* **type inequality system** *of t and σ is*

$$\begin{aligned}
\mathcal{I}(t,\sigma) = \{\tau^{\epsilon} \leq \sigma\} \cup \quad & \{\tau^{\zeta.i} \leq \tau_i^{\zeta} \mid \textit{Position } \zeta.i \textit{ in } t \textit{ is non-variable}\} \cup \\
& \{u^x \leq \tau_i^{\zeta} \mid \textit{Position } \zeta.i \textit{ in } t \textit{ is variable } x\}.
\end{aligned}$$

A **solution** *of $\mathcal{I}(t,\sigma)$ is a type substitution Θ such that $dom(\Theta) \cap pars(\sigma) = \emptyset$ and for each $\tau \leq \tau' \in \mathcal{I}(t,\sigma)$, the inequality $\tau\Theta \leq \tau'\Theta$ holds.*

A solution Θ to $\mathcal{I}(t,\sigma)$ is **principal** *if for every solution $\tilde{\Theta}$ for $\mathcal{I}(t,\sigma)$, there exists a Θ' such that for each $\tau \leq \tau' \in \mathcal{I}(t,\sigma)$, we have $\tau\tilde{\Theta} \leq \tau\Theta\Theta'$ and $\tau'\tilde{\Theta} \leq \tau'\Theta\Theta'$.*

So for each subterm $f(\ldots,g(\ldots),\ldots)$ of t, the type inequality system says that the range type of g must be less than or equal to the ith argument type of f, where $g(\ldots)$ is in the ith position.

If Θ is a solution for $\mathcal{I}(t,\sigma)$, by Prop. 2, for every type substitution Θ', we have that $\Theta\Theta'$ is also a solution for $\mathcal{I}(t,\sigma)$. The following proposition follows from the rules in Table 2 and Def. 5.

Proposition 8 *Let t be a term and σ a type. If $U \vdash t :\leq \sigma$ for some variable typing U, then there exists a solution Θ for $\mathcal{I}(t,\sigma)$ (called the* **solution for $\mathcal{I}(t,\sigma)$ corresponding to** *U) such that for each subterm t' in position ζ in t, we have*

- *$U \vdash t' : \tau^{\zeta}\Theta$, if t' is non-variable,*
- *$U \vdash t' : u^{x}\Theta$, if $t' = x$ and $x \in \mathcal{V}$.*

The following lemma says that if t is an instance of s, then a solution to the type inequality system for t is also a solution for the type inequality system for s.

Lemma 9 *Consider two terms s and t such that $vars(s) \cap vars(t) = \emptyset$ and $s\theta = t$ for some idempotent θ, and suppose that $\_ \vdash s :\leq \sigma$ and $\_ \vdash t :\leq \sigma$. If Θ_t is a solution of $\mathcal{I}(t,\sigma)$, then*

$$\tilde{\Theta}_s = \Theta_t \cup \{u^{x}/\tau^{\zeta}\Theta_t \mid s \text{ has } x \text{ in position } \zeta\}$$

is a solution of $\mathcal{I}(s,\sigma)$.

Proof: For the inequality $\tau^{\epsilon} \leq \sigma$ and for each $\tau^{\zeta.i} \leq \tau_i^{\zeta} \in \mathcal{I}(s,\sigma)$ such that s has a non-variable term in $\zeta.i$, we have that the same inequality is also in $\mathcal{I}(t,\sigma)$, and so Θ_t, and consequently $\tilde{\Theta}_s$, is a solution for it.

For each $u^{x} \leq \tau_i^{\zeta} \in \mathcal{I}(s,\sigma)$ (i.e., s has variable x in $\zeta.i$), we have a corresponding inequality $\tau^{\zeta.i} \leq \tau_i^{\zeta}$ in $\mathcal{I}(t,\sigma)$. Since $\tau^{\zeta.i}\Theta_t \leq \tau_i^{\zeta}\Theta_t$ is true and $\tau^{\zeta.i}\Theta_t = u^{x}\tilde{\Theta}_s$, it follows that $u^{x}\tilde{\Theta}_s \leq \tau_i^{\zeta}\tilde{\Theta}_s$ is true. □

Example 8 *Let $s = [\mathtt{x},\mathtt{z}]$ and $t = [\mathtt{x},[\mathtt{y}]]$ and $\sigma = \mathtt{Anylist}$. A solution for $\mathcal{I}(t,\sigma)$ is*

$$\Theta_t = \{\mathtt{u}^{\mathtt{y}}/\mathtt{u}^{2.1}, \mathtt{u}^{2.1.2}/\mathtt{u}^{2.1}, \mathtt{u}^{\epsilon}/\mathtt{Anylist}, \mathtt{u}^{2.2}/\mathtt{Anylist}, \mathtt{u}^{\mathtt{x}}/\mathtt{Anylist}, \mathtt{u}^{2}/\mathtt{Anylist}\}$$

