Well-Typed Logic Programs Are not Wrong

Pierre Deransart, Jan Georg Smaus

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Well-Typed Logic Programs Are not Wrong

Pierre Deransart — Jan–Georg Smaus

N° 4082

December 2000

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4082--FR+ENG

Well-Typed Logic Programs Are not Wrong

Pierre Deransart* , Jan–Georg Smaus†

Thème 2 — Génie logiciel
et calcul symbolique
Projet Contraintes

Rapport de recherche n° 4082 — December 2000 — 21 pages

Abstract: We consider prescriptive type systems for logic programs (as in Gödel or Mercury). In such systems, the typing is *static*, but it guarantees an operational property: if a program is "well-typed", then all derivations starting in a "well-typed" query are again "well-typed". This property has been called *subject reduction*. We show that this property can also be phrased as a property of the *proof-theoretic* semantics of logic programs, thus abstracting from the usual operational (top-down) semantics. This proof-theoretic view leads us to questioning a condition which is usually considered necessary for subject reduction, namely the *head condition*. It states that the head of each clause must have a type which is a variant (and not a proper instance) of the declared type. We provide a more general condition, thus reestablishing a certain symmetry between heads and body atoms. The condition ensures that in a derivation, the types of two unified terms are themselves unifiable. We discuss possible implications of this result. We also discuss the relationship between the head condition and *polymorphic recursion*, a concept known in functional programming.

Key-words: Logic programming, (prescriptive) type system, subject reduction, polymorphism, head condition, derivation tree, polymorphic recursion

This paper is the complete version of a paper presented at FLOPS 2001 [8]. It contains all proofs omitted there for space reasons.

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

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

Les programmes logiques bien typés ont tout bon

Résumé : On étudie ici les sytèmes de typage prescriptif (à la Gödel ou Mercury) pour la programmation en logique. Dans de tels systèmes, le typage est *statique*, mais il garantit une propriété opérationnelle: si un programme est "bien typé", alors tous les buts obtenus par dérivation d'un but "bien typé" sont eux-même "bien typés". Le système est alors dit *stable par réduction* ("subject reduction"). Nous montrons dans ce papier que cette propriété de stabilité est en fait aussi *déclarative*. Cette vue déclarative nous conduit à reconsidérer une condition habituellement admise et nécéssaire pour la stabilité par réduction, dite *condition de tête* ("head condition"). Cette condition stipule que le type des têtes des clauses d'un programme "bien typé" doit être une variante (et non une instance quelconque) du type déclaré de son prédicat. Il est alors possible de formuler des conditions plus générales qui rétablissent une certaine symétrie entre les têtes et les atomes du corps, et qui garantissent en fait que dans toutes dérivations les types des termes unifiables sont eux aussi unifiables. On discute enfin les implications possibles d'un tel résultat, en particulier la relation entre la condition de tête et la *recursion polymorphique*, un concept connu dans la programmation fonctionnelle.

Mots-clés : programmation en logique, système de type (prescriptif), "subject reduction", polymorphisme, condition de tête, arbre de dérivation, recursion polymorphique

1 Introduction

Prescriptive types are used in logic programming (and other paradigms) to restrict the underlying syntax so that only "meaningful" expressions are allowed. This allows for many programming errors to be detected by the compiler. Moreover, it ensures that once a program has passed the compiler, the types of arguments of predicates can be ignored at runtime, since it is guaranteed that they will be of correct type. This has been turned into the famous slogan [20, 21]

> Well-typed programs cannot go wrong.

Adopting the terminology from the theory of the λ-calculus [30], 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. Translated to logic programming, this means that resolving a "well-typed" query with a "well-typed" clause will always result in a "well-typed" query, and so the successive queries obtained during a derivation are all "well-typed".

From this observation, it is clear that subject reduction is a property of the *operational* semantics of a logic program, i.e., SLD resolution [17]. In this paper, we show that it is also a property of the proof-theoretic semantics based on *derivation trees*. This is obtained by showing that using "well-typed" clauses, only "well-typed" derivation trees can be constructed, giving rise to the new slogan:

> Well-typed programs *are* not wrong.

The *head condition*, which is a condition on the program (clauses) [13], is usually considered to be crucial for subject reduction. The second objective of this paper is to analyse the head condition in this new light and open the field for generalisations, of which we introduce one.

The head condition, also called *definitional genericity* [16], states that the types of the arguments of a clause head must be a variant[1] (and not a proper instance) of the declared type of the head predicate. This condition imposes a distinction between "definitional" occurrences (clause heads) and "applied" occurrences (body atoms) of a predicate. In contrast, the proof-theoretic view of subject reduction we propose reestablishes a certain symmetry between the different occurrences. By this generalisation, the class of programs for which subject reduction is guaranteed is enlarged.

This paper is organised as follows. Section 2 contains some preliminaries. Section 3 introduces our proof-theoretic notion of subject reduction. Section 4 gives conditions for subject reduction, and in particular, a generalisation of the head condition. In Section 5, we discuss, in the light of these results, the usefulness of the head condition and its generalisation. We also exhibit an interesting relationship between the head condition and *polymorphic recursion* [15]. Section 6 concludes by mentioning possible applications of these results.

[1] A variant is obtained by renaming the type parameters in a type.

2 Preliminaries

We assume familiarity with the standard concepts of logic programming [17]. To simplify the notation, a vector such as $o_1, \ldots, o_m$ is often denoted by $\bar{o}$. The restriction of a substitution θ to the variables in a syntactic object o is denoted as $\theta\restriction_o$, and analogously for type substitutions (see Subsec. 2.2). The relation symbol of an atom a is denoted by $Rel(a)$.

When we refer to a *clause in a program*, we usually 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 Q' is **derived from** a query Q, denoted $Q \rightsquigarrow Q'$, if $Q = a_1, \ldots, a_m$, $Q' = (a_1, \ldots, a_{k-1}, B, a_{k+1}, \ldots, a_m)\theta$, and $h \leftarrow B$ is a clause (in a program usually clear from the context) such that h and a_k are unifiable with MGU θ. A **derivation** $Q \rightsquigarrow^* Q'$ is defined in the usual way. Given a program P, the **immediate consequence operator** T_P is defined by $T_P(M) = \{h\theta \mid h \leftarrow a_1, \ldots, a_m \in P,\ a_1\theta, \ldots, a_m\theta \in M\}$.

2.1 Derivation Trees

A key element of this work is the proof-theoretic semantics of logic programs based on derivation trees [6]. We recall some important notions and basic results.

Definition 1 *An* **instance name** *of a clause C is a pair of the form $\langle C, \theta\rangle$, where θ is a substitution.*

Definition 2 *Let P be a program. A* **derivation tree** *for P is a labelled ordered tree [6] such that:*

1. *Each leaf node is labelled by $\bot$ or an instance name $\langle C, \theta\rangle$ of a clause[2] in P; each non-leaf node is labelled by an instance name $\langle C, \theta\rangle$ of a clause in P.*
2. *If a node is labelled by $\langle h \leftarrow a_1, \ldots, a_m, \theta\rangle$, where $m \geq 0$, then this node has m children, and for $i \in \{1, \ldots, m\}$, the ith child is labelled either $\bot$, or $\langle h' \leftarrow B, \theta'\rangle$ where $h'\theta' = a_i\theta$.*

Nodes labelled $\bot$ are **incomplete**, *all other nodes are* **complete**. *A derivation tree containing only complete nodes is a* **proof tree**.

To define the semantics of logic programs, it is useful to associate an atom with each node in a derivation tree in the following way.

