Logic Programming with Constructor-based Type Constraints

Hans-Joachim Goltz
National Research Centre for Informatics
and Information Technology, GMD-FIRST Berlin
Rudower Chaussee 5, D – 12489 Berlin, Germany
email: goltz@first.gmd.de

Abstract

A concept of types and type sorts for logic programming is introduced, where types and type sorts are regarded as constraints. Instead of defining "well-sorted" terms and substitutions, type constraints and type sort constraints are defined. The approach is based on term models. A type is interpreted as a set of object terms and a type sort is interpreted as a set of types. Parameters of types can be object variables, object terms, and type variables. Types are defined by means of type rules which can be regarded as rules for the generation of the elements belonging to the corresponding type. The definitions of types and type sorts can be transformed into definite clauses and the least Herbrand model of these definite clauses are used for the semantics of the types and type sorts. Type constraint processing is based on SLD-resolution with a specific computation rule. Furthermore, computing with type constraints and type sort constraint is discussed.

1 Introduction

Structuring the universe of discourse is a successful method in the field of knowledges representation and automated reasoning. The universe is subdivided into a certain number of subuniverses. Subuniverses may contain one another, may partially overlap, or may be disjoint. The introduction of types and sorts provides an appropiate means for such a structuring of the universe. Much research has been devoted to this task, and several proposals for various type systems have been made. Several approaches are described in [14] and further approaches are discussed, for example, in [13], [1], and [16]. The majority of the approaches are based on many-sorted logic or order-sorted logic.

Constraint logic programming (CLP) is an elegant generalization of logic programming where the basic operation of logic programming, the unification, is replaced by the more general concept of constraint solving over a computation domain. The general scheme CLP($\mathcal{D}$) of constraint logic programming was developed in [9]. In [3] a predicate logic with restricted quantifiers is discussed and a resolution principle for clauses with constraints was proposed, where unification is replaced by testing constraints for satisfiability over a restriction theory. Furthermore, in this book, a sort theory is also mentioned as an example of such a restriction theory. Further examples of papers dealing with constraints and types are [2], [12], and [17].

In this paper a concept of types and type sorts for logic programming is introduced, where types and type sorts are regarded as constraints. Instead of defining "well-sorted" terms and substitutions, we define type constraints and type sort constraints. The approach is based on term models. We assume that the universe of discourse can be represented by ground data terms. Data terms (also called constructor terms) are generated by contructor symbols and variables. Constructor symbols are function symbols which are interpreted by themselves and do not have any operational semantics. Ground data terms are data terms without any variables. A data term with variables can be regarded as a representation of a specific subset of the universe of discourse. However, generally, not every ground data term can be assigned to an element of the universe of discourse. The construction of data terms can be restricted if typed variables are used instead of general variables, where a type characterizes a specific set of data terms. For the representation of types data terms are used. Thus we have to distinguish between two kinds of constructor symbols: constructor symbols for the representation of the basic objects of the universe and constructor symbols for the representation of types. In the sequel we use the names "*object constructor*" and "*type constructor*" for these kinds of constructor symbols. Furthermore, data terms generated only by object constructors and variables are called object terms, and data terms for the representation of types are called type terms.

Types can be defined by type rules of the form $s \; :=> \; t \; \longleftarrow \; \mathcal{C}$, where s is a type term, t is an object term and $\mathcal{C}$ is a set of type constraints (expressions of the form $x : r$). We use the special binary relation ":" in order to determine the elements belonging to a type. For example, the type of natural numbers can be defined by the following type rules: $nat :=> suc(x) \longleftarrow x : nat$ and $nat :=> 0$ (nat is a type constructor and suc is a object constructor). The concept that a type is specified by means of the elements belonging to this type is especially emphasized by the chosen syntax. Type rules can be regarded as rules for the generation of the elements belonging to the corresponding type.

Since types are regarded as constraints, $x : s_1$ and $x : s_2$ with different types s_1, s_2 could be solutions of a type constraint $t : s$, where x is a variable occuring in t. This means that the elements belonging to these types s_1 and s_2 satisfy the type constraint $t : s$.

We introduce type sorts so that the set of types itself can be structured. For example, type sorts may be introduced for types representing subsets of numbers, for types representing sets of lists, and for types representing specific object terms. A type sort does not contain any parameter and is interpreted as a set of types. A subset relation can exist between some type sorts. We assume that there is a type sort *types* which is interpreted as the set of all types. The introduction of a type sort for a set of types has more expressive power than a type representing the union of this set. There are better possibilities of structuring the universe by the introduction of type sorts.

We show that the definitions of types and type sorts can be transformed into definite clauses. Therefore, the semantics of the types and type sorts can be based on the least Herbrand model of the constraint theory consisting of these definite clauses. An important operation with types is the generation of the intersection of two types. If computing with types is regarded as constraint processing, then it is sufficient to generate an element which belongs to the intersection (or to prove that such an element does not exist). Type constraint processing is based on SLD-resolution with a specific computation rule. Type checking, i.e. checking if a type constraint without any variable is satisfiable, is shown to be decidable.

The type system discussed in this paper differs mainly from other approaches of type systems by the following points:

- Types are strictly regarded as constraints and are integrated into constraint processing.
- Besides type variables, object terms and object variables can be parameters of types.
- The set of types is structured by type sorts (note that the introduction of type sorts is mentioned in [8]).

Usually, the definitions of types have to satisfy certain conditions in order to guarantee good properties of computation. In the presented approach these conditions are syntactical conditions which can be easily checked.