(in Ex. 9 it will be shown how this solution is obtained). Now

$$\begin{aligned}\mathcal{I}(s,\sigma) = \quad & \{\mathtt{u}^{\mathtt{z}} \leq \mathtt{u}^{2}, \mathtt{List}(\mathtt{u}^{2.2}) \leq \mathtt{List}(\mathtt{u}^{2}), \mathtt{u}^{\mathtt{x}} \leq \mathtt{u}^{\epsilon}, \mathtt{List}(\mathtt{u}^{2}) \leq \mathtt{List}(\mathtt{u}^{\epsilon}),\\ & \mathtt{List}(\mathtt{u}^{\epsilon}) \leq \mathtt{Anylist}\}.\end{aligned}$$

By Lemma 9, $\tilde{\Theta}_s = \Theta_t \cup \{\mathtt{u}^{\mathtt{z}}/\mathtt{List}(\mathtt{u}^{2.1})\}$ is a solution for $\mathcal{I}(s,\sigma)$.

In the next subsection, we present an algorithm, based on [6], which computes a principal solution to a type inequality system, provided t is linear. In Subsec. 4.3, our interest in principal solutions will become clear.

4.2 Computing a Principal Solution

The algorithm transforms the inequality system, thereby computing bindings to parameters which constitute the solution. It is convenient to consider system of both *inequalities*, and *equations* of the form $u = \tau$. The inequalities represent the current type inequality system, and the equations represent the substitution accumulated so far. We use $\leqq$ for $\leq$ or $=$.

Definition 6 *A system is* **left-linear** *if each parameter occurs at most once on the left hand side of an equation/inequality. A system is* **acyclic** *if it does not have a subset* $\{\rho_1 \leqq \sigma_1, ..., \rho_n \leqq \sigma_n\}$ *with* $pars(\sigma_i) \cap pars(\rho_{i+1}) \neq \emptyset$ *for all* $1 \leq i \leq n-1$, *and* $pars(\sigma_n) \cap pars(\rho_1) \neq \emptyset$.

Proposition 10 *If* t *is a linear term, then any inequality system* $\mathcal{I}(t, \sigma)$ *is acyclic and left-linear.*

Proof: Consider a non-variable position ζ in t. There is exactly one inequality in $\mathcal{I}(t, \sigma)$ with τ^ζ as left-hand side. Moreover, τ^ζ is a flat type (declared range type of a function), thus linear, and (because of indexing the parameters in τ^ζ by ζ) has no parameters in common with any other left-hand side of $\mathcal{I}(t, \sigma)$.

Now consider a position ζ where t has the variable x. Because of the linearity of t, there is exactly one inequality in $\mathcal{I}(t, \sigma)$ with u^x as left-hand side.

Let $\{\rho_1 \leq \sigma_1, ..., \rho_n \leq \sigma_n\}$ be a subset of $\mathcal{I}(t, \sigma)$ with $pars(\sigma_i) \cap pars(\rho_{i+1}) \neq \emptyset$ for all $1 \leq i \leq n-1$. By the definition of $\mathcal{I}(t, \sigma)$, if $\rho_1 = u^x$ for some variable x or if $\sigma_n = \sigma$, then $pars(\sigma_n) \cap pars(\rho_1) = \emptyset$. If however $\rho_1 \leq \sigma_1$ is $\tau^{\zeta.j} \leq \tau_j^\zeta$ and $\rho_n \leq \sigma_n$ is $\tau^{\xi.l} \leq \tau_l^\xi$ for some positions $\zeta.j$ and $\xi.l$, then ξ is a prefix of ζ, and so, since we use the positions to index the parameters, $pars(\sigma_n) \cap pars(\rho_1) = \emptyset$. □

Example 7 makes it also intuitively clear that assuming linearity of t is crucial for the above proposition.

We now give the algorithm for computing principal solutions as a set of rules for simplifying a set of inequalities and equations. A *solved form* is a system I containing only equations of the form $I = \{u_1 = \tau_1, ..., u_n = \tau_n\}$ where the parameters u_i are all different and have no other occurrence in I. Note that the substitution $\{u_1/\tau_1, ..., u_n/\tau_n\}$ associated to a solved form is trivially a principal solution.

Definition 7 *Given a type inequality system* $\mathcal{I}(t, \sigma)$, *where* t *is linear, the* **type inequality algorithm** *applies the following simplification rules:*

(1) $\{K(\tau_1, ..., \tau_m) \leq K'(\tau'_1, ..., \tau'_n)\} \cup I \longrightarrow \{\tau_{\iota(i)} \leq \tau'_i\}_{i=1,..,n} \cup I$
if $K \leq K'$ *and* $\iota = \iota_{K,K'}$

(2) $\{u \leq u\} \cup I \longrightarrow I$

(3) $\{u \leq \tau\} \cup I \longrightarrow \{u = \tau\} \cup I[u/\tau]$
if $\tau \neq u$, $u \notin vars(\tau)$.

