Subject Reduction of Logic Programs as Proof-Theoretic Property

Pierre Deransart[*] and Jan–Georg Smaus[†]

February 6, 2002

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 study more general conditions, thus reestablishing a certain symmetry between heads and body atoms. The underlying idea is that in a derivation, terms should only be unified if their types are unifiable. We discuss possible implications of our results.

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 [Mil78, MO84]

Well-typed programs cannot go wrong.

Adopting the terminology from the theory of the λ-calculus [Tho91], 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"

[*]INRIA-Rocquencourt, BP105, 78153 Le Chesnay Cedex, France, `pierre.deransart@inria.fr`

[†]Institut für Informatik, Universität Freiburg, 79110 Freiburg, Germany, `smaus@informatik.uni-freiburg.de`

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 [Llo87]. In this article, we show that it is also a property of the proof-theoretic semantics based on *derivation trees*. More precisely, we show that using "well-typed" clauses, only "well-typed" derivation trees can be constructed. We might turn this into the new slogan

Well-typed programs *are* not wrong.

The type system we consider here is a system with parametric polymorphism. In such a system, predicates etc. can have a type that contains "variables" (we use the term *parameters*). For example, a predicate of type `list(U)` can be applied to an argument of type `list(int)` or `list(string)` and so forth.

The *head condition*, also called *definitional genericity* [LR91], is a condition on the program (clauses) [HT92]. It is usually considered to be crucial for subject reduction, and one might have the impression that it is a necessary condition. It 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. Our proof-theoretic view of subject reduction leads us to questioning this distinction between "definitional" occurrences (clause heads) and "applied" occurrences (body atoms) of a predicate. The second objective of this article is thus to look for more general conditions. We argue that conditions for subject reduction should be based on *type unifiability*, meaning that the types of terms that might be unified in a derivation are themselves unifiable. In particular, we will present a decidable condition for subject reduction which reestablishes a certain symmetry between the different occurrences. Thus the class of programs for which subject reduction is guaranteed is enlarged.

This article is organised as follows. Section 2 contains some preliminaries. Section 3 introduces our proof-theoretic notion of subject reduction, shows that it is equivalent to the usual operational one, and that it is undecidable. Section 4 gives sufficient conditions for subject reduction, and in particular, a generalisation of the head condition. Section 5 is a discussion of our results.

This article is based on an earlier conference paper [DS01]. The present article contains the proofs and a more extended discussion, e.g. some comparisons with functional programming. Concerning the technical content, the results on undecidability are new.

2 Preliminaries

We assume familiarity with the standard concepts of logic programming [Llo87]. 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 $\theta\lceil_o$, 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

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

the context. A **query** is a sequence of atoms. A query Q' is **derived from** a query Q, denoted $Q \rightsquigarrow Q'$, if $Q = a_1, \ldots, a_m$, and $h \leftarrow B$ is a clause (in a program usually clear from the context) such that h and some a_k are unifiable with MGU θ, and $Q' = (a_1, \ldots, a_{k-1}, B, a_{k+1}, \ldots, a_m)\theta$. A **derivation** $Q \rightsquigarrow^* Q'$ is defined in the usual way. Given a program P, the **immediate consequence operator** T_P is defined by $T_P(M) = \{h\theta \mid h \leftarrow a_1, \ldots, a_m \in P,\ a_1\theta, \ldots, a_m\theta \in M\}$. Note that unlike in similar definitions in the literature, θ is not constrained to be grounding or most general or anything of the like.

2.1 Derivation Trees

A key element of this work is the proof-theoretic semantics of logic programs based on derivation trees [DM93]. We recall some important notions and basic results.

Definition 2.1 An **instance name** of a clause C is a pair of the form $\langle C, \theta \rangle$, where θ is a substitution.

Definition 2.2 Let P be a program. A **derivation tree** for P is a labelled ordered tree [DM93] such that:

1. Each leaf node is labelled by $\perp$ 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 by $\perp$, or $\langle h' \leftarrow B, \theta' \rangle$ where $h'\theta' = a_i\theta$.

Nodes labelled $\perp$ 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 2.3 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 $\perp$, 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 $head(T)$.

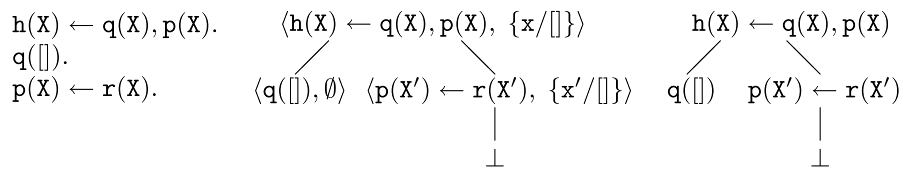
Derivation trees are obtained by grafting instances of clauses of a program. To describe this construction in a general way, we define the following concept.

Definition 2.4 Let P be a program. A **skeleton** for P is a labelled ordered tree such that:

1. Each leaf node is labelled by $\perp$ 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 by $\perp$, or $h' \leftarrow B$ where $Rel(h') = Rel(a_i)$.

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



```
h(X) ← q(X), p(X).
q([]).
p(X) ← r(X).
```

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

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 2.5 Let S be a skeleton. We define

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

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 2.6 [DM93, Prop. 2.1] Let S be a skeleton. A derivation tree based on S exists if and only if S is proper.

Theorem 2.7 [DM93, 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 $\langle C, \theta\lceil_C \rangle$. 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 2.8 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 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 2.9 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. $Q \leadsto^* Q'$ if and only if Q' is the sequence of node atoms of incomplete nodes of a most general derivation tree for $P \cup \{\texttt{go} \leftarrow Q\}$ with head go, visited left to right.

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 [HL94] and Mercury [SHC96].

Table 1: Rules defining a typed language

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

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*[3] [HT92]: $pars(\tau_1,\ldots,\tau_m)\subseteq pars(\tau)$; 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 article, 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 Γ. In rules *(Program)* and *(Queryset)*, it is tacitly assumed that the clauses, resp. queries, are variable-disjoint. This justifies using the notation $\_\vdash\ldots$ for "there exists Γ such that $\Gamma\vdash\ldots$". 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) [Tho91]. 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

[3]We shall discuss this condition after Lemma 2.16.

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ \Gamma\vdash\bar{t}:\bar{\tau}\end{matrix}}{\Gamma\vdash p(\bar{t})\ \mathit{Atom}} \quad \dfrac{\dfrac{\begin{matrix}\vdots\\ \Gamma\vdash\bar{t}_1:\bar{\tau}_1\end{matrix}}{\Gamma\vdash p_1(\bar{t}_1)\ \mathit{Atom}}\ \cdots\ \dfrac{\begin{matrix}\vdots\\ \Gamma\vdash\bar{t}_m:\bar{\tau}_m\end{matrix}}{\Gamma\vdash p_m(\bar{t}_m)\ \mathit{Atom}}}{\Gamma\vdash p_1(\bar{t}_1),\ldots,p_m(\bar{t}_m)\ \mathit{Query}}}{\Gamma\vdash p(\bar{t})\leftarrow p_1(\bar{t}_1),\ldots,p_m(\bar{t}_m)\ \mathit{Clause}}$$

Figure 2: Proving a type 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* [Bei95].

Definition 2.10 Consider a judgement $\Gamma \vdash p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_m(\bar{t}_m)$ *Clause*, and a proof of this judgement containing judgements $\Gamma \vdash \bar{t} : \bar{\tau}$, $\Gamma \vdash \bar{t}_1 : \bar{\tau}_1$, ..., $\Gamma \vdash \bar{t}_m : \bar{\tau}_m$ (see Fig. 2) such that $(\bar{\tau}, \bar{\tau}_1, \ldots, \bar{\tau}_m)$ is most general wrt. all such proofs for all possible choices of Γ. We call $(\bar{\tau}, \bar{\tau}_1, \ldots, \bar{\tau}_m)$ the **most general type** of $p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_m(\bar{t}_m)$.