We assume the reader to be familiar with the usual notations and basic results of the foundations of logic programming and constraint logic programming (see e.g. [11], [9, 10]).

2 Notation

The set of function symbols $\mathbb{F}$ is subdivided into symbols for object constructors $\mathbb{F}_o$, symbols for type constructors $\mathbb{F}_t$ and 0-ary symbols for type sorts $\mathbb{F}_s$. These subsets are supposed to be pairwise disjoint. We use the notation f/n if we want to express that a function symbol f is associated with the arity n. In the following we suppose that $\mathbb{F}$ is given and finite, $types \in \mathbb{F}_s$, and that both $\mathbb{F}_o$ and $\mathbb{F}_t$ contain at least one function symbol of arity 0. Two sorts of variables are introduced: a denumerable set $\mathbb{V}_o = \{x, y, z, x_1, y_1, z_1, x_2, \ldots\}$ of *object variables* and a denumerable set $\mathbb{V}_t = \{\alpha, \alpha_1, \alpha_2, \ldots\}$ of *type variables* (throughout this paper we use these notations for variables). The union of these sets of variables is denoted by $\mathbb{V}$ and we use the notation of an object variable, if we want to mark a variable of any sort. If Ex is a syntactical expression, the set of (object and type) variables which are contained in it is denoted by $var(Ex)$. If a set $V \subseteq \mathbb{V}$ of variables and a set $F \subseteq \mathbb{F}$ of function symbols are given, the set of first order terms $Tm(V, F)$ is defined as usual. Elements of $Tm(\mathbb{V}_o, \mathbb{F}_o)$ are called *object terms* and denoted by $\mathcal{O}$.

Lists are special object terms. We use the notation $[t_1, \ldots, t_n]$ for a list with the elements $t_1, \ldots, t_n$. Then $[t \mid z]$ is the usual notation of a list with the first element t. The empty list is denoted by $[\,]$.

The set $\mathcal{T}$ of *type terms* is the set of terms $h(t_1, \ldots, t_n)$ satisfying the following conditions: $h/n \in \mathbb{F}_t$ and $t_1, \ldots, t_n \in \mathbb{V}_t \cup \mathcal{T} \cup \mathcal{O}$. We say a term is ground if there is not any variable in it. The set of *ground object terms* is denoted by $\mathcal{O}_0$ and the set of *ground type terms* by $\mathcal{T}_0$. A *type constraint* is a syntactical expression of the form $t : s$, where one of the following conditions must be satisfied: $t \in \mathcal{O}$ and $s \in \mathbb{V}_t \cup \mathcal{T}$ **or** $t \in \mathbb{V}_t \cup \mathcal{T}$ and $s \in \mathbb{F}_s$. A *variable constraint* is a type constraint $x : s$ with $x \in \mathbb{V}_o \cup \mathbb{V}_t$ and $x \notin var(s)$. We use a tilde to denote a sequence of terms. For example, $\tilde{t}$ denotes the terms $t_1, \ldots, t_n$. $\exists\phi$ denotes the existential closure of a formula ϕ.

3 Specifications of Type Sorts and Types

A definition of a type consists of a declaration of this type and a set of type rules (the definition rules). The declaration of a type determines the types and type sorts, respectively, of the arguments and specifies the type sort to which the type belongs. The binary relation ":" is used for determining the elements belonging to a type. The syntax is chosen mainly for two reasons. The definition of the special relation ":" is syntactically different from definitions of other relations. The concept that types are specified by means of the elements belonging to these types is especially emphasized. Thus type rules can also be regarded as rules for the generation of the elements belonging to the corresponding type.

Definition 1 *A declaration of a type* $h/n \in \mathbb{F}_t$ *is an expression* **type** $h : s_1 \times \ldots \times s_n \rightarrow b$, *where* $b \in \mathbb{F}_s$ *and* $s_1, \ldots, s_n \in \mathbb{F}_s \cup \mathcal{T}_0$.

A declaration **type** $h : s_1 \times \ldots \times s_n \rightarrow b$ means that the arguments of a type term $h(t_1, \ldots, t_n)$ have to belong to the corresponding types $s_1, \ldots, s_n$ and that a type represented by a term $h(t_1, \ldots, t_n)$ belongs to the type sort b.

Definition 2 *A type rule of a type is an expression* $s \;{:=>}\; u \;\longleftarrow\; \mathcal{C}$ *such that the following conditions are satisfied (x denotes any variable):*

(1) $s = h(t_1, \ldots, t_n)$ *with* $h \in \mathbb{F}_t$, $t_1, \ldots, t_n \in \mathcal{O} \cup \mathcal{T}_0 \cup \mathbb{V}_t$, *and* $u \in \mathcal{O}$;

(2) $\mathcal{C}$ *is a set of variable constraints with* $var(\mathcal{C}) \subseteq var(u) \cup var(s)$;

(3) if $x : r \in \mathcal{C}$, *then* $x \in var(u) \setminus var(s)$, $var(r) \subseteq var(s)$, *and there is not any type variable which occurs more than once in* r *(if* $var(u) \subseteq var(s)$, *then* $\mathcal{C} = \emptyset$*);*

(4) if u *is a variable, then* $\mathcal{C} = \{u : r\}$, $r \notin \mathbb{V}_t$, *and the type constructor* h *does not occur in* r;

(5) if $x : r \in \mathcal{C}$ *is a variable constraint which contains the type constructor* h, *then* $r = h(r_1, \ldots, r_k)$, h *does not occur in any argument* r_i *of* r, *and for each* $i \leq k$ *holds* $r_i \in \mathbb{V}_t$ *or* $var(r_i) \cap \mathbb{V}_t = \emptyset$.