(4) $\{\tau \leq u\} \cup I \longrightarrow \{u = Max(\tau)\} \cup I[u/Max(\tau)]$
if $\tau \notin V$, $u \notin vars(Max(\tau))$ *and* $u \notin vars(l)$ *for any* $l \leq r \in \Sigma$.

Intuitively, left-linearity of $\mathcal{I}(t, \sigma)$ is crucial because it renders the binding of a parameter (point (3)) unique.

Example 9 *Consider* $\mathcal{I}([x, [y]], \texttt{Anylist})$ *as in Ex. 7. The initial* I *is given by the inequality system in the example (where the type substitutions are removed). Applying (1) three times,*

we have

$$I = \{\mathtt{u}^{\mathtt{y}} \leq \mathtt{u}^{2.1}, \mathtt{u}^{2.1.2} \leq \mathtt{u}^{2.1}, \mathtt{List}(\mathtt{u}^{2.1}) \leq \mathtt{u}^{2}, \mathtt{u}^{2.2} \leq \mathtt{u}^{2}, \mathtt{u}^{\mathtt{x}} \leq \mathtt{u}^{\epsilon}, \mathtt{u}^{2} \leq \mathtt{u}^{\epsilon}\}.$$

Applying (3) five times, we have

$$I = \{\mathtt{u}^{\mathtt{y}} = \mathtt{u}^{2.1}, \mathtt{u}^{2.1.2} = \mathtt{u}^{2.1}, \mathtt{List}(\mathtt{u}^{2.1}) \leq \mathtt{u}^{\epsilon}, \mathtt{u}^{2.2} = \mathtt{u}^{\epsilon}, \mathtt{u}^{\mathtt{x}} = \mathtt{u}^{\epsilon}, \mathtt{u}^{2} = \mathtt{u}^{\epsilon}\}.$$

Applying (4) once, we have

$$I = \{\mathtt{u}^{\mathtt{y}} = \mathtt{u}^{2.1}, \mathtt{u}^{2.1.2} = \mathtt{u}^{2.1}, \mathtt{u}^{\epsilon} = \mathtt{Anylist}, \mathtt{u}^{2.2} = \mathtt{Anylist}, \mathtt{u}^{\mathtt{x}} = \mathtt{Anylist}, \mathtt{u}^{2} = \mathtt{Anylist}\}.$$

Proposition 11 *Given a type inequality system $\mathcal{I}(t, \sigma)$, where t is linear, the type inequality algorithm terminates with either a solved form, in which case the associated substitution is a principal solution, or a non-solved form in which case the system has no solution.*

Proof: Termination is proved by remarking that the sum of the sizes of the terms in left-hand sides of inequalities strictly decreases after each application of a rule.

By Prop. 10 the initial system is left-linear and acyclic, and one can easily check that each rule preserves the left-linearity as well as the acyclicity of the system.

Furthermore each rule preserves the satisfiability of the system and its principal solution if one exists. Indeed rules (1) and (2) preserve all solutions by definition of the subtyping order. Rule (3) replaces a parameter u by its upper bound τ. As the system is left-linear this computes the principal solution for u, and thus preserves the principal solution of the system if one exists. Rule (4) replaces a parameter u 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 principal solution for u and thus preserves the principal solution of the system if it exists.

Now consider a normal form I' for I. If I' contains a non variable pair $\tau \leq \tau'$ irreducible by (1), then I', and hence I, have no solution. Similarly I' has no solution if it contains an inequality $u \leq \tau$ with $u \in vars(\tau)$ or an inequality $\tau \leq u$ with $u \in vars(\mathrm{M}ax(\tau))$ (Prop. 5). In the other cases, by irreducibility and acyclicity, I' contains no inequality, hence I' is in solved form and the substitution associated to I' is a principal solution for I. □

The next lemma says that principality is stable under instantiation of types.

Lemma 12 *Let $\mathcal{I}(t, \sigma)$ be a type inequality system, where t is linear, and Θ' a type substitution such that $dom(\Theta') \subseteq pars(\sigma)$ and $ran(\Theta') \cap pars(\mathcal{I}(t, \sigma)) = \emptyset$. If Θ is a principal solution of $\mathcal{I}(t, \sigma)$, then $\Theta\Theta'$ is a principal solution of $\mathcal{I}(t, \sigma\Theta')$.*

