# Towards a Type Discipline for Answer Set Programming

Camillo Fiorentini, Alberto Momigliano, and Mario Ornaghi

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy
{fiorenti,momiglia,ornaghi}@dsi.unimi.it

**Abstract.** We argue that it is high time that types had a beneficial impact in the field of Answer Set Programming and in particular Disjunctive Datalog as exemplified by the *DLV* system. Things become immediately more challenging, as we wish to present a type system for *DLV-Complex*, an extension of DLV with uninterpreted function symbols, external implemented predicates and types. Our type system owes to the seminal polymorphic type system for Prolog introduced by Mycroft and O'Keefe, in the formulation by Lakshman and Reddy. The most innovative part of the paper is developing a *declarative grounding* procedure which is at the same time appropriate for the operational semantics of ASP and able to handle the new features provided by DLV-Complex. We discuss the soundness of the procedure and evaluate informally its success in reducing, as expected, the set of ground terms. This yields an automatic reduction in size and numbers of (non isomorphic) models. Similar results could have only been achieved in the current untyped version by careful use of generator predicates in lieu of types.

**Keywords:** Answer set programming, type checking, grounding, many sorted interpretation.

## 1 Introduction

The advantages of static type checking for programming languages are almost universally recognized and well-understood, although types managed to make it into logic programming somewhat belatedly [21]. From the pioneering paper [18] advocating the introduction of types in Prolog, different approaches emerged, belonging roughly to two main camps: the descriptive and the prescriptive one. The former aims at capturing some aspects of a program behavior, for example its success set, up to much more complex static and dynamic properties as with CiaoPP [12].

We favor the *prescriptive* approach, where types are an integral part of the program's meaning, thus allowing the user to discard ill-formed elements at compile time. This helps the developer to proceed in a more disciplined way, being able to receive early feedback about mistakes that may be hard to find, especially in a purely declarative setting in the non-infrequent case where the answer would be merely "no".

The type theory of logic programming has significantly developed in the passing years, following the ever-increasing role of types in the general theory of programming languages – see the type system of Mercury [14] and λProlog [19] for a modern take to prescriptive typing. Still, for the sake of this paper, we will adapt to disjunctive

S. Berardi, F. Damiani, and U. de'Liguoro (Eds.): TYPES 2008, LNCS 5497, pp. 117–135, 2009.

logic programing a rather elementary SML-like polymorphic discipline; in addition we will enrich it semantically, following Lakshman & Reddy's approach [15], where typed logic programs are interpreted over first order many sorted structures, see also [13].

Our objective here is to advocate the usefulness of prescriptive typing for Answer Set Programming (ASP) [9] in general. ASP is a form of declarative programming based on the stable models semantics [10]. It has been proposed as a language for knowledge representation and as a tool for formalizing and solving hard search problems [1]. The main idea is to reduce a search problem encoded with a (disjunctive) logic program $P$ to the problem of computing stable models of $P$ using an answer set solver, namely a system for generating stable models, such as Smodels [22] and DLV [16]. In this sense ASP's operational semantics is very different from the usual SLD(NF)-resolution [17]. This is also why typing ASP is interesting and less banal that at first sight: the original paper by Mycroft and O'Keefe was entirely syntactical and so the vast majority of the ensuing research was intrinsically biased towards standard SLD(NF)-resolution.

Although some papers in the ASP literature use a many sorted language or deals with object-oriented features — we refer to Sect. 6 for a detailed discussion — it is safe to say that types have never been *integrated* into ASP; in fact the above papers do not consider a *many-sorted model theory*, nor any form of *parametric* polymorphism. The present paper introduces static type-checking in DLV-Complex [5], an extension of DLV with uninterpreted function symbols – hence we are leaving the comfort of Datalog – as well with external types and predicates requiring an oracle to be computable. In other terms, one may exploit external sources of knowledge. To take into account this possibility, we consider programs over *grounding structures*. The latter provide a declarative semantics for grounding, as they play the role of pre-interpretations [17] in presence of an external implementation of data types and related operations.

The paper is organized as follows: we begin in Section 2 by briefly illustrating the role of typing in ASP through examples. Section 3 and 4 give a static semantics to the language both in terms of types and many-sorted interpretations. In Section 5 we extend the above analysis to DLV-Complex's programs and offer a rational reconstruction of type-driven grounding. After discussing related work and summing up the results in Section 6, we add a short Appendix (A) explaining the basics of ASP for the reader unfamiliar with this topic.

## 2 An Informal Introduction to Typed DLV

For the sake of this paper, an ASP program $P$ is a set of rules (or clauses) of the form:

$$A_1 \vee \cdots \vee A_n \leftarrow B_1, \ldots, B_k, \mathit{not}\ B_{k+1}, \ldots, \mathit{not}\ B_m$$

where $A_1, \ldots, A_n, B_1, \ldots, B_m$ are atoms, i.e., formulas of the form $p(\boldsymbol{t})$, where $\boldsymbol{t}$ stands for a sequence $t_1, \ldots, t_j$ of terms. Usually *not* is referred to as "negation as failure" or "default negation" and $\vee$ as "epistemic disjunction".

*Example 1.* The following `pencil problem` is a simple non-typed ASP program, which illustrates the meaning of default negation.

```
color(red). color(blue). pencil(p1).  pencil(p2).     % facts
nice(X) :- pencil(X), color(Y), not ugly(Y).          % clause c1
ugly(Y) :- color(Y), pencil(X), not nice(X).          % clause c2
```

By rule `c1`, we can infer that a pencil `X` is nice if we do not have evidence for `ugly(Y)` (i.e., if we can assume `not ugly(Y)` as a default). Symmetrically, by rule `c2`, we can infer `ugly(Y)` if we do not have evidence for `nice(X)`. Let us assume that "`not ugly(Y)`" holds. We infer that all the pencils are nice, i.e., we have the answer set {`nice(p1),nice(p2)`}. Symmetrically, assuming "`not nice(X)`" we obtain the answer set {`ugly(red),ugly(blue)`}.

As shown by the example, an ASP program $P$ represents a problem that may have multiple solutions, represented by its answer sets. The answer set semantics is defined in terms of *grounding*: let $grnd(C)$ be the set of the ground instances of a clause $C$, obtained by substituting its variables with ground terms of the language underlying $P$. We define $grnd(P) = \cup_{C \in P} grnd(C)$. An answer set is a set of ground literals. The answer sets of $P$ are the stable models of $grnd(P)$ (see the Appendix for the formal definitions).

An ASP solver computes the stable models in two stages: firstly it builds an optimized version of $grnd(P)$, namely a set $grnd^{\bullet}(P)$ of ground clauses with the same semantics of $grnd(P)$. Then it generates the answer sets of $grnd^{\bullet}(P)$.

*Example 2.* In DLV, the `pencil` program of Example 1 grounds to:

```
% EDB facts:
color(red).  color(blue).  pencil(p1).  pencil(p2).
% Residual ground instantiation of the program:
ugly(red)  :- not nice(p1).      ugly(red)  :- not nice(p2).
ugly(blue) :- not nice(p1).      ugly(blue) :- not nice(p2).
nice(p1)  :- not ugly(red).      nice(p1)  :- not ugly(blue).
nice(p2)  :- not ugly(red).      nice(p2)  :- not ugly(blue).
```

We point out that only 12 simplified clauses of the 36 of $grnd$(`pencil`) are generated, called the *residual ground instantiation*. For example, `ugly(red) :- color(red), pencil(p1), not nice(p1)` is simplified to `ugly(red) :- not nice(p1)`, because the facts `color(red)` and `pencil(p1)` are true in every model. In contrast, the clause `ugly(p1) :- color(p1), pencil(p1), not nice(p1)` has not been generated because the atom `color(p1)` in the body is false in every stable model. We get the answer sets:

```
{color(red), color(blue), pencil(p1), pencil(p2), nice(p1), nice(p2)}
{color(red), color(blue), pencil(p1), pencil(p2), ugly(red), ugly(blue)}
```

Now we come to the main issue of this paper. Looking at the example, one easily recognizes that the optimizations performed by the grounding algorithm, henceforth indicated by *GA*, are related to the implicit presence of two types of objects: pencils and colors. Such types are *interpreted in the same way in all stable models*. We believe that making typing explicit has several advantages. In particular, the programmer has a better control on the grounding process, because a type-driven grounder will not generate clauses corresponding to type information, thus yielding smaller $grnd^{\bullet}(P)$ and smaller answer

sets.[1] Furthermore there is a clear-cut distinction between *data base facts*, which represent relevant pieces of information to be shown in answer sets, and *typing constraints*, which only have the role of narrowing grounding, as exemplified next.

