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

Jan–Georg Smaus[1], François Fages[2], and Pierre Deransart[2]

[1] CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, jan.smaus@cwi.nl
[2] INRIA-Rocquencourt, BP105, 78153 Le Chesnay Cedex, France,
{francois.fages, pierre.deransart}@inria.fr

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.

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 [7] and Mercury [15] 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 [17], 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 [8, 9]. 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*, we might want to declare $int \leq real$, i.e., the set of integers is a subset of the set of reals. More generally, subtype relations such as $list(u) < term$ make it possible to type Prolog meta-programming predicates [5], as shown in Ex. 1.4 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. The following example explains this informally, using an ad hoc syntax.

* A long version of this paper, containing all proofs, is available in [14].

Example 1.1. Assume 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, whereas the result (the "output") may be greater than expected. This is best illustrated by a counterexample, which is obtained by introducing *reference* types.

Example 1.2. Suppose we have a function $f : real\ ref \to real$ defined by $let\ f(x) = x := 3.14;\ return\ x$. 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, it is the main problem for subject reduction in logic programming with subtypes.

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 [9, 10].[1]

Example 1.3. In analogy to Ex. 1.1, suppose `Sqrt`/2 and `Fact`/2 are predicates of declared type (`Real`,`Real`) and (`Int`, `Int`), respectively. Consider the program

```
Fact(3,6).
Sqrt(6,2.45).
```

and the derivations

```
Fact(3,x), Sqrt(x,y) ⇝ Sqrt(6,y) ⇝ □
Sqrt(6,x), Fact(x,y) ⇝ Fact(2.45,y)
```

In the first derivation, all arguments 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 2.45 to `Fact` has type `Real`, which is greater than the declared type. The atom `Fact`(2.45, `y`) is illegal, and so we do not have subject reduction.

Here we address this problem by giving a fixed direction of dataflow to logic programs, i.e., 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 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. 1.3, let the first argument of each predicate be input and the second be output. In both derivations, `x` has type `Int`. For `Fact`(3, `x`), this is exactly the declared type, and so the condition is fulfilled for the first derivation. For `Sqrt`(6, `x`), the declared type is `Real`, and so the condition is violated.

[1] Note however that the first formulation of subject reduction [10] was incorrect [8].

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 [11]. In Sec. 6, we discuss the conditions on programs.

There are few works on prescriptive type systems for logic programs with subtyping [3–6, 8]. Hill and Topor [8] give a result on subject reduction for systems without subtyping, and study general type systems with subtyping. However their results on the existence of principal typings turned out to be wrong [3]. Beierle [3] shows the existence of principal typings for systems with subtype relations between constant types, and provides type inference algorithms. Beierle and also Hanus [6] do not claim subject reduction for their systems. Fages and Paltrinieri [5] have shown a weak form of subject reduction for constraint logic programs with subtyping, where equality constraints replace substitutions in the execution model.

The idea of introducing modes to ensure subject reduction for logic programs was proposed previously by Dietrich and Hagl [4]. 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), subtype relations between type constructors of different arities are forbidden.

Example 1.4. Assume types $\mathtt{Int}$, $\mathtt{String}$ and $\mathtt{List(u)}$ defined as usual, and a type $\mathtt{Term}$ that contains all terms (so all types are subtypes of $\mathtt{Term}$). Moreover, assume $\mathtt{Append}$ as usual with declared type $(\mathtt{List(u)}, \mathtt{List(u)}, \mathtt{List(u)})$, and a predicate $\mathtt{Functor}$ with declared type $(\mathtt{Term}, \mathtt{String})$, which gives the top functor of a term. In our formalism, we could show subject reduction for the query $\mathtt{Append([1], [], x)}$, $\mathtt{Functor(x, y)}$, whereas this is not possible in [4] because the subtype relation between $\mathtt{List(Int)}$ and $\mathtt{Term}$ 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. 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 use the type system of [5]. 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 any other variables in the context. A query is a sequence of atoms. A query Q' is a **resolvent of** a query Q and a clause $h \leftarrow B$ if $Q = a_1, \ldots, a_m$, $Q' = (a_1, \ldots, a_{k-1}, B, a_{k+1}, \ldots, a_m)\theta$, and h and a_k are unifiable with MGU θ. **Resolution steps** and **derivations** are defined in the usual way.

