Directional types for logic programs and the annotation method

Johan Boye, Jan Maluszynski

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Directional types for logic programs and the annotation method

Johan Boye , Jan Małuszyński

N ° 2471

Janvier 1995

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

ISSN 0249-6399

Directional types for logic programs and the annotation method

Johan Boye* , Jan Małuszyński**

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Chloe

Rapport de recherche n°2471 — Janvier 1995 — 44 pages

Abstract: A *directional type* for a Prolog program expresses certain properties of the operational semantics of the program.

This paper shows that the annotation proof method, proposed by Deransart for proving declarative properties of logic programs, is also applicable for proving correctness of directional types. In particular, the sufficient correctness criterion of *well-typedness* by Bronsard et al, turns out to be a specialization of the annotation method. The comparison shows a general mechanism for construction of similar specializations, which is applied to derive yet another concept of well-typedness. The usefulness of the new correctness criterion is shown on examples of Prolog programs, where the traditional notion of well-typedness is not applicable.

We further show that the new well-typing condition can be applied to different execution models. This is illustrated by an example of an execution model where unification is controlled by directional types, and where our new well-typing condition is applied to show the absence of deadlock.

Key-words: logic programming, directional types, program correctness

(Résumé : tsvp)

A short version of this report will appear in the Proceedings of the International Conference on Logic Programming, Tokyo, June 1995

*Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, johbo@ida.liu.se

**Jan.Maluszynski@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Types flèchés pour programmes logiques et méthode de preuve par annotation

Résumé : *Types flèchés* pour un programme Prolog expriment certaines propriétés operationelles de celui-ci.

Cet article montre que la méthode de preuve par annotation, proposée par Deransart pour prouver des propriétés déclaratives de programmes logiques, est également appliquable pour démontrer la correction des types flèchés. En particulier, le critère de *bon typage* proposé par Bronsard *et al*, se trouve être un cas particulier de la méthode de preuve par annotation. La comparaison révèle un mécanisme général pour concevoir des spécialisations similaires. Celui-ci est appliqué pour définir un nouveau critère de bon typage. L'utilité de ce nouveau critère est démontrée sur des exemples de programmes Prolog, auxquels l'ancienne notion de bon typage n'est pas appliquable.

Nous montrons aussi que ce nouveau critère peut être appliqué à differents modèles d'exécution. Ceci est illustré par un modèle d'exécution où la résolution d'équations est contrôlée par des types flèchés, et où notre nouveau critère de bon typage est utilisé pour démontrer l'absence de "deadlock".

Mots-clé : programmation en logique, types flèchés, correction de programmes

1 Introduction

Recently there has been a growing interest in the notion of *directional types* for Prolog programs [12, 35, 4, 6, 1, 13, 33, 38]. This kind of prescriptive typing describes the intended ways of calling the program, as well as the user's intuition of how the program behaves when called as prescribed. Together with some methods and tools for type checking, directional types may provide a good support for program validation.

This paper shows that directional types has two aspects. One of them is declarative and can be discussed regardless of the computation rule as long as the operational semantics is sound, while the other is related to the computation rule. This view allows us to put the concepts of *well-typing* appearing in the literature in a broader perspective, and relate it to the annotation method for proving declarative properties of logic programs proposed by Deransart (see e.g. [22]). This facilitates systematic development of similar sufficient conditions for correctness of directional types, also for other computation rules. In particular, we introduce yet another well-typing condition, and illustrate its power on examples which cannot be handled by the well-typing of [12, 4]. We show that it also gives a sufficient condition of non-suspension for a model of computation when the declared types are used for control.

The idea of directional types is to describe the computational behaviour of Prolog programs by associating an *input* and an *output* assertion to every predicate. The input assertion puts a restriction on the form of the arguments of the predicate in the initial atomic goals. The output assertion describes the form of the arguments at success, given that the predicate is called as specified by its input assertion. Thus a directional type is a specification of the input-output behaviour of a program executed under the Prolog computation rule.

A directional type can also be used as a description of the way the program predicates are called during the computation. In that case, assuming that the initial query satisfies its input assertion, it is expected that every intermediate call in the computation will also satisfy its input assertion. In that case the directional type can also be seen as a resolution invariant.

As an example, consider the `append/3` predicate:

```
append([], X, X).
append([E|L], R, [E|LR]) :- append(L, R, LR).
```

When this predicate is used to concatenate two lists, we call it with the two first arguments bound to the two lists. Upon success the third argument is bound to the resulting list. The form of the third argument at call is not restricted. Also we are not concerned about the form of the first two arguments at success. This use of the predicate may be described by the following notation:

`append/3`: ($\downarrow List$, $\downarrow List$, $\uparrow List$)

where the two first argument positions (marked with $\downarrow$) are considered as *input* positions, and the third argument (marked with $\uparrow$) is considered an *output* position. A more general concept

of directional type (e.g. in [12]) requires specification of input and output assertion for every argument of the predicate. Thus a directional type specification for an n-ary predicate p would consist of two vectors of type expressions, each of them of length n. Clearly, the restricted concept of directional type is a special case of the more general one; the example shown above can be reformulated as follows:

`append/3`: $\downarrow(List,\ List,\ Any)\ \uparrow(List,\ List,\ List)$

A given directional type may or may not be correct in the sense that it properly describes the actual computational behaviour. The directional type in the example above is correct; if `append/3` is called with the two first arguments bound to lists, then the third argument will be bound to a list upon success. Moreover, in every intermediate call, the two first arguments are bound to lists.

As pointed out above, the correctness of directional types has two aspects:

- the *input-output* correctness: whenever the call of a predicate satisfies the input assertion then the call instantiated by any computed answer substitution satisfies the output assertion.
- the *call* correctness: whenever the call of a predicate satisfies the input assertion then also any other call in this computation will satisfy the input assertion.

A directional type intended to cover both aspects can be seen as a special case of assertion in the sense of [25], where a sound (and complete [24]) method for proving correctness of such assertions has been presented. However, the method is quite complicated in general case. The focus of recent research on directional types has been on defining sufficient conditions for correctness. For example, the *well-typing* condition of [4, 12] is sufficient to ensure both input output correctness and call correctness of a given directional type and it is used as a basis for automatic type checking in [1, 38]. However, it is not applicable to directional types which are input-output correct but not call correct. This kind of directional types may be particularly interesting for programs using the power of the logical variable, as illustrated in the examples of Section 3 and Section 7.

In this paper we discuss the two aspects of directional types separately. Notice that input-output correctness is a property which does not depend on the computation rule R, since the SLD-resolution computes the same answers, regardless of R. This observation allows us to abstract from the computation rule and to study the problem in the framework of declarative semantics. In particular, we show that for the types closed under substitution input-output correctness of a program can be proved by the *annotation method* originating from [20, 19] and discussed more recently in [22]. It turns out that the well-typedness condition of [12, 4] can be seen as a specialization of the annotation method, even though it has been derived for Prolog computation rule. With this perspective we obtain immediately another specialization of the annotation method, which allows us to prove input-output correctness of directional types which are not call-correct under Prolog computation rule. This sufficient condition for input-output correctness is called *sharing-based well-typedness*, or briefly *S-well-typedness*.

For studying call correctness of a program we introduce a concept of *well-typed position of a clause*. It allows us to identify a subset of the positions of the predicates for which a given directional type is a resolution invariant.

We also discuss directional types as a means for controlling execution of logic programs through a delay mechanism. The idea is to postpone unification of those arguments of the goal which do not satisfy the prescribed type. The principle is reminiscent of that used in existing Prolog systems, like e.g. [14], where an atomic subgoal can be delayed if some of its arguments are not sufficiently instantiated. However, our delays concern unification of single arguments, while Prolog suspends resolution steps. We define formally an execution mechanism, called *type-driven resolution* (or simply *T-resolution*), based on this idea. A mechanism similar to T-resolution has been used in our previous work to achieve declarative integration of logic programs with external procedures [9, 30].

The programs executed under T-resolution may deadlock, in the sense that no further step of computation is possible, even though the set of delayed unifications is not empty. We show that if for the directional type used for controlling the execution the program is S-well-typed, then every computation starting with a correctly typed goal is deadlock-free.

The paper is organized as follows.

Section 2 summarizes the basic notions relevant for the presentation of the results. In particular the notion of well-typing is presented following [12, 4]. The concepts of input-output correctness and call correctness of a gived directional type are formally defined.

Section 3 discusses an example of a program with a directional type which is input-output correct but not call correct for LD-resolution. Thus the program is not well-typed under this directional type. This motivates an attempt to search for a better sufficient condition for checking input-output correctness.

Section 4 outlines the annotation proof method following [22] and shows that the input-output correctness of a directional type is equivalent to the correctness of an annotation corresponding to that type. It shows also that the well-typedness condition of [4, 12] can be seen as a specialization of the annotation method. It introduces the concept of S-well-typedness as another specialization of the method and illustrates its use on the example of Section 3.

Section 5 discusses the problem of call correctness of a given directional type for a given computation rule. The question considered is for which input arguments of the program predicates a given directional type is a resolution invariant for a given computation rule. For the Prolog computation rule a sufficient test for answering this question is provided.

Section 6 presents T-resolution. It is shown that no computation of a S-well-typed program executed by T-resolution will result in a non-empty set of delayed unifications. This can be seen as a kind of deadlock-freeness: the delayed unification will always be resolved, unless the computation loops or fails. This theorem contributes to static analysis of programs with delays, which seems to be an important topic.

Section 7 illustrates the usefulness of the concept of S-well-typed program by some examples.

Section 8 discusses relations to other work. Conclusions and future work are outlined in **Section 9**.

2 Preliminaries

2.1 Types

Adopting to a popular view (e.g. Apt [4]), we define a *type* to be a decidable set of terms closed under substitution. In particular, in the examples we will use the following types:

Any	the set of all terms	
Ground	the set of ground terms	
List	[] | [*Any* | *List*]	(lists)
GrList	[] | [*Ground* | *GrList*]	(ground lists)
BinTree	*void* | *tree*(*Any*, *BinTree*, *BinTree*)	(binary trees)
GrBinTree	*void* | *tree*(*Ground*, *GrBinTree*, *GrBinTree*)	(ground binary trees)

A *typed term* is an object $t : T$, where t is a term, and T is a type. A *directional type* for an n-ary predicate `p` is a pair of vectors of types of length n, denoted

$$\texttt{p}\colon \downarrow(T_1, ..., T_n)\ \uparrow(T'_1, ..., T'_n)$$

For each $i = 1, ..., n$, T_i will be called the *input type* of the i-th position, while T'_i will be called the *output type* of the i-th position. In most examples considered in this paper we use a restricted kind of directional types, where for è each argument only an input or an output type is specified, but not both. Formally, a *restricted directional type* for an n-ary predicate `p` is an n-tuple, associating every predicate position of `p` with a direction ($\downarrow$ or $\uparrow$) and with a type. The argument positions associated with $\downarrow$ ($\uparrow$) are called the *input* (the *output*) positions of `p`.

A restricted directional type is to be understood as a special case of directional type, where for every i either T'_i and T_i are identical types or T_i is the type *Any*. An atom $p(t_1, ..., t_n)$ is said to be *correctly typed in its i-th position* iff $t_i \in T'_i$.

For instance,

$$\texttt{append/3}\colon (\downarrow List,\ \downarrow List,\ \uparrow List)$$

is a directional type for the `append/3` predicate and the atom `append/3([],Y,[1,2]` is correctly typed in its first and third positions.

A directional type is a specification of the computational behaviour of the program under SLD-resolution with a fixed computation rule (typically under LD-resolution). This is captured by a notion of *correctly typed* program for a given computation rule.

Definition 2.1 Let R be a computation rule. If for every atom A which is correctly typed in its input positions:

- all atoms selected in every SLD-derivation of a given program P under R starting from A are correctly typed in their input positions, and if
- for every computed answer σ, $A\sigma$ is correctly typed in its output positions,

then the program P is *correctly typed* for R. Alternatively, we say that the directional type is *correct* for P and R.

If the first of the conditions is satisfied the directional type is said to be *call correct* for P and R. If the second condition is satisfied the directional type is said to be *input-output correct* (*IO correct*) for P. □

By the independence of the SLD-resolution of the computation rule we obtain at once that the IO correctness of a directional type for P does not depend on R, what justifies the definition.