Example 2.11 Consider function $\mathtt{nil}_{\to\mathtt{list(U)}}$ and clause $C \equiv \mathtt{p} \leftarrow \mathtt{X} = \mathtt{nil}$. For $\Gamma = \{\mathtt{X} : \mathtt{list(V)}\}$, the judgement $\Gamma \vdash C$ *Clause* can be proven using the judgements $\Gamma \vdash \mathtt{X} : \mathtt{list(V)}$ and $\Gamma \vdash \mathtt{nil} : \mathtt{list(V)}$, and the vector $(\mathtt{list(V)}, \mathtt{list(V)})$ is most general wrt. all such proofs and all choices of Γ. Thus $((), (\mathtt{list(V)}, \mathtt{list(V)}))$ is the most general type of C.

For several purposes, it is useful to consider programs obtained from usual typed programs by replacing each term by its (most general) type. This is based on the previous definition.

Definition 2.12 Given a clause $C = p(\bar{t}) \leftarrow p_1(\bar{t}_1), \ldots, p_m(\bar{t}_m)$ with most general type $(\bar{\tau}, \bar{\tau}_1, \ldots, \bar{\tau}_m)$, we call $p(\bar{\tau}) \leftarrow p_1(\bar{\tau}_1), \ldots, p_m(\bar{\tau}_m)$ the **type clause corresponding to** C. Given a program P, the **type program corresponding to** P is obtained by replacing each clause with the corresponding type clause.

The following is a standard concept for typed logic programming.

Definition 2.13 If $\Gamma \vdash x_1 = t_1, \ldots, x_m = t_m$ *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 $(\{x_1/t_1, \ldots, x_m/t_m\}, \Gamma)$ is a **typed (term) substitution**.

We shall need three fundamental lemmas introduced in [HT92].[4]

Lemma 2.14 [HT92, Lemma 1.2.8] Let $\Gamma \subseteq \Gamma'$ be variable typings and Θ a type substitution. If $\Gamma \vdash t : \sigma$, then $\Gamma'\Theta \vdash t : \sigma\Theta$. Moreover, if $\Gamma \vdash A$ *Atom* then $\Gamma'\Theta \vdash A$ *Atom*, and likewise for queries and clauses.

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

[4]Note that some results in [HT92] 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.

Now consider $\Gamma \vdash f_{\tau_1 \ldots \tau_m \to \tau}(t_1, \ldots, t_m) : \sigma$ 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 $\Gamma \vdash t_i : \tau_i\Theta'$ for each $i \in \{1, \ldots, m\}$. By the inductive hypothesis, $\Gamma'\Theta \vdash t_i : \tau_i\Theta'\Theta$ for each $i \in \{1, \ldots, m\}$, and hence by Rule *(Func)*, $\Gamma'\Theta \vdash f_{\tau_1 \ldots \tau_m \to \tau}(t_1, \ldots, t_m) : \tau\Theta'\Theta$.

The rest of the proof is now trivial. □

Lemma 2.15 [HT92, Lemma 1.4.2] Let (θ, Γ) be a typed substitution. If $\Gamma \vdash t : \sigma$ then $\Gamma \vdash t\theta : \sigma$. Moreover, if $\Gamma \vdash A\ Atom$ then $\Gamma \vdash A\theta\ Atom$, and likewise for queries and clauses.

Proof: The proof is by structural induction. For the base case, suppose $\Gamma \vdash x : \sigma$ 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, $\Gamma \vdash t : \sigma$.

Now consider $\Gamma \vdash f_{\tau_1 \ldots \tau_m \to \tau}(t_1, \ldots, t_m) : \sigma$ 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 $\Gamma \vdash t_i : \tau_i\Theta'$ for each $i \in \{1, \ldots, m\}$. By the inductive hypothesis, $\Gamma \vdash t_i\theta : \tau_i\Theta'$ for each $i \in \{1, \ldots, m\}$, and hence by Rule *(Func)*, $\Gamma \vdash f_{\tau_1 \ldots \tau_m \to \tau}(t_1, \ldots, t_m)\theta : \tau\Theta'$.

The rest of the proof is now trivial. □

Lemma 2.16 [HT92, Thm. 1.4.1] Let E be a set (conjunction) of equations such that for some variable typing Γ, we have $\Gamma \vdash E\ Query$. Suppose θ is an MGU of E. Then (θ, Γ) is a typed substitution.

Proof: We show that the result is true when θ is computed using the well-known Martelli-Montanari algorithm [MM82] which works by transforming a set of equations into a set of the form required in the definition of a typed substitution. Let $E_0 = E$. Only the following two transformations are considered here. The others are trivial.

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 $\Gamma \vdash E_k\ Query$ and E_{k+1} is obtained by either of the above transformations, then $\Gamma \vdash E_{k+1}\ Query$. For (1), this follows from Lemma 2.15.

For (2), suppose $\Gamma \vdash E_k\ Query$ 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 $\Gamma \vdash f(t_1, \ldots, t_m) =_{u,u} f(s_1, \ldots, s_m)\ Atom$, and hence by Rule *(Atom)*, $\Gamma \vdash f(t_1, \ldots, t_m) : u\Theta$ and $\Gamma \vdash f(s_1, \ldots, s_m) : u\Theta$ 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 $\Gamma \vdash t_i : \tau_i\Theta_t$ and $\Gamma \vdash s_i : \tau_i\Theta_s$. Since $pars(\tau_i) \subseteq pars(\tau)$, it follows that $\tau_i\Theta_t = \tau_i\Theta_s$. Therefore $\Gamma \vdash t_i = s_i\ Atom$, and so $\Gamma \vdash E_{k+1}\ Query$. □

Note how in the above proof, the transparency condition is essential to ensure that subarguments in corresponding positions have identical types, so that each variable can only become instantiated to a term of its own type. Hill gives examples explaining this [Hil93]. The condition was ignored in [MO84].

From the point of view of functional programming, the transparency condition may seem very restrictive: for example, it would be ludicrous to forbid a function that computes the length of a list, which would naturally have the type `list(U)` $\rightarrow$ `int` and hence violate the transparency condition. However, one must bear in mind that in logic programming, the word "function" has a meaning different from that of functional programming. A *function* in logic programming is sometimes called *term constructor* in functional programming, and a *term* in logic programming is sometimes called *constructor term* or *data term*.

In logic programming, unification of terms and thus instantiation of logical variables is at the heart of the computation mechanism, and as we have seen in the proof above, it is crucial that terms that are unified have the same type. This is comparable in functional programming to the matching of an argument pattern in a function definition against the actual arguments of a function call. For example, `length` could be defined in Miranda [Tho95] as

```
fun length []      = 0
  | length (x::xs) = 1 + length (xs)
```

Note that a pattern such as (`x` :: `xs`) is required to be a constructor term, and that for term constructors, transparency is required. Thus the matter is not so different in functional programming after all.

Nevertheless, it has been observed that the transparency condition can be quite restrictive, in particular in the context of meta-programming [Hil93, HL89].

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 is undecidable.

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 3.1 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 Γ such that for each node atom a of T, $\Gamma \vdash a$ *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 $\mathcal{Q} = \{Q \mid \_ \vdash Q \textit{ Query}\}$. The following theorem states a certain equivalence between the two notions.

Theorem 3.2 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.

Proof: The first statement is a straightforward consequence of Prop. 2.9 (2).