Definition 3 *Let T be a derivation tree. For each node n in T, the* **node atom** *of n, denoted $atom(n)$, is defined as follows: If n is labelled $\langle h \leftarrow B, \theta\rangle$, then $h\theta$ is the node atom of n; if n is labelled $\bot$, and n is the ith child of its parent labelled $\langle h \leftarrow a_1, \ldots, a_m, \theta\rangle$, then $a_i\theta$ is the node atom of n. If n is the root of T then $atom(n)$ is the* **head of** *T, denoted $head(T)$.*

[2] Recall that C is renamed apart from any other clause in the same tree.

Derivation trees are obtained by grafting instances of clauses of a program. To describe this construction in a general way, we define the following concept.

Definition 4 *Let P be a program. A* **skeleton (tree)** *for P is a labelled ordered tree such that:*

1. *Each leaf node is labelled by $\bot$ or a clause in P, and each non-leaf node is labelled by a clause in P.*
2. *If a node is labelled by $h \leftarrow a_1, \ldots, a_m$, where $m \geq 0$, then this node has m children and for $i \in \{1, \ldots, m\}$, the ith child is labelled either $\bot$, or $h' \leftarrow B$, where $Rel(h') = Rel(a_i)$.*

The **skeleton of a tree** T, denoted $Sk(T)$, is the skeleton obtained from T by replacing each label $\langle C, \theta \rangle$ with C. Conversely, we say that T is a **derivation tree based on** $Sk(T)$.

Definition 5 *Let S be a skeleton. We define*

$$Eq(S) = \{a_i = h' \mid \begin{array}{l} \text{there exist complete nodes } n, n' \text{ in } S \text{ such that} \\ \bullet\ n' \text{ is the } i\text{th child of } n, \\ \bullet\ n \text{ is labelled } h \leftarrow a_1, \ldots, a_m, \\ \bullet\ n' \text{ is labelled } h' \leftarrow B\} \end{array}$$

Abusing notation, we frequently identify the set of equations with the conjunction or sequence of all equations contained in it. If $Eq(S)$ has a unifier then we call S a **proper** *skeleton.*

Proposition 1 *[6, Prop. 2.1] Let S be a skeleton. A derivation tree based on S exists if and only if S is proper.*

Theorem 2 *[6, Thm. 2.1] Let S be a skeleton and θ an MGU of $Eq(S)$. Let $D(S)$ be the tree obtained from S by replacing each node label C with the pair $\langle C, \theta{\restriction_C} \rangle$. Then $D(S)$ is a most general derivation tree based on S (i.e., any other derivation tree based on S is an instance of $D(S)$).*

Example 1 *Figure 1 shows a program, one of its derivation trees, and the skeleton of the derivation tree.*

To model derivations for a program P and a query Q, we assume that P contains an additional clause $\mathsf{go} \leftarrow Q$, where go is a new predicate symbol.

We recall the following straightforward correspondences between derivations, the T_P-semantics and derivation trees.

Proposition 3 *Let P be a program. Then*

1. *$a \in lfp(T_P)$ if and only if $a = head(T)$ for some proof tree T for P,*
2. *$Q \rightsquigarrow^* Q'$ if and only if Q' is the sequence of node atoms of incomplete nodes of a most general derivation tree for $P \cup \{\mathsf{go} \leftarrow Q\}$ with head go, visited left to right.*

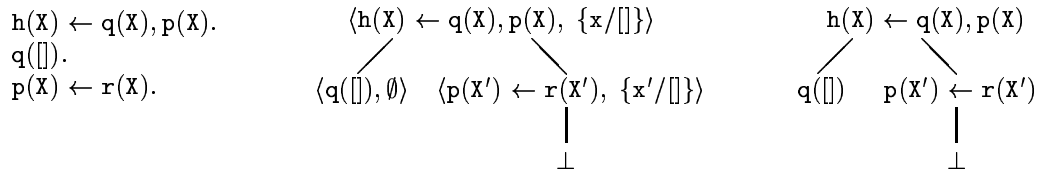

Figure 1: A program, a derivation tree and its skeleton

2.2 Typed Logic Programming

We assume a type system for logic programs with parametric polymorphism but without subtyping, as realised in the languages Gödel [12] or Mercury [28].

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 **type substitution** is an idempotent mapping from parameters to types which is the identity almost everywhere. The set of parameters in a syntactic object o is denoted by $pars(o)$.

We assume a denumerable set $\mathcal{V}$ of **variables**. The set of variables in a syntactic object o is denoted by $vars(o)$. A **variable typing** is a mapping from a finite subset of $\mathcal{V}$ to $\mathcal{T}$, written as $\{x_1 : \tau_1, \ldots, x_m : \tau_m\}$.

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 $(\tau_1, \ldots, \tau_m, \tau)$, where m is the arity of f, $(\tau_1, \ldots, \tau_m) \in \mathcal{T}^m$, and τ satisfies the *transparency condition* [13]: $pars(\tau_1, \ldots, \tau_m) \subseteq pars(\tau)$; for each $p \in \mathcal{P}$, the declared type has the form $(\tau_1, \ldots, \tau_m)$, where m is the arity of p and $(\tau_1, \ldots, \tau_m) \in \mathcal{T}^m$. We often indicate the declared types by writing $f_{\tau_1 \ldots \tau_m \to \tau}$ and $p_{\tau_1 \ldots \tau_m}$, however we assume that the parameters in $\tau_1, \ldots, \tau_m, \tau$ are fresh for each occurrence of f or p. We assume that there is a special predicate symbol $=_{u,u}$ where $u \in \mathcal{U}$.

Throughout this paper, we assume $\mathcal{K}$, $\mathcal{F}$, and $\mathcal{P}$ arbitrary but fixed. The **typed language**, i.e. a language of terms, atoms etc. based on $\mathcal{K}$, $\mathcal{F}$, and $\mathcal{P}$, is defined by the rules in Table 1. All objects are defined relative to a variable typing U, and $\_ \vdash \ldots$ stands for "there exists U such that $U \vdash \ldots$". The expressions below the line are called **type judgements**.

Formally, a proof of a type judgement is a tree where the nodes are labelled with judgements and the edges are labelled with rules (e.g. see Fig. 2) [30]. From the form of the rules, it is clear that in order to prove any type judgement, we must, for each occurrence of a term t in the judgement, prove a judgement $\ldots \vdash t : \tau$ for some τ. We now define the most general such τ. It exists and can be computed by *type inferencing algorithms* [2].

Definition 6 *Consider a judgement $U \vdash p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_m(\bar{t}_m)$ Clause, and a proof of this judgement containing judgements $U \vdash \bar{t} : \bar{\tau}$, $U \vdash \bar{t}_1 : \bar{\tau}_1$, ..., $U \vdash \bar{t}_m : \bar{\tau}_m$ (see Fig. 2) such that $(\bar{\tau}, \bar{\tau}_1, \ldots, \bar{\tau}_m)$ is most general (wrt. all such proofs). We call $(\bar{\tau}, \bar{\tau}_1, \ldots, \bar{\tau}_m)$ the* **most general type** *of $p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_m(\bar{t}_m)$* **wrt.** *$U$.*