Proof: Suppose Θ is computed by the algorithm of Def. 7, and that $I_1, \ldots, I_m$ is the sequence of systems of this computation, i.e. Θ is equal to I_m viewed as a substitution. By Def. 5, $dom(\Theta) \cap pars(\sigma) = \emptyset$. In particular, this means that no system I_j ($j \in \{1, \ldots, m\}$) contains an inequality $\tau \leq u$ where $u \in pars(\sigma)$ and τ is not a parameter. It is easy to

see that $I_1\Theta', \ldots, I_m\Theta'$ is a computation of the algorithm for $\mathcal{I}(t, \sigma\Theta')$, and hence $\Theta\Theta'$ (i.e. $I_m\Theta'$ viewed as a substitution) is a principal solution of $\mathcal{I}(t, \sigma\Theta')$. □

4.3 Principal Variable Typings

The existence of a principal solution Θ of a type inequality system $\mathcal{I}(t, \sigma)$ and Prop. 8 motivate defining the variable typing U such that Θ is exactly the solution of $\mathcal{I}(t, \sigma)$ corresponding to U.

Definition 8 *Let $\_ \vdash t :\leq \sigma$, and Θ be a principal solution of $\mathcal{I}(t, \sigma)$. A variable typing U is* **principal** *for t and σ if $U \supseteq \{x : u^x\Theta \mid x \in vars(t)\}$.*

By the definition of a principal solution of $\mathcal{I}(t, \sigma)$ and Prop. 8, if U is a principal variable typing for t and σ, then for any U' such that $U'(x) > U(x)$ for some $x \in vars(t)$, we have $U' \not\vdash t :\leq \sigma$. (since U' corresponds to an instantiation of the u^x's that is not a solution of $\mathcal{I}(t, \sigma)$). The following is a corollary of Lemma 12.

Corollary 13 *If U is a principal variable typing for t and σ, then $U\Theta$ is a principal variable typing for t and $\sigma\Theta$.*

The following key lemma states conditions under which a substitution obtained by unifying two terms is indeed ordered.

Lemma 14 *Let s and t be terms, s linear, such that $U \vdash s :\leq \rho$, $U \vdash t :\leq \rho$, and there exists a substitution θ such that $s\theta = t$. Suppose θ is a minimal matcher, i.e. $dom(\theta) \subseteq vars(s)$. Suppose U is principal for s and ρ. Then there exists a type substitution Θ such that for $U' = U\Theta\restriction_{\mathcal{V}ars(s)} \cup U\restriction_{\mathcal{V}\setminus\mathcal{V}ars(s)}$, we have that (θ, U') is an ordered substitution.*

Proof: Since θ is a minimal matcher, we have

$$\theta = \{x/t' \mid \exists\zeta.x \text{ is subterm of } s \text{ in } \zeta,\ t' \text{ is subterm of } t \text{ in } \zeta\}.$$

It remains to be shown that there exists a type substitution Θ such that (θ, U') as defined above is an ordered substitution. Let Θ_s be the solution of $\mathcal{I}(s, \rho)$ corresponding to U, and Θ_t be the solution of $\mathcal{I}(t, \rho)$ corresponding to U (see Prop. 8). Note that since U is principal for s and ρ, Θ_s is a principal solution. By Lemma 9, $\tilde{\Theta}_s = \Theta_t \cup \{u^x/\tau^\zeta\Theta_t \mid s \text{ has variable } x \text{ in position } \zeta\}$ is a solution of $\mathcal{I}(s, \rho)$, and moreover, since Θ_s is a principal solution of $\mathcal{I}(s, \rho)$, there exists a type substitution Θ such that for each τ occurring (on a left-hand side or right-hand side) in $\mathcal{I}(s, \rho)$,

$$\tau\tilde{\Theta}_s \leq \tau\Theta_s\Theta. \tag{1}$$

In particular, let x be a variable occurring in s in position ζ, and let t' be the subterm of t in position ζ. By Prop. 8, $U' \vdash t' : \tau^\zeta\Theta_t$. By Def. 8, $x/u^x\Theta_s \in U$, and so by Rule *(Var)*,

$U' \vdash x : u^x\Theta_s\Theta$. Since by definition of $\tilde{\Theta}_s$, $\tau^\zeta\Theta_t = u^x\tilde{\Theta}_s$, we also have $U' \vdash t' : u^x\tilde{\Theta}_s$, and so by (1), the condition in Def. 3 is fulfilled. Since the choice of x was arbitrary, the result follows. □

Example 10 *Consider the term vectors (since Lemma 14 generalises in the obvious way to term vectors)* $s = (3, \mathtt{x})$ *and* $t = (3, 6)$*, let* $\rho = (\mathtt{Int}, \mathtt{Int})$ *and* $U_s = \{\mathtt{x} : \mathtt{Int}\}$*,* $U_t = \emptyset$ *(see Ex. 3). Note that* U_s *is principal for* s *and* ρ*, and so* $(\{\mathtt{x}/6\}, U_s \cup U_t)$ *is an ordered substitution (*Θ *is empty).*