2.2 Well-typing

Bronsard et al. [12] give a sufficient condition for a program to be correctly typed for the Prolog computation rule, namely that the program is *well-typed*. For our setting we follow its presentation by Apt [4]. Before explaining this concept, we introduce the notion of *type judgement*.

Let $s_1, \ldots, s_n, t$ be terms, and $S_1, \ldots, S_n, T$ be types. A *type judgement* has the form

$$s_1 : S_1 \ \wedge \ \ldots \ \wedge \ s_n : S_n \ \Rightarrow \ t : T$$

The judgement is true, written

$$\models s_1 : S_1 \ \wedge \ \ldots \ \wedge \ s_n : S_n \ \Rightarrow \ t : T$$

if, for all i from 1 to n, and for all substitutions σ, $\sigma(s_i) \in S_i$ implies that $\sigma(t) \in T$.

The problem of proving a type judgement true is undecidable in the general case. In all examples considered here however, it will be obvious whether type judgements are true or not.

To simplify the notation, we will throughout this section write an atom as $p(\mathbf{u} : \mathbf{U}, \mathbf{t} : \mathbf{T})$, where $\mathbf{u : U}$ is a sequence of typed terms filling in the input positions of p, and $\mathbf{t : T}$ is a sequence of terms filling in the output positions of p.

Definition 2.2

- A clause $p_0(\mathbf{i_0} : \mathbf{I_0}, \mathbf{o_0} : \mathbf{O_0}) :- p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$ is *well-typed* if, for all j from 1 to n:

$$\models \mathbf{i_0} : \mathbf{I_0} \ \wedge \ \mathbf{o_1} : \mathbf{O_1} \ \wedge \ \ldots \ \wedge \ \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \ \Rightarrow \ \mathbf{i_j} : \mathbf{I_j}$$

and if

$$\models \mathbf{i_0} : \mathbf{I_0} \ \wedge \ \mathbf{o_1} : \mathbf{O_1} \ \wedge \ \ldots \ \wedge \ \mathbf{o_n} : \mathbf{O_n} \ \Rightarrow \ \mathbf{o_0} : \mathbf{O_0}$$

- A program is well-typed if each of its clauses is well-typed.

□

Thus a clause is well-typed if

- the types of the terms filling in the *input* positions of a body atom can be deduced from the types of the terms filling in the *input* positions of the head and the *output* positions of the preceding body atoms, and if
- the types of the terms filling in the *output* positions of the head can be deduced from the types of the terms filling in the *input* positions of the head and the *output* positions of the body atoms.

To show that the first clause of `append/3` is well-typed, we have to prove that

$$([], X) : (List, List) \Rightarrow X : List$$

which is obviously true. To show that the second clause is well-typed, we have to prove that

$$([E|L], R) : (List, List) \Rightarrow (L, R) : (List, List)$$

and that

$$([E|L], R) : (List, List) \quad \wedge \quad LR : List \Rightarrow [E|LR] : List$$

Both of these type judgements are easily proven true; thus the `append/3` program is well-typed.

The following theorem is stated in [12]. A proof can be found in [6].

Theorem 2.3 Every well-typed program is correctly typed.

2.3 Proof trees

We now summarize a uniform framework for discussing both the operational and the declarative semantics of definite programs. This will allow us to discuss separately the IO correctness, and to relate the results to the problem of call correctness. This will also allow us to discuss SLD-resolution under various computation rules and execution with delays in a uniform way. The framework originates from Deransart and Małuszyński [22], and is based on the notion of proof tree, similar to that used by Clark [15].

In our view, the resolution process can be viewed as the stepwise construction of a *skeleton* (by "pasting" together instances of clauses), intertwined with equation solving (unification). We first give a preliminary definition:

Definition 2.4 For a given program P a *skeleton* is a tree defined as follows:

- if G is an (atomic) initial query, then the node labeled $(G, \bot)$ is a skeleton;

- if S_1 is a skeleton, then S_2 is a skeleton if S_2 can be obtained from S_1 by means of the following extension operation:
  1. choose a node n in S_1, labeled $(A, \bot)$;
  2. choose a clause $A_0 :- A_1, \ldots, A_k$ in P, such that A and A_0 have the same predicate symbol and the same arity;
  3. change n's label into $(A, \sigma(A_0))$, (where σ is a renaming to fresh variables), and add k children to n, labeled $(\sigma(A_1), \bot), \ldots, (\sigma(A_k), \bot)$.

A node is *incomplete* if its label contains $\bot$, and *complete* otherwise. A skeleton is incomplete if it contains an incomplete node, and complete otherwise. □

Definition 2.5 The *set of equations associated to the node* n is denoted by $E(n)$, and is defined as follows:

- if n is an incomplete node, then $E(n) = \emptyset$;
- if n is labeled with $(p(s_1, \ldots, s_k), p(t_1, \ldots, t_k))$, then $E(n) = \{s_1 = t_1, \ldots, s_k = t_k\}$.

□

For example, an LD-resolution step corresponds to choosing the leftmost node n (in preorder of the skeleton), expanding it as described in definition 2.4, and computing the solved form of $E(n)$ (which is equivalent to computing an idempotent unifier [29]). A *proof tree* is a complete skeleton, together with a solution of all the associated equations. The mgu of the equations is a canonical solution, representing all solutions of the equations.

Every successful LD-derivation corresponds to a proof tree and its computed answer substitution is determined by the mgu of the set of equations of the underlying skeleton.

One of the advantages of this view on operational semantics is that we can make fine-grained adjustments to the resolution process. For instance, for some node n, we may choose not to solve all equations in $E(n)$ at once (this corresponds to partly delaying unification). This is in fact exactly what we will do in the type of resolution introduced in Sect. 6.

Note that this scheme applies also to constraint logic programs, in which case solutions to the equations have to be computed by a constraint solver.

The declarative semantics of a program P can also be defined in terms of the proof trees. For every proof tree t denote by $r(t)$ the atom labeling the root of the skeleton and by $\sigma(t)$ a solution of the equations associated to the skeleton, i.e. any unifier of the equations. Then the set

$\mathcal{PT}_P = \{r(t)\sigma(t) | t \text{ is a proof tree of } P\}$

consists of all atomic logical consequences of P can be considered as a declarative semantics of the program.

Notice that this set may include non-ground atoms and that it is closed under substitution.

2.4 Dependencies

For the rest of this section, we assume that we have some unambiguous way of referring to the atoms in the program, and let A be the atom $p(t_1, \ldots, t_k)$ in some clause C. The argument positions in A are denoted by $A(1), \ldots, A(k)$.

Definition 2.6 The set of *clause positions in* C is defined as

$$\bigcup_{A \text{ is an atom in } C} \{A(i) \mid 1 \leq i \leq arity(A)\}$$

□

If no confusion can arise, we will refer to "clause positions" simply as "positions". We will not always make a distinction between clause positions and terms filling in clause positions, i.e. we may make statements like "$A(i)$ is a variable" instead of "the term filling in $A(i)$ is a variable".

Note that, in the type judgements concerning well-typedness of a clause (Definition 2.2) the terms occurring in the consequents originate from the output positions in the head, or from the input positions of the body. For convenience, we introduce a name for these positions:

Definition 2.7 $A(i)$ is an *exporting* clause position of C if either

- A is the head of C, and the i:th argument of p is an output position, or
- A is a body atom in C, and the i:th argument of p is an input position.

A clause position is *importing* if it is not exporting. We use the expressions $imp(C)$ and $exp(C)$ to denote the set of importing and exporting clause positions of C, respectively. □

Within a clause C, we think of data as flowing from the importing positions to the exporting positions. This is reflected by the following definition.

Definition 2.8 For each clause C, its *local dependency relation* $\rightsquigarrow_C$ is defined as follows:

$$A(i) \rightsquigarrow_C B(j)$$

iff $A(i) \in imp(C), B(j) \in exp(C)$, and $A(i)$ and $B(j)$ have at least one common variable. □

To model the dataflow in a complete skeleton T, we construct a *compound dependency graph* $\rightsquigarrow_T$ by "pasting together" the local dependency graphs for the clauses used in T. More precisely:

Definition 2.9 Let T be a complete skeleton, and let n be a node in T, labeled with $(p(s_1,\ldots,s_n), p(t_1,\ldots,t_n))$. Then n has k *node positions* (one for each equation $s_i = t_i$), denoted $n(1),\ldots n(k)$.

Let n_1 and n_2 be nodes in T, labeled (A_1, B_1) and (A_2, B_2) respectively. We define

$$n_1(i) \rightsquigarrow_T n_2(j)$$

if one of the following cases apply:

- n_1 is the father of n_2 (thus B_1 is the head of some clause C, and A_2 is a body atom in C), and $B_1(i) \rightsquigarrow_C A_2(j)$
- n_1 and n_2 are siblings (thus A_1 and A_2 are body atoms in the same clause C), and $A_1(i) \rightsquigarrow_C A_2(j)$
- n_1 is the son of n_2 (thus B_2 is the head of some clause C, and A_1 is a body atom in C), and $B_2(i) \rightsquigarrow_C A_1(j)$
- n_1 and n_2 is the same node, and $A_1(i) \rightsquigarrow_C A_1(j)$ (where C is the clause in which A_1 is a body atom), or $B_1(i) \rightsquigarrow_D B_1(j)$ (where D is the clause in which B_1 is the head).

□

Every node position is associated with an equation. The idea of the *type-driven* resolution introduced in Sect. 6 is that we solve these equations in accordance with the $\rightsquigarrow_T$ relation. Therefore it is absolutely essential that the $\rightsquigarrow_T$ relation is a partial ordering. This motivates us to introduce the following concept.

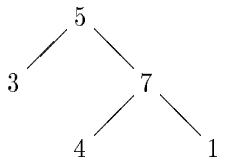
Definition 2.10 If the relation $\rightsquigarrow_T$ is a partial ordering for every skeleton T, then the program is said to be *non-circular*. □

The non-circularity concept stems originally from the field of attribute grammars. [28]. It is well-known that this property is decidable (see e.g. [21] or [22]).

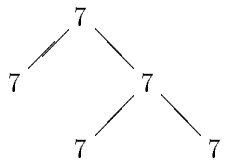
3 An informal example

In the literature on directional types, specification of the input-output behaviour of the program is often stated as the main objective (see e.g. [1]). However, as already pointed out, the well-typing condition is a sufficient condition both for IO correcnesst and call correctness. Thus, if a given program is IO correct but not call correct, the well-typing criterion is inapplicable to prove IO correctness. We claim that such typings often are of practical interest, especially for programs using incomplete data structures. We will now give an example of a correctly IO typed program which is not well typed. The reader may find more examples in Sect. 7.

Consider the following task: Given a binary tree T whose nodes are labeled with integers, compute a binary tree with the same structure as T, but where every node is labeled with the maximal integer in T. For example, given the tree



we would like the answer



Conceptually this is a two-pass problem; first traverse T to find the maximal integer n, and then construct the output tree where every node is labeled with n. However, the following program solves the problem in one pass.

```
maxtree(Tree, NewTree) :-
        maxtree(Tree, Max, [], Labels, NewTree),
        max(Labels, Max).

maxtree(void, _, L, L, void).
maxtree(tree(Lbl,Left,Right), Max, In, [Lbl|Out], tree(Max,NLeft,NRight)) :-
        maxtree(Left, Max, In, IO, NLeft),
        maxtree(Right, Max, IO, Out, NRight).
```

During the execution of the program `maxtree/5` traverses the input tree, collects all labels in a list, and builds a new tree where all nodes are labeled with the same logical variable. This variable is then unified with the maximal label, as computed by `max/2` The definition of `max/2` is straightforward and therefore omitted.