For the second statement, assume $\Gamma \vdash Q$ *Query*, let $\xi = Q \rightsquigarrow^* Q'$, and T be the derivation tree for $P \cup \{\mathtt{go} \leftarrow Q\}$ corresponding to ξ (by Prop. 2.9 (2)).

By hypothesis, there exists a variable typing Γ' such that for each *incomplete* node n of T, we have $\Gamma' \vdash atom(n)$ *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 $\tilde{\xi} = \mathtt{go} \rightsquigarrow \tilde{Q}'$ be the derivation for $\tilde{P} \cup \{\mathtt{go} \leftarrow Q, Q\}$ 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 Γ' such that $\Gamma' \vdash \tilde{Q}'$ *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 $\Gamma' \vdash a$ *Atom*. Since the choice of Q was arbitrary, P has subject reduction. $\square$

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 3.3 Let $\mathcal{K} = \{\mathtt{int}/0, \mathtt{list}/1\}$, $\mathcal{F} = \{\mathtt{0}_{\to\mathtt{int}}, \mathtt{1}_{\to\mathtt{int}}, \ldots, \mathtt{nil}_{\to\mathtt{list(U)}}, \mathtt{cons}_{\mathtt{U},\mathtt{list(U)}\to\mathtt{list(U)}}\}$, and $\mathcal{P} = \{\mathtt{p}_{\mathtt{list(int)}}, \mathtt{r}_{\mathtt{list(U)}}\}$. Throughout the article, we will use the usual (e.g., Prolog) list notation [_|_]. Let P be

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

For each derivation $\mathtt{p(X)} \rightsquigarrow^* Q'$, we have $Q' = \mathtt{p(Y)}$ or $Q' = \mathtt{r(Y)}$ for some $\mathtt{Y} \in \mathcal{V}$, and so $\{\mathtt{Y} : \mathtt{list(int)}\} \vdash \mathtt{p(Y)}$ *Query* or $\{\mathtt{Y} : \mathtt{list(U)}\} \vdash \mathtt{r(Y)}$ *Query*. Therefore P has operational subject reduction wrt. $\{\mathtt{p(X)}\}$. Yet the derivation trees for P have heads $\mathtt{p(Y)}$, $\mathtt{p([Y])}$, $\mathtt{p([[Y]])}$ etc., and $\_ \nvdash \mathtt{p([[Y]])}$ *Query*.

3.2 Undecidability of Subject Reduction

We show that it is undecidable whether a program has subject reduction. Note that the question of decidability arises because we assume a standard type system for logic programs except that we do not initially impose the head condition. We do not believe that the question has been studied in this form before. Usually, a particular type system will be designed taking several considerations into account, and one of them is that the system should have subject reduction, which has to be proven in each particular case.

To show undecidability, we encode a Turing machine (TM) as a typed program in such a way that the TM terminates if and only if the program does not have subject reduction. This construction combines two programs.

The first encodes the TM itself and has two essential properties:

- The TM terminates iff the program has a complete proper skeleton;
- the program trivially has subject reduction.

The second program has the following two essential properties:

- All its skeletons are trivially proper;
- a skeleton for the program is "well-typed" if and only if the skeleton is *not* complete.

By combining the two programs, we obtain a program which has a complete skeleton, and hence does not have subject reduction, if and only if the TM terminates. Thereby it follows that if subject reduction was decidable, then the halting problem for the TM would be decidable. We now give the details.

Recall that a TM is a tuple $(K, \Sigma, \delta, \mathsf{s})$ where

- K is of a finite set of states not containing the halt state h,
- Σ is an alphabet containing the blank symbol $\#$ but not containing the symbols L and R,
- the transition function δ is a function from $(K \times \Sigma)$ to $(K \cup \{\mathsf{h}\}) \times (\Sigma \cup \{\mathsf{L}, \mathsf{R}\})$,
- $\mathsf{s} \in K$ is the initial state.

Here L, R stand for left, right move, respectively. A *configuration* is a member of

$$(K \cup \{\mathsf{h}\}) \times \Sigma^* \times \Sigma \times (\Sigma^* \cdot (\Sigma \setminus \{\#\}) \cup \{\epsilon\}).$$

Note that we only consider computations starting with an empty tape, but it is well-known that this is no loss of generality as far as decidability is concerned. See [LP81] for further details.

To encode a TM M as a typed program P_M, we define $\mathcal{K} = \{\mathtt{symbol}/0, \mathtt{list}/1\}$ and $\mathcal{F} \supset \{\sigma_{\to\mathtt{symbol}} \mid \sigma \in \Sigma \cup \{\mathsf{L}, \mathsf{R}\}\}$ in addition to the usual list functions. Moreover, we define $\mathcal{P} = \{q_{\mathtt{list(symbol),symbol,list(symbol)}} \mid q \in K \cup \{\mathsf{h}\}\}$. A configuration of the TM is encoded as an atom $q(l, \sigma, r)$, where q is the state, l is a list representing the part of the tape to the left of the head, σ is the symbol under the head, and r is a list representing the part of the tape to the right of the head. The list l represents the symbols on the tape in reverse order, so that the head of l represents the square next to the head, as for r.

The program clauses model the transition function δ as shown in Table 2. The asymmetry between the cases for L and R is due to the fact that the tape is open-ended only to the right.

The following proposition states the essential properties of P_M. The first statement follows immediately from the construction. The second can be seen in some straightforward ad-hoc way.

Proposition 3.4 *Let M be a Turing machine and P_M be the program as defined above. M halts when started in configuration $(\mathsf{s}, \epsilon, \#, \epsilon)$ if and only if P_M has a complete derivation tree whose skeleton has head $\mathsf{s}([], \#, [])$.*

Moreover, P_M has subject reduction wrt. $\{\mathsf{s}([], \#, [])\}$.

We now construct the second program P'. Let $\mathcal{K} = \{\mathtt{colour}, \mathtt{shape}\}$, $\mathcal{F} = \{\mathtt{red}_{\to\mathtt{colour}}, \mathtt{square}_{\to\mathtt{shape}}\}$, $\mathcal{P} = \{\mathtt{s}_{\mathtt{colour}}, \mathtt{q}_{\mathtt{U}}, \mathtt{h}_{\mathtt{U}}\}$, and the clauses be as follows

```
s(X) <- q(X).     q(X) <- q(X).     q(X) <- h(X).     h(square).
```

Table 2: The modelling of δ as a logic program

For each q, σ such that $\delta(q,\sigma) = (q', \sigma')$ with $\sigma' \in \Sigma$:	$q(\mathtt{L}, \sigma, \mathtt{R}) \leftarrow q'(\mathtt{L}, \sigma', \mathtt{R})$.
For each q, σ such that $\delta(q,\sigma) = (q', \mathsf{L})$:	$q([\mathtt{S}\vert\mathtt{L}], \sigma, \mathtt{R}) \leftarrow q'(\mathtt{L}, \mathtt{S}, [\sigma\vert\mathtt{R}])$.
For each q, σ such that $\delta(q,\sigma) = (q', \mathsf{R})$:	$q(\mathtt{L}, \sigma, [\mathtt{S}\vert\mathtt{R}]) \leftarrow q'([\sigma\vert\mathtt{L}], \mathtt{S}, \mathtt{R})$.
	$q(\mathtt{L}, \sigma, []) \leftarrow q'([\sigma\vert\mathtt{L}], \#, [])$.
For the halt state:	$\mathtt{h}(\mathtt{L}, \mathtt{S}, \mathtt{R})$.

In P', a derivation tree becomes ill-typed as soon as one adds the leaf $\mathtt{h}(\mathtt{square})$, since this forces the head to be $\mathtt{s}(\mathtt{square})$. This is stated in the following proposition.