Table 1: Rules defining a typed language

(Var)	$\{x:\tau,\ldots\}\vdash x:\tau$	
(Func)	$\dfrac{U\vdash t_1{:}\tau_1\Theta\ \cdots\ U\vdash t_m{:}\tau_m\Theta}{U\vdash f_{\tau_1\ldots\tau_m\to\tau}(t_1,\ldots,t_m){:}\tau\Theta}$	Θ is a type substitution
(Atom)	$\dfrac{U\vdash t_1{:}\tau_1\Theta\ \cdots\ U\vdash t_m{:}\tau_m\Theta}{U\vdash p_{\tau_1\ldots\tau_m}(t_1,\ldots,t_m)\ \mathit{Atom}}$	Θ is a type substitution
(Query)	$\dfrac{U\vdash A_1\ \mathit{Atom}\ \cdots\ U\vdash A_m\ \mathit{Atom}}{U\vdash A_1,\ldots,A_m\ \mathit{Query}}$	
(Clause)	$\dfrac{U\vdash A\ \mathit{Atom}\quad U\vdash Q\ \mathit{Query}}{U\vdash A\leftarrow Q\ \mathit{Clause}}$	
(Program)	$\dfrac{\_\vdash C_1\ \mathit{Clause}\ \cdots\ \_\vdash C_m\ \mathit{Clause}}{\_\vdash\{C_1,\ldots,C_m\}\ \mathit{Program}}$	
(Queryset)	$\dfrac{\_\vdash Q_1\ \mathit{Query}\ \cdots\ \_\vdash Q_m\ \mathit{Query}}{\_\vdash\{Q_1,\ldots,Q_m\}\ \mathit{Queryset}}$	

$$\dfrac{\dfrac{\vdots}{U\vdash\bar{t}:\bar{\tau}}\qquad\dfrac{\dfrac{\vdots}{U\vdash\bar{t}_1:\bar{\tau}_1}}{U\vdash p_1(\bar{t}_1)\ \mathit{Atom}}\ \cdots\ \dfrac{\dfrac{\vdots}{U\vdash\bar{t}_m:\bar{\tau}_m}}{U\vdash p_m(\bar{t}_m)\ \mathit{Atom}}}{\dfrac{U\vdash p(\bar{t})\ \mathit{Atom}}{U\vdash p(\bar{t})\leftarrow p_1(\bar{t}_1),\ldots,p_m(\bar{t}_m)\ \mathit{Clause}}\qquad U\vdash p_1(\bar{t}_1),\ldots,p_m(\bar{t}_m)\ \mathit{Query}}$$

Figure 2: Proving a type judgement

Moreover, consider the variable typing U' and the proof of the judgement $U'\vdash p(\bar{t})\leftarrow p_1(\bar{t}_1),\ldots,p_m(\bar{t}_m)$ Clause containing judgments $U'\vdash\bar{t}:\bar{\tau}$, $U'\vdash\bar{t}_1:\bar{\tau}_1$, …, $U'\vdash\bar{t}_m:\bar{\tau}_m$ such that $(\bar{\tau},\bar{\tau}_1,\ldots,\bar{\tau}_m)$ is most general (wrt. all such proofs and all possible U'). We call $(\bar{\tau},\bar{\tau}_1,\ldots,\bar{\tau}_m)$ the **most general type** *of $p(\bar{t})\leftarrow p_1(\bar{t}_1),\ldots,p_m(\bar{t}_m)$.*

The following example explains the difference between the most general type wrt. a fixed variable typing, and the most general type as such.

Example 2 *Consider function* $\mathtt{nil}_{\to\mathtt{list(U)}}$ *and clause* $C=\mathtt{p}\leftarrow\mathtt{X}{=}\mathtt{nil},\mathtt{nil}{=}\mathtt{nil}$. *Fixing* $U=\{\mathtt{X}:\mathtt{list(int)}\}$, *the judgement* $U\vdash C$ *Clause can be proven using the judgements* $U\vdash\mathtt{X}:\mathtt{list(int)}$ *and then* $U\vdash\mathtt{nil}:\mathtt{list(int)}$ *for* each *occurrence of* `nil`. *It can also be proven using the judgements* $U\vdash\mathtt{X}:\mathtt{list(int)}$ *and then* $U\vdash\mathtt{nil}:\mathtt{list(int)}$ *(for the first occurrence of* `nil`*) and then* $U\vdash\mathtt{nil}:\mathtt{list(V)}$ *(for the other two occurrences of* `nil`*). In the latter case, we obtain* $(\mathtt{list(int)},\mathtt{list(int)},\mathtt{list(V)},\mathtt{list(V)})$, *the most general type*

of C wrt. U. Moreover, $(\texttt{list(V')}, \texttt{list(V')}, \texttt{list(V)}, \texttt{list(V)})$ is the most general type of C (choose $U' = \{\texttt{X} : \texttt{list(V')}\}$).

Definition 7 *If $U \vdash x_1 = t_1, \ldots, x_m = t_m$ Query where $x_1, \ldots, x_m$ are distinct variables and for each $i \in \{1, \ldots, m\}$, t_i is a term distinct from x_i, then $(\{x_1/t_1, \ldots, x_m/t_m\}, U)$ is a* **typed (term) substitution**.

We shall need three fundamental lemmas introduced in [13].[3]

Lemma 4 *[13, Lemma 1.2.8] Let U be a variable typing and Θ a type substitution. If $U \vdash t : \sigma$, then $U\Theta \vdash t : \sigma\Theta$. Moreover, if $U \vdash A$ Atom then $U\Theta \vdash A$ Atom, and likewise for queries and clauses.*

Proof: The proof is by structural induction. For the base case, suppose $U \vdash x : \sigma$ where $x \in \mathcal{V}$. Then $x : \sigma \in U$ and hence $x : \sigma\Theta \in U\Theta$. Thus $U\Theta \vdash x : \sigma\Theta$.

Now consider $U \vdash f_{\tau_1 \ldots \tau_m \to \tau}(t_1, \ldots, t_m) : \sigma$ where the inductive hypothesis holds for $t_1, \ldots, t_m$. By Rule *(Func)*, there exists a type substitution Θ' such that $\sigma = \tau\Theta'$ and $U \vdash t_i : \tau_i\Theta'$ for each $i \in \{1, \ldots, m\}$. By the inductive hypothesis, $U\Theta \vdash t_i : \tau_i\Theta'\Theta$ for each $i \in \{1, \ldots, m\}$, and hence by Rule *(Func)*, $U\Theta \vdash f_{\tau_1 \ldots \tau_m \to \tau}(t_1, \ldots, t_m) : \tau\Theta'\Theta$.

The rest of the proof is now trivial. □

Lemma 5 *[13, Lemma 1.4.2] Let (θ, U) be a typed substitution. If $U \vdash t : \sigma$ then $U \vdash t\theta : \sigma$. Moreover, if $U \vdash A$ Atom then $U \vdash A\theta$ Atom, and likewise for queries and clauses.*

Proof: The proof is by structural induction. For the base case, suppose $U \vdash x : \sigma$ where $x \in \mathcal{V}$. If $x\theta = x$, there is nothing to show. If $x/t \in \theta$, then by definition of a typed substitution, $U \vdash t : \sigma$.

Now consider $U \vdash f_{\tau_1 \ldots \tau_m \to \tau}(t_1, \ldots, t_m) : \sigma$ where the inductive hypothesis holds for $t_1, \ldots, t_m$. By Rule *(Func)*, there exists a type substitution Θ' such that $\sigma = \tau\Theta'$, and $U \vdash t_i : \tau_i\Theta'$ for each $i \in \{1, \ldots, m\}$. By the inductive hypothesis, $U \vdash t_i\theta : \tau_i\Theta'$ for each $i \in \{1, \ldots, m\}$, and hence by Rule *(Func)*, $U \vdash f_{\tau_1 \ldots \tau_m \to \tau}(t_1, \ldots, t_m)\theta : \tau\Theta'$.