Note that upon success of `maxtree/5`, the fifth argument is bound to a non-ground binary tree. The only variable of the tree is being instantiated to an integer by `max/2`, so that upon success of `maxtree/2`, the second argument is bound to a ground binary tree. Thus, the most precise correct directional type of the program (using the types in Sect. 2.1) is as follows:

$$\begin{aligned}
&\texttt{maxtree/2} : (\downarrow \mathit{GrBinTree},\ \uparrow \mathit{GrBinTree})\\
&\texttt{maxtree/5} : (\downarrow \mathit{GrBinTree},\ \downarrow \mathit{Any},\ \downarrow \mathit{GrList},\ \uparrow \mathit{GrList},\ \uparrow \mathit{BinTree})\\
&\texttt{max/2} : (\downarrow \mathit{GrList},\ \uparrow \mathit{Ground})
\end{aligned}$$

However, the clause defining `maxtree/2` is not well-typed, since the type judgement

$$\begin{aligned}
\mathit{Tree} : \mathit{GrBinTree} \quad &\wedge\\
(\mathit{Labels}, \mathit{NewTree}) : (\mathit{GrList}, \mathit{BinTree}) \quad &\wedge\\
\mathit{Max} : \mathit{Ground} \quad &\Rightarrow\\
\mathit{NewTree} : \mathit{GrBinTree}&
\end{aligned}$$

is not true. The problem lies with the variable $\mathit{NewTree}$: one can not conclude that $\mathit{NewTree}$ is a ground binary tree just from the fact that it is a binary tree.

Since the program is not well-typed, theorem 2.3 is inapplicable. It is also inapplicable to the directional type:

$\texttt{maxtree/2} : (\downarrow GrBinTree,\ \uparrow GrBinTree)$
$\texttt{maxtree/5} : (\downarrow GrBinTree,\ \downarrow Ground,\ \downarrow GrList,\ \uparrow GrList,\ \uparrow GrBinTree)$
$\texttt{max/2} : (\downarrow GrList,\ \uparrow Ground)$

In this case the first clause is not well-typed, since the judgement $Tree : GrBinTree \Rightarrow Max : Ground$ is not true. This directional type is indeed not correct, since it is not call correct under LD-resolution: the `maxtree/5` predicate is called with the second argument being a variable, not a ground term. However, the directional type is IO correct, as will be shown by the method presented in Sect. 4.

4 Proving IO correctness

As already pointed out the problem whether a given directional type is IO correct or not is independent of a particular computation rule under which the program is to be executed. Thus the problem can be discussed in terms of proof trees of a program rather than in terms of computations. We now show that the method for proving properties of proof trees introduced in [19] and presented more recently in [22] can also be used for proving IO correctness of directional types. For making the relation explicit we introduce a new notation for directional types.

4.1 Annotations

A directional type `p/n`: $\downarrow(T_1, ..., T_n)\ \uparrow(T'_1, ..., T'_n)$ will be alternatively represented as a pair of formulae

$$\langle \psi_1(p_1) \wedge ... \wedge \psi_n(p_n),\ \psi'_1(p_1) \wedge ... \wedge \psi'_n(p_n) \rangle$$

interpreted on the universe of the terms of the program, where $p_1, ..., p_n$ are the only free variables of the formulae, referring to the argument positions of the predicate. For example the directional type

`append/3`: $(\uparrow List,\ \uparrow List,\ \downarrow List)$

will be represented as

$$\langle any(append_1) \wedge any(append_2) \wedge list(append_3),\ list(append_1) \wedge list(append_2) \wedge list(append_3) \rangle$$

Here *any* and *list* are unary predicates of the specification language, interpreted on the domain of terms. Thus, a directional type $T = \langle \psi, \psi' \rangle$ for a predicate p specifies two sets of atoms, $S_1(T)$ and $S_2(T)$, consisting of the elements of the form $p(t_1, t_2, t_3)$, which satisfy, respectively, ψ and ψ' in the interpretation considered. For example, for the directional type T above, $S_1(T)$ consists of all atoms of the form $append(t_1, t_2, t_3)$, where t_1 and t_2 are

arbitrary terms and t_3 is a list. The conjuncts of the formulae will be called the *assertions* of the directional type.

Recall the restrictions on directional types:

- every assertion is unary, i.e. it has exactly one free variable, and there is a one-one mapping between the assertions of each formula and the argument positions of the predicate.
- the interpretation of the specification language is such that the set of the terms specified by every assertion is a type, i.e. it is a decidable set of terms closed under substitution.
- the formulae observe a kind moding: for each i the i-th assertion of the first formula is either $any(p_i)$, or it is identical to the i-th assertion of the second formula.

Lifting these restrictions will give us a (syntactic) concept of predicate *annotation*.

Definition 4.1 Let L be a first order logical language with an interpretation $\mathcal{I}$. An *assertion* for an n-ary predicate p is a formula whose free variables are in the set $\{p_1, ..., p_n\}$. An *annotation* Δ for p is any pair $\langle Inh, Syn\rangle$, where Inh and Syn are finite sets of assertions, called respectively the *inherited* assertions and the *synthesised* assertions of Δ.

□

In the sequel we assume that L is interpreted on a term domain. Thus, as in the case of directional type, for a given interpretation $\mathcal{I}$ the annotation defines two sets of atoms denoted by $S1_\Delta$ and $S2_\Delta$. An annotation Δ is said to be *closed under substitution* iff for every term t in Si_Δ $(i = 1, 2,)$ and for every substitution σ, $t\sigma$ is in Si_Δ. For example, consider the following annotation of `append/3`:

$\langle\{list(append_3)\}, \{list(append_3), list(append_2), \forall x(elem(x, append_2) \rightarrow elem(x, append_3)\}\rangle$

Assume that the predicate *list* is interpreted in $\mathcal{I}$ as the set of lists, and the relation *elem* holds for x and y iff x is an element of the list y. Under this interpretation:

- $S1$ consists of all atoms of the form $append(t_1, t_2, t_3)$, where t_1, t_2 are arbitrary terms and t_3 is a list,
- $S2$ consists of all atoms of the form $append(t_1, t_2, t_3)$ such that t_1 is an arbitrary term, and t_2, t_3 are lists such that each element of t_2 appears also as an element of t_3.

Clearly, a directional type is a specific case of an annotation.

4.2 Program annotations

As discussed above, an annotation Δ of a predicate is a specification of two sets of atoms $S1_\Delta$ and $S2_\Delta$. We now discuss the use of annotations for specification of logic programs. The idea is to compare some semantics of a program P with the sets of atoms which are

specified by the annotation. In this context we will consider two kinds of the semantics: the input-output semantics and the declarative semantics $\mathcal{PT}_P$. The former will be used to extend for arbitrary annotation the notion of input-output correctness defined for directional types in Section 2.1. The latter is the semantics used in the annotation method.

Definition 4.2 The *input-output semantics* $\mathcal{IO}_P$ of P is a function which maps an arbitrary set of atoms I into the set of all atoms $g\sigma$ such that g is in I, and σ is a computed answer substitution for the goal $\leftarrow g$ under SLD-resolution.

An annotation Δ is input-output correct for P iff $\mathcal{IO}_P(S1_\Delta) \subseteq S2_\Delta$. □

Suppose *var* is a unary predicate of the metalanguage L, such that $var(t)$ is true in $\mathcal{I}$ whenever t is a variable. Then the annotation

$$\langle\{var(append_1), (var(append_2), list(append_3)\}, \{list(append_2), \forall x(elem(x, append_2) \rightarrow elem(x, append_3)\}\rangle$$

is input-output correct for the `append` program.

Notice that the notion of input-output correctness of an annotation which is a directional type, reduces to the notion of input-output correctness of the directional type, discussed in Section 2.1.

We now relate the annotations to the proof-theoretic semantics. The proof-theoretic semantics $\mathcal{PT}_P$ of a program P is the set of the root labels of all proof trees of P. Thus it corresponds to all atoms provable from P. An annotation may be used to state a property of the subset of $\mathcal{PT}_P$ obtained by the intersection with a given set of atoms. In that case the inherited assertions of the annotation specify the restriction, while the synthesised assertions state the property. This is captured by the following definition:

Definition 4.3 An annotation Δ is *success correct* for P iff $(\mathcal{PT}_P \cap S1_\Delta) \subseteq S2_\Delta$. □

For example the annotation

$$\langle\{list(append_3)\}, \{list(append_2), list(append_3), \forall x(elem(x, append_2) \rightarrow elem(x, append_3)\}\rangle$$

is success correct and describes an interesting property of the `append` program.

On the other hand, the annotation

$$\langle\{var(append_1), list(append_3)\}, \{nat(append_3)\}\rangle$$

is (trivially) success correct for the `append` program, whatever is the relation *nat*, since in no proof tree of the program the atom labeling the root has a variable as the first argument and a list as the third argument. Assume now that *nat* is the set of terms representing natural numbers: $\{zero, s(zero), ...\}$. In that case the example annotation is not input-output correct, since a call of `append` whose first argument is a variable and whose third argument is a list, may succeed but no instance of a list is in *nat*. Hence, in general, the input-output correctness is not implied by the success correctness of the annotation. However, for the annotations closed under substitutions both types of correctness coincide.

Theorem 4.4 Let Δ be an annotation closed under substitution. Δ is input-output correct for a program P iff it is success correct for P.

Proof : Assume Δ is not success correct. Then there exists an atom g in $S1_\Delta$ which is the root label of a proof tree and which does not belong to $S2_\Delta$. Then, by the definition of proof tree and by the completeness of the SLD-resolution, the empty substitution is a computed answer substitution for the goal $\leftarrow g$. Hence Δ is not input-output correct.

Assume Δ is not input-output correct. Then there exists an atom g in $S1_\Delta$ such that a substitution σ is a computed answer substitution for $\leftarrow g$, but $g\sigma$ is not in $S2_\Delta$. By the definition, $g\sigma$ is the root label of a proof tree. As $S1_\Delta$ is closed under substitution it includes $g\sigma$. Hence Δ is not success correct. □

Notice that the proof of the "if" case does not use the assumption that Δ is closed under substitution. However, for input-output correct Δ not closed under substitution, the success correctness may sometimes reduce to the trivial case where no root label of a proof is in $S1_\Delta$.

The theorem applies in particular to the annotations being directional types. Thus a method for proving success correctness of annotations can also be applied for proving correctness of directional types.

4.3 The Annotation Method

We now survey briefly the complete method for proving success correctness of annotations introduced in [20] for attribute grammars and adapted for the case of logic programs in [19]. The method is called *the annotation method*. Its more extensive presentation can be found in [22].

Let Δ be an annotation. To show that it is success correct one has to check for every proof tree that if its root label $p(t_1, ..., t_n)$ is in $S1_\Delta$ then it is also in $S2_\Delta$. This corresponds to checking validity of an implication of the form

$\psi_1 \wedge ... \wedge \psi_k \Rightarrow \psi'_1 \wedge ... \wedge \psi'_m$

where ψ_i's are the inherited assertions of p in Δ and ψ'_j's are the synthesised assertions of p in Δ, instantiated by the substitution $\{p_1/t_1, ..., p_n/t_n\}$

The idea of the annotation method is to consider the structure of the proof trees. Each of them is constructed from instances of the clauses of the program. The body atoms of a clause included in a proof tree give rise to root labels of the subtrees of the tree. As the number of program clauses is finite the idea is then to check that the above mentioned condition holds for the head atom of the clause, provided that it holds for every body atom. But also the condition for every of the body atoms is an implication constructed as discussed above. The interesting case is when the predecessor of the implication hold, since otherwise the implication holds trivially. Call the predecessor assertions of the head implication and the successor assertions of the body implications the *premiss* assertions of the clause. The remaining assertions of the implications will be called the *conclusion* assertions of the clause.

To achieve the local proof it suffices to show that every conclusion assertion follows from some premiss assertions of the clause.

For example consider the following annotation for the `append` program:

$\langle\{list(append_1), list(append_2)\}, \{list(append_3\}\rangle$

Then for the clause

```
append([A|X], Y, [A|Z]) :- append(X,Y,Z).
```

there will be two instances of the annotation:

$\langle\{list([A|X]), list(Y)\}, \{list([A|Z]\}\rangle$
$\langle\{list(X), list(Y)\}, \{list(Z)\}\rangle$

The premiss assertions are:

(a) $list([A|X])$, (b) $list(Y)$, (c) $list(Z)$

The remaining assertions are the conclusion assertions:

(1) $list([A|Z])$, (2) $list(X)$, (3) $list(Y)$

The verification of a clause consists in proving each of the conclusion assertions from (some of) the premiss assertions. More precisely, what is proved are universally quantified implications. For each of them the predecessor is a conjunction of some premiss assertions and the successor is a conclusion assertion.