Proposition 3.5 Let T be a most general derivation tree for P' whose skeleton has head $\mathtt{s}(\mathtt{X})$. Then T is complete if and only if there does not exist a variable typing Γ such that for each node atom a of T, we have $\Gamma \vdash a\ Atom$.

Moreover, any skeleton for P' is proper.

We now combine the two programs to a program P_M^+ in a straightforward way. We associate $\mathtt{s}$ in P' with $\mathtt{s}$ in P_M, $\mathtt{q}$ in P' with any predicate q in P_M such that $q \in K \setminus \{\mathtt{s}\}$, and $\mathtt{h}$ in P' with $\mathtt{h}$ in P_M. For the combined program, we define $\mathcal{K}$ and $\mathcal{F}$ simply to be the union of the respective sets above, and

$$\begin{aligned}\mathcal{P} = \ & \{\mathtt{s}_{\mathtt{list(symbol)},\mathtt{symbol},\mathtt{list(symbol)},\mathtt{colour}}\} \cup \\ & \{q_{\mathtt{list(symbol)},\mathtt{symbol},\mathtt{list(symbol)},\mathtt{U}} \mid q \in (K \setminus \{\mathtt{s}\}) \cup \{\mathtt{h}\}\}\end{aligned}$$

The program clauses are obtained from the *non-fact* clauses in Table 2 by adding $\mathtt{X}$ (assuming this is a fresh variable) as last argument to each atom. Moreover, the fact clause is replaced with $\mathtt{h}(\mathtt{L}, \mathtt{S}, \mathtt{R}, \mathtt{square})$.

Any skeleton in P_M corresponds in an obvious way to a skeleton in P_M^+.

Theorem 3.6 It is undecidable whether a program P has subject reduction for a set of queries $\mathcal{Q}$.

Proof: Assume that it is decidable whether a program P has subject reduction for a set of queries $\mathcal{Q}$. Consider a TM M and the corresponding program P_M^+ as defined above. By Propositions 3.4 and 3.5, the following three statements are equivalent:

- M halts when started in configuration $(\mathtt{s}, \epsilon, \#, \epsilon)$;
- P_M^+ has a complete derivation tree whose skeleton has head $\mathtt{s}([], \#, [], \mathtt{X})$;
- P_M^+ does not have subject reduction wrt. $\{\mathtt{s}([], \#, [], \mathtt{X})\}$.

By the assumption that subject reduction is decidable, it thus follows that the halting problem for M is decidable, which is a contradiction. $\square$

4 Type Unifiability

In this section, we argue that the unifiability of types is the key to ensuring subject reduction.

Recall Def. 2.12. By obvious analogy, we can treat type clauses and programs like ordinary clauses and programs. We call **type skeleton** (resp., **type derivation tree**, **type equation set**) a skeleton (resp., derivation tree, equation set) that is obtained from a type program.

In particular, given a skeleton S for a program P, the **type skeleton corresponding to** S is obtained by replacing each node label C_n in S with the type clause corresponding to C_n.[5] Clearly, this type skeleton is a skeleton of the type program corresponding to P.

4.1 Weak Type Unifiability

Definition 4.1 Let $\_ \vdash P\ \mathit{Program}$ and $\_ \vdash \mathcal{Q}\ \mathit{Queryset}$. We say that P has **weak type unifiability wrt.** $\mathcal{Q}$ if for each proper skeleton S of $P \cup \{\mathtt{go} \leftarrow Q\}$ with head $\mathtt{go}$, where $Q \in \mathcal{Q}$, the *type* skeleton corresponding to S is proper.

Weak type unifiability is a sufficient condition for subject reduction.

Theorem 4.2 Let $\_ \vdash P\ \mathit{Program}$ and $\_ \vdash \mathcal{Q}\ \mathit{Queryset}$. If P has weak type unifiability wrt. $\mathcal{Q}$, then P has subject reduction wrt. $\mathcal{Q}$.

Proof: Let S be an arbitrary proper skeleton for $P \cup \{\mathtt{go} \leftarrow Q\}$ with head $\mathtt{go}$, where $Q \in \mathcal{Q}$, and let TS be the corresponding type skeleton. Let $\theta = MGU(Eq(S))$ and $\Theta = MGU(Eq(TS))$. For each node n in S, labelled $p(\bar{t}) \leftarrow p_1(\bar{t}_1), \dots, p_m(\bar{t}_m)$ in S and $p(\bar{\tau}) \leftarrow p_1(\bar{\tau}_1), \dots, p_m(\bar{\tau}_m)$ in TS, let Γ_n be the variable typing such that $\Gamma_n \vdash (\bar{t}, \bar{t}_1, \dots, \bar{t}_m) : (\bar{\tau}, \bar{\tau}_1, \dots, \bar{\tau}_m)$. Let

$$\Gamma = \bigcup_{n \in S} \Gamma_n \Theta.$$

Since the variables are renamed apart for each node label, Γ is a variable typing.

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. 2.5). Consider also the equation $p(\bar{\sigma}) = p(\bar{\sigma}') \in Eq(TS)$ corresponding to the pair n, n' in TS. Note that $\Gamma_n \vdash \bar{s} : \bar{\sigma}$ and $\Gamma_{n'} \vdash \bar{s}' : \bar{\sigma}'$. By Lemma 2.14, $\Gamma \vdash \bar{s} : \bar{\sigma}\Theta$ and $\Gamma \vdash \bar{s}' : \bar{\sigma}'\Theta$. Moreover, since $\Theta = MGU(Eq(TS))$, we have $\bar{\sigma}\Theta = \bar{\sigma}'\Theta$. Therefore $\Gamma \vdash \bar{s} = \bar{s}'\ \mathit{Atom}$. Since the same reasoning applies for any equation in $Eq(S)$, by Lemma 2.16, (θ, Γ) is a typed substitution.

Consider a node n'' in S with node atom a. Since $\Gamma_{n''} \vdash a\ \mathit{Atom}$, by Lemma 2.14, $\Gamma \vdash a\ \mathit{Atom}$. and by Lemma 2.15, $\Gamma \vdash a\theta\ \mathit{Atom}$. Therefore P has subject reduction wrt. $\mathcal{Q}$. $\square$

Example 4.3 Figure 3 shows a proper skeleton and the corresponding *non-proper* type skeleton for the program in Ex. 3.3.

In contrast, let $\mathcal{K}$ and $\mathcal{F}$ be as in Ex. 3.3, 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 corresponding type

[5]Recall that the variables/parameters are renamed apart for each node label in the (type) skeleton.

$$\begin{tikzcd}
\mathtt{go} \leftarrow \mathtt{p(X)} \arrow[d, dash] \\
\mathtt{p(X')} \leftarrow \mathtt{r(X')} \arrow[d, dash] \\
\mathtt{r([X''])} \leftarrow \mathtt{r(X'')} \arrow[d, dash] \\
\mathtt{r([X'''])} \leftarrow \mathtt{r(X''')}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathtt{go} \leftarrow \mathtt{p(list(int))} \arrow[d, dash] \\
\mathtt{p(list(int))} \leftarrow \mathtt{r(list(int))} \arrow[d, dash] \\
\mathtt{r(list(U''))} \leftarrow \mathtt{r(U'')} \arrow[d, dash] \\
\mathtt{r(list(U'''))} \leftarrow \mathtt{r(U''')}
\end{tikzcd}$$

Figure 3: A skeleton and the corresponding *non-proper* type skeleton for Ex. 3.3

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

Figure 4: A program used to illustrate proper type skeletons

clauses are given as comments. 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`.

The next example shows that weak type unifiability is not a necessary condition for subject reduction.