The rest of the proof is now trivial. □

Lemma 6 *[13, Thm. 1.4.1] Let E be a set (conjunction) of equations such that for some variable typing U, we have $U \vdash E$ Query. Suppose θ is an MGU of E. Then (θ, U) is a typed substitution.*

Proof: We show that the result is true when θ is computed using the well-known Martelli-Montanari algorithm [19] which works by transforming a set of equations $E = E_0$ into a set of the form required in the definition of a typed substitution. Only the following two transformations are considered here. The others are trivial.

[3] Note that some results in [13] have been shown to be faulty (Lemmas 1.1.7, 1.1.10 and 1.2.7), although we believe that these mistakes only affect type systems which include subtyping.

1. If $x = t \in E_k$ and x does not occur in t, then replace all occurrences of x in all other equations in E with t, to obtain E_{k+1}.
2. If $f(t_1, \ldots, t_m) = f(s_1, \ldots, s_m) \in E_k$, then replace this equation with $t_1 = s_1, \ldots, t_m = s_m$, to obtain E_{k+1}.

We show that if $U \vdash E_k$ *Query* and E_{k+1} is obtained by either of the above transformations, then $U \vdash E_{k+1}$ *Query*. For (1), this follows from Lemma 5.

For (2), suppose $U \vdash E_k$ *Query* and $f(t_1, \ldots, t_m) = f(s_1, \ldots, s_m) \in E_k$ where $f = f_{\tau_1 \ldots \tau_m \to \tau}$. By Rule *(Query)*, we must have $U \vdash f(t_1, \ldots, t_m) =_{u,u} f(s_1, \ldots, s_m)$ *Atom*, and hence by Rule *(Atom)*, $U \vdash f(t_1, \ldots, t_m) : u\Theta$ and $U \vdash f(s_1, \ldots, s_m) : u\Theta$ for some type substitution Θ. On the other hand, by Rule *(Func)*, $u\Theta = \tau\Theta_t$ and $u\Theta = \tau\Theta_s$ for some type substitutions Θ_s and Θ_t, and moreover for each $i \in \{1, \ldots, m\}$, we have $U \vdash t_i : \tau_i\Theta_t$ and $U \vdash s_i : \tau_i\Theta_s$. Since $pars(\tau_i) \subseteq pars(\tau)$, it follows that $\tau_i\Theta_t = \tau_i\Theta_s$.[4] Therefore $U \vdash t_i = s_i$ *Atom*, and so $U \vdash E_{k+1}$ *Query*. □

3 Subject Reduction for Derivation Trees

We first define subject reduction as a property of derivation trees and show that it is equivalent to the usual operational notion. We then show that subject reduction requires that the types of all unified terms are themselves unifiable.

3.1 Proof-Theoretic and Operational Subject Reduction

Subject reduction is a well-understood concept, yet it has to be defined formally for each system. We now provide two fundamental definitions.

Definition 8 *Let* $\_ \vdash P$ *Program and* $\_ \vdash \mathcal{Q}$ *Queryset. We say* P *has* **(proof-theoretic) subject reduction wrt.** $\mathcal{Q}$ *if for every* $Q \in \mathcal{Q}$*, for every most general derivation tree* T *for* $P \cup \{\texttt{go} \leftarrow Q\}$ *with head* go*, there exists a variable typing* U' *such that for each node atom* a *of* T*,* $U' \vdash a$ *Atom.*

P *has* **operational subject reduction wrt.** $\mathcal{Q}$ *if for every* $Q \in \mathcal{Q}$*, for every derivation* $Q \rightsquigarrow^* Q'$ *of* P*, we have* $\_ \vdash Q'$ *Query.*

The reference to $\mathcal{Q}$ is omitted if $\mathcal{Q} = \{Q \mid \_ \vdash Q \text{ Query}\}$. The following theorem states a certain equivalence between the two notions.

Theorem 7 *Let* $\_ \vdash P$ *Program and* $\_ \vdash \mathcal{Q}$ *Queryset. If* P *has subject reduction wrt.* $\mathcal{Q}$*, then* P *has operational subject reduction wrt.* $\mathcal{Q}$*. If* P *has operational subject reduction, then* P *has subject reduction.*

[4] Note how the transparency condition is essential to ensure that subarguments in corresponding positions have identical types. This condition was ignored in [21].

Proof: The first statement is a straightforward consequence of Prop. 3 (2).

For the second statement, assume $U \vdash Q$ *Query*, let $\xi = Q \rightsquigarrow^* Q'$, and T be the derivation tree for $P \cup \{\texttt{go} \leftarrow Q\}$ corresponding to ξ (by Prop. 3 (2)).

By hypothesis, there exists a variable typing U' such that for each *incomplete* node n of T, we have $U' \vdash atom(n)$ *Atom*. To show that this also holds for *complete* nodes, we transform ξ into a derivation which "records the entire tree T". This is done as follows: Let $\tilde{P}$ be the program obtained from P by replacing each clause $h \leftarrow B$ with $h \leftarrow B, B$. Let us call the atoms in the second occurrence of B *unresolvable*. Clearly $\_ \vdash h \leftarrow B, B$ *Clause* for each such clause.

By induction on the length of derivations, one can show that $\tilde{P}$ has operational subject reduction. For a single derivation step, this follows from the operational subject reduction of P.

Now let $\tilde{\xi} = \texttt{go} \rightsquigarrow \tilde{Q}'$ be the derivation for $\tilde{P} \cup \{\texttt{go} \leftarrow Q, Q\}$ using in each step the clause corresponding to the clause used in ξ for that step, and resolving only the resolvable atoms. First note that since $\tilde{P}$ has operational subject reduction, there exists a variable typing U' such that $U' \vdash \tilde{Q}'$ *Query*. Moreover, since the unresolvable atoms are not resolved in $\tilde{\xi}$, it follows that $\tilde{Q}'$ contains exactly the non-root node atoms of T. This however shows that for each node atom a of T, we have $U' \vdash a$ *Atom*. Since the choice of Q was arbitrary, P has subject reduction. □

The following example shows that in the second statement of the above theorem, it is crucial that P has operational subject reduction wrt. *all* queries.

Example 3 *Let* $\mathcal{K} = \{\texttt{list}/1, \texttt{int}/0\}$, $\mathcal{F} = \{\texttt{nil}_{\rightarrow \texttt{list(U)}}, \texttt{cons}_{\texttt{U,list(U)} \rightarrow \texttt{list(U)}}, -1_{\rightarrow \texttt{int}}, 0_{\rightarrow \texttt{int}}, \ldots\}$, $\mathcal{P} = \{\texttt{p}_{\texttt{list(int)}}, \texttt{r}_{\texttt{list(U)}}\}$, *and* P *be*

```
p(X) <- r(X).                    r([X]) <- r(X).
```

For each derivation $\texttt{p(X)} \rightsquigarrow^* Q'_0$, *we have* $Q'_0 = \texttt{p(Y)}$ *or* $Q'_0 = \texttt{r(Y)}$ *for some* $\texttt{Y} \in \mathcal{V}$, *and so* $\{\texttt{Y} : \texttt{list(int)}\} \vdash \texttt{p(Y)}$ *Query or* $\{\texttt{Y} : \texttt{list(U)}\} \vdash \texttt{r(Y)}$ *Query. Therefore* P *has operational subject reduction wrt.* $\{\texttt{p(X)}\}$. *Yet the derivation trees for* P *have heads* $\texttt{p(Y)}$, $\texttt{p([Y])}$, $\texttt{p([[Y]])}$ *etc., and* $\_ \not\vdash \texttt{p([[Y]])}$ *Query.*

3.2 Unifiability of Types and Subject Reduction

We now lift the notion of skeleton to the type level.