Example 1 Two type definitions are given: the type of finite natural numbers $\{0, suc(0), \ldots, x\}$ and the type of lists with a certain number of elements (the definition of $nat/0$ is given in Section 1).

type $f\_nat : nat \rightarrow number\_types$
$f\_nat(x) \;{:=>}\; x$
$f\_nat(suc(x)) \;{:=>}\; y \;\longleftarrow\; y : f\_nat(x)$

type $list : nat \times types \rightarrow list\_types$
$list(0, \alpha) \;{:=>}\; [\,]$
$list(suc(x), \alpha) \;{:=>}\; [y \mid z] \;\longleftarrow\; y : \alpha,\; z : list(x, \alpha)$ $\triangleleft$

Let $h(t_1,\ldots,t_n)$ $:=>$ u $<\!\!-\!\!-$ $\mathcal{C}$ be a type rule. If the argument t_i of $h(t_1,\ldots,t_n)$ is a type parameter, then t_i is a type variable or a ground type term (condition (1) of Definition 2). The conditional part $\mathcal{C}$ of this type rule cannot introduce any new variable (condition (2)). The conditional part $\mathcal{C}$ contains variable constraints only for such variables which belong to u and do not belong to $h(t_1,\ldots,t_n)$ (condition (3)). Condition (5) of Definition 2 restricts the possibilities of recursive definitions.

Note that a type rule s $:=>$ $x <\!\!-\!\!-$ $x : r$ states that the interpretation of r is a subset of the interpretation of s.

A type can also be defined by means of a set definition ($s := \{t_1,\ldots,t_m\}$, where $s \in \mathbb{F}_t$ and $t_1,\ldots,t_m$ are 0-ary function symbols of $\mathbb{F}_o$) or by means of a union of types (s $:=$ s_1 $++$ $\ldots$ $++$ s_m with $s_1,\ldots,s_m \in \mathcal{T}$ and $var(\{s_1,\ldots,s_m\}) \subseteq var(s)$). However, such a definition can be transformed into a set of type rules. A definition of a type by means of a set definition $s := \{u_1,\ldots,u_m\}$ can be transformed into the type rules $s :=> u_1$, ..., $s :=> u_m$. A definition of a type by means of $s := s_1 ++ \ldots ++ s_m$ can be transformed into the type rules $s :=> x <\!\!-\!\!- x\!:\!s_1$, ..., s $:=>$ x $<\!\!-\!\!-$ $x\!:\!s_m$, where x is a new variable (x does not occur in $s, s_1,\ldots,s_m$). Therefore, in the following, we can suppose that all types are defined by means of type rules.

Example 2 Examples of type definitions by means of a set definition and a union definition are:

type $ford : \rightarrow$ $vehicle\_types$
$ford := \{$ford_1, ford_2, ford_3, ford_4 $\}$

type $car : \rightarrow$ $vehicle\_types$
$car :=$ $ford$ $++$ $opel$ $++$ $mercedes$ $++$ bmw

These definitions can be transformed into the type rules:

$ford$ $:=>$ ford_1
$ford$ $:=>$ ford_2
$ford$ $:=>$ ford_3
$ford$ $:=>$ ford_4

car $:=>$ x $<\!\!-\!\!-$ $x\!:\!ford$
car $:=>$ x $<\!\!-\!\!-$ $x\!:\!opel$
car $:=>$ x $<\!\!-\!\!-$ $x\!:\!mercedes$
car $:=>$ x $<\!\!-\!\!-$ $x\!:\!bmw$ $\triangleleft$

We introduce type sorts so that the set of types itself can be structured. A type sort is interpreted as a set of types. A type sort does not contain any parameter, but a subset relation can exist between some type sorts. We assume that there is a type sort $types$ which is interpreted as the set of all types.

Definition 3 *A specification of type sorts consists of a set* $\mathbb{F}_s^b \subseteq \mathbb{F}_s$ *and a set of union definitions* $a := b_1$ $++$ $\ldots$ $++$ b_k *with* $a \in \mathbb{F}_s \setminus (\mathbb{F}_s^b \cup \{\,types\,\})$ *and* $b_1,\ldots,b_k \in \mathbb{F}_s$ *. If* $\mathbb{F}_s \setminus \{\,types\,\} \neq \emptyset$*, then* $types \notin \mathbb{F}_s^b$ *. The elements of* $\mathbb{F}_s^b$ *are called basic type sorts.*

Example 3 A specification of type sorts is:
$\mathbb{F}_s^b = \{\,number\_types, list\_types, vehicle\_types, person\_types, country\_types\,\}$,
$basic\_object\_types := vehicle\_types$ $++$ $person\_types$ $++$ $country\_types$. $\triangleleft$

The introduction of a type sort for a set of types has more expressive power than a type representing the union of this set. There are better possibilities of structuring the universe by the introduction of type sorts.