In contrast, let $s = (6, \mathtt{x})$ *and* $t = (6, 2.449)$*, let* $\rho = (\mathtt{Real}, \mathtt{Real})$ *and* $U_s = \{\mathtt{x} : \mathtt{Int}\}$*,* $U_t = \emptyset$*. Then* U_s *is not principal for* s *and* ρ *(the principal variable typing would be* $\{\mathtt{x}/\mathtt{Real}\}$*), and indeed, there exists no* Θ *such that* $(\{\mathtt{x}/2.449\}, U_s\Theta \cup U_t)$ *is an ordered substitution.*

5 Nicely Typed Programs

In the previous section, we have seen that *matching*, *linearity*, and *principal variable typings* are crucial to ensure that unification yields ordered substitutions (see Lemma 14). In this section, we define three corresponding conditions on programs and the execution model.

We will generalise concepts defined for *terms* in the previous section, to term *vectors*. In particular, we consider principal variable typings for a term vector $\bar{t}$ and a type vector $\bar{\sigma}$ (Def. 8). Also, Lemma 14 generalises to term vectors in the obvious way (conceptually, one could think of introducing special functors into the typed language so that any vector can be represented as an ordinary term).

First, we define modes, which are a common concept used for verification [1]. For a predicate p/n, a **mode** is an atom $p(m_1, \ldots, m_n)$, where $m_i \in \{I, O\}$ for $i \in \{1, \ldots, n\}$. Positions with I are called **input positions**, and positions with O are called **output positions** of p. We assume that a fixed mode is associated with each predicate in a program. To simplify the notation, an atom written as $p(\bar{s}, \bar{t})$ means: $\bar{s}$ is the vector of terms filling the input positions, and $\bar{t}$ is the vector of terms filling the output positions.

Definition 9 *Consider a derivation step where* $p(\bar{s}, \bar{t})$ *is the selected atom and* $p(\bar{w}, \bar{v})$ *is the renamed apart clause head. The equation* $p(\bar{s}, \bar{t}) = p(\bar{w}, \bar{v})$ *is* **solvable by moded unification** *if there exist substitutions* θ_1*,* θ_2 *such that* $\bar{w}\theta_1 = \bar{s}$ *and* $vars(\bar{t}\theta_1) \cap vars(\bar{v}\theta_1) = \emptyset$ *and* $\bar{t}\theta_1\theta_2 = \bar{v}\theta_1$*.*

A derivation where all unifications are solvable by moded unification is a **moded** *derivation.*

Moded unification is a special case of *double matching*. How moded derivations are ensured is not our problem here, and we refer to [2]. Note that the requirement of moded derivations is stronger than *input-consuming derivations* [17] where it is only required that the MGU does not bind $\bar{s}$.

Definition 10 *A query $Q = p_1(\bar{s}_1, \bar{t}_1), \ldots, p_n(\bar{s}_n, \bar{t}_n)$ is* **nicely moded** *if $\bar{t}_1, \ldots, \bar{t}_n$ is a linear vector of terms and for all $i \in \{1, \ldots, n\}$*

$$vars(\bar{s}_i) \cap \bigcup_{j=i}^{n} vars(\bar{t}_j) = \emptyset. \tag{2}$$

The clause $C = p(\bar{t}_0, \bar{s}_{n+1}) \leftarrow Q$ is **nicely moded** *if Q is nicely moded and*

$$vars(\bar{t}_0) \cap \bigcup_{j=1}^{n} vars(\bar{t}_j) = \emptyset. \tag{3}$$

A program is **nicely moded** *if all of its clauses are nicely moded.*

An atom $p(\bar{s}, \bar{t})$ is **input-linear** *if $\bar{s}$ is linear,* **output-linear** *if $\bar{t}$ is linear.*

Definition 11 *Let*

$$C = p_{\bar{\tau}_0, \bar{\sigma}_{n+1}}(\bar{t}_0, \bar{s}_{n+1}) \leftarrow p^1_{\bar{\sigma}_1, \bar{\tau}_1}(\bar{s}_1, \bar{t}_1), \ldots, p^n_{\bar{\sigma}_n, \bar{\tau}_n}(\bar{s}_n, \bar{t}_n)$$

be a clause. If C is nicely moded, $\bar{t}_0$ is input-linear, and there exists a variable typing U such that $U \vdash C$ Clause, and for each $i \in \{0, \ldots, n\}$, U is principal for $\bar{t}_i$ and $\bar{\tau}'_i$, where $\bar{\tau}'_i$ is the instance of $\bar{\tau}_i$ used for deriving $U \vdash C$ Clause, then we say that C is **nicely typed**.

A query $U_Q : Q$ is **nicely typed** *if the clause* `Go` $\leftarrow Q$ *is nicely typed. A program is* **nicely typed** *if all of its clauses are nicely typed.*

We can now state the main result.