Definition 9 *Let* $\_ \vdash P$ *Program and* S *be a skeleton for* P. *The* **type skeleton corresponding to** S *is a tree obtained from* S *by replacing each node label* $C_n = p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_m(\bar{t}_m)$ *with* $p(\bar{\tau}) \leftarrow p_1(\bar{\tau}_1), \ldots, p_m(\bar{\tau}_m)$, *where* $(\bar{\tau}, \bar{\tau}_1, \ldots, \bar{\tau}_m)$ *is the most general type of* C_n.[5] *For a type skeleton* TS, *the* **type equation set** $Eq(TS)$ *and a* **proper** *type skeleton are defined as in Def. 5.*

[5] Recall that the variables in C_n and the parameters in $\bar{\tau}, \bar{\tau}_1, \ldots, \bar{\tau}_m$ are renamed apart from other node labels in the same (type) skeleton.

The following theorem states that in order to obtain subject reduction, terms must be unified only if their types are also unifiable.

Theorem 8 *Let* $\_ \vdash P$ *Program and* $\_ \vdash \mathcal{Q}$ *Queryset. P has subject reduction wrt. $\mathcal{Q}$ iff for each proper skeleton S of $P \cup \{\mathtt{go} \leftarrow Q\}$ with head* go*, where $Q \in \mathcal{Q}$, the* type *skeleton corresponding to S is proper.*

Proof: For the "$\Rightarrow$" direction suppose that P has subject reduction wrt. $\mathcal{Q}$, and let S be an arbitrary proper skeleton for $P \cup \{\mathtt{go} \leftarrow Q\}$ with head go, where $Q \in \mathcal{Q}$. Let $\theta = MGU(Eq(S))$.

By hypothesis, there exists a variable typing U' such that for each node atom a of S, we have $U' \vdash a\theta$ *Atom*, and so for each label (clause) C of S, we have $U' \vdash C\theta$ *Clause*. Suppose $\theta = \{x_1/s_1, \ldots, x_k/s_k\}$. For each $i \in \{1, \ldots, k\}$, because of renaming apart the variables, x_i occurs in exactly one C in S, and moreover, there exists σ_i such that a judgement $U' \vdash s_i : \sigma_i$ is contained in the proof of the judgement $U' \vdash C\theta$ *Clause* (see Subsec. 2.2). Let $U'' = U' \cup \{x_1 : \sigma_1, \ldots, x_k : \sigma_k\}$. It is easy to see that $U'' \vdash C$ *Clause*. Now by Def. 6, the most general type of C *wrt.* U'' is an instance of the most general type of C. Because of renaming apart the parameters in TS, it follows that there exists a type substitution Θ which solves $Eq(TS)$.

We now show the "$\Leftarrow$" direction. Let S be an arbitrary proper skeleton for $P \cup \{\mathtt{go} \leftarrow Q\}$ with head go, where $Q \in \mathcal{Q}$. Let $\theta = MGU(Eq(S))$ and $\Theta = MGU(Eq(TS))$. For each node n in S, labelled $p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_m(\bar{t}_m)$ in S and $p(\bar{\tau}) \leftarrow p_1(\bar{\tau}_1), \ldots, p_m(\bar{\tau}_m)$ in TS, let U_n be the variable typing such that $U_n \vdash (\bar{t}, \bar{t}_1, \ldots, \bar{t}_m) : (\bar{\tau}, \bar{\tau}_1, \ldots, \bar{\tau}_m)$. Let

$$U = \bigcup_{n \in S} U_n\Theta.$$

Consider a pair of nodes n, n' in S such that n' is a child of n, and the equation $p(\bar{s}) = p(\bar{s}') \in Eq(S)$ corresponding to this pair (see Def. 5). Consider also the equation $p(\bar{\sigma}) = p(\bar{\sigma}') \in Eq(TS)$ corresponding to the pair n, n' in TS. Note that $U_n \vdash \bar{s} : \bar{\sigma}$ and $U_{n'} \vdash \bar{s}' : \bar{\sigma}'$. By Lemma 4, $U \vdash \bar{s} : \bar{\sigma}\Theta$ and $U \vdash \bar{s}' : \bar{\sigma}'\Theta$. Moreover, since $\Theta = MGU(Eq(TS))$, we have $\bar{\sigma}\Theta = \bar{\sigma}'\Theta$. Therefore $U \vdash p(\bar{s}) = p(\bar{s}')$ *Atom*. Since the same reasoning applies for any equation in $Eq(S)$, by Lemma 6, (θ, U) is a typed substitution.

Consider a node n'' in S with node atom a. Since $U_{n''} \vdash a$ *Atom*, by Lemma 4, $U \vdash a$ *Atom*. and by Lemma 5, $U \vdash a\theta$ *Atom*. Therefore P has subject reduction wrt. $\mathcal{Q}$. □

Example 4 *Figure 3 shows a proper skeleton and the corresponding* non-proper *type skeleton for the program in Ex. 3.*

In contrast, let $\mathcal{K}$ and $\mathcal{F}$ be as in Ex. 3, and $\mathcal{P} = \{\mathtt{app}_{\mathtt{list(U),list(U),list(U)}}, \mathtt{r}_{\mathtt{list(int)}}\}$. *Let P be the program shown in Fig. 4. The most general type of each clause is indicated as comment. Figure 5 shows a skeleton S and the corresponding type skeleton TS for P. A solution of $Eq(TS)$ is obtained by instantiating all parameters with* int.

Figure 3: A skeleton and the corresponding *non-proper* type skeleton for Ex. 3

```
app([],Ys,Ys).                  %app(list(U),list(U),list(U))
app([X|Xs],Ys,[X|Zs]) <-        %app(list(U),list(U),list(U))
  app(Xs,Ys,Zs).                %app(list(U),list(U),list(U))

r([1]).                         %r(list(int))

go <-
  app(Xs,[],Zs),                %app(list(int),list(int),list(int))
  r(Xs).                        %r(list(int))
```

Figure 4: A program used to illustrate type skeletons

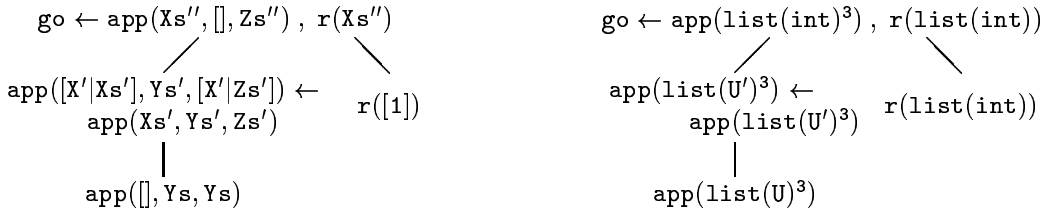

Figure 5: A skeleton and the corresponding type skeleton for Ex. 4

4 Conditions for Subject Reduction

By Thm. 8, a program has subject reduction whenever for each proper skeleton, the corresponding type skeleton is also proper. A very general sufficient condition consists in ensuring that *any* type skeleton is proper. We call this property **type unifiability**. Arguably, type unifiability is in the spirit of prescriptive typing, since subject reduction should be independent of the unifiability of terms, i.e., success or failure of the computation. However this view has been challenged in the context of higher-order logic programming [22].

We conjecture that both subject reduction and type unifiability are undecidable. Proving this is a topic for future work.

4.1 The Head Condition

The head condition is the standard way [13] of ensuring type unifiability.