In our example

(1) follows from (c), $\mathcal{I} \models \forall A, Z(list(Z) \Rightarrow list([A|Z])$

(2) follows from (a), $\mathcal{I} \models \forall A, X(list([A|X]) \Rightarrow list(X))$

(3) follows from (b). $\mathcal{I} \models \forall Y(list(Y) \Rightarrow list(Y))$

A proof done for the assertions of a clause holds also for the assertions of any clause instance, since the assertions are closed under substitution. Thus, for any proof tree of the program the correctness of annotation of its root should in principle follow from the proofs done for its constituent clauses. For example, consider a proof tree of the `append` program with the following skeleton:

```
append([A|X],Y,[A|Z])
|
append(  X,  Y,  Z  ),
append( [],  V,  V  )
```

and such that the arguments of its root label satisfy the inherited assertions of `append`. To show that its root label satisfies the synthesised assertion of the annotation it suffices to combine the local proofs for the constituent clauses.

(1) $\models list([A\|X])$	the inherited assertion of the root
(2) $\models list(Y)$	the inherited assertion of the root
(3) $X = []$	definition of the proof tree
(4) $Y = V$	definition of the proof tree
(5) $Z = V$	definition of the proof tree
(6) $\models list(V)$	by (2) and (4)
(7) $\models list(Z)$	by (5) and (6)
(8) $\models list(Z) \Rightarrow list([A\|Z])$	local proof for a constituent clause
(9) $\models list([A\|Z])$	by (7) and (8)

There is, however, a danger that combination of the local proofs may lead to a circular argumentation for some proof trees. For example, consider the program:

```
q(X) :- p(X,X).
p(X,X).
```

with the annotation:

$\langle\{ground(p_1)\}\{ground(q_1), ground(p_2)\}\rangle$

Intuitively this annotation says that if an atom $q(t)$ is the root label of a proof tree then t must be ground, which is not true. The conditions which are to be checked are as follows. For the first clause we get:

$\models \forall X(ground(X) \Rightarrow ground(X))$
This condition will be generated twice: for the only position of the head and for the first position of the body atom.

The same condition will be obtained for the second clause. The condition is trivially satisfied. Thus, the conclusion assertions of each of the clauses are implied by their premiss assertions. However, in the only proof tree of this program, the combination of the verification conditions of its clauses gives the statement $\forall X(ground(X) \Leftrightarrow ground(X))$, which does not allow to conclude that X is indeed ground. We now discuss the circularity phenomenon more abstractly.

Construction of a proof for a conclusion assertion of a clause uses some premiss assertions of this clause. We say that the conclusion assertion *depends* on these premiss assertions. Thus, in our first example (1) depends on (c), (2) depends on (a), and (3) depends on (b). Notice that different proofs may give rise to different dependencies. For example, if the set of premisses contains the assertions $list([A|X])$ and $list([B|X])$ then the conclusion $list(X)$ may be obtained by each of the premisses.

Generally we may consider an arbitrary relation between the premisses and the conclusions of a clause, which may or may not properly indicate the premisses sufficient for proving each conclusion. A *logical dependency scheme (LDS)* for a given program P and an annotation Δ is a family of such relations, indexed by the clauses of P. The relation for a particular clause can be represented by a graph spanned on the tree representing a clause.

The nodes of the graph are the assertions of the clause. Thus a tree node is associated with the nodes representing the assertions of its atom. The arcs of the dependency graph are determined by a given LDS. Any proof tree is obtained by pasting together instances of the clauses. Hence, by pasting together the copies of the dependency graphs of the clauses we obtain a dependency graph of a skeleton. The mechanism is identical to that described by the definition of the relation $\rightsquigarrow_T$, so that we skip a more formal presentation. An LDS is said to be *non-circular* iff the dependency graph of any skeleton has no loops. This property is decidable and has been studied in the context of attribute grammars (see e.g. [21]). The techniques proposed therein can be directly applied for checking non-circularity of a given LDS.

Intuitively, an LDS can be seen as a plan for proving success correctness of an annotation. For each clause the LDS shows the premiss assertions which are to be used for proving its conclusion assertions. The first question is whether the local proofs can be achieved according to this plan. The other one is whether for some skeleton the combination of the local proofs may give a circular reasoning, as illustrated in the example above.

For a given program an annotation is said to be *sound* iff there exists a non-circular LDS such that each conclusion assertion is implied by the premiss assertions on which it depends in the LDS.

The essence of the annotation method is captured by the following theorem, originating from [22] [1].

Theorem 4.5 An annotation $\Delta = (Inh, Syn)$ is success correct for a program P iff there exists an annotation $\Delta' = (Inh', Syn')$ sound for P and such that $S1_\Delta \subseteq S1_{\Delta'}$ and $S2_{\Delta'} \subseteq S2_\Delta$

The annotation Δ' plays the role of a generalization lemma, which may be needed to prove Δ. However, we are only interested in sufficient conditions for success correctness. Such conditions can be obtained by defining sound but incomplete specializations of the annotation method. The first step in that direction is to assume $\Delta' = \Delta$. Thus, the restriction focuses on the cases when the inductive proof can be achieved without any lemma. Further simplification consists in assigning to each program and annotation a particular LDS, which may or may not be sound. Thus, the application of theorem 4.5 reduces to checking whether for given program P and annotation Δ the associated LDS is sound or not. If yes, Δ is success correct for P, otherwise no information is provided by the check. Since success correctness is equivalent to IO correctness for the directional types (Theorem 4.4), this approach applies also to type checking.

In the particular case of directional types of [4] there is a one-one correspondence between the assertions of the annotation and the positions of the predicates. Thus, the premiss assertions of a clause correspond to the input positions of the head and to the output positions of the body, while the conclusion assertions correspond to the remaining positions. The well-typing condition of [4, 12] (see also Section 2.1) requires that for every clause the

[1] A slightly different phrasing is caused by the fact that [22] considers only one notion of correctness of the annotation, namely the success correctness

type of an output position in a body atom is implied by the type of the input positions of the head and by the types of the output positions of the preceding body atoms. It requires also that the type of an output position of the head is implied by the types of the input positions of the head and by the types of the output positions of all body atoms. Thus, for given program and directional type, it defines a priori a logical dependency scheme. This kind of dependency is an *L-dependency scheme*, according to the terminology used in attribute grammars (see e.g. [21]) and it is known to be non-circular. Thus, well-typing requires satisfaction of the verification conditions connected with a non-circular LDS determined by a given program and a given type annotation. It is hence a specialization of the annotation method.

By Theorem 4.4 we obtain at once that well-typed programs are IO correct. The additional result that they are also call correct does not follow automatically, since the theorem concerns only IO correctness. On the other hand, the theorem may allow for proving IO correctness of directional types which are not call correct. We now develop another specialization of the theorem, applicable to such directional types.

4.4 S-well-typed programs

We will derive yet another sufficient condition for IO correctness of a directional type, considered as an annotation. To do this we put a restriction on the use of the annotation method similar to that used for well-typing: we assume $\Delta' = \Delta$ and we define a priori an LDS for given program and directional type considered as annotation. This LDS is, however, different from that used by well-typing. We now present a rationale for defining the LDS.

For a given clause there is a one-one correspondence between the type assertions and the arguments of the predicates. Consider a conclusion assertion ψ in a clause corresponding to an argument position p of the clause. For construction of the LDS one may consider all premiss assertions of the clause. However for reducing the risk of circularity it is better to restrict a priori the logical dependencies. Therefore we propose to assume that ψ depends only on those premisses whose corresponding positions share variables with p. This suggestion is justified by the observation that for every valuation for which all these premiss assertions are satisfied, the logical values of the remaining premisses are irrelevant for the satisfaction of ψ. In other words, we propose to use the family $\rightsquigarrow_C$ of the local dependency relations, determined for a given program by a given directional type, (see Section2.4) as the LDS to be considered.

We formalize the proposed idea by the following notion of *sharing-based-well-typing* or *S-well-typing* where the imposed a priori logical dependency scheme is based on sharing of variables between positions of the clauses:

Definition 4.6
Let $\mathcal{T}$ be a directional type of a program P and let C be a clause of P. Denote by $\rightsquigarrow_C$ the dependency relation determined by $\mathcal{T}$ on the positions of C.

- For a given exporting position e, in C:
  - let t be the term occurring in C on e, and let T be the type associated to e by $\mathcal{T}$.
  - let $i_1, ..., i_k$ be all importing positions of C such that $i_j \rightsquigarrow_C e$, and let $t_1, ..., t_n$ be the terms on these positions of C typed, respectively, $T_1, ..., T_n$ by $\mathcal{T}$.

  The position e is *S-well-typed* iff

$$\models t_1 : T_1 \ \wedge \ \ldots \ \wedge \ t_k : T_k \ \Rightarrow \ t : T$$

- The clause C is *S-well-typed* iff all its exporting positions are S-well-typed.
- The program P is *S-well-typed* iff it is non-circular (Definition 2.10) and all its clauses are S-well-typed.

□

Thus, the annotation induced by an S-well-typing $\mathcal{T}$ is sound for P, since the LDS induced by $\rightsquigarrow_C$ of P is non-circular, and every conclusion assertion of a clause is implied by the premiss assertions on which it depends. Hence, by theorem 4.5 and by theorem 4.4 we obtain at once:

Theorem 4.7 Every S-well-typed program is correctly IO typed.

We now illustrate the definition for the `maxtree` program of Section 3 with the directional type:

$$\texttt{maxtree/2} : (\downarrow \mathit{GrBinTree}, \uparrow \mathit{GrBinTree})$$
$$\texttt{maxtree/5} : (\downarrow \mathit{GrBinTree}, \downarrow \mathit{Ground}, \downarrow \mathit{GrList}, \uparrow \mathit{GrList}, \uparrow \mathit{GrBinTree})$$
$$\texttt{max/2} : (\downarrow \mathit{GrList}, \uparrow \mathit{Ground})$$

As discussed in Section 3 the program is not well-typed with this directional type. We now show that it is S-well-typed, hence that it is IO correct.

According to the definition, the program is S-well-typed iff every clause satisfies the local verification conditions and the program is non-circular. For a given clause, every conclusion assertion gives rise to one verification condition. The conclusion assertions are associated with the exporting positions of the clause. To make things explicit we quote the program and we underline the exporting positions of each clause

```
maxtree(Tree, NewTree) :-
       maxtree(Tree, Max, [], Labels, NewTree),
       max(Labels, Max).

maxtree(void, _, L, L, void).
maxtree(tree(Lbl,Left,Right), Max, In, [Lbl|Out], tree(Max,NLeft,NRight)) :-
       maxtree(Left, Max, In, Out1, NLeft),
       maxtree(Right, Max, Out1, Out, NRight).
```

We now give explicitly the verification conditions for the clauses. The conditions will be stated for every exporting position in the order of its appearance in the clause. All conditions are easily proven true.

$\models$ `NewTree`: $GrBinTree \Rightarrow$ `NewTree`: $GrBinTree$

$\models$ `Tree`: $GrBinTree \Rightarrow$ `Tree`: $GrBinTree$

$\models$ `Max`: $Ground \Rightarrow$ `Max`: $Ground$

$\models true \Rightarrow$ `[]`: $GrList$

$\models$ `Labels`: $GrList \Rightarrow$ `Labels`: $GrList$

For the second clause we get:

$\models$ `L`: $GrList \Rightarrow$ `L`: $GrList$

$\models true \Rightarrow$ `void`: $GrBinTree$

For the third clause we have:

$\models$ `tree(Lbl,Left,Right)`: $GrBinTree \wedge$ `Out`: $GrList \Rightarrow$ `[Lbl|Out]`: $GrList$

$\models$ `Max`: $Ground \wedge$ `NLeft`: $GrBinTree \wedge$ `NRight`: $GrBinTree \Rightarrow$ `tree(Max,NLeft,NRight`: $GrBinTree$

$\models$ `tree(Lbl,Left,Right)`: $GrBinTree \Rightarrow$ `Left`: $GrBinTree$

$\models$ `Max`: $Ground \Rightarrow$ `Max`: $Ground$

$\models$ `In`: $GrList \Rightarrow$ `In`: $GrList$

$\models$ `tree(Lbl,Left,Right)`: $GrBinTree \Rightarrow$ `Right`: $GrBinTree$

$\models$ `Max`: $Ground \Rightarrow$ `Max`: $Ground$

$\models$ `Out1`: $GrList \Rightarrow$ `Out1`: $GrList$

For checking S-well-typedness it is now sufficient to check non-circularity of the program. An automatic checker would discover that the scheme is *strongly non-circular*, hence non-circular. For discussion of the concept of strong non-circularity see e.g. [21].