Example 4.4 Let $\mathcal{K} = \{\mathtt{int}/0, \mathtt{char}/0, \mathtt{list}/1, \mathtt{c}/1\}$, $\mathcal{F} = \{\mathtt{1}_{\to\mathtt{int}}, \ldots, \mathtt{'a'}_{\to\mathtt{char}}, \ldots, \mathtt{nil}_{\to\mathtt{list(U)}}, \mathtt{cons}_{\mathtt{U},\mathtt{list(U)}\to\mathtt{list(U)}}, \mathtt{f}_{\mathtt{list(U)},\mathtt{list(int)},\mathtt{list(char)}\to\mathtt{c(U)}}\}$, $\mathcal{P} = \{\mathtt{p}_{\mathtt{c(U)}}\}$, $P = \{\mathtt{p(f(X,X,[])).}\}$, $Q = \mathtt{p(f(Y,[],Y))}$. Then the only skeleton of $P \cup \{\mathtt{go} \leftarrow Q\}$ is proper, the corresponding type skeleton is not proper, and yet P has subject reduction wrt. $\{Q\}$, as shown in Fig. 6.

The above example exhibits a interesting phenomenon. One would expect that instantiating a term constrains its possible types. This is not generally true. For $\Gamma = \{\mathtt{X} : \mathtt{list(int)}, \mathtt{Y} : \mathtt{list(char)}\}$, we have $\Gamma \vdash \mathtt{f(X,X,[])} : \mathtt{c(char)}$ and $\Gamma \vdash \mathtt{f(Y,[],Y)} : \mathtt{c(int)}$. The types of the terms are not even unifiable, but for their common instance we have $\Gamma \vdash \mathtt{f([],[],[])} : \mathtt{c(U)}$. The point is that multiple variable occurrences must all have the same type, whereas in $\mathtt{f([],[],[])}$, the occurrences of $[]$ are independent. We wonder if this phenomenon has been observed before.

As with our discussion of the transparency condition, a small digression to functional programming is in order here. What becomes apparent in the above example is that in our type system, only declared function and predicate symbols (which would be called constants in functional programming terminology) can be used polymorphically, whereas variables must always be used monomorphically. This is in contrast to an expression like **let** $\mathtt{X} = []$ **in** $\mathtt{f(X,X,X)}$ in functional

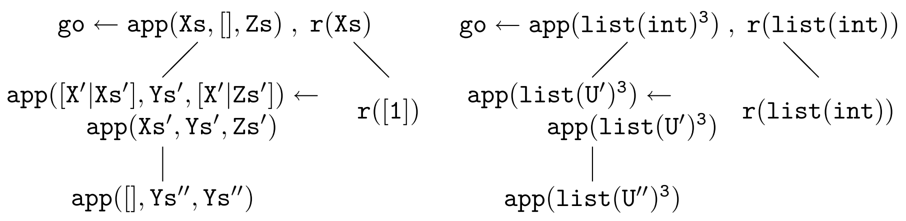

go ← app(Xs, [], Zs) , r(Xs)
app([X′|Xs′], Ys′, [X′|Zs′]) ← app(Xs′, Ys′, Zs′)
r([1])
app([], Ys″, Ys″)
go ← app(list(int)³) , r(list(int))
app(list(U′)³) ← app(list(U′)³)
r(list(int))
app(list(U″)³)

Figure 5: A skeleton and the corresponding *proper* type skeleton for Ex. 4.3

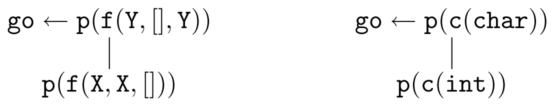


Figure 6: Subject reduction in spite of a *non-proper* type skeleton

programming, where we assume f to be typed as in the above example. In this expression, X is used with three different type instances.

The contrast is again due to the central role that instantiation of logical variables plays in logic programming. Subject reduction relies on the fact that the well-typing of objects is stable under instantiation of types and terms, as formalised in Lemmas 2.14 and 2.15. We cannot allow the term f(X, X, X) since generally, instances of it will not be well-typed, with the only exception of f([], [], []), where we use the polymorphic constant [].

We now give an example of a program that has weak type unifiability for an appropriate set of queries. The example is interesting because it falls out of any conditions for subject reduction we shall present later.

Example 4.5 It has been observed that higher-order constructs in logic programming create difficulties for typing. Consider a generic $\mathtt{apply}_{\mathtt{string},\mathtt{U},\mathtt{U}}$ predicate where the first argument is the name of an ordering predicate and the other arguments are elements that the ordering predicate is applied to. Such a predicate could be useful to define a generic sorting predicate, where the ordering relation is passed as argument. The definition of apply may include clauses like

```
apply("int_order",X,Y) <- less(X,Y).
apply("char_order",X,Y) <- alph(X,Y).
```

where $\mathtt{less}_{\mathtt{int},\mathtt{int}}$ is the "<"-relation and $\mathtt{alph}_{\mathtt{char},\mathtt{char}}$ is the alphabetical order. This program has weak type unifiability wrt. $\mathcal{Q} = \{\mathtt{apply}(\texttt{"int\_order"}, x, y) \mid x, y \text{ integers}\} \cup \{\mathtt{apply}(\texttt{"char\_order"}, x, y) \mid x, y \text{ characters}\}$.

The lesson learnt from such an example is not new but we shall briefly summarise it here: The example is very simple and even if one looks at it in a slightly bigger context (e.g. defining a sorting predicate as mentioned), it is possible to see in a simple ad-hoc way that such a program has weak type unifiability and hence subject reduction for an appropriate set of queries. However, one also sees that subject reduction depends critically on the actual values that arguments take (here, the value of the first argument of apply_order). In general, establishing subject reduction in such cases must be extremely difficult, and we conjecture that it is undecidable.

However, we have no proof of the undecidability of weak type unifiability.

4.2 Strong Type Unifiability

We now define a stronger notion of type unifiability where the condition of "term unifiability" is discarded. It is based on a property of a particular class of programs studied previously by Bouquard [Bou92].

Definition 4.6 Let P be a program and $\mathcal{Q}$ be a set of queries. We say that P has the **full success property wrt.** $\mathcal{Q}$ if each skeleton S of $P \cup \{\mathtt{go} \leftarrow Q\}$ with head $\mathtt{go}$, where $Q \in \mathcal{Q}$, is proper.

Definition 4.7 Let $\_ \vdash P$ *Program* and $\_ \vdash \mathcal{Q}$ *Queryset*, and P^*, $\mathcal{Q}^*$ be the corresponding type program and set of type queries[6]. We say that P has **strong type unifiability wrt.** $\mathcal{Q}$ if P^* has the full success property wrt. $\mathcal{Q}^*$.

Clearly, strong type unifiability implies weak type unifiability.

Arguably, strong 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. Therefore, we believe that the efforts concerning proofs of subject reduction should focus on establishing this property. However this view has been challenged in the context of higher-order logic programming [NP92].

We now show that strong type unifiability is undecidable, just as subject reduction itself. This is based on two facts:

- it is undecidable whether a program has the full success property;
- type programs (Def. 2.12) are *not* a restricted class of programs. Hence, the undecidability for general programs applies also to type programs.

Theorem 4.8 It is undecidable whether a program has strong type unifiability.

Proof: The first point (it is undecidable whether a program has the full success property) was already conjectured in [Bou92]. A result from [DLP$^+$96] solves the conjecture. There it is shown that the halting problem is undecidable even for a logic program with one goal of the form $\leftarrow p(r)$ and one (recursive) clause of the form $p(s) \leftarrow p(t)$. Such a program does not halt if and only if all its skeletons are proper, which is shown as follows: if it does not halt, it has an infinite proper skeleton, obtained by grafting infinitely many copies of the clause $p(s) \leftarrow p(t)$. But any skeleton for the program is necessarily a sub-skeleton of this infinite skeleton and hence also proper. Conversely, if all skeletons for the program are proper, then this holds in particular for the infinite skeleton obtained by grafting infinitely many copies of the clause $p(s) \leftarrow p(t)$. By definition, all skeletons being proper is equivalent to the full success property. Hence the full success property is also undecidable.