Definition 10 *A clause* $C = p_{\bar{\tau}}(\bar{t}) \leftarrow B$ *fulfills the* **head condition** *if its most general type has the form* $(\bar{\tau}, \ldots)$.

Note that by the typing rules in Table 1, clearly the most general type of C must be $(\bar{\tau}, \ldots)\Theta$ for some type substitution Θ. Now the head condition states that the type of the head arguments must be the declared type of the predicate, or in other words, $\Theta\uparrow_{\bar{\tau}} = \emptyset$. It has been shown previously that typed programs fulfilling the head condition have operational subject reduction [13, Theorem 1.4.7]. By Thm. 7, this means that they have subject reduction.

4.2 Generalising the Head Condition

To reason about the existence of a solution for the equation set of a type skeleton, we give a sufficient condition for unifiability of a finite set of term equations.

Proposition 9 *Let* $E = \{l_1 = r_1, \ldots, l_m = r_m\}$ *be a set of oriented equations, and assume an order relation on the equations such that* $l_1 = r_1 \rightarrow l_2 = r_2$ *if* r_1 *and* l_2 *share a variable. E is unifiable if*

1. *for all* $1 \leq i < j \leq m$, r_i *and* r_j *have no variable in common, and*
2. *the graph of* $\rightarrow$ *is a partial order, and*
3. *for all* $i \in \{1, \ldots, m\}$, l_i *is an instance of* r_i.

In fact, the head condition ensures that $Eq(TS)$ meets the above conditions for any type skeleton TS. The equations in $Eq(TS)$ have the form $p(\bar{\tau}_a) = p(\bar{\tau}_h)$, where $\bar{\tau}_a$ is the type of an atom and $\bar{\tau}_h$ is the type of a head. Taking into account that the "type clauses" used for constructing the equations are renamed apart, all the head types (r.h.s.) have no parameter in common, the graph of $\rightarrow$ is a tree isomorphic to TS, and, by the head condition,$\bar{\tau}_a$ is an instance of $\bar{\tau}_h$. In the next subsection, we show that by decomposing each equation $p(\bar{\tau}_a) = p(\bar{\tau}_h)$, one can refine this condition.

4.3 Semi-generic Programs

In the head condition, all arguments of a predicate in clause head position are "generic" (i.e. their type is the declared type). One might say that all arguments are "head-generic". It is thus possible to generalise the head condition by partitioning the arguments of each predicate into those which stay head-generic and those which one requires to be generic for body atoms. The latter ones will be called *body-generic*. If we place the head-generic arguments of a clause head and the body-generic arguments of a clause body on the right

hand sides of the equations associated with a type skeleton, then Condition 3 in Prop. 9 is met.

The other two conditions can be obtained in various ways, more or less complex to verify (an analysis of the analogous problem of not being subject to occur check (NSTO) can be found in [6]). Taking into account the renaming of "type clauses", a relation between two equations amounts to a shared parameter between a generic argument (r.h.s.) and a non-generic argument (l.h.s.) of a clause. We propose here a condition on the clauses which implies that the equations of any skeleton can be ordered.

In the following, an atom written as $p(\bar{s}, \bar{t})$ means: $\bar{s}$ and $\bar{t}$ are the vectors of terms filling the head-generic and body-generic positions of p, respectively. The notation $p(\bar{\sigma}, \bar{\tau})$, where σ and τ are types, is defined analogously.

Definition 11 *Let* $\_ \vdash P$ *Program and* $\_ \vdash C$ *Clause where*

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

and Θ *the type substitution such that* $(\bar{\tau}_0, \bar{\sigma}_{m+1}, \bar{\sigma}_1, \bar{\tau}_1, \ldots, \bar{\sigma}_m, \bar{\tau}_m)\Theta$ *is the most general type of* C. *We call* C **semi-generic** *if*

1. *for all* $i, j \in \{0, \ldots, m\}$, $i \neq j$, $pars(\tau_i\Theta) \cap pars(\tau_j\Theta) = \emptyset$,
2. *for all* $i \in \{1, \ldots, m\}$, $pars(\bar{\sigma}_i) \cap \bigcup_{i \leq j \leq m} pars(\bar{\tau}_j) = \emptyset$,
3. *for all* $i \in \{0, \ldots, m\}$, $\tau_i\Theta = \tau_i$.

A query Q *is* **semi-generic** *if the clause* go $\leftarrow Q$ *is semi-generic. A program is* **semi-generic** *if each of its clauses is semi-generic.*

Note that semi-genericity has a strong resemblance with *nicely-modedness*, where head-generic corresponds to input, and body-generic corresponds to output. Nicely-modedness has been used, among other things, to show that programs are free from unification [1]. Semi-genericity serves a very similar purpose here. Note also that a typed program which fulfills the head condition is semi-generic, where all argument positions are head-generic.

The following theorem states subject reduction for semi-generic programs.

Theorem 10 *Every semi-generic program* P *has subject reduction wrt. the set of semi-generic queries.*

Proof: Let Q be a semi-generic query and TS a type skeleton corresponding to a skeleton for $P \cup \{\mathsf{go} \leftarrow Q\}$ with head go. Each equation in $Eq(TS)$ originates from a pair of nodes (n, n_i) where n is labelled $C = p(\bar{\tau}_0, \bar{\sigma}_{m+1}) \leftarrow p_1(\bar{\sigma}_1, \bar{\tau}_1), \ldots, p_m(\bar{\sigma}_m, \bar{\tau}_m)$ and n_i is labelled $C_i = p_i(\bar{\tau}'_i, \bar{\sigma}'_i) \leftarrow \ldots$, and the equation is $p_i(\bar{\tau}'_i, \bar{\sigma}'_i) = p_i(\bar{\sigma}_i, \bar{\tau}_i)$. Let Eq' be obtained from $Eq(TS)$ by replacing each such equation with the two equations $\bar{\sigma}_i = \bar{\tau}'_i$, $\bar{\sigma}'_i = \bar{\tau}_i$. Clearly Eq' and $Eq(TS)$ are equivalent. Because of the renaming of parameters for each node and since TS is a tree, it is possible to define an order $\dashrightarrow$ on the equations in Eq' such that for

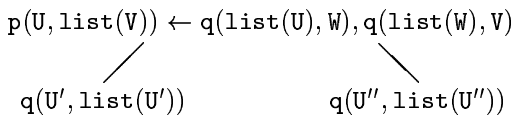

Figure 6: A type skeleton for a semi-generic program

each label C defined as above, $\bar{\sigma}_1 = \bar{\tau}'_1 \dashrightarrow E_1 \dashrightarrow \bar{\sigma}'_1 = \bar{\tau}_1 \dashrightarrow \ldots \dashrightarrow \bar{\sigma}_m = \bar{\tau}'_m \dashrightarrow E_m \dashrightarrow \bar{\sigma}'_m = \bar{\tau}_m$, where for each $i \in \{1, \ldots, m\}$, E_i denotes a sequence containing all equations e with $pars(e) \cap pars(C_i) \neq \emptyset$.

We show that Eq' fulfills the conditions of Prop. 9. By Def. 11 (1), Eq' fulfills condition 1. By Def. 11 (2), it follows that $\rightarrow$ is a subrelation of $\dashrightarrow$, and hence Eq' fulfills condition 2. By Def. 11 (3), Eq' fulfills condition 3.

Thus $Eq(TS)$ has a solution, so TS is proper, and so by Thm. 8, P has subject reduction wrt. the set of semi-generic queries. □

The following example shows that our condition extends the class of programs that have subject reduction.