Table 1. The subtyping order on types

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

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.

A **type substitution** Θ is a mapping from parameters to types. The **domain** of Θ is denoted by $dom(\Theta)$, the parameters in its range by $ran(\Theta)$. The set of parameters in a syntactic object o is denoted by $pars(o)$.

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 associate with each pair $K/m \leq K'/m'$ an injection $\iota_{K,K'} : \{1, \ldots, m'\} \to \{1, \ldots, m\}$ 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.

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

Proposition 2.2. For each type σ, the set $\{\tau \mid \sigma \leq \tau\}$ has a maximum, which is denoted by $Max(\sigma)$.

For Prop. 2.2, it is crucial that $K/m \leq K'/m'$ implies $m \geq m'$. For example, if we allowed for `Emptylist/0` $\leq$ `List/1`, then we would have `Emptylist` $\leq$ `List`(τ) for all τ, and so Prop. 2.2 would not hold. Note that the possibility of "forgetting" type parameters, as in `List/1` $\leq$ `Anylist/0`, may provide solutions to inequalities of the form `List(u)` $\leq$ `u`, e.g. `u` $=$ `Anylist`. However, we have:

Proposition 2.3. 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 $u \in vars(Max(\tau))$.

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 $vars(o)$. 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* [8]: $pars(\tau_1, \ldots, \tau_n) \subseteq pars(\tau)$; 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 $(\tau_1, \ldots, \tau_n) \in \mathcal{T}^n$.

Table 2. The type system.

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

The declared types are indicated by writing $f_{\tau_1 \ldots \tau_n \to \tau}$ and $p_{\tau_1 \ldots \tau_n}$. We assume that there is a special predicate symbol $=_{\mathtt{u},\mathtt{u}}$ where $\mathtt{u} \in \mathcal{U}$.

We assume that $\mathcal{K}$, $\mathcal{F}$, and $\mathcal{P}$ are fixed by declarations in a **typed program**, where the syntactical details are insignificant for our results. In examples we loosely follow Gödel syntax [7].

A **variable typing** is a mapping from a finite subset of $\mathcal{V}$ to $\mathcal{T}$, written as $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$. The restriction of a variable typing U to the variables in o is denoted as $U\lceil_o$. The type system, which defines terms, atoms etc. relative to a variable typing U, consists of the rules shown in Table 2.

3 The Subtype and Instantiation Hierarchies

3.1 Modifying Variable Typings

We now show that 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.

Definition 3.1. 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 $\tau_i \leq \tau_i'$. We write $U' \supseteq\!\leq U$ if there exists a variable typing U'' such that $U' \supseteq U''$ and $U'' \leq U$.

Lemma 3.1. 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 $U' \vdash Q\ Query$.

3.2 Typed Substitutions

Typed substitutions are a fundamental concept for typed logic programs.

Definition 3.2. 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**.

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 3.1. Consider again Ex. 1.3. Taking $U = \{\mathtt{x} : \mathtt{Int}, \mathtt{y} : \mathtt{Int}\}$, we have $U \vdash \mathtt{x} : \mathtt{Int}$, $U \vdash \mathtt{2.45} : \mathtt{Real}$, and hence $U \vdash \mathtt{x} = \mathtt{2.45}$ *Atom*. So $(\{\mathtt{x}/\mathtt{2.45}\}, U)$ is a typed substitution. Now $U \vdash \mathtt{Fact(x,y)}$ *Atom*, but $U \not\vdash \mathtt{Fact(2.45,y)}$ *Atom*. The type of x is too small to accommodate for instantiation to 2.45.

Definition 3.3. A typed (term) substitution $(\{x_1/r_1, \ldots, x_n/r_n\}, U)$ 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 $U \vdash r_i : \sigma_i$ and $\sigma_i \leq \tau_i$.