*Example 3.* A typed program is (informally) declared as a unit. We use a concrete syntax inspired by [15]. The typed version of the program of Example 1 is:

```
unit pencils.
type pencil --> p1 ; p2.       % Cf. SML's datatype pencil = p1 | p2;
type color --> red ; blue.
pred nice(pencil), ugly(color).
prog
   nice(X) :- not ugly(Y).
   ugly(X) :- not nice(Y).
```

The *enumerated* type `color` is interpreted as the set of constants {`red, blue`}, and `pencil` as {`p1,p2`}. The `pred` keyword introduces a set of predicate symbols and their declared types. By `nice(pencil)` we indicate that `nice` requires an argument of type `pencil` to be a well-typed predicate. Finally, we have the program section, whose clauses are well-typed. Of course we do not need to declare the type of the variables, as they can be inferred. In the example, the rules for clauses in Sect. 5 would yield these well-typing judgments of the form $\Gamma \vdash H \leftarrow B$ : clause .

$$X : pencil, Y : color \;\vdash\; nice(X) \leftarrow not\ ugly(Y) \;: \text{clause}$$
$$X : color, Y : pencil \;\vdash\; ugly(X) \leftarrow not\ nice(Y) \;: \text{clause}$$

We will explicitly connect the concrete and abstract syntax in Example 6. A typed *GA* will use the variable contexts $X : color, Y : pencil$ instead of the predicates `pencil(X)` and `color(Y)` in the untyped program. The difference is that the untyped *GA* catalogues `color(red)`, `color(blue)`, ... as "external data base" (`EDB`) facts and puts them in the answer sets, as shown in Example 2, since it has no mean to distinguish type and data-base information. In contrast, a typed *GA* would *generate* only the residual grounding instantiation of Example 2, which has the more concise answer sets {`nice(p1)`, `nice(p2)`} and {`ugly(red)`, `ugly(blue)`}.

We remark that our model-theory fix the interpretation of external types, functions as well as of equality, but not of program predicates. The latter depends on the program and may have a variety of models. In pure logic programming, types, functions and equality are interpreted on term models. Equality is axiomatized by Clark's Equality Theory (CET) [17]. For the predicates defined in our example, CET includes the standard equality axioms and the freeness axioms $\neg(\texttt{red} = \texttt{blue})$ and $\neg(\texttt{p1} = \texttt{p2})$. A set of equations has a solution in CET iff a unifier exists. Thus standard unification is a sound and complete decision method for CET. If we add non-free functions, we no longer have pure term models and the unification algorithm cannot be applied as such.

*Example 4.* The following program uses the externally implemented type `#int` of integers, where we use the notation `#` to indicate external types and predicates, as in DLV-Complex [5].

[1] We remark that constructions such as `#hide` in Smodels, may not show the type information, but the latter is still present during the generation of the model.

```
unit usr.
type #int.
type usr --> john ; ted.
func age: usr -> #int.
     age(john) = 12.
     age(ted) = 15.
pred older(usr,usr,#int).
prog
    older(P1,P2,A) :- age(P1) = age(P2)+A, A > 0.
```

The domain of the external type `#int` is predefined, together with the usual operations on it. In our example, the function `age` is defined by two equations, although it could be an arbitrarily complex computable function. Both `age` and the predefined $+$, $=$ , $>$ are used in the body of the program rule. An ASP system should be able to solve constraint problems, preferably at the grounding stage. If this were the case, the user would not have to specify a `#maxint` and the `older` clause would not have to be grounded up to that.

The next example uses parametric polymorphism. We adopt the usual Prolog notation for lists `[_|_]` as well as using `'A` for the (quantified) type variable $\alpha$ (following SML's notation).

*Example 5.* We assume that `#list` is an external type constructor, while `rec` is internal.

```
unit store.
type  usr --> john ; ted.
type  #int.
type  rec('A,'B) --> r('A,'B).
type  #list('A).
pred  is_user(usr), stored_at(#int,'A), store(#list(rec(#int,'A))).
prog
      is_user(U) :- stored_at(K,U).
      stored_at(K,E) :- store(S), #member(r(K,E),S).
      store([r(1,john),r(2,ted)]).
```

Note that for every instantiation `[`$T$`/'A]`, the constructor `#list(`$T$`)` yields an external implementation of finite lists with elements of type $T$, together with a set of pre-defined operations such as `#member('A,#list('A))`.

Since the signature of a DLV-Complex program consists of two distinct parts, one containing functions and (external) data predicates, the other (internal) predicates, we present the type system, the abstract syntax and the intended model semantics in three steps over the next three Sections. We firstly introduce a many-sorted first order data signature $\Sigma^*$ induced by a polymorphic type system. Then we define the *intended data models* as suitable $\Sigma^*$-interpretations. Finally, we introduce the program signature, DLV-Complex programs and their intended models.

## 3 The Type and Term Language

We start from the type system. Types are defined inductively applying a type constructors $c$ (#$c$) to *internal* and *external* base types. Internal atomic types include a unit type

and a constant "o" denoting the type of propositions. To stay inside first-order logic, "o" can only appear at the right of an arrow type. Terms are obtained from logical variables $X$ and function symbols. Signatures give kinds to (any) type constructors and types to internal function symbols and external predicates. The latter two can be used polymorphically.

$$\begin{array}{rrcl}
\text{Base types} & T_0 & ::= & \alpha \mid \text{o} \mid \text{unit} \mid \#\mathit{int} \mid \#\mathit{string} \ldots \\
\text{Compound types} & T_{n+1} & ::= & c(\boldsymbol{T_n}) \mid \#c(\boldsymbol{T_n}) \\
\text{Types} & T & ::= & \bigcup_{n\geq 0} T_n \\
\text{Signature} & \Sigma & ::= & \emptyset \mid \Sigma, c : \boldsymbol{type} \to \text{type} \mid \Sigma, \#c : \boldsymbol{type} \to \text{type} \\
 & & & \mid\ \Sigma, f : \forall \boldsymbol{\alpha}.\, \boldsymbol{T} \to T' \mid \Sigma, \#p : \forall \boldsymbol{\alpha}.\, \boldsymbol{T} \to \text{o} \\
\text{Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, \alpha : \text{type} \mid \Gamma, X : T \\
\text{Terms} & t & ::= & X \mid f_{\boldsymbol{T}}(\boldsymbol{t}) \\
\text{Atoms} & A & ::= & \#p(\boldsymbol{t}) \mid \boldsymbol{t} = \boldsymbol{t'}
\end{array}$$

Before we discuss type checking for the above syntactic categories (Fig. 1), some common notation and conventions: tokens in bold such as $\boldsymbol{T}$ stand for a sequence, i.e. $T_1, \ldots, T_n$. We view $\Sigma$ and $\Gamma$ as sets and we use $\Gamma, e$ for $\Gamma \cup \{e\}$ with $e \notin \Gamma$. We will often write $\alpha$ instead of $\alpha : \text{type}$. $\Gamma \vdash_\Sigma J_1, \ldots, J_n$ abbreviates $\Gamma \vdash_\Sigma J_1, \ldots, \Gamma \vdash_\Sigma J_n$. We often write $\Gamma \vdash J$, leaving the signature $\Sigma$ understood. In the rule $sf$, $f$ is new in $\Sigma$. In the rules for Declarations, Types, Terms and Atoms, $\Gamma$ is assumed to be a well-formed context.

In the rules $fd$ and $pd$, $[\boldsymbol{I}/\boldsymbol{\alpha}]$ indicates the substitution of the variables $\boldsymbol{\alpha}$ by the types $\boldsymbol{I}$. It is worth remarking that while in our type systems constants can be parameterized by type *schemata*, rather than closed (simple) types, we do not support *impredicative* polymorphism. Indeed, we can think of this still as form of prenex polymorphism, where type schema can be seen as axiom schema in first-order logic.