We now address the second point. We show that type programs are, syntactically speaking, not a restricted class of programs. Any program could be a type program. This implies that any problem undecidable for arbitrary programs must remain undecidable for type programs. To this end we show how an arbitrary (a priori untyped) logic program P can be typed in such a way that the corresponding type program P^* is isomorphic to P.

[6]Defined in obvious analogy to a type clause, Def. 2.12.

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

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

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

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

Figure 7: Three potential solutions for Ex. 4.9

Let $\mathcal{F}$ (resp., $\mathcal{P}$) be the set of function (resp., predicate) symbols occurring in P, and $\mathcal{V}$ a denumerable set of variables including the variables occurring in P. We define the type language in such a way that each term is essentially identical to its type, except that we use ˜ to mark types. Thus we define $\mathcal{K} = \{\tilde{f}/m \mid f/m \in \mathcal{F}\}$. Further, we declare the type of each $f/m \in \mathcal{F}$ to be $u_1, \ldots u_m \to \tilde{f}(u_1, \ldots u_m)$, and the type of each $p/m \in \mathcal{P}$ to be $p_{u_1,\ldots u_m}$. Note that this type language respects the transparency condition. Finally, let $\mathcal{U} = \{\tilde{x} \mid x \in \mathcal{V}\}$. Let us denote by $\tilde{t}$ the syntactic object obtained from t by marking each symbol in t with ˜. It is easy to see that in the thus defined language, $\_ \vdash t : \tilde{t}$, i.e., terms are essentially (except for the ˜ markers) identical to their types, and the type program P^* corresponding to P is "identical" to P. $\square$

We now give an example where we can show subject reduction based on strong type unifiability. It is a programming task for which we present three solutions. The example is interesting because two of those solutions fall out of the scope of decidable conditions for subject reduction we shall present later.

Example 4.9 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 $\{\sigma \mid \exists s, t.\ s \text{ is subterm of } t,\ \_ \vdash s : \sigma,\ \_ \vdash t : \mathtt{t}(\mathtt{U})\}$ is infinite, or in words, there are infinitely many types that a subterm of a term of type $\mathtt{t}(\mathtt{U})$ can have. This property of the type $\mathtt{t}(\mathtt{U})$ is very unusual. In [SHK00], a condition is considered (the *Reflexive Condition*) which rules out this situation.

Now consider the predicate $\mathtt{fgs}/2$ 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 $\mathcal{P} = \{\mathtt{fgs1}_{\mathtt{int,t(U)}}, \mathtt{gs1}_{\mathtt{int,t(U)}}, \mathtt{fgs2}_{\mathtt{int,t(U)}}, \mathtt{fgs3}_{\mathtt{int,t(U)}}, \mathtt{fs1}_{\mathtt{int,t(U),int}}, \mathtt{fs2}_{\mathtt{int,t(U),int}}, \mathtt{gs2}_{\mathtt{int,t(U),t(V)}}, \mathtt{fgs3\_aux}_{\mathtt{int,t(U),t(U)}}\}$.

All three programs have subject reduction wrt. $\mathcal{Q} = \{\mathtt{fgs}j(i, \mathtt{Y}) \mid i \in \mathtt{int}\}$ where $j = 1, 2, 3$ as applicable. In fact, all three programs have strong type unifiability wrt. $\mathcal{Q}$. For example the type programs for the first and third solution are shown in Fig 8.

For the above example, one can easily show strong type unifiability in an ad-hoc way. In general however, finding decidable sufficient conditions for strong type unifiability with low complexity is not a trivial question, although the head

```
fgs1(int,t(U)) <-
  fs1(int,t(U),int).

fs1(int,t(U),int) <-
  fs1(int,t(t(U)),int).
fs1(int,t(U),int) <-
  gs1(int,t(U)).

gs1(int,t(t(U))) <-
  gs1(int,t(U)).
gs1(int,t(U)).
```

```
fgs3(int,t(U)) <-
  fgs3_aux(int,t(U),t(U)).

fgs3_aux(int,t(U),t(U)) <-
  fgs3_aux(int,t(t(U)),t(t(U))).
fgs3_aux(int,t(U),t(U)).
```

Figure 8: Two type programs for Ex. 4.9

condition is one such condition. In [Bou92, DM93] some guidelines based on NSTO (not subject to occur check) tests may be used to find some decidable condition. It shows in particular that there are tests of subject reduction for the above programs. This will not be developed further here. It seems clear however by these studies that the complexity of such tests will remain quite high (at least exponential). In the rest of this section we will consider conditions that are formulated at the clause level, i.e. they can be verified for each clause independently.

4.3 The Head Condition

The head condition is the standard way [HT92] of ensuring strong type unifiability.

Definition 4.10 A clause $C = p_{\bar{\tau}}(\bar{t}) \leftarrow B$ fulfils 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, $\Theta\lceil_{\bar{\tau}} = \emptyset$. It has been shown previously that typed programs fulfilling the head condition have operational subject reduction [HT92, Thm. 1.4.7]. By Thm. 3.2, this means that they have subject reduction.

Consider again Ex. 4.9. Only the third solution fulfils the head condition (see Fig. 8). The other two versions do not. For this 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.

Note that the third solution uses *polymorphic recursion*, a concept previously discussed for functional programming [KTU93]: In the recursive clause for `fgs3_aux`, the arguments of the recursive call have type (`int`, `t(t(U))`, `t(t(U))`), while the arguments of the clause head have type (`int`, `t(U)`, `t(U)`). If we wrote a function corresponding to `fgs3_aux` in Miranda, Haskell 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 or Haskell, 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 [Kah96]. Example 4.9 suggests that there is a certain symmetry between polymorphic recursion and violations of the head condition.

4.4 A Generalisation: Semi-generic Programs

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 4.11 Let $E = \{l_1 = r_1, \dots, l_m = r_m\}$ be a set of oriented equations. E is unifiable if

1. if $i \neq j$, then r_i and r_j have no variable in common, and
2. there exists a partial order $\rightarrow$ on the equations such that if r_i and l_j share a variable, then $i \neq j$ and $l_i = r_i \rightarrow l_j = r_j$, and
3. for all $i \in \{1, \dots, m\}$, l_i is an instance of r_i.

Proof: Without loss of generality, assume that the equations in E are indexed in a way that is compatible with $\rightarrow$, i.e., $l_i = r_i \rightarrow l_j = r_j$ implies $i \leq j$. The proof is by induction on m. We have to strengthen the inductive hypothesis: we claim that there is a unifier θ of E whose domain contains only variables occurring in some r_i, i.e., $\theta = \theta\lceil_{r_1,\dots,r_m}$. Moreover, θ is *relevant*, i.e., it does not contain any variables not contained in E.

The base case $m = 0$ is trivial.

Now suppose that the hypothesis holds for $E = \{l_1 = r_1, \dots, l_m = r_m\}$, where the unifier of E is θ. Consider a further equation $l_{m+1} = r_{m+1}$ such that $E \cup \{l_{m+1} = r_{m+1}\}$ meets the assumptions of the statement.

By conditions 1 and 2, r_{m+1} and E have no variable in common, and hence by the inductive hypothesis that θ is relevant, we have $r_{m+1}\theta = r_{m+1}$. By condition 2, r_{m+1} and l_{m+1} have no variable in common, and again by the inductive hypothesis, r_{m+1} and $l_{m+1}\theta$ have no variable in common. This and condition 3 implies that $l_{m+1}\theta$ is an instance of r_{m+1}, and so there is a (minimal) substitution θ' such that $r_{m+1}\theta' = l_{m+1}\theta$.