We now show that expressions stay "well-typed" when ordered substitutions are applied [8, Lemma 1.4.2].

Lemma 3.2. 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.

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 U. We will show that if U is, in a certain sense, maximal, then it is guaranteed that the typed substitution is ordered.

We first formalise paths leading to subterms of a term.

Definition 4.1. 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 4.1. 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 write $\_ \vdash t :\leq \sigma$ if there exist U and σ' such that $U \vdash t : \sigma'$ and $\sigma' \leq \sigma$. To derive $U \vdash t_1 = t_2$ *Atom*, clearly the last step has the form

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

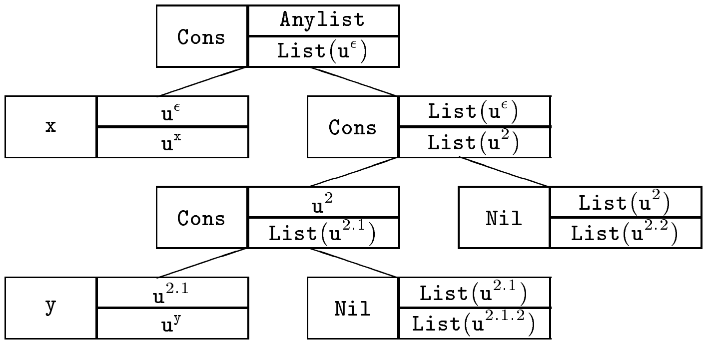

Fig. 1. The term $[\mathtt{x}, [\mathtt{y}]]$ and associated inequalities

So 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 then 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 of t with respect to σ?

Example 4.2. Let $\mathtt{List}/1, \mathtt{Anylist}/0 \in \mathcal{K}$ where $\mathtt{List}(\tau) \leq \mathtt{Anylist}$ for all τ, and $\mathtt{Nil}_{\rightarrow \mathtt{List(u)}}, \mathtt{Cons}_{\mathtt{u},\mathtt{List(u)} \rightarrow \mathtt{List(u)}} \in \mathcal{F}$. Consider the term $[\mathtt{x}, [\mathtt{y}]]$ (in usual list notation) depicted in Fig. 1, and let $\sigma = \mathtt{Anylist}$. Each functor in $[\mathtt{x}, [\mathtt{y}]]$ is introduced using Rule *(Func)*. E.g., any type of $\mathtt{Nil}$ in position 2.1.2 is necessarily an instance of $\mathtt{List}(\mathtt{u}^{2.1.2})$, its declared type.[2] To derive that $\mathtt{Cons}(\mathtt{y}, \mathtt{Nil})$ is a typed term, this instance must be smaller than some instance of the second declared argument type of $\mathtt{Cons}$ in position 2.1, i.e., $\mathtt{List}(\mathtt{u}^{2.1})$.

So in order to derive that $[\mathtt{x}, [\mathtt{y}]]$ is a term of a type smaller than $\mathtt{Anylist}$, we need an instantiation of the parameters such that for each box (position), the type in the *lower* subbox is smaller than the type of the *upper* subbox.

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 4.2. 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 $f^{\zeta}_{\tau^{\zeta}_1, \ldots, \tau^{\zeta}_{n^{\zeta}} \rightarrow \tau^{\zeta}}$ (assuming that the parameters in $\tau^{\zeta}_1, \ldots, \tau^{\zeta}_{n^{\zeta}}, \tau^{\zeta}$ are fresh, say by indexing them with ζ). For each variable $x \in vars(t)$, we introduce a parameter u^x (so $u^x \notin pars(\sigma)$). The **type inequality system** of t and σ is

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

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

[2] We use the positions as superscripts to parameters in order to obtain fresh copies.

$\mathcal{I}(t,\sigma)$ 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 $\tau'\tilde{\Theta} \leq \tau'\Theta\Theta'$.

Proposition 4.1. 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 $t' \notin \mathcal{V}$, and $U \vdash t' : u^{t'}\Theta$ if $t' \in \mathcal{V}$.

In the next subsection, we present an algorithm, based on [5], 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 equalities represent the substitution accumulated so far. We use $\leqq$ for $\leq$ or $=$.