Example 4 Let us consider the type definition:

type $vehicle\_list :$ $vehicle\_types$ $\rightarrow$ $list\_types$
$vehicle\_list(\alpha)$ $:=>$ $[\,]$
$vehicle\_list(\alpha)$ $:=>$ $[y\,|\,z]$ $<\!\!-\!\!-$ $y\!:\!\alpha$, $z\!:\!vehicle\_list(\alpha)$

Supposed that *car* and *truck* are types of the type sort *vehicle_types* and that there is not any type s of the type sort *vehicle_types* such that s contains elements of the types *car* and *truck*. Then a list containing elements of *car* and *truck* cannot belong to the type *vehicle_list*(α). However, if there is a type $vehicle := car ++ truck$, then such a list belongs to the type *vehicle_list*(α). $\triangleleft$

We assume that type sorts and types are hierarchically defined (i.e. a type can be only used in a type definition, if this type is already defined), where a direct recursive definition of types is allowed. This assumption can be formally defined by a function η of $\mathbb{F}$ into the natural numbers such that the following conditions are satisfied:

(C0) If $f \in \mathbb{F}_o$, $h \in \mathbb{F}_t$, and $b \in \mathbb{F}\backslash\{\, types \,\}$, then $\eta(f) < \eta(h) < \eta(b) < \eta(types)$.

(C1) If $b := b_1 \;{+}{+}\; \ldots \;{+}{+}\; b_k$ is a definition of type sorts, then $\eta(b_i) < \eta(b)$ for each $i \le k$.

(C2) If **type** $h : s_1 \times \ldots \times s_n \;\rightarrow\; b$ is a declaration of some type, then $\eta(g) < \eta(h)$ for each $g \in \mathbb{F}_t$ occuring in $s_1, \ldots, s_n$.

(C3) If $h(t_1, \ldots, t_n) \;:=>\; u \;<\!\!-\!\!-\; \mathcal{C}$ is a type rule, then

- $\eta(g) < \eta(h)$ for each $g \in \mathbb{F}_t$ contained in $t_1, \ldots, t_n$;
- $\eta(g) < \eta(h)$ for each $g \in \mathbb{F}_t$ which occurs in $\mathcal{C}$ and is different from h.

4 Theory of Type Constraints

This section describes how given specifications of types and type sorts can be transformed into a set of definite clauses called type clauses. Furthermore, the semantics of types and type sorts are defined.

If $b := b_1 \;{+}{+}\; \ldots \;{+}{+}\; b_k$ is a definition of some type sort, then the following clauses are generated: $\alpha : b \;<\!\!-\!\!-\; \alpha : b_1$, ..., $\alpha : b \;<\!\!-\!\!-\; \alpha : b_k$. Moreover the clause $\alpha : types \;<\!\!-\!\!-\; \alpha : b$ is added for each type sort b which does not occur in the left hand side of some definition of a type sort.

If $h(t_1, \ldots, t_n) :=> u \;<\!\!-\!\!-\; y_1 : r_1, \;\ldots,\; y_k : r_k$ is a type rule of a type definition and **type** $h : s_1 \times \ldots \times s_n \;\rightarrow a$ is the declaration belonging to this definition, then the following definite clause is generated: $u : h(t_1, \ldots, t_n) \;<\!\!-\!\!-\; y_1 : r_1, \;\ldots,\; y_k : r_k,\; t_1 : s_1, \;\ldots,\; t_n : s_n$. In addition the clause $h(x_1, \ldots, x_n) : a \;<\!\!-\!\!-\; x_1 : s_1, \;\ldots,\; x_n : s_n$ is generated for the declaration.

Example 5 The following clauses are generated by the definition of $list/2$ (see Example 1):

$$list(x, \alpha) : list\_types \;<\!\!-\!\!-\; x : nat,\; \alpha : types$$
$$[\,] : list(0, \alpha) \;<\!\!-\!\!-\; \alpha : types$$
$$[y \mid z] : list(suc(x), \alpha) \;<\!\!-\!\!-\; y : \alpha,\; z : list(x, \alpha),\; suc(x) : nat,\; \alpha : types \qquad \triangleleft$$

A type clause $u : h(\tilde{t}) \;:=>\; u \;<\!\!-\!\!-\; \mathcal{C}$ is said to be recursive, if the type constructor h occurs in $\mathcal{C}$. In the other case this clause is called nonrecursive type clause.

The described transformation can be improved for recursive type clauses such that type checking of the arguments of a recursive type is only carried out in the first call and not in a recursive call. If the definition of $h/n \in \mathbb{F}_t$ is recursive and **type** $h : s_1 \times \ldots \times s_n \;\rightarrow a$ is the corresponding declaration, then $\mathbb{F}_t$ is extended by a new symbol h_{rec}/n of a type constructor, the definite clause $x : h(x_1, \ldots, x_n) <\!\!-\!\!-\; x : h_{rec}(x_1, \ldots, x_n),\; x_1 : s_1, \;\ldots,\; x_n : s_n$ is added, and for each type rule $h(\tilde{t}) :=> u \;<\!\!-\!\!-\; y_1 : r_1, \;\ldots,\; y_k : r_k$ the type clause $u : h_{rec}(\tilde{t}) \;<\!\!-\!\!-\; y_1 : \delta(r_1), \;\ldots,\; y_k : \delta(r_k)$ is generated, where $\delta(r_i)$ is generated from r_i by the replacement of h by h_{rec} (i.e. the types of the arguments $\tilde{t}$ are not checked). Note that the type function h_{rec}/n may occur in these type clauses, only.