*Example 6.* The concrete syntax of Example 5 corresponds to the signature:

$$\begin{array}{l}
\mathit{usr} : \text{type},\ \#\mathit{int} : \text{type},\ \mathit{john} : \mathit{usr},\ \mathit{ted} : \mathit{usr},\ 0 : \#\mathit{int},\ + : \#\mathit{int}, \#\mathit{int} \to \#\mathit{int}, \ldots \\
\mathit{rec} : \text{type}, \text{type} \to \text{type},\ r : \forall \alpha, \beta. \alpha, \beta \to \mathit{rec}(\alpha, \beta),\ \#\mathit{list} : \text{type} \to \text{type}, \\
\mathit{nil} : \forall \alpha. \#\mathit{list}(\alpha),\ \mathit{cons} : \forall \alpha. \alpha, \#\mathit{list}(\alpha) \to \#\mathit{list}(\alpha),\ \#\mathit{member} : \forall \alpha. \alpha, \#\mathit{list}(\alpha) \to o \ldots
\end{array}$$

The system in Fig. 1 is an extension with rules for external sources of knowledge of the one presented in [15]. One difference however is in the judgments "$f_{\boldsymbol{I}}$ decl" and "$\#p_{\boldsymbol{I}}$ decl", which introduce *indexed* function and predicate declarations. In [15] instead, indexes are only used at the semantic level, to provide a first-order interpretation of polymorphic declarations. Indexes can be reconstructed as we will discuss in Sect. 4.

Some relevant admissible properties of the type system are listed below, where $fv(J)$ indicates the set of the variables occurring free in $J$ and $\Gamma|fv(J)$ the restriction of $\Gamma$ to terms containing variables in $fv(J)$. Here we have $J$ range over Decl, Typs, Trms and Atms.

- uq) Uniqueness of typing: $\Gamma \vdash_\Sigma t : T$ and $\Gamma \vdash_\Sigma t : T'$ implies $T = T'$, modulo $\alpha$-conversion.
- str) Strengthening: if $\Gamma \vdash_\Sigma J$ then every variable free in $J$ is declared in $\Gamma$. Moreover, $\Gamma|fv(J) \vdash_\Sigma J$.

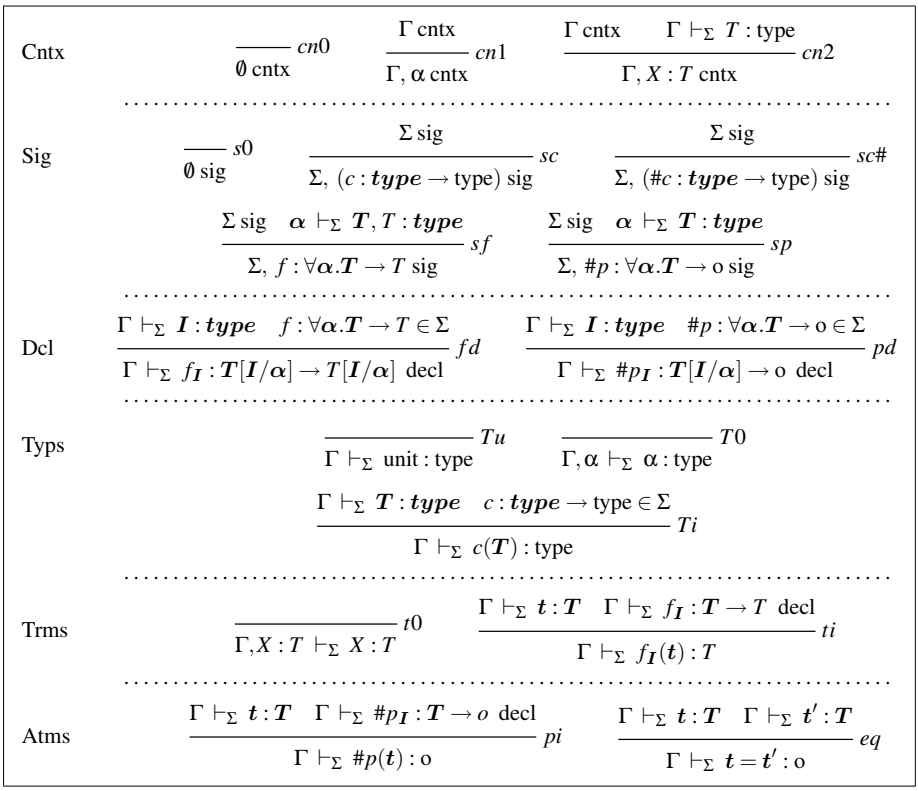


**Fig. 1.** Type checking rules

wk) Weakening: if $\Gamma \subseteq \Gamma'$ and $\Gamma \vdash_\Sigma J$, then $\Gamma' \vdash_\Sigma J$.

sbi) Substitution: we implicitly assume the $\alpha$-conversions needed to avoid capture and, for $\theta \equiv [T/\alpha]$, $\Gamma\theta$ is defined as: $\emptyset\theta = \emptyset$, $(\Gamma, \alpha : \text{type})\theta = (\Gamma\theta),\ \alpha : \text{type}$ and $(\Gamma,\ X : T')\theta = (\Gamma\theta),\ X : (T'\theta)$.

$$\frac{\Gamma, \alpha \vdash_\Sigma T : \text{type} \qquad \Gamma \vdash_\Sigma T' : \text{type}}{\Gamma[T'/\alpha] \vdash_\Sigma T[T'/\alpha] : \text{type}}\ sb1 \qquad \frac{\Gamma, X : T' \vdash_\Sigma t : T \qquad \Gamma \vdash_\Sigma t' : T'}{\Gamma \vdash_\Sigma t[t'/X] : T}\ sb2$$

## 4 Grounding Structures

We firstly introduce a many-sorted first order signature $\Sigma^*$. Then we define grounding structures as suitable $\Sigma^*$-interpretations. A polymorphic signature $\Sigma$ yields an infinite many sorted signature $\Sigma^*$ in the following way:

- Every ground type $T$ is a sort in $\Sigma^*$;
- for every $f : \forall \boldsymbol{\alpha}.\boldsymbol{T} \to T$ such that $\emptyset \vdash_\Sigma f_{\boldsymbol{I}} : \boldsymbol{T}[\boldsymbol{I}/\boldsymbol{\alpha}] \to T[\boldsymbol{I}/\boldsymbol{\alpha}]$ decl is provable, $f_{\boldsymbol{I}} : \boldsymbol{T}[\boldsymbol{I}/\boldsymbol{\alpha}] \to T[\boldsymbol{I}/\boldsymbol{\alpha}]$ is a function declaration in $\Sigma^*$;

– for every $\#p : \forall \boldsymbol{\alpha}.\boldsymbol{T} \to \mathrm{o}$ such that $\emptyset \vdash_\Sigma \#p_{\boldsymbol{I}} : \boldsymbol{T}[\boldsymbol{I}/\boldsymbol{\alpha}] \to o$ decl is provable, $\#p_{\boldsymbol{I}} : \boldsymbol{T}[\boldsymbol{I}/\boldsymbol{\alpha}] \to o$ is a predicate declaration in $\Sigma^*$.

$\Sigma^*$-interpretations are defined in a way similar to [13]:

**Definition 1.** *A $\Sigma^*$-interpretation is a function $\iota$ such that:*

– *Every $T$ in $\Sigma^*$ is mapped to a set $\iota(T)$, called the* domain *of $T$.*
– *Every $f_{\boldsymbol{I}} : \boldsymbol{T} \to T$ in $\Sigma^*$ is mapped to a function $\iota(f_{\boldsymbol{I}}) : \iota(\boldsymbol{T}) \to \iota(T)$.*
– *Every $\#p_{\boldsymbol{I}} : \boldsymbol{T} \to \mathrm{o}$ in $\Sigma^*$ is mapped to a relation $\iota(\#p_{\boldsymbol{I}}) \subseteq \iota(\boldsymbol{T})$.*

We are interested in a fixed interpretation G, called *grounding structure* or *pre-interpretation*.[2] The grounding structure of a program is defined by its type and function sections and the predefined types thereby declared.