5 Call correctness under LD-resolution

We now consider the problem of call-correctness. It may turn out that for a given directional type the input assertions of certain predicate positions are call invariants under a given computation rule, while the others are not. In this section we give a sufficient condition for an input position to be a call invariant. We restrict our discussion to the Prolog computation rule, but the idea presented can also be extended to other computation rules.

Reconsider the `maxtree` program in Sect. 3 with the second directional type, that is:

$$\texttt{maxtree/2} : (\downarrow \mathit{GrBinTree},\ \uparrow \mathit{GrBinTree})$$
$$\texttt{maxtree/5} : (\downarrow \mathit{GrBinTree},\ \downarrow \mathit{Ground},\ \downarrow \mathit{GrList},\ \uparrow \mathit{GrList},\ \uparrow \mathit{GrBinTree})$$
$$\texttt{max/2} : (\downarrow \mathit{GrList},\ \uparrow \mathit{Ground})$$

When executed with LD-resolution, in every call to the recursive clause for `maxtree/5`, the first and third position (but not the second) are correctly typed. Upon success, the fourth position (but not the fifth) is correctly typed. Intuitively, the reason is that the dataflow to these positions follows the execution order of LD-resolution. We say that these positions are *well-typed*.

Definition 5.1 Let P be a program. $\mathcal{W}$ is a set of *well-typed* clause positions in P if it satisfies:

(1) Let H be a head of some clause in P. If $H(i)$ is an input position, and $H(i) \in \mathcal{W}$, then for all body atoms B that unify with H, $B(i) \in \mathcal{W}$.

(2) Let B be a body atom in some clause in P. If $B(i)$ is an output position, and $B(i) \in \mathcal{W}$ then for all heads H that unify with B, $H(i) \in \mathcal{W}$.

(3) Let $A_j(i)$ be an input position in the clause $H :- A_1, \ldots, A_n$. If $A_j(i) \in \mathcal{W}$, then its type can be inferred from the types of input positions in H which are elements of $\mathcal{W}$, and the types of output positions in $A_1, \ldots, A_{j-1}$ which are elements of $\mathcal{W}$.

(4) Let $H(i)$ be an output position in the clause $H :- A_1, \ldots, A_n$. If $H(i) \in \mathcal{W}$, then its type can be inferred from the type of input positions in H which are elements of $\mathcal{W}$, and the types of output positions in $A_1, \ldots, A_n$ which are elements of $\mathcal{W}$. □

□

It is easily realized that there exists a *largest* set of well-typed clause positions. Let S_1 and S_2 be two sets of well-typed clause positions, and suppose $S_1 \cup S_2$ is not a set of well-typed clause positions. For example, suppose case (1) is violated. Then there exists an input position $H(i)$ in some head H, such that $H(i) \in S_1 \cup S_2$, but $B(i) \notin S_1 \cup S_2$, for some body atom B that unifies with H. Obviously, this implies that either S_1 or S_2 contains $H(i)$ but not $B(i)$, which contradicts our assumption that S_1 and S_2 are sets of well-typed clause positions. For case (2)–(4) we reason similarly, proving that $S_1 \cup S_2$ is indeed a set of well-typed clause positions. Thus there exists a largest set of well-typed clause positions. A clause position of P will be called *well-typed* if it belongs to this set.

Definition 5.2 Let p be a predicate, and let $A_1, \ldots A_n$ be all atoms in P which have p as a predicate symbol. The i:th position of p is *well-typed* if $A_1(i), A_2(i), \ldots A_n(i)$ all are well-typed.

□

Let us exemplify definition 5.2 on the `maxtree` program. Consider the clause defining `maxtree/2`. The second clause position of the first body atom is *not* well-typed. Since this position is a input position in the body, we check case (3). We note that the type of the term filling in this position (`Max`) cannot be inferred from the types of the importing clause positions in the head.

Now consider the recursive clause for `maxtree/5`. The second position in the head is not well-typed, since (1) is not satisfied. This is due to that the position considered in the previous paragraph is not well-typed. As a consequence, the fifth position in the head, and the second position in the two body atoms are not well-typed, and so on.

The `maxtree` program, with its well-typed clause positions underlined, is shown below. (By also considering the definition of `max/2`, we would perhaps be able to show that also the second clause position of `max/2` is well-typed).

```
maxtree(Tree, NewTree) :-
       maxtree(Tree, Max, [], Labels, NewTree),
       max(Labels, Max).

maxtree(void, _, L, L, void).
maxtree(tree(Lbl,Left,Right), Max, In, [Lbl|Out], tree(Max,NLeft,NRight)) :-
       maxtree(Left, Max, In, Out1, NLeft),
       maxtree(Right, Max, Out1, Out, NRight).
```

We conclude that the first argument of `maxtree/2`, the first, third and fifth arguments of `maxtree/5`, and the first argument of `max/2` are well-typed.

Definition 5.3 Given a directional type $\mathcal{T}$ of a program P, we obtain $\mathcal{T}_W$, the *strongest well-typing compatible with* $\mathcal{T}$, as follows: For every predicate position e:

- if e is well-typed under $\mathcal{T}$, then it is given the same type by $\mathcal{T}_W$ as by $\mathcal{T}$;
- otherwise, e is given the type Any by $\mathcal{T}_W$.

□

Recall the `maxtree` program, and let $\mathcal{T}$ be the previous directional type. Then $\mathcal{T}_W$ is the following directional type:

`maxtree/2`: $(\downarrow GrBinTree, \uparrow Any)$

`maxtree/5`: $(\downarrow GrBinTree, \downarrow Any, \downarrow GrList, \uparrow GrList, \uparrow BinTree)$

`max/2`: $(\downarrow GrList, \uparrow Any)$

Theorem 5.4 Let $\mathcal{T}$ be a directional type for P. Then $\mathcal{T}_W$ is a well-typing for P.

Proof : Let C be the clause $H :- A_1, ...A_n$, and consider the position $A_k(i)$. If this position is importing, we do not need to consider it. If it is exporting, then according to definition 2.2 we must be able to infer its type from the types of importing positions in $H, A_1, \ldots, A_{k-1}$. This is obviously possible if the type of $A_k(i)$ is Any; thus the only interesting case is when $A_k(i)$ has some type different from Any. By the construction of $\mathcal{T}_W$, this case will only arise if $A_k(i)$ is a well-typed clause position under $\mathcal{T}$.

If $A_k(i)$ is a well-typed clause position, then according to case (3) of definition 5.1, we can infer its type from well-typed importing clause positions in $H, A_1, \ldots, A_{k-1}$. Since these positions are typed the same way by $\mathcal{T}$ and $\mathcal{T}_W$, it follows immediately that we can infer the type of $A_k(i)$ from the types of importing positions in $H, A_1, \ldots, A_{k-1}$.

For exporting positions in H we reason similarly, thus proving that $\mathcal{T}_W$ is indeed a well-typing for P. □

We now state some immediate corollaries of the above theorem.

Corollary 5.5 Let P be a program, and let G be an atom which is correctly typed in its input positions. Then for every computed answer substitution σ, $\sigma(G)$ is correctly typed in its well-typed output positions.

Corollary 5.6 Let P be a program, and let G be an atom which is correctly typed in its input positions. Let $H :- A_1, \ldots, A_n$ be a clause in P. Then in every LD-derivation starting from G: if the query $\sigma(A_j, \ldots, A_n, B_1, \ldots, B_m)$ is reached, then the well-typed input positions in $\sigma(A_j)$ are correctly typed.

The idea behind the notion of well-typed position is related to the annotation method. The call-correctness concerns properties of incomplete nodes of derivation trees. The computation rule defines a class of incomplete derivation trees which will be constructed during the computations. The property of an input argument of an incomplete node can be proved by the annotation method provided that this argument logically depends only on the arguments of some complete nodes of the (incomplete) tree. Thus the problem is which of the input arguments of the program predicates are always fully determined in every incomplete derivation tree which can be constructed under the considered computation rule. The concept of well-typed position gives a sufficient condition identifying such arguments for Prolog computation rule.

6 Type-driven resolution

Several existing Prolog systems (like e.g. [14]) provide delaying primitives, which allow for suspension of resolution of the selected goal until it is sufficiently instantiated, for example until some of its arguments are ground. The non-suspension condition can often be formalized as a type of the atom, typically the groundness of some argument is required. This section

presents a model of computation where directional types are used for controlling execution. This is formalized as a notion of *type-driven resolution* (*T-resolution* for short).

The delays in Prolog concern resolution steps: a selected atom which does not fulfill an imposed condition is not resolved. In contrast to that, the selected atom in our execution mechanism is first compared with the head of a clause. As result a set of equations is obtained, one equation per argument position of the atom. The delay mechanism we are going to discuss concerns unification of the equations in this set. A non-suspension condition is used for controlling which of them may be selected for unification. The remaining ones are kept suspended until they become sufficiently instantiated. Thus, the usual resolution step is divided into the equation generation phase and the equation solving phase.

Notice, that Prolog execution with delays is still an SLD-resolution, though with a dynamic computation rule. In contrast to that, our model of computation is not an SLD resolution.

In contrast to Prolog with delays the equation generation phase in our model is never suspended and a selected atom of the goal is always replaced by the body atoms of some clause. Thus, the computation may reach a state where the goal is empty and the set of the suspended unifications is non-empty. Such a situation may be seen as a kind of deadlock. We show in this Section that S-well-typedness is a sufficient condition for a program to be deadlock-free.

6.1 T-resolution

We now give a formal definition of T-resolution. We consider logic programs with restricted directional types. Thus, the i-th argument position of a program predicate p/n is always classified as an input or as an output and has associated a type T_i. We first introduce some auxiliary concepts.

Definition 6.1 [Query] A *query* is a either the atom **fail** or a pair $(G; E)$, where G is a sequence of atoms, and E is a set of equations. For an initial query (given by the user), we require that $E = \emptyset$. □

Definition 6.2 [Eligible equation] Let p/n be a predicate and let $p(s_1, \ldots, s_n) = p(t_1, \ldots, t_n)$ be an equation. The equation $s_j = t_j$ is *eligible* if either

- the j:th position of p is an input, and $\models s_j : T_j$, or
- the j:th position of p is an output, and $\models t_j : T_j$.

□

Definition 6.3 [T-derivative] Let $Q \equiv (G; E)$ be a query. A *T-derivative* Q' is a query obtained from Q as follows:

1. If E contains a trivial equation $t = t$, then $Q' \equiv (G; E - \{t = t\})$;

2. Otherwise, if E contains a non-trivial equation of the form $p(s_1, \ldots, s_n) = p(t_1, \ldots, t_n)$, where $s_j = t_j$ is an eligible equation, and s_j and t_j unify with mgu σ, then $Q' \equiv (\sigma(G), \sigma(E))$. If s_j and t_j do not unify then $Q' \equiv$ **fail**.
3. Otherwise (if no eligible equation is found in E), if $G \equiv A_1, \ldots, A_k$, where $A_1 \equiv p(s_1, \ldots, s_n)$, and $H :- B_1, \ldots B_n$ is a (renamed) clause, where $H \equiv p(t_1, \ldots, t_n)$, then
$$Q' \equiv (B_1, \ldots, B_m, A_2, \ldots A_k \ ; \ E \cup \{p(s_1, \ldots, p_n) = p(t_1, \ldots, t_n)\})$$
4. Otherwise, Q has no T-derivative.

□

A more elaborate concept of the notion of T-derivative might have required unifiability of the delayed equations. This would correspond to the usual satisfiability requirement for the accumulated constraints in a constraint system. The main result of this section extend easily for such a modified version of T-resolution.