By a simple argument about the independence of θ and θ', it follows that $\theta\theta'$ is a unifier of $E \cup \{l_{m+1} = r_{m+1}\}$, and $\theta\theta' = \theta\theta'\lceil_{r_1,\dots,r_m,r_{m+1}}$, and $\theta\theta'$ does not contain any variables not contained in $E \cup \{l_{m+1} = r_{m+1}\}$. □

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 $\bar{\tau}_h$.

We now show that by decomposing each equation $p(\bar{\tau}_a) = p(\bar{\tau}_h)$, one can refine this condition.

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. 4.11 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 NSTO can be found in [DM93]). Taking into account the renaming of type clauses, a relation between two equations (as in Prop. 4.11) 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 $p(\bar{h}, \bar{b})$ means: $\bar{h}$ and $\bar{b}$ are the vectors of terms filling the head-generic and body-generic positions of p, respectively. The notation $p(\bar{\chi}, \bar{\beta})$, where $\bar{\chi}$ and $\bar{\beta}$ are type vectors, is defined analogously. When denoting a type clause, it is convenient to use one letter, say τ, for "generic" positions (head-generic in the head or body-generic in the body) and another, say σ, for "non-generic" positions.

Definition 4.12 Let $\_ \vdash C$ *Clause* such that the type clause corresponding to C is $p(\bar{\tau}_0, \bar{\sigma}_{m+1}) \leftarrow p_1(\bar{\sigma}_1, \bar{\tau}_1), \ldots, p_m(\bar{\sigma}_m, \bar{\tau}_m)$. We call C **semi-generic** if

1. for all $i, j \in \{0, \ldots, m\}$, $i \neq j$, $pars(\tau_i) \cap pars(\tau_j) = \emptyset$,
2. for all $j \in \{1, \ldots, m\}$, $pars(\bar{\sigma}_j) \cap \bigcup_{j \leq i \leq m} pars(\bar{\tau}_i) = \emptyset$,
3. for all $i \in \{0, \ldots, m\}$, the vector τ_i is the declared type of the body-generic positions of p_i.

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

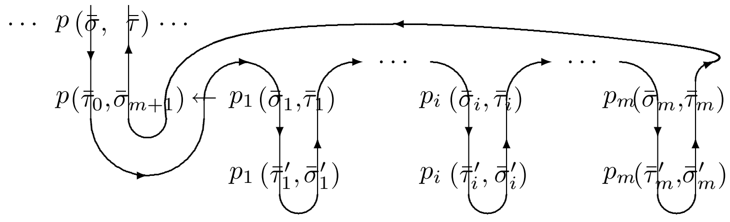
Technically, semi-genericity resembles *nicely-modedness*, where head-generic corresponds to input, and body-generic corresponds to output. Among other things, nicely-modedness has been used to show that programs are free from unification [AE93]. Semi-genericity serves a similar purpose here. However, the analogy should be treated with care. First, there are slight technical differences, which we do not want to elaborate. More importantly, one should not expect that for a given program, the head-generic (resp., body-generic) arguments coincide with the ones one would intuitively consider to be input (resp., output).

Note also that a typed program which fulfils the head condition is semi-generic, where all argument positions are head-generic.

The following theorem states subject reduction for semi-generic programs. It is based on the fact that semi-genericity implies strong type unifiability, thereby weak type unifiability, and hence subject reduction.

Theorem 4.13 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 $P \cup \{\mathtt{go} \leftarrow Q\}$ with head $\mathtt{go}$. We will order the equations in $Eq(TS)$ in such a way that Prop. 4.11 is applicable.



$\cdots\ p(\bar\sigma,\ \bar\tau)\cdots$
$p(\bar\tau_0,\bar\sigma_{m+1})\leftarrow p_1(\bar\sigma_1,\bar\tau_1)$
$p_i(\bar\sigma_i,\bar\tau_i)$
$p_m(\bar\sigma_m,\bar\tau_m)$
$\cdots$
$\cdots$
$p_1(\bar\tau'_1,\bar\sigma'_1)$
$p_i(\bar\tau'_i,\bar\sigma'_i)$
$p_m(\bar\tau'_m,\bar\sigma'_m)$

Figure 9: Illustrating the proof of Thm. 4.13

In particular, we show how the clauses originating from one particular clause can be ordered. Consider a node n labelled $C = p(\bar\tau_0, \bar\sigma_{m+1}) \leftarrow p_1(\bar\sigma_1, \bar\tau_1), \ldots, p_m(\bar\sigma_m, \bar\tau_m)$. To simplify the notation, we assume that n is not the root and that all children of n are complete. Otherwise, the reasoning applies a fortiori.

For each $i \in \{1, \ldots, m\}$, let the ith child of n be labelled $p_i(\bar\tau'_i, \bar\sigma'_i) \leftarrow \ldots$, and assume that n is the kth child of its parent, whose node label has $p(\bar\sigma, \bar\tau)$ as kth body atom.

The equations in $Eq(TS)$ that might share parameters with C are

$$p(\bar\sigma, \bar\tau) = p(\bar\tau_0, \bar\sigma_{m+1}), p_1(\bar\sigma_1, \bar\tau_1) = p_1(\bar\tau'_1, \bar\sigma'_1), \ldots, p_m(\bar\sigma_m, \bar\tau_m) = p_m(\bar\tau'_m, \bar\sigma'_m).$$

We decompose, orient and order these equations as follows:

$$\bar\sigma = \bar\tau_0 \to \bar\sigma_1 = \bar\tau'_1 \to \bar\sigma'_1 = \bar\tau_1 \to \ldots \to \bar\sigma_m = \bar\tau'_m \to \bar\sigma'_m = \bar\tau_m \to \bar\sigma_{m+1} = \bar\tau \quad (*)$$

This is illustrated in Fig. 9. Since TS is a tree, it is clearly possible to decompose, orient and order all equations in $Eq(TS)$ in such a way that for each clause, $(*)$ holds. We interpret $\to$ as a partial order on equations in the obvious way. Let Eq' be the obtained equation system, which is clearly equivalent to $Eq(TS)$.

We show that Eq' fulfils the conditions of Prop. 4.11 (where types are regarded as terms in the obvious way). By Def. 4.12 (1) and the renaming of parameters for each node, Eq' fulfils condition 1. By Def. 4.12 (2), if $\bar\tau_i$ and $\bar\sigma_j$ (using the notations as above) share a parameter, then $i < j$, and thus $\_ = \bar\tau_i \to \bar\sigma_j = \_$. Hence Eq' fulfils condition 2. By Def. 4.12 (3), Eq' fulfils condition 3.

Thus $Eq(TS)$ has a solution, so TS is proper, and so P has strong type unifiability wrt. $\mathcal{Q}$. Since strong type unifiability implies weak type unifiability, by Thm. 4.2, 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 4.14 Suppose $\mathcal{K}$ and $\mathcal{F}$ define lists as usual (see Ex. 3.3). Let $\mathcal{P} = \{\mathtt{p}_{\mathtt{U,V}}, \mathtt{q}_{\mathtt{U,V}}\}$ 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 types in generic positions are U, W, V; all generic arguments have the declared type (condition 3);

$$\begin{tikzcd}[column sep=small]
& \mathtt{p(U,list(V))} \leftarrow \mathtt{q(list(U),W)}, \arrow[dl, dash] & \mathtt{q(list(W),V)} \arrow[dr, dash] & \\
\mathtt{q(U',list(U'))} & & & \mathtt{q(U'',list(U''))}
\end{tikzcd}$$

Figure 10: A type skeleton for a semi-generic program

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. 10.