Example 6 Using this improvement the following clauses are generated by the definition of $list/2$ (see Examples 1 and 5):

$$list(x, \alpha) : list\_types \;<\!\!-\!\!-\; x : nat,\; \alpha : types$$
$$y : list(x, \alpha) \;<\!\!-\!\!-\; y : list_{rec}(x, \alpha),\; x : nat,\; \alpha : types$$
$$[\,] : list_{rec}(0, \alpha) \;<\!\!-\!\!-$$
$$[y \mid z] : list_{rec}(suc(x), \alpha) \;<\!\!-\!\!-\; y : \alpha,\; z : list_{rec}(x, \alpha) \qquad \triangleleft$$

For each object variable x (and type variable α, respectively) occuring in some type clause, there is a type constraint $t:s$ in the body of this clause such that $x \in var(t)$ and $s \in \mathbb{V}_t \cup \mathcal{T}$ (and $\alpha \equiv t$, $s \in \mathbb{F}_s$, respectively). Moreover, there is not any type constraint $x:b$ of some clause such that $x \in \mathbb{V}_0$ and $b \in \mathbb{F}_s$. Thus different notations (and sorts) of both kinds of variables are not necessary. Whether a variable occuring in some type clause is an object variable or a type variable can be deduced from syntactical information on this type rule. It is easy to define a binary relation $=$ by means of definite clauses such that, for the least Herbrand model $\mathcal{H}^=$ of this set of clauses, $t = u$ is valid in $\mathcal{H}^=$ if, and only if $t, u \in \mathcal{O} \cup \mathcal{T} \cup \mathbb{F}_s$ and u is identical to t. Consequently, we can consider one-sorted logic. However, substitutions are defined in a restricted way and take into account the deduced sorts of variables. A *substitution* σ is defined by a mapping from $\mathbb{V}$ to $Tm(\mathbb{V}, \mathbb{F})$ such that the set $\{x \in \mathbb{V} \mid \sigma(x) \neq x\}$ is finite, $x \in \mathbb{V}_o$ implies $\sigma(x) \in \mathcal{O}$, and $\alpha \in \mathbb{V}_t$ implies $\sigma(\alpha) \in \mathcal{T}$.

The semantics of types and type sorts are determined by the least Herbrand model $\mathcal{H}^t$ of the definite clauses generated from the definitions of these types and type sorts. The set $\{ t \in \mathcal{O}_0 \mid t:s \in \mathcal{H}^t \}$ can be regarded as the interpretation of some type $s \in \mathcal{T}_0$. Analogously, the set $\{ s \in \mathcal{T}_0 \mid s:a \in \mathcal{H}^t \}$ can be regarded as the interpretation of a type sort $a \in \mathbb{F}_s$.

The union of the set of type clauses generated from specifications of type sorts and types, and the set of clauses for defining the relation $=$ is called constraint theory. The least Herbrand model of the constraint theory is the union of the Herbrand models $\mathcal{H}^t$ and $\mathcal{H}^=$.

5 Logic Programming with Type Constraints

A logic program with type constraints consists of definite clauses with type declarations for each predicate and of specifications of type sorts and types. The definite clauses and the corresponding type declarations for the predicates can be transformed into clauses with type constraints each being of the form $H \leftarrow \mathcal{C} \diamond B_1, \ldots, B_n$, where H (called head) and $B_1, \ldots, B_n$ (called body) are atomic formulas, and $\mathcal{C}$ is a set of type constraints and equations. The type constraints belonging to $\mathcal{C}$ are generated from the type declarations of the predicates occuring in the body, where we assume that a predicate can only be called, if the arguments satisfy the types given in the type declaration of this predicate. A goal is a clause without head. Thus we can assume that a set of clauses with type constraints and a constraint theory consisting of type clauses and equations are given. Using type inference the number of type constraints can be reduced (in many cases drastically).

Example 7 We regard a part of a program. Let s_1, s_2, s_3, s_4 be defined types and $p/2, q_1/1, q_2/1$ be predicates with the type declarations **pred** $p:\ s_1 \times s_2$, **pred** $q_1 :\ s_3$, **pred** $q_2 :\ s_4$.

Furthermore, let $p(x, y) \longleftarrow q_1(x), q_2(y)$ be a clause of the definition of $p/2$. This definite clause can be transformed into the following clause with type constraints:

$$p(x, y) \longleftarrow x:s_3, y:s_4 \diamond q_1(x), q_2(y)$$

If the interpretation of s_1 is a subset of the interpretation of s_3, then this clause can be simplified to:

$$p(x, y) \longleftarrow y:s_4 \diamond q_1(x), q_2(y)$$

The goal $\longleftarrow p(t, u)$ is transformed into a goal with type constraints: $\longleftarrow x:s_1, y:s_2 \diamond p(x, y)$ $\lhd$

Let $\mathcal{H}$ be the least Herbrand model of the given constraint theory. A derivation step takes a goal

$$\leftarrow \ \mathcal{C}_A \diamond A_1, \ldots, A_i, \ldots, A_n \quad ,$$

chooses some A_i of the body of this goal, takes a variant of a clause from the program

$$H \ \leftarrow \ \mathcal{C}_B \diamond B_1, \ldots, B_m \quad ,$$

and returns a new goal

$$\leftarrow \ \mathcal{C}_A \cup \mathcal{C}_B \cup \{H = A_i\} \diamond A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n \, ,$$

if the constraints $\mathcal{C}_A \cup \mathcal{C}_B \cup \{H = A_i\}$ are satisfiable in $\mathcal{H}$.

A derivation sequence is a possibly infinite sequence of goals connected by derivation steps. A derivation sequence is called succesful if it is finite and its last goal contains only constraints.