Definition 6.4 [T-derivation] A *T-derivation* is a sequence of queries $Q_1, Q_2, \ldots$, such that Q_{j+1} is a T-derivative of Q_j. Consider a finite T-derivation which ends with a query for which no T-derivative exists. The T-derivation is:

- *successful* if it ends with $(\epsilon; \emptyset)$;
- *deadlocked* if it ends with $(\epsilon; E)$, where E is a non-empty set of equations;
- *failed* otherwise, i.e. if it ends with **fail** or a query of the form $(G; E)$, where G is a non-empty sequence.

□

For successful derivations we can compute answers in the ordinary way by composing all the substitutions obtained in the derivation. The soundness of T-resolution follows directly from the soundness of LD-resolution, since the same equations are solved, albeit possibly in a different order. T-resolution is not complete since some derivations may deadlock, but theorem 6.5 constitutes a restricted completeness result.

Using the "proof tree view" on resolution, a successful derivation corresponds to the case where we can construct a complete skeleton and solve all the associated equations. A deadlocked derivation corresponds to the case where we can construct a complete skeleton, but there is at least one equation which cannot be selected for solving, due to that the terms therein are not instantiated to the right type.

We illustrate this resolution process on an example, Reconsider the `maxtree` program in Sect. 3. We now type it as follows:

$$\texttt{maxtree/2} : (\downarrow \mathit{GrBinTree}, \uparrow \mathit{GrBinTree})$$
$$\texttt{maxtree/5} : (\downarrow \mathit{GrBinTree}, \downarrow \mathit{Ground}, \downarrow \mathit{GrList}, \uparrow \mathit{GrList}, \uparrow \mathit{GrBinTree})$$
$$\texttt{max/2} : (\downarrow \mathit{GrList}, \uparrow \mathit{Ground})$$

Consider the initial query

```
(maxtree(tree(5,void,void), N); {}
```

We resolve it against the only possible clause, but we keep one equation unsolved, yielding the query:

```
(maxtree(tree(5,void,void),Max1,[],Labels1,NewTree1,
 max(Labels1,Max1);
{ N=NewTree1 }
```

We can depict this with the incomplete proof tree shown in figure 1. (Two terms stacked upon each other indicate an unsolved equation.) We continue by resolving the leftmost atom in the query (expand the node N_1).

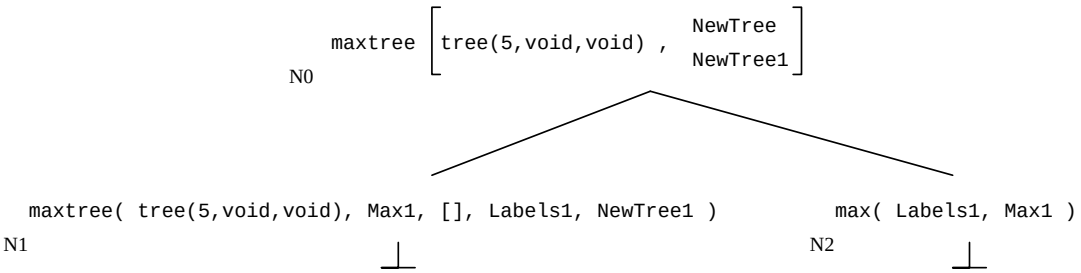


Figure 1: An incomplete proof tree

Some derivation steps later we obtain the tree shown in figure 2.

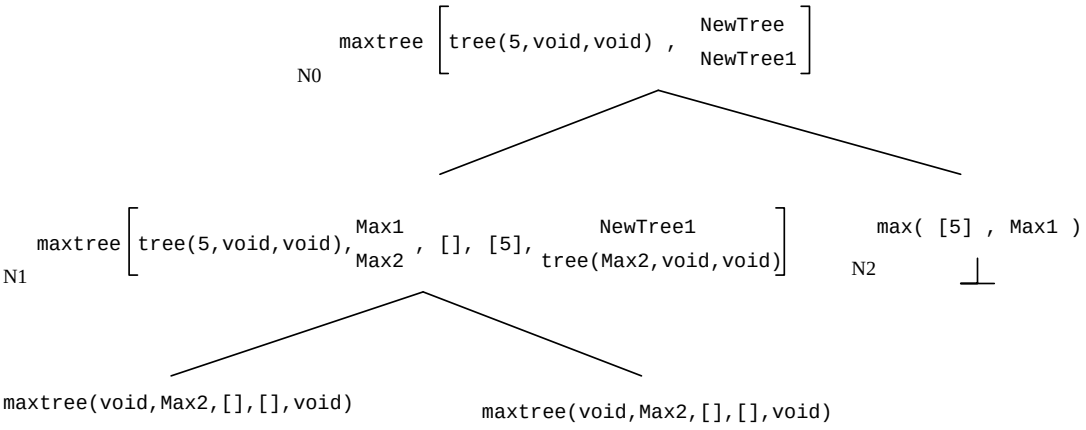


Figure 2: Another incomplete proof tree

When we now resolve the atom `max([5], Max1)`, the variable `Max1` becomes instantiated to 5. We can now solve the equation `Max1=Max2` at node N_1, since `Max1` now is instantiated to

the right type (*Ground*). We can then solve the equation `NewTree1=tree(Max2,void,void)` at node N_2, and finally the equation `NewTree=NewTree1` at node N_0.

Hopefully this example has conveyed the general idea of type-driven resolution: unification is performed argumentwise in "dataflow order".

6.2 A sufficient condition for deadlock-freeness

The possibility of deadlock when executing a program with T-resolution, raises the question if it is possible to detect the cases where T-resolution really computes all answers, i.e. where deadlock does not occur. It turns out that the notion of S-well-typedness is a sufficient condition for that.

Theorem 6.5 Let P be a program which is S-well-typed, and let G be an atom which is correctly typed in its input positions. Then no T-derivation starting from $(G;\emptyset)$ will deadlock.

Proof : Assume the contrary. Then starting from G, we can construct a complete skeleton T such that at least one equation will never be selected for solving.

Since P is non-circular, the $\leadsto_T$ relation is a partial ordering. We will prove by induction on $\leadsto_T$ that it is possible to select equations until we reach **fail**, or until all equations are selected and solved. Hence it is impossible to construct a deadlocked derivation.

(Base step) The minimal elements of $\leadsto_T$ are **(1)** the input positions at the root of the T, and **(2)** the positions associated with an equation $c = t$, or $t = c$, where c is a term occurring on an exporting position of a clause D and not sharing variables with any importing positions of D. occurring at an exporting position $A(i)$ in some clause D. In case (1), by assumption the input positions of the initial query are correctly typed, so these equations can be selected. In case (2), the verification condition for c has empty set of premisses, hence c is correctly typed, and the equation $c = t$ can be selected.

(Induction step) Let n_i be a node position associated with the equation $s = t$, where s is a term occurring at an exporting position in some clause D. Let $dep(n_i)$ be the set of all node positions related to n_i under the $\leadsto_T$ ordering. By the induction hypothesis, all equations at the positions in $dep(n_i)$ can be selected. If any of these equations is unsolvable, we have reached **fail**. Otherwise all of these equations can be solved, and since by assumption the program is S-well-typed, the type of s can be inferred from the types of the terms computed at the nodes in $dep(n_i)$. Thus s is correctly typed, and the equation $s = t$ can be selected. □

The following theorem is a direct corollary of the previous results and definitions.

Theorem 6.6 Let P be a program which is S-well-typed, and let g be an atom which is correctly typed in its input positions. Then for every answer σ computed by T-resolution, $g\sigma$ is correctly typed in its output positions.

Proof :
As T-resolution is sound, the result follows by Theorem 4.7. □

We illustrate the use of T-resolution by another example modelling a version of producer-consumer process. A producer p may freely produce some items i which are to be consumed by a consumer r. The consumer writes a confirmation c for every consumed item. The confirmation is read by the producer. The process terminates after the producer has finished the production and the consumer has consumed all produced items. For achieving the simulation effect we use the following types:

- $IListStr$ consists of all terms of the form $[i|t]$ where i is a constant (representing one produced item) and t is arbitrary term.
- $CListStr$ consists of all terms of the form $[c|t]$ where c is a constant (representing a confirmation of consumption) and t is arbitrary term.
- C is the singleton $\{c\}$.
- I is the singleton $\{i\}$.

The producer and the consumer are binary predicates. Their arguments reflect their information about the state of production and the state of consumption.

$\mathtt{p} : (\uparrow IListStr, \downarrow CListStr)$
$\mathtt{r} : (\downarrow IListStr, \uparrow CListStr)$
$\mathtt{read} : (\downarrow \{C\})$
$\mathtt{write} : (\downarrow \{I\}, \uparrow \{C\})$

$\mathtt{pc} : -\mathtt{p}(\mathtt{X}, \mathtt{X1}), \mathtt{r}(\mathtt{X}, \mathtt{X1}).$
$\mathtt{p}([\mathtt{i}, \mathtt{i}|\mathtt{T}], [\mathtt{X1}, \mathtt{X2}|\mathtt{T1}]) : - \ \mathtt{read}(\mathtt{X1}), \mathtt{p}([\mathtt{i}|\mathtt{T}], [\mathtt{X2}|\mathtt{T1}]).$
$\mathtt{read}(\mathtt{c}).$
$\mathtt{r}([\mathtt{X1}, \mathtt{X2}|\mathtt{T}], [\mathtt{Y1}, \mathtt{Y2}|\mathtt{T1}]) : - \ \mathtt{write}(\mathtt{X1}, \mathtt{Y1}), \mathtt{r}([\mathtt{X2}|\mathtt{T}], [\mathtt{Y2}|\mathtt{T1}]).$
$\mathtt{write}(\mathtt{i}, \mathtt{c}).$
$\mathtt{p}([\mathtt{i}], [\mathtt{X}]) : - \ \mathtt{read}(\mathtt{X}).$
$\mathtt{r}([\mathtt{X}], [\mathtt{Y}]) : - \ \mathtt{write}(\mathtt{Y}).$

This program is S-well-typed hence its execution until T-resolution starting with the goal $\leftarrow \mathtt{pc}$ is deadlock-free. Such an execution simulates the interaction of the producer and the consumer described above. If the actual goal contains a call of the producer, then the first argument of this call can be resolved. Thus the producer can produce freely until it terminates. The communication with the consumer is obtained at the top level of the derivation tree. Each produced item causes further instantiation of the list structure bound to X and

releases a possibility for one consumption step, that is one delayed equation corresponding to the first argument of r in some node becomes eligible. This releases the possibility of writing a confirmation in the second argument of the same node. The confirmation is passed back to the top level of the tree and releases the possibility of reading the confirmation by the producer.

7 Examples

In this section we give some more examples of programs which are not well-typed (given an "intuitive" typing), but which are S-well-typed. The programs of Section 7.1 and 7.2 could be made well-typed under the considered directional types by changing the order of the body atoms in some clauses. The programs originate from the literature, thus we are interested in proving IO correctness of their directional types without transforming them. This does not apply to the program of Section 7.3. It cannot be made well-typed under the considered directional type by rearrangement of body atoms of the clauses.

7.1 Flattening a list

A program flattening a list using difference-lists is given in the book by Sterling and Shapiro [37]. Below we give a version of the program where instead of using difference lists, we use two arguments.

```
flatten(Xs, Ys) :- flatten(Xs, Ys, []).

flatten([X|Xs], Ys, Zs) :-
        flatten(X, Ys, Ys1),
        flatten(Xs, Ys1, Zs).
flatten(X, [X|Xs], Xs) :-
        simple(X), \+ X=[].
flatten([], Xs, Xs).
```

We would like to prove that if we call `flatten/2` with the first argument bound to a list, then the second argument will be bound to a list upon success. However, to type the predicate `flatten/3` in a way that reflects how the program executes under LD-resolution, is not so easy. The most precise we can do is:

$$\texttt{flatten/2} : (\downarrow List,\ \uparrow List)$$
$$\texttt{flatten/3} : (\downarrow Any,\ \uparrow Any,\ \downarrow Any)$$

The reader can easily verify that under this directional type the program is not well-typed, so theorem 2.3 is inapplicable here.

We now type the program in a way that reflects the dataflow in the program:

$$\texttt{flatten/2} : (\downarrow List,\ \uparrow List)$$
$$\texttt{flatten/3} : (\downarrow Any,\ \uparrow List,\ \downarrow List)$$

This time the program is S-well-typed, thus the second argument of `flatten/2` will indeed be bound to a list upon success.