As another example, suppose now that $\mathcal{K}$ and $\mathcal{F}$ define list and integers, and consider the predicate `r/2` specified as $\mathtt{r}(1, []), \mathtt{r}(2, [[]]), \mathtt{r}(3, [[[]]]) \ldots$. 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 $(\mathtt{int}, \mathtt{list(U)})$ 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. This intermingling means that the *type* of the second argument depends on the *value* of the first. However, `r` is in perfect analogy to `gs1` in Ex. 4.9. Here at least the specification of `fgs` cannot be accused of such intermingling.

In fact, Ex. 4.9 seems to be a promising candidate for an example which shows the interest in the class of semi-generically typed programs. But unfortunately, the first two programs, which violate the head condition, are not semi-generic. 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)`).

The fact that in the above program, the head condition is violated no matter what the declared type is raises the question of type inference already mentioned in Sec. 4.3 [Tiu90]. In analogy to functional programming languages such as ML, one could envisage that the types of the predicate symbols could be inferred automatically, rather than declared by the user [LR91] (nota bene: the predicate symbols, not the function symbols!). We do not address this question here, but note that while it may seem that having user-declared types naturally goes together with types being prescriptive, this is not necessarily so. In functional programming languages, it is common that types are inferred automatically and yet prescriptive.

Program Class	**Decidability**
Typed programs	
$\cup$ (Ex. 3.3)	
Subject reduction	undecidable
$\cup$ (Ex. 4.4)	
Weak type unifiability	?
$\cup$ (Ex. 4.5)	
Strong type unifiability	undecidable
$\cup$ (Ex. 4.9)	
Semi-generic	decidable
$\cup$ (Ex. 4.14)	
Head condition	decidable

Table 3: Hierarchy of classes of programs with subject reduction

5 Discussion

5.1 Summary of the results

We introduced fives classes of typed logic programs with subject reduction, where each class is properly included in the subsequent classes: programs which satisfy the head condition, semi-generic programs, programs with strong type unifiability, with weak type unifiability, programs with subject reduction. This is summarised in Table 3, where we refer to examples showing that each inclusion is proper.

We established (un)decidability results for all the classes except weak type unifiability. We conjecture this class is also undecidable. A proof should probably be based on a careful analysis of the relation between term and corresponding type unification. In spite of being undecidable and only a sufficient condition for subject reduction, we believe that strong type unifiability is the right basis for searching for decidable conditions for subject reduction. It expresses a condition for subject reduction that is independent of failure and success of the computation, which is the view one usually takes in prescriptive typing.

5.2 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 strong 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* [Mil78], 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 [LR96]. 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 [O'K90].

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 [Han92] 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, but so far, we have not been able to identify a "useful", non-contrived, example which clearly shows the interest in the class of semi-generic programs.

5.3 Conclusion

In this article 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. 3.2). We also showed that subject reduction is undecidable (Thm. 3.6).

We have argued that the key to subject reduction lies in ensuring that types of unified terms are unifiable. Formally, unifiability of types was based on *type skeletons*, obtained from skeletons by replacing terms by their types. We have defined *weak* type unifiability, meaning that any type skeleton corresponding to a proper skeleton is proper, and *strong* type unifiability, meaning that any type skeleton, regardless of whether it corresponds to a proper skeleton, is proper. We have shown that strong type unifiability is undecidable, too.

Our approach has several potential applications:

- It facilitates studying the semantics of typed programs by simplifying its formulation in comparison to other works (e.g. [LR91]). 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 [BGLM94, DM93, FLMP89].

- The approach may enhance program analysis based on abstract interpretation. Proper type skeletons could also be modelled by fixpoint operators [CLMV99, CC77, GDL95]. Abstract interpretation for prescriptively typed programs has been studied by [RB01, SHK00], 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 [TDD97], 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 type-correctness criteria [DM98], 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 strong type unifiability which refines the head condition (Thm. 4.13). Several observations arise from this:

- Def. 4.12 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 strong type unifiability contains at least an NSTO condition, one could also benefit from the refinements proposed for the NSTO check [DM93]. Such further refined conditions should, in particular, be fulfilled by all solutions of Ex. 4.9.

We have also studied *operational* subject reduction for type systems with subtyping [SFD00]. As future work, we want to integrate that work with the *proof-theoretic* view of subject reduction of this article. Also, we want to design more refined tests for strong type unifiability, and we want to study the relationship between the head condition and polymorphic recursion.

Acknowledgements

We thank François Fages and Erik Poll for interesting discussions, Michael Hanus for the idea of Ex. 4.5, and the anonymous referees for their valuable comments. The second author was supported by an ERCIM fellowship.

References

[AE93] 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.

[Bei95] 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.

[BGLM94] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: theory and applications. *Journal of Logic Programming*, 19/20:149–197, 1994.

[Bou92] J.-L. Bouquard. *Etude des rapports entre Grammaires Attribuées et Programmation en Logique: application au test d'occurrence et à l'analyse statique.* PhD thesis, University of Orléans, 1992. in French.

[CC77] 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.

[CLMV99] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.

[DLP$^+$96] P. Devienne, P. Lebègue, A. Parrain, J.-C. Routier, and J. Würtz. Smallest Horn clause programs. *Journal of Logic Programming*, 27(3):227–267, 1996.

[DM93] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* MIT Press, 1993.

[DM98] 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/`.

[DS01] P. Deransart and J.-G. Smaus. Well-typed logic programs are not wrong. In H. Kuchen and K. Ueda, editors, *Proceedings of the 5th International Symposium on Functional and Logic Programming*, volume 2024 of *LNCS*, pages 280–295. Springer-Verlag, 2001.

[FLMP89] 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.

[GDL95] 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.

[Han92] M. Hanus. *Logic Programming with Type Specifications*, chapter 3, pages 91–140. 1992. In [Pfe92].

[Hil93] P. M. Hill. The completion of typed logic programs and SLDNF-resolution. In A. Voronkov, editor, *Proceedings of the Fourth International Conference on Logic Programming and Automated Reasoning*, volume 698 of *LNCS*, pages 182–193. Springer-Verlag, 1993.

[HL89] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In H. Abramson and M. H. Rogers, editors, *Proceedings of the 1988 International Workshop on Meta-Programming in Logic*, pages 23–51. MIT Press, 1989.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[HT92] P. M. Hill and R. W. Topor. *A Semantics for Typed Logic Programs*, chapter 1, pages 1–61. 1992. In [Pfe92].

[Kah96] 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.

[KTU93] 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. Title wrongly given in table of contents: Type *recursion* in the presence of polymorphic recursion.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[LP81] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[LR91] 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.

[LR96] 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.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[MO84] A. Mycroft and R. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[NP92] G. Nadathur and F. Pfenning. *Types in Higher-Order Logic Programming*, chapter 9, pages 245–283. 1992. In [Pfe92].

[O'K90] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[Pfe92] F. Pfenning, editor. *Types in Logic Programming.* MIT Press, 1992.

[RB01] O. Ridoux and P. Boizumault. Typed static analysis: Application to the groundness analysis of typed prolog. *Journal of Functional and Logic Programming*, 2001(4), 2001.

[SFD00] 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*, pages 214–226. Springer-Verlag, 2000.

[SHC96] 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.

[SHK00] 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. Long version appeared as Report 2000.06, University of Leeds.

[TDD97] 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.

[Tho91] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[Tho95] S. Thompson. *Miranda: The Craft of Functional Programming.* Addison-Wesley, 1995.

[Tiu90] J. Tiuryn. Type inference problems: A survey. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 105–120. Springer-Verlag, 1990.