Definition 4.3. 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 $\{\tau_1 \leqq \sigma_1, ..., \tau_n \leqq \sigma_n\}$ with $pars(\sigma_i) \cap pars(\tau_{i+1}) \neq \emptyset$ for all $1 \leq i \leq n-1$, and $pars(\sigma_n) \cap pars(\tau_1) \neq \emptyset$.

Proposition 4.2. If t is a linear term, then any inequality system $\mathcal{I}(t,\sigma)$ is acyclic and left-linear.

By looking at Ex. 4.2, it should be intuitively clear that assuming linearity of t is crucial for the above proposition.

We now give the algorithm. 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.

Definition 4.4. Given a type inequality system $\mathcal{I}(t,\sigma)$, where t is linear, the **type inequality algorithm** applies the following simplification rules:

(1) $\{K(\tau_1, ..., \tau_m) \leq K'(\tau'_1, ..., \tau'_{m'})\} \cup I \longrightarrow \{\tau_{\iota(i)} \leq \tau'_i\}_{i=1,..,m'} \cup I$
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.

Proposition 4.3. 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.

4.3 Principal Variable Typings

The existence of a principal solution Θ of a type inequality system $\mathcal{I}(t,\sigma)$ and Prop. 4.1 motivate defining the variable typing U such that Θ is exactly the solution of $\mathcal{I}(t,\sigma)$ corresponding to U.

Definition 4.5. 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. 4.1, 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 $U' \not\vdash t :\leq \sigma$.

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

Lemma 4.4. 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 U is principal for s and ρ. Then there exists a type substitution Θ such that for $U' = U\Theta\restriction_{vars(s)} \cup U\restriction_{\mathcal{V}\setminus vars(s)}$, we have that (θ, U') is an ordered substitution.

Example 4.3. Consider the term vectors (since Lemma 4.4 generalises in the obvious way to term vectors) $s = (\mathtt{3}, \mathtt{x})$ and $t = (\mathtt{3}, \mathtt{6})$, let $\rho = (\mathtt{Int}, \mathtt{Int})$ and $U_s = \{\mathtt{x} : \mathtt{Int}\}$, $U_t = \emptyset$ (see Ex. 1.3). 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 $s = (\mathtt{6}, \mathtt{x})$ and $t = (\mathtt{6}, \mathtt{2.45})$, 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 $\{\mathtt{x/Real}\}$), and indeed, there exists no Θ such that $(\{\mathtt{x/2.45}\}, U_s\Theta \cup U_t)$ is an ordered substitution.

5 Nicely Typed Programs

So far we have seen that *matching*, *linearity*, and *principal variable typings* are crucial to ensure that unification yields ordered substitutions. Note that those results generalise in the obvious way from terms to term vectors. We now define three corresponding conditions on programs and the execution model.

First, we define modes [1]. For $p/n \in \mathcal{P}$, 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 (resp. O) are called **input** (resp. **output**) **positions** of p. We assume that a mode is associated with each $p \in \mathcal{P}$. The notation $p(\bar{s}, \bar{t})$ means that $\bar{s}$ (resp. $\bar{t}$) is the vector of terms filling the input (resp. output) positions of $p(\bar{s}, \bar{t})$. Moded unification is a special case of *double matching* [2].

Definition 5.1. Consider a resolution 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**.

Definition 5.2. 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 $i \in \{1, \ldots, n\}$

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

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

$$vars(\bar{t}_0) \cap \bigcup_{j=1}^{n} vars(\bar{t}_j) = \emptyset. \tag{2}$$

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

Definition 5.3. 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 $U \vdash C\ \mathit{Clause}$, 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 $U \vdash C\ \mathit{Clause}$, then we say that C is **nicely typed**. A query $U_Q : Q$ is **nicely typed** if the clause $\mathtt{Go} \leftarrow Q$ is nicely typed.

We can now state the main result.

Theorem 5.1. 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.

Example 5.1. Consider again Ex. 1.3. 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. 4.3). 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.