Theorem 15 (Subject reduction) *Let C and Q be a nicely typed clause and query. If Q' is a resolvent of C and Q where the unification of the selected atom and the clause head is solvable by moded unification, then Q' is nicely typed.*

Proof: By [3, Lemma 11], Q' is nicely moded. Let U_C and U_Q be the variable typings used to type C and Q, respectively (in the sense of Def. 11).

Let $p_{\bar{\sigma},\bar{\tau}}(\bar{s}, \bar{t}) \in Q$ be the selected atom and $C = p(\bar{w}, \bar{v}) \leftarrow \mathbf{B}$. By Rule *(Headatom)*, $U_C \vdash (\bar{w}, \bar{v}) :\leq (\bar{\sigma}, \bar{\tau})$. Moreover, $U_Q \vdash (\bar{s}, \bar{t}) :\leq (\bar{\sigma}, \bar{\tau})\Theta$ for some type substitution Θ. Let $U = U_Q \cup U_C\Theta$. Note that since $\mathrm{v}vars(C) \cap \mathrm{v}vars(Q) = \emptyset$, U is a variable typing. By Lemma 6, we have $U \vdash \mathbf{B}$ *Query* and $U \vdash p(\bar{w}, \bar{v})$ *Atom* (but not necessarily $U \vdash C$ *Clause*, because of the special rule for head atoms) and in particular, $U \vdash (\bar{w}, \bar{v}) :\leq (\bar{\sigma}, \bar{\tau})\Theta$.

Since C is nicely typed, it follows by Cor. 13 that U is principal for $\bar{w}$ and $\bar{\sigma}\Theta$. Moreover by assumption of moded unification, there exists a substitution θ_1 such that $\bar{w}\theta_1 = \bar{s}$. We assume θ_1 is *minimal*, i.e. $dom(\theta_1) \subseteq \mathrm{v}vars(\bar{w})$. By Lemma 14, there exists a variable typing U' such that (θ_1, U') is an ordered substitution, and moreover $U'\restriction_{\mathcal{V}\setminus \mathrm{v}vars(\bar{w})} = U\restriction_{\mathcal{V}\setminus \mathrm{v}vars(\bar{w})}$. Therefore by Lemma 7, $U' \vdash \mathbf{B}\theta_1$ *Query* and $U' \vdash Q\theta_1$ *Query*. In particular, $U' \vdash \bar{v}\theta_1 :\leq \bar{\tau}\Theta$.

Now since Q is nicely typed and $U'\restriction_{\mathrm{v}vars(Q)} = U_Q\restriction_{\mathrm{v}vars(Q)}$, U' is principal for $\bar{t}$ and $\bar{\tau}\Theta$. Moreover by assumption of moded unification, there exists a minimal substitution θ_2 such

that $\bar{t}\theta_2 = \bar{v}\theta_1$. By Lemma 14, there exists a variable typing U'' such that (θ_2, U'') is an ordered substitution, and moreover $U''\restriction_{\mathcal{V}\setminus \mathrm{V}ars(\bar{t})} = U'\restriction_{\mathcal{V}\setminus \mathrm{V}ars(\bar{t})}$. Therefore by Lemma 7, $U'' \vdash \mathbf{B}\theta_1\theta_2$ *Query* and $U'' \vdash Q\theta_1\theta_2$ *Query*. Hence by Rule *(Query)*, $U'' \vdash Q'$ *Query*. Finally, $U''\restriction_{\mathcal{V}\setminus(\mathrm{V}ars(\bar{w})\cup \mathrm{V}ars(\bar{t}))} = U\restriction_{\mathcal{V}\setminus(\mathrm{V}ars(\bar{w})\cup \mathrm{V}ars(\bar{t}))}$ and so by the linearity conditions and (2) in Def. 10, it follows that

- if $\bar{t}'$ is an output argument vector in Q, other than $\bar{t}$, and $\bar{\tau}'$ is the instance of the declared type of $\bar{t}'$ used for deriving $U_Q \vdash Q$ *Query*, then $U''\restriction_{\mathrm{V}ars(\bar{t}')} = U_Q\restriction_{\mathrm{V}ars(\bar{t}')}$, $\theta_1\theta_2\restriction_{\mathrm{V}ars(\bar{t}')} = \emptyset$, and hence U'' is a principal variable typing for $\bar{t}'\theta_1\theta_2$ and $\bar{\tau}'$,
- analogously, if $\bar{t}'$ is an output argument vector in $\mathbf{B}$, and $\bar{\tau}'$ is the instance of the declared type of $\bar{t}'$ used for deriving $U_C \vdash C$ *Clause*, then $U''\restriction_{\mathrm{V}ars(\bar{t}')} = U_C\Theta\restriction_{\mathrm{V}ars(\bar{t}')}$, $\theta_1\theta_2\restriction_{\mathrm{V}ars(\bar{t}')} = \emptyset$, and hence, by Cor. 13, U'' is a principal variable typing for $\bar{t}'\theta_1\theta_2$ and $\bar{\tau}'\Theta$.