Example 5 *Suppose $\mathcal{K}$ and $\mathcal{F}$ define lists as usual (see Ex. 3). Let $\mathcal{P} = \{\mathtt{p}_{\mathtt{U},\mathtt{V}}, \mathtt{q}_{\mathtt{U},\mathtt{V}}\}$ and assume that for* `p`, `q`, *the first argument is head-generic and the second argument is body-generic. Consider the following program.*

```
p(X,[Y]) <-                 %p(U,list(V)) <-
  q([X],Z), q([Z],Y).       %  q(list(U),W), q(list(W),V).
q(X,[X]).                   %q(U,list(U)).
```

This program is semi-generic. E.g. in the first type clause the terms in generic positions are `U`, `W`, `V`*; all generic arguments have the declared type (condition 3); they do not share a parameter (condition 1); no generic argument in the body shares a parameter with a non-generic position to the left of it (condition 2). A type skeleton is shown in Fig. 6.*

As another example, suppose now that $\mathcal{K}$ and $\mathcal{F}$ define list and integers, and consider the predicate `r`/2 *specified as* `r(1,[])`, `r(2,[[]])`, `r(3,[[[]]])` .... *Its obvious definition would be*

```
r(1,[]).
r(J,[X]) <-  r(J-1,X).
```

One can see that this program must violate the head condition no matter what the declared type of `r` *is. However, assuming declared type* (`int`, `list(U)`) *and letting the second argument be body-generic, the program is semi-generic.*

One can argue that in the second example, there is an intermingling of the typing and the computation, which contradicts the spirit of prescriptive typing. However, as we discuss in the next section, the situation is not always so clearcut.

5 What is the Use of the Head Condition?

The above results shed new light on the head condition. They allow us to view it as just one particularly simple condition guaranteeing type unifiability and consequently subject reduction and "well-typing" of the result, and hence a certain correctness of the program. This raises the question whether by generalising the condition, we have significantly enlarged the class of "well-typed" programs.

However, the head condition is also sometimes viewed as a condition inherent in the type system, or more specifically, an essential characteristic of *generic* polymorphism, as opposed to *ad-hoc* polymorphism. Generic polymorphism means that predicates are defined on an infinite number of types and that the definition is independent of a particular instance of the parameters. Ad-hoc polymorphism, often called *overloading* [20], means, e.g., to use the same symbol + for integer addition, matrix addition and list concatenation. Ad-hoc polymorphism is in fact forbidden by the head condition.

One way of reconciling ad-hoc polymorphism with the head condition is to enrich the type system so that types can be passed as parameters, and the definition of a predicate depends on these parameters [18]. Under such conditions, the head condition is regarded as natural.

So as a second, more general question, we discuss the legitimacy of the head condition briefly, since the answer justifies the interest in our first question.

In favour of the head condition, one could argue (1) that a program typed in this way does not compute types, but only propagates them; (2) that it allows for separate compilation since an imported predicate can be compiled without consulting its definition; and (3) that it disallows certain "unclean" programs [23].

In reality, these points are not, strictly speaking, fundamental arguments in favour of the head condition. Our generalisation does not necessarily imply a confusion between computation and typing (even if the result type does not depend on the result of a computation, it may be an instance of the declared type). Moreover, if the type declarations of the predicates are accompanied by declarations of the head- and body-generic arguments, separate compilation remains possible. Finally, Hanus [11] does not consider the head condition to be particularly natural, arguing that it is an important feature of logic programming that it allows for *lemma generation.*

We thus believe that the first question is, after all, relevant. So far, we have not been able to identify a "useful", non-contrived, example which clearly shows the interest in the class of semi-generically typed programs. The following example demonstrates the need for a generalisation, but also the insufficiency of the class defined in Def. 11.

Example 6 *Let* $\mathcal{K} = \{\mathtt{t}/1, \mathtt{int}/0\}$ *and*

$$\mathcal{F} = \{-1_{\to \mathtt{int}}, 0_{\to \mathtt{int}}, \ldots, \mathtt{c}_{\to \mathtt{t(U)}}, \mathtt{g}_{\mathtt{U} \to \mathtt{t(U)}}, \mathtt{f}_{\mathtt{t(t(U))} \to \mathtt{t(U)}}\}.$$

For all $i \geq 0$*, we have* $\_ \vdash \mathtt{g}^i(\mathtt{c}) : \mathtt{t}^{i+1}(\mathtt{U})$ *and* $\_ \vdash \mathtt{f}^i(\mathtt{g}^i(\mathtt{c})) : \mathtt{t}(\mathtt{U})$*. This means that the set* $\{\sigma \mid \exists s, t.\ s$ *is subterm of* t*,* $\_ \vdash s : \sigma$*,* $\_ \vdash t : \mathtt{t}(\mathtt{U})\}$ *is infinite, or in words, there are*

```
fgs1(I,Y) <-
  fs1(I,Y,I).

fs1(I,f(X),J) <-
  fs1(I-1,X,J).
fs1(0,X,J) <-
  gs1(J,X).

gs1(0,c).
gs1(J,g(X)) <-
  gs1(J-1,X).
```

```
fgs2(I,Y) <-
  fs2(I,Y,I).

fs2(I,f(X),J) <-
  fs2(I-1,X,J).
fs2(0,X,J) <-
  gs2(J,X,c).

gs2(0,X,X).
gs2(J,X,Y) <-
  gs2(J-1,X,g(Y)).
```

```
fgs3(I,X) <-
  fgs3_aux(I,c,X).

fgs3_aux(0,X,X).
fgs3_aux(I,X,f(Y)) <-
  fgs3_aux(I-1,g(X),Y).
```

Figure 7: Three potential solutions for Ex. 6

infinitely many types that a subterm of a term of type t(U) *can have. This property of the type* t(U) *is very unusual. In [27], a condition is considered (the* Reflexive Condition*) which rules out this situation.*

Now consider the predicate fgs/2 *specified as* $\mathtt{fgs}(i, \mathtt{f}^i(\mathtt{g}^i(\mathtt{c})))$ *(*$i \in \mathbb{N}$*). Figure 7 presents three potential definitions of this predicate. The declared types of the predicates are given by* $\mathcal{P} = \{\mathtt{fgs1}_{\mathtt{int,t(U)}}, \mathtt{gs1}_{\mathtt{int,t(U)}}, \mathtt{fgs2}_{\mathtt{int,t(U)}}, \mathtt{fgs3}_{\mathtt{int,t(U)}}, \mathtt{fs1}_{\mathtt{int,t(U),int}}, \mathtt{fs2}_{\mathtt{int,t(U),int}}, \mathtt{gs2}_{\mathtt{int,t(U),t(V)}}, \mathtt{fgs3\_aux}_{\mathtt{int,t(U),t(U)}}\}$*. The first solution is the most straightforward one, but its last clause does not fulfill the head condition. For the second solution, the fact clause* gs2(0,x,x). *does not fulfill the head condition. The third program fulfills the head condition but is the least obvious solution.*

For the above example, the head condition is a real restriction. It prevents a solution using the most obvious algorithm, which is certainly a drawback of any type system. We suspected initially that it would be impossible to write a program fulfilling the specification of fgs without violating the head condition.

Now it would of course be interesting to see if the first two programs, which violate the head condition, are semi-generic. Unfortunately, they are not. We explain this for the first program. The second position of gs1 must be body-generic because of the second clause for gs1. This implies that the second position of fs1 must also be body-generic because of the second clause for fs1 (otherwise there would be two generic positions with a common parameter). That however is unacceptable for the first clause of fs1 (X has type t(t(U)), instance of t(U)).