*Example 7.* Recall the signature $\Sigma$ of Example 5. The intended grounding structure is the interpretation G defined as follows.

– Base type *usr*: $\mathrm{G}(usr) = \{john, ted\}$, $\mathrm{G}(john) = john$, $\mathrm{G}(ted) = ted$.
– Base type *#int*: it is interpreted according to its external implementation.
– Level 1 type $rec(\#int, usr)$: $\mathrm{G}(rec(\#int, usr))$ is the set generated by $r_{\#int,usr}$ starting from the interpretations of level 0 $\mathrm{G}(\#int)$ and $\mathrm{G}(usr)$. A representation is:
  - $\mathrm{G}(rec(\#int, usr)) = \{r_{\#int,usr}\} \times \mathrm{G}(\#int) \times \mathrm{G}(usr)$;
  - $\mathrm{G}(r_{\#int,usr})$ is the map $\langle i \in \mathrm{G}(\#int), u \in \mathrm{G}(usr)\rangle \mapsto \langle r_{\#int,usr}, i, u\rangle$.

  By abuse of notation, we represent triples $\langle r_{\#int,usr}, i, u\rangle$ as "terms" $r(i,u)$.
– Level 1 type $\#list(usr)$: $\mathrm{G}(\#list(usr))$ is the structure of the finite lists with elements from $\mathrm{G}(usr)$. The predefined operations are interpreted according to their implementation.
– ... and so on. We remark that the set-theoretic representation applies at each level.

We now move to the evaluation of a term with respect to an *assignment* in an interpretation.

**Definition 2.** *Let $\iota$ be a $\Sigma^*$-interpretation and $\Gamma$ a context. An* assignment *a for $\Gamma$ in $\iota$ maps every $\beta$ of $\Gamma$ to a sort $a(\beta)$ of $\Sigma^*$ and every $X : T$ of $\Gamma$ to a value $v \in \iota(T\,a)$, where $T\,a$ is the sort of $\Sigma^*$ obtained by grounding each variable $\beta$ occurring in $T$ by the sort $a(\beta)$.*

Given $\iota$, we denote by $\mathrm{ass}(\Gamma, \iota)$ the set of all the assignments for $\Gamma$ in $\iota$. Take $\Gamma \vdash_\Sigma t : T$ and $a \in \mathrm{ass}(\Gamma, \iota)$. The evaluation of $t$ with respect to $a$, denoted $t \rightsquigarrow_a v$, is performed according to the following rules:

$$\frac{}{X \rightsquigarrow_a a(X)}(ev1) \qquad \frac{\boldsymbol{t} \rightsquigarrow_a \boldsymbol{v} \qquad \Gamma_\Sigma \vdash f_{\boldsymbol{I}} : \boldsymbol{T} \to T \text{ decl}}{f_{\boldsymbol{I}}(\boldsymbol{t}) \rightsquigarrow_a \iota(f_{\boldsymbol{I}}\,a)(\boldsymbol{v})}(ev2)$$

**Theorem 1.** *Let $\Sigma$ be a signature, $\iota$ a $\Sigma^*$-interpretation, $a \in ass(\Gamma, \iota)$ and $\Gamma \vdash_\Sigma t : T$. Then there is a unique $v \in \iota(T\,a)$ such that $t \rightsquigarrow_a v$.*

[2] Our notion of pre-interpretation generalizes Lloyd's [17] interpreting #-predicates.

Theorem 1 follows by the existence and uniqueness of the $\Gamma \vdash_\Sigma f_{\boldsymbol{I}} : \boldsymbol{T} \to T$ decl judgment and as such depends on the indexing of function symbols via a simple table look up "modulo matching" over $\Sigma$.

*Example 8.* Let $\Sigma$ be the signature of Ex. 5 and $\Gamma = \{\alpha, \beta : \text{type}, X : \alpha, Y : \beta\}$. We have that $\Gamma \vdash_\Sigma r(X,Y) : rec(\alpha,\beta)$. Let G be the interpretation defined in Ex. 7 and $a \in \text{ass}(\Gamma, \text{G})$ such that $a(\alpha) = \#int$, $a(\beta) = usr$, $a(X) = 5$ and $a(Y) = ted$. Then $r(X,Y) \rightsquigarrow_a \langle r_{\#int,usr}, 5, ted \rangle$.

Finally, the truth relation $\iota \models_a A$ for a well-typed atom $A$ in $\Gamma$ and $a \in \text{ass}(\Gamma, \iota)$ is defined by cases on $A$:

- $\iota \models_a \#p_{\boldsymbol{I}}(\boldsymbol{t})$ iff $\boldsymbol{t} \rightsquigarrow_a \boldsymbol{v}$ and $\boldsymbol{v} \in \iota(p_{\boldsymbol{I}})$;
- $\iota \models_a t_1 = t_2$ iff $t_j \rightsquigarrow_a v$ for $j = 1, 2$.

## 5 Program Grounding

A *program* signature is of the form $\Sigma \cup \Pi$ where $\Pi$ is a set of *program predicate* declarations. We remark that $\Sigma$ still accounts for the external predicates $\#p$. As we saw in Section 2, the latter are interpreted on top of the grounding structure G defined in the type and function section of the program at hand. To formalize this, we resort to G-expansions. We start with $\Sigma \cup \Pi$ as above: the associated many sorted signature $(\Sigma \cup \Pi)^*$ has the form $\Sigma^* \cup \Pi^*$, where $\Pi^*$ contains the ground predicate declarations $p_{\boldsymbol{I}} : \boldsymbol{T} \to \text{o}$. A grounding structure G is a $\Sigma^*$-interpretation and a G-expansion is a $(\Sigma^* \cup \Pi^*)$-interpretation $\iota$ such that $\iota | \Sigma^* = \text{G}$, i.e. the *reduct* of $\iota$ w.r.t. $\Sigma^*$. We can represent $\iota$ as the set

$$H_\iota \;=\; \{\, \langle p_{\boldsymbol{I}}, \boldsymbol{v} \rangle \mid p_{\boldsymbol{I}} : \boldsymbol{T} \to \text{o} \in \Pi^* \text{ and } \boldsymbol{v} \in \iota(p_{\boldsymbol{I}}) \,\}$$

By abuse of notation, we write $p(\boldsymbol{v})$ instead of $\langle p_{\boldsymbol{I}}, \boldsymbol{v} \rangle$ and call it a G-*atom*. We dub the set of G-atoms the G-*base*, playing a role analogous to the Herbrand base. In this way, $H_\iota$ can be thought as a kind of "G-answer set".

*Example 9.* Back to program of Example 5 and the related grounding structure of Example 7. The following interpretation $\iota$ is the minimum model of the program (the unit "store"), seen as a set of G-atoms:

$$\{\, store([r(1, john), r(2, ted)]), stored\_at(1, john), stored\_at(2, ted), is\_user(john), is\_user(ted) \,\}$$

The loose notation $store([r(1, john), r(2, ted)]), stored\_at(1, john), \ldots$ means that

$$\langle cons, \langle r, 1, john, \rangle, \langle cons, \langle r, 2, ted \rangle, nil \rangle \rangle \in \text{G}(store_{usr}),\;\; \langle 1, john \rangle \in \text{G}(stored\_at_{usr}), \ldots$$

where $cons = cons_{rec(\#int,usr)}$ and $nil = nil_{rec(\#int,usr)}$.

Now we can formally introduce programs and enrich the syntax judgments of the previous section. The form of a clause is $A_1 \vee \cdots \vee A_n \leftarrow B_1, \ldots, B_m$, where $A_1, \ldots, A_n$ are program atoms and $B_1, \ldots, B_m$ are literals, i.e., atoms, external predicates, negated atoms or negated external predicates. For the sake of simplicity, we do not consider here classical negation, see the appendix.

- Atom $A$ ::= ... (as before).
- Program Atom $PA$ ::= $p(\boldsymbol{t})$, with $p$ defined in $\Pi$.
- Negated Atom $NA$ ::= $not\ A \mid not\ PA$ .
- Literal $B$ ::= $A \mid PA \mid NA$ .
- Head $HD$ ::= $A_1 \vee \cdots \vee A_n$; for $n = 0$, $HD \equiv false$.
- Body $BD$ ::= $B_1, \cdots, B_m$; for $m = 0$, $BD \equiv true$.
- Clause $C$ ::= $HD \leftarrow BD$ .