So we have shown that Q' is nicely moded, U'' is a variable typing such that $U'' \vdash Q'$ *Query*, and the principality requirement on U'' is fulfilled. Thus Q' is a nicely-typed query. □

To conclude, we state subject reduction as a property of an entire derivation.

Corollary 16 *Any derivation for a nicely typed program P and a nicely typed query Q contains only nicely typed queries.*

Example 11 *Consider again Ex. 3. The program is nicely typed, where the declared types are given in that example, and the first position of each predicate is input, and the second output. Both queries are nicely moded. The first query is also nicely typed, whereas the second is not (see also Ex. 10). For the first query, we have subject reduction, for the second we do not have subject reduction.*

6 Discussion

In this paper, we have proposed criteria for ensuring subject reduction for typed logic programs with subtyping under the untyped execution model. Our starting point was a comparison between functional and logic programming: In functional programs, there is a clear notion of dataflow, whereas in logic programming, there is no such notion a priori, and arguments can serve as input arguments and output arguments. This difference is the source of the difficulty of ensuring subject reduction for logic programs. We thus coped with the problem by introducing modes into a program, so that there is a fixed direction of dataflow.

To understand better the numerous conditions for ensuring subject reduction, it is useful to distinguish roughly between four kinds of conditions: (1) "basic" type conditions on the program (Sec. 2), (2) conditions on the execution model (Def. 9), (3) mode conditions on the program (Def. 10), (4) "additional" type conditions on the program (Def. 11). We will refer to this distinction below.

Concerning (1), our notion of subtyping deserves discussion. Approaches differ with respect to conditions on the *arities* of type constructors for which there is a subtype relation. Beierle [4] assumes that the (constructor) order is only defined for type constants, i.e. constructors of arity 0. Thus we could have $\texttt{Int} \leq \texttt{Real}$, and so by extension $\texttt{List(Int)} \leq \texttt{List(Real)}$, but not $\texttt{List(Int)} \leq \texttt{Tree(Real)}$. Many authors assume that only constructors of the same arity are comparable. Thus we could have $\texttt{List(Int)} \leq \texttt{Tree(Real)}$, but not $\texttt{List(Int)} \leq \texttt{Anylist}$. We assume, as [6], that if $K \leq K'$, then the arity of K' must not be greater that the arity of K. Other authors have been vague about justifying their choice, suggesting that one could easily consider modifications. We think that this choice is crucial for the existence of principal types. In particular, if one allowed for comparing constructors of arbitrary arities, then the existence of a maximum above any type (Prop. 3) would not be guaranteed.

The PAN type system has been proposed in [12] and described in detail in [20]. It is argued there that comparisons between constructors of arbitrary arity should be allowed in principle, and that the subtype relation should be defined by a relation between argument positions of constructors, similar to our ι (see Table 1). However, we believe that this construction is flawed: It is claimed that under some simple conditions, the subtyping relation implies a *subset* relation between the sets of terms represented by the types, while in fact, their formalism would allow for $\texttt{NonemptyList(Int)} \leq \texttt{List(String)}$ (where those types are declared as expected) even though the set of non-empty integer lists is not a subset of the set of string lists. They define *extensional type bases*, essentially meaning typed languages where also the converse holds, i.e., the subtyping relation exactly corresponds to the subtype relation. Nothing is said about decidability of this property, although the formalism heavily relies on this concept. Furthermore the very example given in order to motivate the need for such a general subtyping relation is *not* extensional.

Technically, what is crucial for subject reduction is that substitutions are *ordered*: each variable is replaced with a term of a smaller type. In Section 4, we give conditions under which unification of two terms yields an ordered substitution: the unification is a matching, the term that is being instantiated is linear and is typed using a *principal* variable typing. The linearity requirement ensures that a principle variable typing exists and can be computed (Subsec. 4.2). The conditions guarantee that the type of each variable x that is being bound to t can be instantiated so that it is greater than the type of t.

In Sec. 5, we show how those conditions on the level of a single unification translate to conditions on the program and the execution model (points 2–4 above). We introduce modes and assume that programs are executed using moded unification (2). This might be explicitly enforced by the compiler by modifying the unification procedure (which would have to yield a runtime error if the atoms are unifiable but violating the mode requirement). Alternatively, it can be verified statically that a program will be executed using moded unification. In particular, nicely moded programs are very amenable to such verification [2]. Moded unification can actually be very beneficial for efficiency, as witnessed by the language Mercury [19]. Apart from that, (3) nicely-modedness states the linearity of the terms being instantiated in a unification. Nicely-modedness is designed so that it is persistent under

resolution steps, provided clause heads are input-linear. Finally, (4) nicely-typedness states that the instantiated terms must be typed using a principal variable typing.