7.2 Quicksort on difference lists

An example of a directional type for a quicksort program using difference lists has been discussed in [13]. For proving that the program transforms an integer list into an integer list an extension to polymorphic directional types has been used. Notice that the result concerns input-output behaviour of the program. In this section we show that it can be obtained by a simple S-well-typing. Below we give a version of the program where instead of using difference lists, we use two arguments.

```
quicksort(X,Y)          :- quicksort-dl(X,Y,[]).
quicksort-dl([],X,X).
quicksort-dl([A|X],Y,Z) :- partition(X,A,L,U),
                           quicksort-dl(U,Y1,Z), quicksort-dl(L,Y,Y1).
```

We assume that `partition` is defined by a set of clauses S-well-typed under the following directional type:

`partition`: $(\downarrow IntList, \downarrow Int, \uparrow IntList, \uparrow IntList)$

Under this assumption we want to show that `quicksort` called with the first argument being an integer list binds on success its second argument to an integer list. This follows immediately from the fact that the program is S-well-typed under the following directional type:

`quicksort`: $(\downarrow IntList, \uparrow IntList)$
`quicksort-dl`: $(\downarrow IntList, \uparrow IntList, \downarrow IntList)$
`partition`: $(\downarrow IntList, \downarrow Int, \uparrow IntList, \uparrow IntList)$

7.3 A small typesetting program

As mentioned in section 2.3, we may also consider definite programs interpreted on some domain other than the term domain. The concept of a S-well-typed program can then be very useful in program analysis, as it extends the groundness analysis method used for such programs in our previous research [10, 11].

In the example below, we assume that the functor `+` is interpreted as a function computing the sum of two integers, and that the functor `size` is interpreted as a function returning the number of characters of its only argument. Operationally, we assume that these interpreted symbols are handled as follows: Whenever an interpreted term, e.g. $s + t$, is unified with another term u, it is checked that s and t both are ground terms. If so, $s + t$ is evaluated, and the result is unified with u. Otherwise, the unification is delayed until both s and t become ground. Thus we can view the operational semantics as being based on a restricted

form of T-resolution (where every predicate argument is typed either as $\downarrow Ground$ or as $\uparrow Ground$), combined with evaluation of interpreted terms. The restriction is that only the arguments including interpreted functors are delayed, while the others are unified without delay, whenever selected. In [9], a condition which guarantees deadlock-freeness of such execution was given. The condition is a special case of S-well-typedness: It uses only the type *Ground* and requires that the interpreted terms appear only at the exporting positions.

Note that in the presence of interpreted terms the concept of T-resolution is very natural, and similar ideas have appeared in several functional logic languages, e.g. in [3, 2, 31, 30].

Consider a simple typesetting program which typesets text tables. The input to the program consists of a description of the text table as a list of lists, for instance:

```
[[This, is, some, text], [Another, line, of, text]]
```

The produced output consists of a list of typesetting commands:

```
[[put(1,1,This), put(1,8,is), put(1,12,some), put(1,16,text)],
 [put(2,1,Another), put(2,8,line), put(2,12,of), put(2,16,text)]]
```

where the two first arguments to `put` represent the line and the indentation on the line. In this case, the output list of typesetting commands represents the table:

This	is	some	text
Another	line	of	text

Note that every column has the width of the longest word of the column. Thus the widths of the individual columns cannot be computed until the whole input has been processed by the program. By using the same programming technique as in the example of section 3, we still obtain an efficient "one-pass" program.

First we consider the predicate `typesetrow/6`, which typesets one row of the table.

(C_1)
```
typesetrow(_, _, [], [], [], []).
```

(C_2)
```
typesetrow(Line, Ind, [Text|Ts], [MaxWid|Ms], [size(Text)|Ss],
               [put(Line, Ind, Text)|Insts]) :-
        typesetrow(Line, Ind+MaxWid, Ts, Ms, Ss, Insts).
```

The arguments represent (from left to right) the current line number, the current indentation on the line, the description of one row of the table (for instance `[This, is, some, text]`), the width of the widest element in each column, the number of characters in each element of the row, and the output list of typesetting instructions. Conceptually, the first four arguments represent the input to the predicate, while the last two arguments represent the output. Thus the natural directional type for `typesetrow` (using only the type *Ground*) would be:

$$\texttt{typesetrow/6} : (\downarrow Ground, \downarrow Ground, \downarrow Ground, \downarrow Ground, \uparrow Ground, \uparrow Ground)$$

or more precisely

$$\texttt{typesetrow/6}:(\downarrow Nat, \downarrow Nat, \downarrow GrList, \downarrow ListofNat, \uparrow Nat, \uparrow ListofCmd)$$

The types used in the more precise directional type are as informally defined in the intuitive explanation above. Thus, Nat is the type of natural numbers, Cmd is the type of the typesetting instructions, and $ListofT$, for T being a type, denotes the type of lists with the elements of type T.

`typesetrow/6` is called from `typesettab/6`:

(C_3)
```
typesettab(_, _, [], X, X, []).
```

(C_4)
```
typesettab(Line,Ind,[Row|Rows],MaxWidths,MaxSoFar,[InstRow|Insts]) :-
        typesetrow(Line, Ind, Row, MaxWidths, Widths, InstRow),
        compute_max(Widths, MaxSoFar, NewMaxSoFar),
        typesettab(Line+1, Ind, Rows, MaxWidths, NewMaxSoFar, Insts).
```

The arguments represent (from left to right) the current line number, the current indentation on the line, the description of the whole table (as a list of lists), the width of the widest element in each column, the width of the widest element in each column in the rows processed so far, and the output list of typesetting instructions. Conceptually, the first, second, third and fifth arguments represent input to the predicate, while the fourth and sixth arguments represent output. Thus a natural directional type is:

$$\texttt{typesettab/6} : (\downarrow Ground, \downarrow Ground, \downarrow Ground, \uparrow Ground, \downarrow Ground, \uparrow Ground)$$

or more precisely:

$$\texttt{typesettab/6} : (\downarrow Nat, \downarrow Nat, \downarrow ListofGrList, \uparrow ListofNat, \downarrow ListofNat, \uparrow ListofListofCmd)$$

We assume that `compute_max` is a predicate that, given two lists of integers $[i_1, \ldots, i_n]$ and $[j_1, \ldots, j_n]$, returns the list $[max(i_1, j_1), \ldots, max(i_n, j_n)]$ (we omit the definition of `compute_max`). Thus the directional type for `compute_max` is:

$$\texttt{compute\_max/6} : (\downarrow Ground, \downarrow Ground, \uparrow Ground)$$

or more precisely:

$$\texttt{compute\_max/6} : (\downarrow ListofNat, \downarrow ListofNat, \uparrow ListofNat)$$

The first directional type describes groundness properties of the program. It tells for instance that if we call the program with the goal

```
?- typesettab(1, 1, [[This,is,some,text],[Another,line,of,text]], _,
   [0,0,0,0], I)
```

then the variable `I` will be ground upon success. This reflects our intention, since `I` is supposed to be bound to the resulting list of typesetting instructions.

The reader may verify that for both directional types considered above the program is not well-typed. On the other hand, for both of them the program is S-well-typed, hence IO correct. Since the interpreted terms appear only at the exporting positions, the first directional type satisfies the deadlock-freeness condition of [9]. The second, more precise, directional type describes an interesting property of the program.

8 Related Work

8.1 General Proof Methods for Run-Time Properties

As pointed out in the introduction, directional type checking for Prolog can be seen as a special case of proving run-time properties of the Prolog program. The early papers addressing this problem are [25, 8, 18]. In the approach of [25], input-output assertions are assigned to all predicates. Correctness of the assertions are proved by showing a verification condition locally for each clause. The method of [8] can be seen as a special case of [25], where the properties described by the assertions are closed under substitution. The input assertions of [25] may express arbitrary relations between the predicate arguments at call and the output assertions may express any relations between the predicate arguments at success and its arguments at call. Thus, the directional types are assertions such that

- both the input and the output assertions are products of types closed under substitution,
- output assertions express only relations between predicate arguments at success but cannot relate them to the arguments at call.

The types of [4], discussed also in this paper, are even more restricted, since for every argument one can only specify either its input type or its output type, but not both. In practice, the restricted directional types are often sufficient to show interesting properties of programs. On the other hand, checking of a non-restricted directional type of a program may be reduced to checking of a restricted directional type of a transformed program. We illustrate this idea by the following example.

Consider the `append` program

```
append([], X, X).
append([E|L], R, [E|LR]) :- append(L, R, LR).
```

with the directional type :

$$\texttt{append:} \downarrow (List, Any, GrList) \uparrow (GrList, GrList, Any)$$

The directional type defines different input and output types for the first argument of the predicate, which thus can be classified neither as an input nor as an output argument. In the

following version of the program the first argument has been duplicated: its first copy plays the role of input while the second one is an output. The execution of `append1` simulates the execution of `append` in an intuitively clear way.

```
append1([],  [], X, X).
append1([E|L],  [E|L1], R, [E|LR]) :- append(L,  L1, R, LR).
```

The following directional type of the transformed program

$append1 : (\downarrow List \uparrow Groundlist \uparrow Groundlist \downarrow Groundlist)$

corresponds to the unrestricted directional type of the original program.

The second above mentioned paper ([18]) on proving runtime properties of Prolog programs assigns assertions to program points. This shows another, still unexplored way of dealing with directional types: by assigning them to occurrences of the predicates in program clauses rather than to predicates. The concept of well-typed position discussed in Section 5 is a step in that direction. However, for large programs it may be rather difficult for the user to specify this kind of directional type.

While the methods mentioned above are specialized for LD-resolution and aim at proving at once both the IO correctness and the call correctness of a given specification, our approach to directional types separates clearly these two aspects. Hence we are able to prove IO correctness of directional types which are not call correct under LD-resolution. The methods mentioned above do not apply to such directional types.

8.2 Related work on directional types

As already pointed out the paper [4] has been a direct inspiration for the way we present our results. We have shown that the well-typing condition, as presented therein, is a specialization of the annotation method. In contrast to the annotation method, where the LDS is a subject of search, the LDS for well-typing is determined by a given program and a given directional type. Our S-well-typing condition is another specialization of the annotation method using a different kind of LDS. The LDS used by well-typing is non-circular by the definition and thus simpler to check than S-well-typing.

S-well-typing does not imply well-typing, since only IO correctness is guaranteed, while well-typing implies also call correctness. On the other hand, well-typing does not imply S-well-typing. One of the reasons is that the dependency relation of a well-typed program may be circular. For example, consider the program:

`q(X) :- p(X,X).`
`p(X,X).` with the directional type:

`q`:$(\downarrow Ground)$
`p`:$(\downarrow Ground, \uparrow Ground)$

The dependency relation of the program is circular, hence the program is not S-well-typed but it is well-typed. Another reason concerns "trivially" well-typed programs, where the premisses of a verification condition are never satisfied. For example, consider the program:

```
p([]) :- q(X).
q(X).
```
with the directional type:
`p:`($\downarrow$ *BinTree*) `q:`($\downarrow$ *List*)

The program is well-typed since:

$$\models BinTree([]) \Rightarrow List(X)$$

However, it is not S-well-typed since the verification condition

$$true \Rightarrow List(X)$$

is not valid.

Most of the papers on directional types consider types closed under substitution. We have shown that under this assumption specializations of the annotation method can be used for type checking. The well-typedness criteria presented in the literature are mostly such specializations of the annotation method. There is no clear reason why the assertions of a directional type should only concern individual arguments of the predicate. This has been pointed out also in [12]. As long as the assertions are closed under substitution, whether they are unary or not, they can be handled by the annotation method. Soundness of any new sufficient condition obtained by the specialization of the annotation method would automatically follow from the soundness of the method.

Types not closed under substitution can generally not be handled by the annotation method. Correctness of such types may still be proved by the method of [25]. Actually the method of [25] is complete [24], so that its verification condition can be seen as the best possible well-typing for Prolog execution rule.

All papers on directional typing known to the authors concern Prolog computation rule and thus are not directly applicable to other execution methods. Our approach makes it possible to discuss various execution principles in one uniform framework. In particular, we are able to prove some interesting properties of LD-execution of programs with incomplete data structures. For such programs it is usually rather difficult to provide non-trivial well-typings. A recent work presented in [13] suggests a polymorphic extension of well-typing to deal with them. It is interesting to note that for the quicksort example, discussed in [13], S-well-typing condition is sufficient as well. Admittedly, the example cannot be handled by the well-typing condition.