To distinguish external and internal program predicates, we split the judgment for well-formed atoms $p(\boldsymbol{t}) : \mathrm{o}^+$ and $\#q(\boldsymbol{t}) : \mathrm{o}^\#$, where $q$ is declared in $\Sigma$ and $p$ in $\Pi$. The corresponding rules are omitted, since they correspondingly split the rule *pi* in Fig. 1. We introduce the judgments $NA : o^-$, $HD$ : head , $BD$ : body and $C$ : clause , where in $B_i : \mathrm{o}^{s_i}$, the superscript $s_i$ may be $+, -, \#$.

$$\frac{\Gamma \vdash_{\Sigma\cup\Pi} A : \mathrm{o}^+}{\Gamma \vdash_{\Sigma\cup\Pi} not\ A : \mathrm{o}^-}\ at^- \qquad \frac{\Gamma \vdash_{\Sigma\cup\Pi} A : \mathrm{o}^\#}{\Gamma \vdash_{\Sigma\cup\Pi} not\ A : \mathrm{o}^-}\ at^-$$

$$\frac{\Gamma \vdash_{\Sigma\cup\Pi} A_1 : \mathrm{o}^+, \ldots, A_n : \mathrm{o}^+}{\Gamma \vdash_{\Sigma\cup\Pi} A_1 \vee \cdots \vee A_n : \mathrm{head}}\ h \qquad \frac{\Gamma \vdash_{\Sigma\cup\Pi} B_1 : \mathrm{o}^{s_1}, \ldots, B_m : \mathrm{o}^{s_m}}{\Gamma \vdash_{\Sigma\cup\Pi} B_1, \ldots, B_m : \mathrm{body}}\ b$$

$$\frac{\Gamma \vdash_{\Sigma\cup\Pi} HD : \mathrm{head} \quad \Gamma \vdash_{\Sigma\cup\Pi} BD : \mathrm{body}}{\Gamma \vdash_{\Sigma\cup\Pi} HD \leftarrow BD : \mathrm{clause}}\ cl$$

Let $\Sigma\cup\Pi$ be a program signature, G a grounding structure and $\iota$ be a G-expansion. The truth relation $\iota \models_a A$ is extended to clauses as usual. Similarly, the standard properties of the type system stated on Page 122 also hold.

Now, we aim to define the grounding of a program $P$ with respect to the structure G, notation $\mathrm{grnd}_\mathrm{G}(P)$, where $P$ consists of a set of clauses $\forall \boldsymbol{\alpha} : \mathrm{type}.\forall \boldsymbol{X} : \boldsymbol{T}.\ C$ such that $\boldsymbol{\alpha} : \mathrm{type}, \boldsymbol{X} : \boldsymbol{T} \vdash C : \mathrm{clause}$ . Let $a \in \mathrm{ass}(\Gamma, \mathrm{G})$; if we write $\Gamma \vdash C$, for "$\Gamma \vdash_\mathrm{G} C : \mathrm{clause}$ ", we can define grounding on the structure of $C$:

$$\begin{aligned}
\mathrm{grnd}_\mathrm{G}(\Gamma \vdash p(\boldsymbol{t}), a) &= p(\boldsymbol{v}) \quad \text{where } \boldsymbol{t} \rightsquigarrow_a \boldsymbol{v} \\
\mathrm{grnd}_\mathrm{G}(\Gamma \vdash not\ A, a) &= not\ \mathrm{grnd}_\mathrm{G}(\Gamma \vdash A, a) \\
\mathrm{grnd}_\mathrm{G}(\Gamma \vdash B_1, \ldots, B_m, a) &= \mathrm{grnd}_\mathrm{G}(\Gamma \vdash B_1, a), \ldots, \mathrm{grnd}_\mathrm{G}(\Gamma \vdash B_m, a) \\
\mathrm{grnd}_\mathrm{G}(\Gamma \vdash A_1 \vee \cdots \vee A_n, a) &= \mathrm{grnd}_\mathrm{G}(\Gamma \vdash A_1, a) \vee \cdots \vee \mathrm{grnd}_\mathrm{G}(\Gamma \vdash A_n, a) \\
\mathrm{grnd}_\mathrm{G}(\Gamma \vdash H \leftarrow B, a) &= \mathrm{grnd}_\mathrm{G}(\Gamma \vdash H, a) \leftarrow \mathrm{grnd}_\mathrm{G}(\Gamma \vdash B, a) \\
\mathrm{grnd}_\mathrm{G}(\Gamma \vdash C) &= \bigcup_{a \in ass(\Gamma, G)} \{\, \mathrm{grnd}_\mathrm{G}(\Gamma \vdash C, a) \,\} \\
\mathrm{grnd}_\mathrm{G}(P) &= \bigcup_{\Gamma \vdash C \in P} \mathrm{grnd}_\mathrm{G}(\Gamma \vdash C)
\end{aligned}$$

We can see $\mathrm{grnd}_\mathrm{G}(P)$ as a set of propositional clauses over the G-base. We have

$$H_\iota \models_{\mathrm{prop}} \mathrm{grnd}_\mathrm{G}(P) \quad \text{iff} \quad \iota \models P \tag{1}$$

where $\models_{\mathrm{prop}}$ is propositional truth. By the equivalence (1), we can define the G-answer sets of $P$ as those of $\mathrm{grnd}_\mathrm{G}(P)$. The latter are defined using the Gelfond-Lifschitz transformation, as explained in the appendix.

We now discuss the interaction between grounding and typing: this brings about some immediate problems. The first one is *ground type reconstruction*, as illustrated by the following example.

*Example 10*

```
pred  p(#list('A)), q(#list('A)), r(#list(#char)), t(#nat).
prog
      p([X|Y]) :- q(Y).
      q(Z) :- r(Z).
      r([]).
      t(3).
```

Now, given the ground instance $p([3]) \leftarrow q([\,])$, the atom $p([3])$ is in the minimal model of the program, which clearly violates the typing discipline. To overcome this problem, we have to assign to $[\,]$ the "right ground type" $T$. A solution is to assign an index to constants, apparently similar to how constants are declared in the simply-typed $\lambda$-calculus to recover a principal type, viz. $inl_B : A \to A \vee B$. In our case, $\emptyset \vdash_\Sigma [\,]_{\#int} : T$ has solution $T = \#list(\#int)$. Using indexed terms, the above clauses become $p([3]) \leftarrow q([\,]_{\#int})$, as well as $q([\,]_{\#char}) \leftarrow r([\,]_{\#char})$ and $r([\,]_{\#char})$. Now the standard ASP semantics correctly works.

The reader may wonder whether function indexing can be avoided and thus reconstructed. Interestingly enough, the dual of the property of *transparency* [11], for which the type declaration of every (internal) function is such that every type variable occurring in the domain also occurs in the range, ensures, in the absence of overloading, index reconstruction. Space prevents us from providing a formal proof.

Any non-trivial type signature is a source of infinity. We draw our inspiration from the way in which [5] deals with computable functions. We view type instantiation and grounding of individual variables as two separate phases. The intuition is that polymorphic predicate definitions behave as *open units*, i.e., groups of clauses *abstracting* type variables. During grounding an open unit is instantiated by grounding the type variables over a finite number of ground types of its signature ($tinst(P)$). Subsequently $grnd^\bullet(tinst(P))$ grounds individual variables. In the absence of a satisfactory notion of modules and their instantiation in the ASP framework yet, we will not formally define $tinst(P)$. Rather, we directly illustrate a case where modularity and type instantiation do not work properly. In this situation, the user has to provide the type instantiation with a so-called "`@with`-directive".