Nicely-modedness has been widely used for verification purposes (e.g. [2]). In particular, the linearity condition on the output arguments is natural: it states that every piece of data has at most one producer. Input-linearity of clause heads however can sometimes be a demanding condition, since it rules out equality tests between input arguments [16, Section 10.2].

Note that introducing modes into logic programming does not mean that logic programs become functional. The aspect of non-determinacy (possibility of computing several solutions for a query) remains.

Even though our result on subject reduction means that it is possible to execute programs without maintaining the types at runtime, there are circumstances where keeping the types at runtime is desirable, for example for memory management or for some extra logical operations like printing, or in higher-order logic programming where the existence and shape of unifiers depends on the types [14].

There is a relationship between our notion of subtyping and *transparency* (see Subsec. 2.2). It has been observed in [10] that transparency is essential for substitutions obtained from unification to be typed. Transparency ensures that two terms of the same type have identical types in all corresponding subterms, e.g. if [1] and [x] are both of type `List(Int)`, we are sure that `x` is of type `Int`. Now in a certain way, allowing for a subtyping relation that "forgets" parameters undermines transparency. For example, we can derive $\{\mathtt{x} : \mathtt{String}\} \vdash [\mathtt{x}] = [1]$ *Atom*, since $\mathtt{List(String)} \leq \mathtt{Anylist}$ and $\mathtt{List(Int)} \leq \mathtt{Anylist}$, even though `Int` and `String` are incomparable. We compensate for this by requiring principal variable typings. The principal variable typing for [x] and `Anylist` contains $\{\mathtt{x} : \mathtt{u}^{\mathtt{x}}\}$, and so $\mathtt{u}^{\mathtt{x}}$ can be instantiated to `Int`. However, our intuition is that whenever this phenomenon ("forgetting" parameters) occurs, requiring principal variable typings is very demanding; but then, if variable typings are not principal, subject reduction is likely to be violated. As a topic for future work, we want to substantiate this intuition by studying examples. In particular, we want to see if the conditions (in particular, assuming principal variable typings) are *too* demanding, in the sense that there are interesting programs that satisfy subject reduction under more general assumptions.

Acknowledgements

We thank Erik Poll and François Pottier for interesting discussions on type systems for functional programming, and the reviewers of the FSTTCS version of this article for their valuable comments. Jan-Georg Smaus was supported by an ERCIM fellowship.

References

[1] K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[2] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science*, LNCS, pages 1–19. Springer-Verlag, 1993.

[3] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V. S. Alagar and M. Nivat, editors, *Proceedings of AMAST'95*, LNCS, pages 66–90. Springer-Verlag, 1995. Invited Lecture.

[4] C. Beierle. Type inferencing for polymorphic order-sorted logic programs. In L. Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 765–779. MIT Press, 1995.

[5] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, LNCS, pages 79–93. Springer-Verlag, 1988.

[6] F. Fages and M. Paltrinieri. A generic type system for CLP($\mathcal{X}$). Technical report, Ecole Normale Supérieure LIENS 97-16, December 1997.

[7] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. In J. Díaz and F. Orejas, editors, *Proceedings of TAPSOFT'89*, LNCS, pages 225–240. Springer-Verlag, 1989.

[8] M. Hanus. *Logic Programming with Type Specifications*, chapter 3, pages 91–140. MIT Press, 1992. In [15].

[9] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[10] P. M. Hill and R. W. Topor. *A Semantics for Typed Logic Programs*, chapter 1, pages 1–61. MIT Press, 1992. In [15].

[11] T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 202–217. MIT Press, 1991.

[12] Martin Müller II, T. Glaß, and K. Stroetmann. PAN — the Prolog analyzer. In R. Cousot and D. A. Schmidt, editors, *Proceedings of the 3rd Static Analysis Symposium*, LNCS, pages 387–388. Springer-Verlag, 1996.

[13] A. Mycroft and R. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[14] G. Nadathur and F. Pfenning. *Types in Higher-Order Logic Programming*, chapter 9, pages 245–283. MIT Press, 1992. In [15].

[15] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[16] J.-G. Smaus. *Modes and Types in Logic Programming.* PhD thesis, University of Kent at Canterbury, 1999.

[17] J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*, pages 335–349. MIT Press, 1999.

[18] J.-G. Smaus, F. Fages, and P. Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. In S. Kapoor and S. Prasad, editors, *Proceedings of the 20th Conference on the Foundations of Software Technology and Theoretical Computer Science*, LNCS. Springer-Verlag, 2000. To appear.

[19] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[20] K. Stroetmann and T. Glaß. A semantics for types in Prolog: The type system of PAN version 2.0. Technical report, Siemens AG, ZFE T SE 1, 81730 München, Germany, 1995.

[21] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

Contents

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399