The concepts of S-well-typedness and of well-typed position generalize conditions proposed in [22] for groundness analysis of definite programs. The data-driven programs of [22] are well-typed programs, such that the only type used is the type *Ground* of all ground terms. Similarly, the simple programs of [22] are S-well-typed programs with the only type

Ground. An extension of the concept of simple program has been used for groundness analysis and for analysis of delays in the language GAPlog [30] integrating logic programs with external procedures in a clean declarative way. A combination of similar kind of groundness analysis with some properties of unification has been used for studying occur-check, e.g. in [7], termination, e.g. in [34], AND-parallelism [23], and the question whether a program can be executed with pattern matching instead of full unification, e.g. in [5].

Some rudimentary concepts of well-typedness are used in the above mentioned papers for static analysis of logic programs. As mentioned above, well-typedness is conceptually rooted in proof techniques and in reasoning about programs. On the other hand a widely accepted approach to program analysis is based on the notion of abstract interpretation. Abstract interpretation techniques have been proposed for many purposes, among others for groundness analysis, e.g. in [16, 17] and for type analysis e.g. in [27].

The relation between proof-based methods and abstract interpretation based methods is not obvious. We note here that the directional types, as discussed in this paper and in the related publications are prescriptive and concern the predicates while the abstract interpretation approach intends to synthesise properties, usually for program points. On the other hand, for a given abstract domain of types the verification conditions of well-typing for an untyped program could probably be seen as fixpoint equations for assigning directional types to the predicates. A more closer investigation of the relations is a subject of future work.

The notion of well-typed program has been extended to constraint logic programs in [38]. A static type checker implemented for a non-trivial constraint logic programming language checks a sufficient condition for well-typing. The checker is based on the ideas of abstract interpretation. The results of the experiments reported are quite promising as concerns both the efficiency of the checker and the coverage of the correctly typed programs by the sufficient test proposed.

The problem of implementation of a type checker for directional types has also been addressed in [1] in an original way, but there is no reference to existing implementation.

9 Discussion and Conclusions

We have separated two aspects of directional types: the input-output characterization of the program, which is independent of the computation rule, and the characterization of the call patterns, which strongly depends on the execution model. We have shown that the input-output aspect can be proved by the annotation method, provided that the types are closed under substitution. We have also shown that the method can be specialized to obtain relatively simple sufficient conditions for IO correctness. As a matter of fact, the well-typing condition proposed in the literature for types closed under substitution can be seen as a specialization of the annotation method. Analysis of the well-typedness condition from this point of view shows a methodology of specialization of the annotation method to relatively simple special cases, with good prospects of automation. On the other hand, it

shows possible ways of modification and generalization of existing well-typedness conditions. The S-well-typedness is an interesting modification obtained in that way.

We also considered directional types as a tool for controlling execution of logic programs. The idea of such execution is to enforce the types specified by a given declaration by argument-wise delaying of unification of the arguments which are not correctly typed. This mechanism is more fine-grained than atom-wise delaying in some Prolog systems. It suspends only the unification of single equations, but it does not suspend the resolution steps as done in the Prolog systems. The idea of delaying the resolution of an equational constraint until it becomes sufficiently instantiated resembles the concept of the *ask* primitive in concurrent constraint programming [36]. We formalized the model of execution by the concept of T-resolution. The T-resolution is not an SLD-resolution.

The T-resolution is sound but it is not complete, even for definite programs, since the computation may deadlock. In particular, if the imposed directional type restricts the least Herbrand model of the program, the elements of the model which are not correctly typed according to the output assertions cannot be computed by T-resolution. We have shown that S-well-typing gives a sufficient condition for deadlock-free execution under T-resolution. The notion of S-well-typing uses a concept of dependency relation similar to that introduced for attribute grammars, and refers to the techniques of attribute grammars for checking properties of this relation.

Data flow in S-well-typed programs is well characterized by the dependency relation. Therefore the delays under T-resolution are predictable in compile time. Consequently, they can be compiled out, at least in some cases, by source-to-source transformations, similar to those described in our previous work [10], where the resulting logic program is executed without delays with the Prolog computation rule. An alternative approach to implementation of S-well-typed programs may rely on scheduling techniques used in attribute evaluators. A proposal for the use of such techniques in logic programming is the multi-pass execution of logic programs discussed [32]. In this way one would achieve the effects of T-resolution by computational mechanisms without dynamic delays. This topic is, however, outside of the scope of this paper.

The usefulness of T-resolution is an open question. In this paper we use it more as an illustration of the thesis that the methods of directional types apply not only to Prolog. It is an interesting question, whether a variant of the technique used here for deriving a sufficient condition for deadlock-freeness of T-resolution may be of interest for concurrent constraint programming. We note also that we used a similar technique for compiling out dynamic delays in the new implementation [26] of the GAPLog language ([30]).

Our future work aims at automated checking of S-well-typedness of programs. This includes two aspects. The non-circularity of the LDS can be automatically checked by the methods used for attribute grammars. The second aspect concerns checking of the verification conditions for the clauses. This is similar to well-typedness, so that the advances in automation of well-typedness can probably be re-used for that purpose. However, well-typedness itself is undecidable in general case so that identification of interesting and tractable special

cases is essential here. The results on this topic reported in the literature seem still to be preliminary.

Another interesting question is, whether the conditions for well-typing and S-well-typing can be used for types which are not closed under substitution. In general, correctness of such directional types can only be proved by taking into consideration a particular computation rule and using a version of the method of [25]. However, under certain syntactic restrictions on the program well-typedness and S-well-typedness could probably still be applicable.

Acknowledgements.

The authors would like to thank many persons who substantially contributed to the present version of this report. The first idea of writing this paper came from discussions with Gilberto Filé. The comments of Pierre Deransart on the initial version allowed us to substantially improve the presentation. We acknowledge gratefully the comments of Wlodek Drabent, Gérard Ferrand and Eric Vétillard.

References

[1] A. Aiken and T. K. Lakshman. Directional type checking of logic programs. To appear in *Proc. of SAS'94*. Springer-Verlag, 1994.

[2] H. Aït-Kaci. An overview of LIFE. In Schmidt, Stogny (eds.) *Next generation information system technology*, pp. 42–58, LNCS 504, Springer-Verlag, 1990.

[3] H. Aït-Kaci, P.Lincoln and R. Nasr. Le Fun: Logic, Equations and Functions. In: *Proc. 1987 SLP*. pp. 17-23. IEEE 1987.

[4] K.R. Apt. Declarative programming in Prolog. In *Proc. of ILPS'93*, pp. 12–35. The MIT Press, 1993.

[5] K.R. Apt and S. Etalle. On the Unification-free Prolog programs. In . *Proc. of the 1993 Conf. on Mathematical Foundations of Computer Science*. Springer-Verlag, 1993.

[6] K.R. Apt and E. Marchiori. *Reasoning about Prolog programs: from modes through types to assertions*. Technical report CS-R9358, CWI Amsterdam, 1993.

[7] K.R. Apt and A. Pellegrini. Why the occur-check is not a problem. In *Proc. of PLILP'92*, pp. 21-48. Springer-Verlag, 1992.

[8] A. Bossi and N. Cocco. Verifying correctness of logic programs, in: *Proc. of TAPSOFT'89*, LNCS 352, pp. 96-110, 1989.

[9] J. Boye. S-SLD-resolution: An Operational Semantics for Logic Programs with External Procedures. In *Proc. PLILP'91*, LNCS 528 pp. 283-393. Springer-Verlag, 1993.

[10] J. Boye. Avoiding Dynamic Delays in Functional Logic Programs. In *Proc. PLILP'93*, pp. 12-27. Springer-Verlag, 1993.

[11] J. Boye, J. Paakki and J. Małuszyński. Synthesis of directionality information for functional logic programs, in: Cousot, P. (ed.) *Proc. of the Workshop on Static Analysis, WSA'93. Springer-Verlag 1993, LNCS 724, 165-177*

[12] F. Bronsard, T. K. Lakshman and U. Reddy. A framework of directionality for proving termination of logic programs. In *Proc. of JICSLP'92*, pp. 321–335. The MIT Press, 1992.

[13] F. Bronsard, T. K. Lakshman and U. Reddy. Type-based reasoning of incomplete data structures. Draft 1994.

[14] M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Boortz and T. Sjöland. *SICStus Prolog user's manual*. SICS, Box 1263, S-164 28 Kista, Sweden.

[15] K. Clark. *Predicate logic as a computational formalism*. Technical report 79/59, Imperial College, London, 1979.

[16] M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In *Proc. of 1993 ISLP*, pp. 114-129. The MIT Press, 1993.

[17] P. Codognet and G. Filé. Computations, abstractions and constraints in logic programs. In *Proc. of the 4th Int. Conf. on Programming Languages*, 1992.

[18] L. Colussi and E. Marchiori. Proving correctness of logic programs using axiomatic semantics. In *Proc. of ICLP'91*, pp. 629–642. The MIT Press, 1991.

[19] D. Courcelle and P. Deransart. Proofs of partial correctness for attribute grammars with application to recursive procedures and logic programming. *Information and Computation 2*(1988).

[20] P. Deransart. Logical Attribute Grammars. In *Proc. of IFIP 83*, pp. 463-46 North-Holland, 1983.

[21] P. Deransart and M. Jourdan and B. Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*. Springer-Verlag, LNCS 323, 1988.

[22] P. Deransart and J. Małuszyński. *A grammatical view on logic programming*. The MIT Press, 1993.

[23] W. Dembiński and J. Małuszyński. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proc. of the SLP85*, pp. 29-39. The IEEE, 1985.

[24] W. Drabent. *On Completeness of the Inductive Assertion Method for Logic Programs.* Unpublished note, 1988. Can be obtained from wdr@ida.liu.se.

[25] W. Drabent and J. Małuszyński. Induction assertion method for logic programs. *Theoretical Computer Science* 59, pp. 133–155, 1988.

[26] F. Eklund and A. Kågedal. Optimization of GAPLog programs. In: *Proceedings of the Tenth Logic Programming Workshop, WLP94*. Berichte des Instituts für Informatik der Universität Zürich, No. 94.10, 1994.

[27] P. Van Hentenryck, A. Cortesi and B. Le Charlier. Type analysis of Prolog using type graphs. in: *Proc. of the ACM-SIGPLAN Conference on Programming Lnaguage Design and Implementation*, Orlando Fl, June 1994.

[28] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory 2*, pp. 127-145, 1968.

[29] J-L. Lassez, M. Maher and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pp. 587-626, Morgan Kaufmann, Los Altos, 1988.

[30] J. Małuszyński, S. Bonnier, J. Boye, A. Kågedal, F. Kluźniak and U. Nilsson. Logic Programs with External Procedures. In *Logic Programming Languages, Constraints, Functions, and Objects*, pp. 21-48. The MIT Press, 1993.

[31] L. Naish. Adding equations to NU-Prolog. In *Proc. of PLILP'93*, pp. 15–26, LNCS 528, Springer-Verlag, 1991.

[32] J. Paakki. *Multi-pass evaluation of functional logic programs.* Research report LiTH-IDA-R-93-02, Department of computer and information science, Linköping university, 1993, a short version in *Proc. of 21th ACM Symposium on Principles of Programming Languages* , Portland, Oregon, 1994.

[33] D. Pedreschi. A proof method for run-time properties of Prolog programs. In *Proc. of ICLP'94*, pp. 584–598. The MIT Press, 1994.

[34] L. Plümer. *Termination proofs for logic programs.* LNAI 446, Springer Verlag, Berlin, 1990.

[35] Y. Rouzaud and L. Nguyen-Phoung. Integrating modes and types into a Prolog type checker. In *Proc. of JISCLP'92*, pp. 85–97. The MIT Press, 1992.

[36] V.A. Saraswat. *Concurrent Constraint Programming Languages.* The MIT Press, 1990.

[37] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[38] E. Vétillard. *Utilisation de Déclarations en Programmation Logique avec Contraintes*. Ph.D. Thesis. Universite D'Aix-Marseille II, Faculte de Sciences de Luminy, 1994.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399