This rule of inference is a special case of the resolution with restricted quantifiers described in [3]. Thus soundness and completeness of our system follow from the results presented in this book.

The strategy of deduction can be modified for reasons of efficiency. In every derivation step the constraints $\mathcal{C}$ can be replaced by simplified constraints $\mathcal{C}_s$, if $\mathcal{H} \models \mathcal{C} \leftrightarrow \mathcal{C}_s$. The complete test for satisfiability of the constraints can be replaced by a checkup of necessary conditions of satisfiability, if the satisfiability of the relevant constraints is proved in the last derivation step and the satisfiability of the other sets of constraints (belonging to the derivation sequence) follows from this proof.

Since a logic program with type constraints can be transformed into definite clauses, the least Herbrand model can be used for the semantics. A ground type term s can be interpreted as the set of ground object terms t such that $t\!:\!s$ belongs to the least Herbrand model. Type sorts can be interpreted analogously.

6 Computing with Type Constraints

For constraint processing we need rules for the test of satisfiability and rules for the simplification of a set of constraints. Since the constraint theory is a set of definite clauses, computing with constraints can be based on SLD-resolution. Because of the assumptions with respect to the type rules, special computation rules can be applied for the SLD-resolution. The subset of equations contained in a set of constraints can be deleted, if there is a most general unifier of these equations (in the other case the constraints cannot be satisfied) and this unifier is applied to the set of constraints. Therefore, we consider only sets of type constraints in this section. Futhermore, we assume that the type terms occuring in a set of type constraints are ground terms. This means that the type parameters of a logic program with type constraints are assumed to be input variables (or that these variables can be bound by abstract interpretation). If a derivation step is used with respect to a type constraint which does not contain any type term with variables, then the new type constraints cannot contain type terms with variables either.

In the following let $\mathcal{R}$ be a finite set of type clauses. The following theorem is devoted to an important property of type checking (i.e. checking if a type constraint without any variable holds).

Theorem 1 *If $t\!:\!s$ is a type constraint with $var(t\!:\!s) = \emptyset$, then it is decidable whether $t\!:\!s$ follows from $\mathcal{R}$ or not.*

Proof (*Sketch*). Let $\mathcal{N}_f$ be the set of ordered tuples of natural numbers, where the first element of a tuple is the greatest number, i.e. $n_i \geq n_j$ for each tuple $(n_1, \ldots, n_k)$ of $\mathcal{N}_f$, if $i \leq j$. An ordering between the elements of $\mathcal{N}_f$ is defined by: $(n_1, \ldots, n_{k_n}) > (m_1, \ldots, m_{k_m})$ iff one of the following conditions is satisfied

- there is an i with $1 \leq i \leq k_n$ and $i \leq k_m$ such that $n_i > m_i$ and $n_j = m_j$ for all j with $1 \leq j < i$ **or**
- $k_n > k_m$ and $n_j = m_j$ for all j with $1 \leq j \leq k_m$.

This binary relation is a linear ordering in $\mathcal{N}_f$ and is well-founded in $\mathcal{N}_f$, i.e. there is no infinite sequence $\{\vec{n}_i\}$ of elements of $\mathcal{N}_f$ such that $\vec{n}_i > \vec{n}_{i+1}$ for all $i \geq 0$.

An ordering for a set of pairs $\{(\vec{n}, \vec{m}) \mid \vec{n}, \vec{m} \in \mathcal{N}_f\}$ is defined by: $(\vec{n}_1, \vec{n}_2) > (\vec{m}_1, \vec{m}_2)$ iff $\vec{n}_1 > \vec{m}_1$ **or** $\vec{n}_1 = \vec{m}_1$ and $\vec{n}_2 > \vec{m}_2$.

Each type constraint is mapped by a function Ψ to a pair of tuples of natural numbers. This function Ψ and the functions $\Phi_t : \mathcal{T}_0 \to \mathcal{N}_f$, $\Phi_c : \mathcal{O}_0 \to \mathcal{N}_f$ are defined by (function η defined in Section 3 is used):

- For each $s \in \mathcal{T}_0$, $\Phi_t(s)$ is an ordered tuple $(n_1, \ldots, n_k) \in \mathcal{N}_f$ such that there is a one-to-one mapping beetween the symbols of type functions occuring in s and the natural numbers of the tuple, where each $h \in \mathbb{F}_t$ occuring in s is mapped to a natural number n_i with $n_i = \eta(h)$.

– For each $t \in \mathcal{O}_0$, $\Phi_c(t)$ is an ordered tuple $(m_1, \ldots, m_k) \in \mathcal{N}_f$ such that there is a one-to-one mapping beetween the object constructors occuring in t and the natural numbers of the tuple, where each $g \in \mathbb{F}_o$ occuring in t is mapped to a natural number m_i with $m_i = \eta(g)$.

– For each $t \in \mathcal{O}_0$ and $s \in \mathcal{T}_0$, $\Psi(t:s) = (\, \Phi_t(s)\,,\, \Phi_c(t)\,)$.

– For each $a \in \mathbb{F}_s$ and $s \in \mathcal{T}_0$, $\Psi(s:a) = (\, \Phi_t(s)\,,\, (\,\eta(a)\,)\,)$.

Then Theorem 1 follows from the proposition: *If* $t:s \longleftarrow u_1:r_1, \ldots, u_k:r_k$ *is a ground instance of a type clause of* $\mathcal{R}^t$, *then holds:* $\Psi(t:s) > \Psi(u_i:r_i)$ *for all* i *with* $1 \leq i \leq k$.