*Example 11*

```
unit choices.
#maxint=1.
type pencil --> p1 ; p2.
type color --> red ; blue.
func  col : pencil -> color.
      col(p1) = red.
      col(p2) = blue.
pred  choose(pencil), ok(#int), nice('A), ugly('B).
prog
```

```
nice(X) v ugly(Y).
choose(X) :- nice(X), not ugly(col(X)).
@with ugly(#int):
ok(X+1) :- nice(X), X < 1.
```

Type inference yields:

$$\text{poly:}\quad \alpha,\beta : \text{type}, X:\alpha, Y:\beta \vdash \mathit{nice}_\alpha(X) \vee \mathit{ugly}_\beta(Y) : \text{head}$$

$$\text{used by: i)}\ X : \mathit{pencil} \vdash \mathit{choose}(X) \leftarrow \mathit{nice}_{\mathit{pencil}}(X), \mathit{not}\ \mathit{ugly}_{\mathit{color}}(\mathit{col}(X)) : \text{clause}$$

$$\text{ii)}\ X : \#\mathit{int} \vdash \mathit{ok}(X+1) \leftarrow \mathit{nice}_{\#\mathit{int}}(X), X < 1 : \text{clause}$$

The type instantiation inferred from i) is $[\mathit{pencil}/\alpha, \mathit{color}/\beta]$, while ii) gives the partial instantiation $[\#\mathit{int}/\alpha]$. The `@with`-directive provides $[\#\mathit{int}/\beta]$. The result of the type instantiation is:

$$\begin{array}{ll}
c_1 & X : \mathit{pencil}, Y : \mathit{color} \vdash \mathit{nice}(X) \vee \mathit{ugly}(Y) : \text{head} \\
c_2 & X, Y : \#\mathit{int} \vdash \mathit{nice}(X) \vee \mathit{ugly}(Y) : \text{head} \\
c_3 & X : \mathit{pencil} \vdash \mathit{choose}(X) \leftarrow \mathit{nice}(X), \mathit{not}\ \mathit{ugly}(\mathit{col}(X)) : \text{clause} \\
c_4 & X : \#\mathit{int} \vdash \mathit{ok}(X+1) \leftarrow \mathit{nice}(X), X < 1 : \text{clause}
\end{array}$$

The inferred contexts will be relevant next, in Example 13. With the directive `@with ugly(#int)`, the instantiations of *nice* and of *ugly* in $c_1$ and $c_2$ do not interfere, i.e. composition is modular. However, the directive `@with ugly(color)` would have set $[\mathit{color}/\beta]$. Then the clauses $c_1$ and $c_2$ would have shared the same $\mathit{ugly}_{\texttt{color}}$ predicate.

A sufficient condition to avoid interference is to impose that the head predicates have different indexed names in different type instantiations. In our example, we could obtain this by declaring a unit `allNiceOrAllUgly`$(\alpha,\beta)$ with $\mathit{nice}, \mathit{ugly} : \forall\alpha,\beta.\alpha \to o$.

Now we explain the grounding phase. We extend the notion of context so that it contains everything related to the external predicates and operations, while the right-hand side of $\vdash$ contains only the program and the free functions. Namely we transform a clause $\Gamma \vdash C$ into an *extended clause* $\Gamma' | \Delta \vdash C'$, where $\Gamma \subseteq \Gamma'$, $\Delta$ is a set of $\Sigma$-formulas, which we call *constraints*, and $C'$ now contains only predicates from $\Pi$ and function symbols that are internal. This can be done by moving external predicates in the constraint context $\Delta$ and by introducing suitable equalities in the usual CLP's way.

*Example 12.* The clauses $c_1,\ldots,c_4$ of Example 11 are transformed into the following extended clauses $e_1, \ldots, e_4$:

$$\begin{array}{ll}
e_1 & X : \mathit{pencil}, Y : \mathit{color} \mid \top \vdash \mathit{nice}(X) \vee \mathit{ugly}(Y) \\
e_2 & X, Y : \#\mathit{int} \mid \top \vdash \mathit{nice}(X) \vee \mathit{ugly}(Y) \\
e_3 & X : \mathit{pencil}, U : \mathit{color} \mid U = \mathit{col}(X) \vdash \mathit{choose}(X) \leftarrow \mathit{nice}(X), \mathit{not}\ \mathit{ugly}(U) \\
e_4 & X, J : \#\mathit{int} \mid J = X+1, X < 1 \vdash \mathit{ok}(J) \leftarrow \mathit{nice}(X)
\end{array}$$

In $e_3$, the term $\mathit{col}(X)$ occurring in $c_3$ is replaced by a new special variable $U : \mathit{color}$ and $U = \mathit{col}(X)$ is inserted in the constraints context. Similarly for $J = X+1$ in $e_4$. Furthermore, the external predicate $X < 1$ has been moved to the constraints context.

Finally, we explain how *GA* uses extended contexts. Take the signature $\Sigma \cup \Pi$ and extended clause $\Gamma|\Delta \vdash C$. Clearly, *GA* may access both the signature and the grounding structure G. Take an assignment $a$ such that $a \in \mathrm{ass}(\Gamma, \mathrm{G})$. Since $\Delta$ contains $\Sigma$-formulas and G is a $\Sigma$-interpretation, *GA* can evaluate the truth relation $\mathrm{G} \models_a \Delta$.

$$\begin{aligned} \mathrm{egrnd}_{\mathrm{G}}(\Gamma \,|\, \Delta \vdash C, a) &= \begin{cases} \mathrm{grnd}_{\mathrm{G}}(\Gamma \vdash C, a) & \text{if } \mathrm{G} \models_a \Delta \\ \top & \text{otherwise} \end{cases} \\ \mathrm{egrnd}_{\mathrm{G}}(\Gamma \,|\, \Delta \vdash C) &= \bigcup_{a \in ass(\Gamma, G)} \{\, \mathrm{egrnd}_{\mathrm{G}}(\Gamma \,|\, \Delta \vdash C, a) \,\} \end{aligned}$$

It has the following property. Let $P$ be a $(\Sigma \cup \Pi)$-program and $P_e$ be the corresponding extended program. The grounding $\mathrm{egrnd}_{\mathrm{G}}(P_e)$ is the union of $\mathrm{egrnd}_{\mathrm{G}}(\Gamma|\Delta \vdash C)$, for $\Gamma|\Delta \vdash C \in P_e$. One can prove that the stable models of $\mathrm{grnd}_{\mathrm{G}}(P)$ coincide with those of $\mathrm{egrnd}_{\mathrm{G}}(P_e)$.

*Example 13.* Take the clause $e_3$ of Example 12. An assignment for its context is $a_1 = [p1/X, red/U]$. Since $\mathrm{G} \models_{a_1} U = col(X)$, we have

$$\mathrm{egrnd}_{\mathrm{G}}(e_3, a_1) \;=\; choose(p1) \leftarrow nice(p1), not\ ugly(red)$$

In contrast, for the assignment $a_2 = [p1/X, blue/U]$, we have $\mathrm{G} \not\models_{a_2} U = col(X)$. Thus $\mathrm{egrnd}_{\mathrm{G}}(e_3, a_2) = \top$, which can be ignored. Since we have finite types, we generate all the possible assignments and, at the end of the process, we obtain:

$$\begin{array}{ll} nice(p1) \vee ugly(red) & nice(p1) \vee ugly(blue) \\ nice(p2) \vee ugly(red) & nice(p2) \vee ugly(blue) \\ nice(0) \vee ugly(0) & nice(0) \vee ugly(1) \\ nice(1) \vee ugly(0) & nice(1) \vee ugly(1) \\ choose(p1) \leftarrow nice(p1), not\ ugly(red) & \\ choose(p2) \leftarrow nice(p2), not\ ugly(blue) & \\ ok(1) \leftarrow nice(0) & \end{array}$$

As the example shows, a type-driven grounder *GA* has the following features:

- In the grounding process, only well-typed instances are generated, viz. in Example 13, $nice(p1) \vee ugly(p1), nice(p1) \vee ugly(p2), \ldots$ are excluded.
- If a variable $X$ functionally depends on other variables $\boldsymbol{Y}$, one value is generated for it, depending on the values generated for $\boldsymbol{Y}$. For instance, in clause $e_3$ of Ex. 12 the variable $U$ functionally depends on $X$, thus in Ex. 13 only the clauses $choose(p1) \leftarrow nice(p1), not\ ugly(red)$ and $choose(p2) \leftarrow nice(p2), not\ ugly(blue)$ are added.

This should facilitate the generation of small ground instances. More importantly, types give the user a better control of the grounding process.