It can however be observed that both programs have subject reduction wrt. the queries $\mathtt{fgs}j(i, \mathtt{Y})$ for $i \in \mathbb{N}$ and $j = 1, 2$. In fact for these queries all type skeletons are proper, but it can be seen that the equations associated with the type skeletons cannot be ordered. This shows that the condition of semi-genericity is still too restrictive.

There is a perfect analogy between gs1 and r in Ex. 5.

To conclude this section, note that our solution to the problem in Ex. 6 uses *polymorphic recursion*, a concept previously discussed for functional programming [15]: In the recursive clause for fgs3_aux, the arguments of the recursive call have type $(\mathtt{int}, \mathtt{t}(\mathtt{t}(\mathtt{U})), \mathtt{t}(\mathtt{t}(\mathtt{U})))$,

while the arguments of the clause head have type $(\texttt{int}, \texttt{t(U)}, \texttt{t(U)})$. If we wrote a function corresponding to `fgs3_aux` in Miranda [31] or ML, the type checker could not infer its type, since it assumes that recursion is monomorphic, i.e., the type of a recursive call is identical to the type of the "head". In Miranda, this problem can be overcome by providing a type declaration, while in ML, the function will definitely be rejected. This limitation of the ML type system, or alternatively, the ML type checker, has been studied by Kahrs [14].

There is a certain duality between the head condition and monomorphic recursion. When trying to find a solution to our problem, we found that we either had to violate the head condition or use polymorphic recursion. For example, in the recursive clause for `gs1`, the arguments of the recursive call have type $(\texttt{int}, \texttt{t(U)})$, while the arguments of the clause head have type $(\texttt{int}, \texttt{t(t(U))})$, which is in a way the reverse of the situation for `fgs3_aux`. Note that this implies a violation of the head condition for *any* declared type of `gs1`. It would be interesting to investigate this duality further.

6 Conclusion

In this paper we redefined the notion of *subject reduction* by using derivation trees, leading to a proof-theoretic view of typing in logic programming. We showed that this new notion is equivalent to the operational one (Thm. 7).

We introduced *type skeletons*, obtained from skeletons by replacing terms with their types. We showed that a program has subject reduction if and only if for each proper skeleton, the type skeleton is also proper. Apart from clarifying the motivations of the head condition, it has several potential applications:

- It facilitates studying the semantics of typed programs by simplifying its formulation in comparison to other works (e.g. [16]). Lifting the notions of derivation tree and skeleton on the level of types can help formulate proof-theoretic and operational semantics, just as this has been done for untyped logic programming with the classical trees [3, 6, 9].
- The approach may enhance program analysis based on abstract interpretation. Proper type skeletons could also be modelled by fixpoint operators [4, 5, 10]. Abstract interpretation for prescriptively typed programs has been studied by [25, 27], and it has been pointed out that the head condition is essential for ensuring that the abstract semantics of a program is finite, which is crucial for the termination of an analysis. It would be interesting to investigate the impact of more general conditions.
- This "proof-theoretic" approach to typing could also be applied for synthesis of typed programs. In [29], the authors propose the automatic generation of lemmas, using synthesis techniques based on resolution. It is interesting to observe that the generated lemmas meet the head condition, which our approach seems to be able to justify and even generalise.
- The approach may help in combining *prescriptive* and *descriptive* approaches to typing. The latter are usually based on partial correctness properties. Descriptive type

systems satisfy certain criteria of type-correctness [7], but subject reduction is difficult to consider in such systems. Our approach is a step towards potential combinations of different approaches.

We have presented a condition for type unifiability which is a refinement of the head condition (Thm. 10). Several observations arise from this:

- Definition 11 is decidable. If the partitioning of the arguments is given, it can be verified in polynomial time. Otherwise, finding a partitioning is exponential in the number of argument positions.
- The refinement has a cost: subject reduction does not hold for arbitrary (typed) queries. The head condition, by its name, only restricts the clause heads, whereas our generalisation also restricts the queries, and hence the ways in which a program can be used.
- As we have seen, the proposed refinement may not be sufficient. Several approaches can be used to introduce further refinements based on abstract interpretation or on properties of sets of equations. Since any sufficient condition for type unifiability contains at least an NSTO condition, one could also benefit from the refinements proposed for the NSTO check [6]. Such further refined conditions should, in particular, be fulfilled by all solutions of Ex. 6.

We have also studied *operational* subject reduction for type systems with subtyping [26]. As future work, we want to integrate that work with the *proof-theoretic* view of subject reduction of this paper. Also, we want to prove the undecidability of subject reduction and type unifiability, and design more refined tests for type unifiability.

Acknowledgements

We thank François Fages for interesting discussions. Jan-Georg Smaus was supported by an ERCIM fellowship.

References

[1] 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*, volume 711 of *LNCS*, pages 1–19. Springer-Verlag, 1993.

[2] 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.

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The *s*-semantics approach: theory and applications. *Journal of Logic Programming*, 19/20:149–197, 1991.

[4] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, volume 1192 of *LNCS*, pages 22–50. Springer-Verlag, 1996.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[6] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* MIT Press, 1993.

[7] P. Deransart and J. Małuszyński. Towards soft typing for CLP. In F. Fages, editor, *JICSLP'98 Post-Conference Workshop on Types for Constraint Logic Programming.* École Normale Supérieure, 1998. Available at `http://discipl.inria.fr/TCLP98/`.

[8] P. Deransart and J.-G. Smaus. Well-typed logic programs are not wrong. In H. Kuchen and K. Ueda, editors, *Proceedings of the Fifth International Symposium on Functional and Logic Programming*, LNCS. Springer-Verlag, 2001.

[9] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[10] R. Giacobazzi, S. K. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *Journal of Logic Programming*, 25(3):191–247, 1995.

[11] M. Hanus. *Logic Programming with Type Specifications*, chapter 3, pages 91–140. In [24].

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

[13] P. M. Hill and R. W. Topor. *A Semantics for Typed Logic Programs*, chapter 1, pages 1–61. In [24].

[14] S. Kahrs. Limits of ML-definability. In H. Kuchen and S. D. Swierstra, editors, *Proceedings of the 8th Symposium on Programming Language Implementations and Logic Programming*, volume 1140 of *LNCS*, pages 17–31. Springer-Verlag, 1996.

[15] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

[16] 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.

[17] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[18] P. Louvet and O. Ridoux. Parametric polymorphism for Typed Prolog and λProlog. In H. Kuchen and S. D. Swierstra, editors, *Proceedings of the 8th Symposium on Programming Language Implementations and Logic Programming*, volume 1140 of *LNCS*, pages 47–61. Springer-Verlag, 1996.

[19] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[20] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

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

[22] G. Nadathur and F. Pfenning. *Types in Higher-Order Logic Programming*, chapter 9, pages 245–283. In [24].

[23] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

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

[25] O. Ridoux, P. Boizumault, and F. Malésieux. Typed static analysis: Application to groundness analysis of Prolog and λProlog. In A. Middeldorp and T. Sato, editors, *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *LNCS*, pages 267–283. Springer-Verlag, 1999.

[26] 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*, volume 1974 of *LNCS*. Springer-Verlag, 2000.

[27] J.-G. Smaus, P. M. Hill, and A. M. King. Mode analysis domains for typed logic programs. In A. Bossi, editor, *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation*, volume 1817 of *LNCS*, pages 83–102, 2000.

[28] 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.

[29] P. Tarau, K. De Bosschere, and B. Demoen. On Delphi lemmas and other memoing techniques for deterministic logic programs. *Journal of Logic Programming*, 30(2):145–163, 1997.

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

[31] Simon Thompson. *Miranda: The Craft of Functional Programming.* Addison-Wesley, 1995.

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