A ground instance of a type clause of $\mathcal{R}^t$ is generated from this type clause and a substition which maps the variables of the clause to ground terms. ◁

Thus a type constraint without any variable can be reduced to "`true`" or "`false`" and it is sufficient to consider type constraints $t:s$ with $s \in \mathcal{T}_0$ and $var(t) \neq \emptyset$. Interesting cases are: "*Is there a computed answer for* $t:s$" and "*Is there a computed answer for* $x:s_1 \wedge x:s_2$ *?*" In general these problems are not decidable. However, if there is not any recursive type clause in $\mathcal{R}$, then these problems are decidable.

If the type clauses satisfy certain conditions, some interesting properties can be deduced. For this, results on terminating logic programs can be also used (see e.g. [15]). For instance, if $x:s$ is a type constraint with $var(s) = \emptyset$ and the type clauses which are needed for the construction of the SLD-tree of $x:s$ are nonrecursive, then this SLD-tree is finite and the elements belonging to s can be generated by a finite number of derivation steps. In the following theorem, sufficient conditions of recursive type clauses are described in order to guarantee that the mentioned problems "$t:s$?" and "$x:s_1 \wedge x:s_2$?" are decidable.

Theorem 2 *Supposed that the following conditions are satisfied for each type function* $h \in \mathbb{F}_t$ *having a recursive type clause in* $\mathcal{R}$*:*

- *If* $f(\tilde{t_1}):h(\tilde{u}) \longleftarrow \mathcal{C}_h$ *is a recursive type clause of* $\mathcal{R}$, *then there is exactly one recursive type constraint* $x:h(\tilde{v}) \in \mathcal{C}_h$.
- *One of the following conditions holds for all recursive type clauses* $f(\tilde{t_1}):h(\tilde{u}) \longleftarrow \mathcal{C}_h$ *of the type constructor* h*:*

  (a) if $x:h(\tilde{v}) \in \mathcal{C}_h$, *then there is an* i *such that* $v_i \in \tilde{v}$ *is a proper subterm of* $u_i \in \tilde{u}$*;*

  (b) if $x:h(\tilde{v}) \in \mathcal{C}_h$, *then* $h(\tilde{u}) \equiv h(\tilde{v})$.

Then the satisfiability of a finite set of type constraints[1] *is decidable.*

Proof. Let $\mathcal{C}$ be a set of type constraints. Because of Theorem 1 we can assume that, for each $t:s \in \mathcal{C}$, it holds: $s \in \mathcal{T}_0$, $var(t) \neq \emptyset$, $t \in \mathcal{O}$. Furthermore, we can assume that the constraints intoduced by applying an inference rule have the same property. Let ϕ be the function defined by $\phi(t:h(\tilde{s})) = \eta(h)$ for each type constraint. Supposed that an inference ruel is applied with respect to some $t:h(\tilde{s}) \in \mathcal{C}$, $\mathcal{C}_{new}$ is the set of type constraints introduced by this derivation step, and that $\phi(t:h(\tilde{s})) = k$. Then $k \geq \phi(u:r)$ for each $u:r \in \mathcal{C}_{new}$ (see definition of η in Section 3) and there is at most one $u:r \in \mathcal{C}_{new}$ with $\phi(u:r) = k$.

If condition (a) is satisfied for h, then there is at most a finite number of derivation steps with respect to $\mathcal{C}_{new}$ such that a type constraint $u:r$ with $\phi(u:r) = k$ is introduced.

Now we suppose that condition (b) is satisfied for h. The computation rule can be modified such that an inference rule must not be applied with respect to $t:h(\tilde{s})$ and to a recursive type rule, if $t \in \mathbb{V}_t$ (otherwise, $\mathcal{C}_{new}$ would contain a variable constraint $x:h(\tilde{s})$ and the derivation step is redundant, see also [15]). Thus there is at most a finite number of derivation steps with respect to $\mathcal{C}_{new}$ such that a type constraint $u:r$ with $\phi(u:r) = k$ is introduced (in this case, $r = h(\tilde{s})$ and u is a proper subterm of t).

Therefore, the number of derivation steps which can be applied is restricted and the proposition follows easily from this. ◁

[1] Note that we suppose that the type terms occuring in this set do not contain any variables

In support of computing with type constraints a binary relation $\rightarrowtail \subseteq \mathbb{F}_t \times \mathbb{F}_o \cup \mathbb{F}_t$ is defined as follows: $h \rightarrowtail g$ holds if and only if $h \in \mathbb{F}_t$, $g \in \mathbb{F}_o \cup \mathbb{F}_t$, and one of the following conditions is satisfied

- $\mathcal{R}$ contains a type clause $g(\tilde{u}) : h(\tilde{t}) \ \longleftarrow \mathcal{C}$;
- $\mathcal{R}$ contains a type clause $x : h(\tilde{t}) \longleftarrow \mathcal{C}$ with $x : g(\tilde{u}) \in \mathcal{C}$.

The relation $\rightarrowtail$ defines a dependency graph (see also [15]). This relation can also been considered as a rewrite system $\mathcal{R}^{\mathbb{F}}$ for function symbols (see [4]). Let $\rightarrowtail^+$ denote the transitive closure of $\rightarrowtail$.