In the previous example, the grounding structure is finite. To complete the picture, we should consider the general case of possibly infinite grounding structures. We will not introduce new ideas, but consider how types could be managed by existing approaches. Specifically, we consider intelligent grounding w.r.t. the class of finitely ground programs [6], denoted by $\mathscr{FG}$. Here we explain the idea informally. Intelligent grounding

is performed starting from a modular decomposition $\langle M_1, M_2, \ldots, M_n \rangle$ of the program $P$. The module $M_1$ does not depend on any other modules and grounds to a $\mathrm{G}_1$ having the same stable models as $M_1$. We use $\mathrm{G}_1$ to build a $\mathrm{G}_2 = ig(M_2, \mathrm{G}_1)$, having the same stable models of $M_1 \cup M_2$. We proceed in this way until we obtain $\mathrm{G}_n = ig(M_n, \mathrm{G}_{n-1})$, which has the same models of $P$. If $\mathrm{G}_n$ is finite, one says that $P$ is an $\mathscr{FG}$-program.

*Example 14.* This contrived example shows a program that can be divided in two modules (we use `a,b,...` as constants of type `#char`).

```
unit show.
type #int, #char, #list('A).
pred select(#list('A)), show(#list('A)).
prog
  select([a]) v select([4]).                        %module m1
  select([b,a]).
  select([X]) :- select([X+X]).
  show(M):- select(L),select(M), #reverse(L,M).     %module m2
```

The module decomposition is $\langle m1, m2 \rangle$. Note that $m2$ is polymorphic. Type inferences yield the following extended clauses:

$$
\begin{array}{lrcl}
l_1 & & \vdash & \mathit{select}([a]) \vee \mathit{select}([4]) \\
l_2 & & \vdash & \mathit{select}([b,a]) \\
l_3 & X, I : \#\mathit{int} \mid I = X + X & \vdash & \mathit{select}([X]) \leftarrow \mathit{select}([I]) \\
l_4 & L, M : \#\mathit{list}(\#\mathit{char}) \mid \#\mathit{reverse}(L,M) & \vdash & \mathit{show}(M) \leftarrow \mathit{select}(L), \mathit{select}(M) \\
l_5 & L, M : \#\mathit{list}(\#\mathit{int}) \mid \#\mathit{reverse}(L,M) & \vdash & \mathit{show}(M) \leftarrow \mathit{select}(L), \mathit{select}(M)
\end{array}
$$

Now we proceed to the intelligent grounding phase, by applying the instantiation algorithm given in [6] and using extended clauses. We firstly ground the module $m1$. We assume that the grounding algorithm uses $a, b, c, \ldots$ as the internal representation for characters, $0, 1, 2, \ldots$ for numbers and $[v_1, \ldots, v_n]$ for lists, so that we do not need to distinguish between values and ground terms. Following [6], we start from the already ground clauses:

$$\vdash \mathit{select}([a]) \vee \mathit{select}([4]) \qquad\qquad \vdash \mathit{select}([b,a])$$

We consider the set $\mathscr{G}$ of atoms $\mathit{select}([a])$, $\mathit{select}([4])$, $\mathit{select}([b,a])$ occurring in the head of the ground clauses of $m1$. We use $\mathscr{G}$ to instantiate $l_3$ by looking for all the grounding substitutions that unify $\mathit{select}([I])$ with one of the atoms in it. We get:

$$X : \#\mathit{int} \mid 4 = X + X \;\vdash\; \mathit{select}([X]) \leftarrow \mathit{select}([4])$$

Differently from Example 13, the *GA* tries to instantiate clauses looking for body atoms that match with atoms in the current $\mathscr{G}$. In this way, the variable $X$ is not instantiated. It has to be grounded by solving the constraint $4 = X + X$. The solution is of course $X = 2$. The result of this step is:

$$
\begin{array}{rcl}
 & \vdash & \mathit{select}([a]) \vee \mathit{select}([4]) \\
 & \vdash & \mathit{select}([b,a]) \\
 & \vdash & \mathit{select}([2]) \leftarrow \mathit{select}([4]) \\
X, I : \#\mathit{int} \mid I = X + X & \vdash & \mathit{select}([X]) \leftarrow \mathit{select}([I])
\end{array}
$$

and the new $\mathscr{G}$ contains also $select([2])$. We iteratively apply the same process, which yields the instantiation $I = 2$ and the equation $2 = X + X$. We get:

$$\begin{array}{rcl} & \vdash & select([a]) \vee select([4]) \\ & \vdash & select([b,a]) \\ & \vdash & select([2]) \leftarrow select([4]) \\ & \vdash & select([1]) \leftarrow select([2]) \\ X, I : \#int \mid I = X + X & \vdash & select([X]) \leftarrow select([I]) \end{array}$$

Now no new atom is added to $\mathscr{G}$ and we stop the grounding process for $m1$. The next step is to use $m1$ to ground $m2$. We just consider one of the grounding operations activated in this phase, that is the instantiation and simplification of $select(L)$, $select(M)$ by $select([b,a])$ in the first clause of $m2$. We get $\cdots \mid \#reverse([b,a],[b,a]) \vdash show([b,a])$, where the body has been simplified because $select([b,a])$ is true. The evaluation of $\#reverse([b,a],[b,a])$ fails and the result is $\top$, i.e., no new clause is generated.

The main differences with respect to [6] are:

- Typing does not even consider ill typed substitutions.
- The treatment of external functions is shifted to the context. The context also contains the external atoms. Thus, the problem of grounding a clause in presence of predicates and operations defined in the grounding structure G is reduced to a constraint solving problem over G. The class of the groundable programs depends both on the grounding strategy and on the class of the constraints that the grounder can solve. For example, we could reconstruct value invention as treated in [5] in terms of programs that guarantee the solvability of a special class of constraints.

## 6 Related Work and Conclusions

The introduction of polymorphism in logic programming has required some care: there are predicate definitions that are semantically non-problematic, yet may lead to run-time errors [15, 11]. A sufficient condition is *definitional genericity*, which requires that the type of a defining occurrence of a predicate must be a renaming of the assigned type signature of the predicate. This ensures the type soundness result of [18], stating that if a program and a goal are well-typed, then at each resolution step variables can only be instantiated to terms consistent with their typing.

Although our system owes its static semantics to Mycroft and O'Keefe's seminal paper, the very different operational semantics of ASP allows us to dispense of those restrictions – as a matter of fact a dual property to transparency is instrumental in inferring type indexes that ensure ground type reconstruction.

Some papers in the ASP literature, e.g. [9, 2] use a many sorted language or deals with object-oriented features. Sorts, however, are merely syntactic sugar, to be translated back into first-order logic, hence they are not given a semantics in terms of many sorted first order signatures, not (polymorphic) type checking is addressed. W.r.t. object orientation OntoDLV [8] has recently extended DLV-Complex, catering to the specification and reasoning on enterprise ontologies; it extends ASP with object-oriented constructs, such as classes, objects, multiple inheritance, sets and lists. The semantics

is based on a notion of well-defined interpretation of a program, which is defined over the Herbrand Universe of the program. Differently from our approach, it lacks a formal notion of "external" structure where to interpret external predicates, moreover "generics" are not considered.

Modern prescriptive types system for logic programming have significantly evolved since the Eighties: for example, Mercury's type system [14] is a fairly complex extension of [18], supporting higher order types as λProlog, but also type classes similar to Haskell, in addition with a form of existential types. Moreover its mode system [20] provides support for subtypes as well as for uniqueness, similar to linear types. We plan to investigate how modes can be used in our approach to deal with grounding, as we have touched upon in Example 13.

Finally we mention the extensive research about polymorphic type inference in relation algebra and (object-oriented) databases, see [7, 4] for examples. However, this is only loosely related to our efforts, as there the aim is to extend a ML-like type system to capture the *principal* type of a program involving database operations or more in general of an expression in relational algebra, where a type is seen as a set of attribute names for a given schema.