It is instructive to divide the numerous conditions we impose into four classes: (1) "basic" type conditions on the program (Sec. 2), (2) conditions on the execution model (Def. 5.1), (3) mode conditions on the program (Def. 5.2), (4) "additional" type conditions on the program (Def. 5.3).

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 [3] 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 $\texttt{List}(\texttt{Int}) \leq \texttt{List}(\texttt{Real})$, but not $\texttt{List}(\texttt{Int}) \leq \texttt{Tree}(\texttt{Real})$. Many authors assume that only constructors of the same arity are comparable. Thus we could have $\texttt{List}(\texttt{Int}) \leq \texttt{Tree}(\texttt{Real})$, but not $\texttt{List}(\texttt{Int}) \leq \texttt{Anylist}$. We assume, as [5], that if $K/m \leq K'/m'$, then $m \geq m'$. We think that this choice is crucial for the existence of principal types.

Stroetmann and Glaß [16] argue that comparisons between constructors of arbitrary arity should be allowed in principle. Their formalism is such that the subtype relation does not automatically correspond to a subset relation. Nevertheless, the formalism heavily relies on such a correspondence, although it is not said how it can be decided. We refer to [14] for more details.

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 gave 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).

In Sec. 5, we showed how those conditions translate to conditions on the program and the execution model. We introduce modes and assume that programs are executed using moded unification (2). This might be explicitly enforced by the compiler, or it might be verified statically [2]. Moded unification can actually be very beneficial for efficiency, as witnessed by the language Mercury [15]. Apart from that, (3) nicely-modedness states the linearity of the terms being instantiated in a unification. 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 [13, 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, printing, or in higher-order logic programming where the existence and shape of unifiers depends on the types [11].

There is a relationship between our notion of subtyping and *transparency* (see Subsec. 2.2). 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 $\texttt{List}(\texttt{Int})$, 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 $\{\texttt{x} : \texttt{String}\} \vdash [\texttt{x}] = [1]\ Atom$, since $\texttt{List}(\texttt{String}) \leq \texttt{Anylist}$ and $\texttt{List}(\texttt{Int}) \leq \texttt{Anylist}$, even though Int and String are incomparable. We compensate for this by requiring principal variable typings. A 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`. Our intuition is that whenever this phenomenon ("forgetting" parameters) occurs, requiring principal variable typings is very demanding; but otherwise, subject reduction is likely to be violated. As a topic for future work, we want to substantiate this intuition by studying examples.

Acknowledgements. We thank Erik Poll and François Pottier for interesting discussions. 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 MFCS*, LNCS, pages 1–19. Springer-Verlag, 1993.
3. C. Beierle. Type inferencing for polymorphic order-sorted logic programs. In L. Sterling, editor, *Proceedings of ICLP*, pages 765–779. MIT Press, 1995.
4. R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In H. Ganzinger, editor, *Proceedings of ESOP*, LNCS, pages 79–93. Springer-Verlag, 1988.
5. F. Fages and M. Paltrinieri. A generic type system for CLP($\mathcal{X}$). Technical report, Ecole Normale Supérieure LIENS 97-16, December 1997.
6. M. Hanus. *Logic Programming with Type Specifications*, chapter 3, pages 91–140. MIT Press, 1992. In [12].
7. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.
8. P. M. Hill and R. W. Topor. *A Semantics for Typed Logic Programs*, chapter 1, pages 1–61. MIT Press, 1992. In [12].
9. 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 ILPS*, pages 202–217. MIT Press, 1991.
10. A. Mycroft and R. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
11. G. Nadathur and F. Pfenning. *Types in Higer-Order Logic Programming*, chapter 9, pages 245–283. MIT Press, 1992. In [12].
12. F. Pfenning, editor. *Types in Logic Programming.* MIT Press, 1992.
13. J.-G. Smaus. *Modes and Types in Logic Programming.* PhD thesis, University of Kent at Canterbury, 1999.
14. J.-G. Smaus, F. Fages, and P. Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. Technical report, INRIA, 2000. Available via CoRR: `http://arXiv.org/archive/cs/intro.html`.
15. 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.
16. 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.
17. Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.