Lemma 1 *Let $h, h_1, h_3 \in \mathbb{F}_t$ be type constructors and let $f \in \mathbb{F}_o$ be a object constructor.*

- *If $h \rightarrowtail^+ f$ does not hold, then $\exists f(\tilde{t}) : h(\tilde{u})$ cannot follow from $\mathcal{R}$.*
- *If there is not any $g \in \mathbb{F}_o$ such that $h_1 \rightarrowtail^+ g$ and $h_2 \rightarrowtail^+ g$, then $\exists (x : h_1(\tilde{u}_1) \wedge x : h_2(\tilde{u}_2))$ cannot follow from $\mathcal{R}$.*

Necessary conditions for satisfiability of type constraints follow from this lemma. Furthermore, the number of type clauses which are applicable to a type constraint can be reduced, if this lemma is used. The relation $\rightarrowtail^+$ depends only on the specifications of the types and type sorts. Therefore, this relation can be generated at compile time and, for each $h \in \mathbb{F}_t$, we can determine the set $\{ f \in \mathbb{F}_o \mid h \rightarrowtail^+ \}$. Note that a corresponding relation can be defined between type sorts and type constructors.

For better control of type constraint processing the following conditions are very useful:

(1) Each type constructor belongs to a basic type sort (elements of $\mathbb{F}_s^b$) and the interpretations of type terms $h_1(\tilde{u}_1)$ and $h_2(\tilde{u}_2)$ are disjoint, if h_1 and h_2 belong to different basic type sorts.

(2) Two different types are disjoint, if these types are defined by means of set definitions.

If condition (1) is satisfied and the type constructors h_1, h_2 belong to different basic type sorts, then $x : h_1(\tilde{u}_1) \wedge x : h_2(\tilde{u}_2)$ cannot be satisfiable. Furthermore, $x : s_1 \wedge x : s_2$ cannot be satisfiable, if conditions (2) holds and s_1, s_2 are different types defined by means of set definitions. Thus, these conditions imply necessary conditions of satisfiability which can be easily checked.

Our strategy for type constraint processing is based on the following steps: firstly, necessary conditions and type constraint without variables are checked; secondly, derivation steps are carried out, if there is only one nonrecursive type clause which can be applied with respect to a type constraint (in these cases backtracking is not necessary); furthermore, other type clauses are applied, where the application of recursive type clauses are restricted (i.e. a complete test of satisfiability can be delayed).

7 Concluding Remarks

Type rules can also be regarded as special rewrite rules (see e.g. [7]). The presented approach can be extended in several directions. The conditional parts of type rules may be extended by constraints which are not type constraints, e.g. $s :=> x \longleftarrow p(x)$ (where $p/1$ is a predicate defined by definite clauses). Such constraints should be decidable and constructive, and the types of the type declaration of such a constraint should not depend on the type which is defined by means of this constraint. The presented type concept can be integrated into constraint logic programming. In such a case it is useful if the conditional part of a type rule can contain constraints of the corresponding domain, e.g. $interval(x, y) :=> z \longleftarrow z : nat, x < z, z < y$ (the computation with types has to be extended). Another interesting extension is the use of predicates connected to an external database in the conditional part of type rules. For instance, the type "*student*" may be defined by $student :=> x \longleftarrow p\_student(x)$, where we assume that the predicate $p\_student/1$ is connected to an external database and that this predicate can generate the elements belonging to the corresponding relation of the database or can check if an element belongs to this relation. Besides of the mentioned extensions, our future research will include the development of improved methods for computing with type constraints. Note that constraint handling systems (see [6, 5]) are a useful tool for a prototyp implementation of the presented approach.

References

[1] H. Ait-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *J. Logic Programming*, 3:185–215, 1986.

[2] H. Ait-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. In *Proc. Int. Conf. Fifth Generation Computer Systems 1992*, pages 1012–1021, Tokyo, 1992. ICOT, IOS Press.

[3] H.-J. Bürckert. *A Resolution Principle for a Logic with Restricted Quantifiers*, volume 568 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, 1991.

[4] N. Dershowitz and G. Sivakumar. Solving goals in equational languages. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 45–55. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1988.

[5] ECRC, München. *ECL^iPS^e 3.4, Extensions User Manual*, 1994.

[6] T. Frühwirth. Constraint simplification rules. Technical report ECRC-92-18, ECRC, München, 1992.

[7] H.-J. Goltz. A constructive type system based on data terms. In D. Pearce and G. Wagner, editors, *Logics in AI - JELIA'92*, volume 633 of *Lecture Notes in Artificial Intelligence*, pages 279–303. Springer-Verlag, Berlin, Heidelberg, 1992.

[8] M. Hanus. Logic programming with type specifications. In *[14]*, pages 91–140, 1992.

[9] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th Principles of Programming Languages*, pages 111–119, Munich, 1987.

[10] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Logic Programming*, 1994. to appear.

[11] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, Heidelberg, 2. edition, 1987.

[12] M. Mamede and L. Monteiro. A constraint logic programming scheme for taxonomic reasoning. In K. Apt, editor, *Logic Programming, Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 255–269, Cambridge, London, 1992. MIT Press.

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

[14] F. Pfenning, editor. *Types in Logic Programming*. Logic Programming Series. The MIT Press, Cambridge (Mass.), London, 1992.

[15] L. Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, 1990.

[16] G. Smolka, W. Nutt, J.A. Goguen, and J. Meseguer. Order-sorted equational computation. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, Rewriting Techniques, pages 297–367. Academic Press, 1989.

[17] T.E. Uribe. Sorted unification using set constraints. In D. Kapur, editor, *Automated Deduction – CADE-11*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 163–177. Springer-Verlag, Berlin, Heidelberg, 1992.