In conclusion, we have presented a polymorphic types system for DLV-Complex and developed a declarative semantics for grounding, while studied its impact on model generation. Although our research is at an early stage, we believe that it shows the potential of types in ASP. A fortiori this should hold true for ASP systems such as Smodels [22] that implements a simpler logic. Beside the well-known advantages of prescriptive typing, in the ASP setting one may reap significant benefits w.r.t. the size of grounding; finally, we have shown that certain constraints can be directly codified in terms of typing, further reducing the size of the search space. This fits with current research aimed at finding a tighter relationship between ASP and constraint solving [2]. Directions for future research include:

- quantitatively evaluate the efficiency of type-driven grounding in terms of size and number of non-isomorphic models;
- extend the type system with other constructs, such as aggregate types, subtyping, etc., in order to account of other features of ASP, e.g. *preference* and *cardinality* constraints [16]. In particular, we speculate that we can elegantly define aggregate types using concrete data as in [3];
- From a practical standpoint, evaluate whether it is feasible to decompile the many-sorted language into untyped first order logic so that the present systems can readily be used or it is more effective to redesign grounding to take into full account the typing discipline.

**Acknowledgments.** This research was partially supported by the MIUR project "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva". We wish to thanks the anonymous referees for having appreciated the potential of the paper, notwithstanding several lacunae and imperfections. Those, we have done our best to fix.

## References


1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2003)
2. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an integration of answer set and constraint solving. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 52–66. Springer, Heidelberg (2005)
3. Bertoni, A., Mauri, G., Miglioli, P.: A characterization of abstract data as model-theoretic invariants. In: Maurer, H.A. (ed.) ICALP 1979. LNCS, vol. 71, pp. 26–37. Springer, Heidelberg (1979)
4. Buneman, P., Ohori, A.: Polymorphism and type inference in database programming. ACM Trans. Database Syst. 21(1), 30–76 (1996)
5. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. Ann. Math. Artif. Intell. 50(3-4), 333–361 (2007)
6. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: Theory and implementation. In: de la Banda, M.G., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 407–424. Springer, Heidelberg (2008)
7. Van den Bussche, J., Waller, E.: Polymorphic type inference for the relational algebra. J. Comput. Syst. Sci. 64(3), 694–718 (2002)
8. Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., Leone, N.: OntoDLV: an ASP-based system for enterprise ontologies. Journal of Logic and Computation (to appear, 2009)
9. Gelfond, M.: Answer sets. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of knowledge representation, vol. 7. Elsevier, Amsterdam (2007)
10. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080 (1988)
11. Hanus, M.: Horn clause programs with polymorphic types: Semantics and resolution. Theor. Comput. Sci. 89(1), 63–106 (1991)
12. Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). Sci. Comput. Programming 58(1-2), 115–140 (2005)
13. Hill, P.M., Topor, R.W.: A semantics for typed logic programs. In: Types in Logic Programming, pp. 1–62. MIT Press, Cambridge (1992)
14. Jefferey, D.: Expressive Type Systems for Logic Programming Languages. PhD thesis, The University of Melbourne (2002)
15. Lakshman, T.L., Reddy, U.S.: Typed PROLOG: A semantic reconstruction of the Mycroft-O'Keefe type system. In: ISLP, pp. 202–217 (1991)
16. Leone, N., et al.: The DLV system for knowledge representation and reasoning. ACM TOCL 7(3), 499–562 (2006)
17. Lloyd, J.W.: Foundations of logic programming (2nd extended ed.). Springer, Heidelberg (1987)
18. Mycroft, A., O'Keefe, R.A.: A polymorphic type system for PROLOG. Artif. Intell. 23(3), 295–307 (1984)
19. Nadathur, G., Qi, X.: Optimizing the runtime processing of types in polymorphic logic programming languages. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS, vol. 3835, pp. 110–124. Springer, Heidelberg (2005)
20. Overton, D., Somogyi, Z., Stuckey, P.J.: Constraint-based mode analysis of Mercury. In: PPDP, pp. 109–120. ACM, New York (2002)
21. Pfenning, F.: Types in logic programming. MIT Press, Cambridge (1992)
22. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artif. Intell. 138(1-2), 181–234 (2002)

## A Syntax and Semantics of DLV

In this section we provide the bare minimum of definitions of the syntax and semantics of DLV [16] (disjunctive Datalog) and we briefly describe its extension with functions [5].

A *term* is either a variable or a constant. An *atom* $a$ is an expression $p(t_1,\ldots,t_n)$, where $p$ is a predicate of arity $n$ and $t_1,\ldots,t_n$ are terms. A *classical literal* is an atom of the form $a$ or $\neg a$, where $\neg$ is the classical negation. A *literal* $l$ has the form $l$ or *not* $l$, where $l$ is a classical literal and *not* represents the *negation as failure* [17]. A *disjunctive rule* $r$ is a formula of the form

$$A_1 \vee \cdots \vee A_n \leftarrow B_1,\ldots,B_k, \mathit{not}\ B_{k+1},\ldots,\mathit{not}\ B_m$$

where $A_1,\ldots,A_n,B_1,\ldots,B_m$ are classical literals, $n \geq 0$ and $m \geq k \geq 0$. The disjunction $A_1 \vee \cdots \vee A_n$ is the *head* of $r$, the conjunction $B_1,\ldots,B_k,\mathit{not}\ B_{k+1},\ldots,\mathit{not}\ B_m$ is the *body* of $r$. We use the following notations: $HD(r) = \{A_1,\ldots,A_n\}$ (the *head literals*), $BD^+(r) = \{B_1,\ldots,B_k\}$ (the *positive body*), $BD^-(r) = \{B_{k+1},\ldots,B_m\}$ (the *negative body*). A rule with empty head (i.e, $n = 0$) is called a *constraint*, a rule with empty body (i.e., $k = m = 0$) is called a *fact*. A rule $r$ is *safe* if any variable occurring in $r$ also appears in $BD^+(r)$. A *DLV program* is a finite set of safe rules. We recall some basics of logic programming: The Herbrand Universe $U_P$ of $P$ is the set of all constants occurring in $P$ (we assume $U_P \neq \emptyset$). The Herbrand Base $B_P$ of $P$ is the set of all the literals constructible from the predicate symbols of $P$ and the constants of $U_P$. For every rule $r$ of $P$, $grnd(r)$ denotes the set of rules obtained by applying all possible substitutions from the variables in $r$ to elements of $U_P$; $grnd(P)$ is the union of the sets $grnd(r)$, for every $r$ of $P$. An *interpretation* $I$ of $P$ is a consistent subset of $B_P$, i.e. $I$ does not contain pairs of classical literals of the form $a$ and $\neg a$. Let $P$ be a positive program (namely, for every rule $r$ of $P$, $BD^-(r) = \emptyset$) and let $I \subseteq B_P$ be an interpretation.

- $I$ is *closed under* $P$ if, for every $r \in grnd(P)$, $BD^+(r) \subseteq I$ implies $HD(r) \cap I \neq \emptyset$.
- $I$ is an *answer set* for $P$ if $I$ is minimal under set inclusion among all the interpretations closed under $P$.

The *reduct* or *Gelfond-Lifschitz* transformation of a ground program $P$ with respect to an interpretation $I \subseteq B_P$ is the positive ground program $P^I$ obtained from $P$ by:

- deleting all rules $r \in P$ such that $BD^-(r) \cap I \neq \emptyset$;
- deleting the negative body from the remaining rules.

An *answer set* for a program $P$ is an interpretation $I \subseteq B_P$ such that $I$ is an answer set of $grnd(P)^I$.

Recently, DLV has been extended to DLV-Complex by introducing functions, sets and lists. This is obtained through the concept of *value invention* [5], which is based on the possibility of using externally defined predicates in the body of clauses. External predicates must be (well) moded, e.g. `#p(i,o)` denotes a call to an external predicates

`p(c,X)` with ground input `c` that will "invent" an output value for `X`. The semantics of DLV programs has to be extended in order to treat external predicates, which are interpreted by means of external oracles. A DLV-Complex program with external predicates is required to be safe and *VI-restricted* [5]. In this case, it is guaranteed that the grounding process will halt with a finite ground program whose answer sets are the expected ones. A function symbol `f(X)` is introduced in DLV-Complex by associating with it a constructor predicate `#f(o,i)` and a destructor predicate `#f(i,o)`. Given a value `c`, `#f(V,c)` invents a value `v` representing the ground term `f(c)`, while `#f(v,X)` reconstructs the value `c`.