DIRECTIONAL TYPES AND THE ANNOTATION METHOD

JOHAN BOYE and JAN MAŁUSZYŃSKI

▷ 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. ◁

1. INTRODUCTION

Recently there has been a growing interest in the notion of *directional types* for logic programs [1, 5, 7, 12, 13, 14, 39, 40, 44]. A directional type 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 article shows that directional types have two aspects. One of them is declarative and can be discussed regardless of the computation model, while the other is related to the computation model. By relating directional types to the *annotation*

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

method for proving declarative properties of logic programs [21], we obtain a better correctness criterion than those existing in the literature (e.g. the *well-typing* condition of [5, 14]). We further demonstrate how directional types can be used for controlling execution in a coroutining fashion. We show that programs satisfying our new correctness condition will never suspend indefinitely when executed this way.

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. 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, it is called 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` : (↓list, ↓list, ↑list)

where the two first argument positions (marked with ↓) are considered as *input* positions, and the third argument (marked with ↑) is considered an *output* position.[1] A given directional type may or may not be correct in the sense that it properly describes the actual computational behaviour. As shown in this article, 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 (for the Prolog computation rule): whenever the call of a predicate satisfies the input assertion, then any succeeding call in this computation will satisfy its input assertion.

The directional type in the `append/3` example is correct in both aspects.

The *well-typing* condition of [5, 14] is sufficient to ensure both input-output correctness and call correctness of a given directional type. 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 in Sect. 3 and Sect. 7.

Since input-output correctness is a property which is independent of the computation rule, we can study the problem in the framework of declarative semantics.

[1] A more general concept of directional type (e.g. in [14]) 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`: ↓ (list, list, any) ↑ (list, list, list)

In particular, we show that for types closed under substitution, input-output correctness of a program can be proved by the *annotation method* [21]. It turns out that the well-typing condition of [5, 14] can be seen as a specialization of the annotation method, even though it has been devised for the 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-typing*, or briefly *S-well-typing*.

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. We formally define an execution mechanism, called *type-driven resolution* (or simply *T-resolution*), based on this idea. T-resolution is sound but not complete in general, since the computation may deadlock. We show that S-well-typing is a sufficient condition for deadlock-free execution under T-resolution.

The article 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 [5, 14]. The concepts of input-output correctness and call correctness of a given 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 and shows that the input-output correctness of a directional type is equivalent to the correctness of an annotation corresponding to that type. The concept of S-well-typing is introduced as a specialization of the annotation method, and illustrated 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 Prolog's 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.

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. Directional types

Adopting to a popular view (e.g. Apt [5]), 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
int	the set of integers $(0, 1, \ldots)$
list	the set of lists
bintree	the set of binary trees
intbintree	the set of integer binary trees

In the case of lists, we assume the existence of the constant [ ] (the empty list), and the binary list constructor. Lists will be written using Prolog syntax. In the case of binary trees, we assume the existence of the constant `void` (the empty tree), and the ternary constructor `tree`. The term $\texttt{tree}(s, t_l, t_r)$ represents the tree whose top node is labeled with s, and where t_l and t_r are the left and right subtrees.

A *directional type* for an n-ary predicate `p` is an n-tuple, associating every argument position of `p` with a direction ($\downarrow$ or $\uparrow$) and a type. The argument positions associated with $\downarrow$ ($\uparrow$) are called the *input* (*output*) positions of `p`. For instance,

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

is a directional type for the `append/3` predicate. The two first positions are input positions, and the last position is an output position.

A *directionally typed* program is a program with directional types associated to all its predicates.

A directional type is a kind of specification, which describes certain expected properties of the program. The program may or may not enjoy these properties. This is reflected by the following formal notion of *correctness*. We first need some auxiliary notions.

A *typed term* is an object $t : T$, where t is a term, and T is a type. An atom $p(t_1 : T_1, ..., t_n : T_n)$ is said to be *correctly typed in its i-th position* iff $t_i \in T_i$. For instance, if

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

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

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

(1) all atoms selected in every SLD-derivation of P via R starting from A are correctly typed in their input positions, and if

(2) for every computed answer substitution σ, $\sigma(A)$ is correctly typed in its output positions,

then the directional type is *correct* for P and R. Alternatively, we say that P is *correctly typed* under R. If condition (1) is satisfied, the typing of the program

is said to be *call correct* for R. If condition (2) is satisfied, the typing of the program is said to be *input-output correct* (*IO correct*). Alternatively, we say that P is *correctly IO typed* . □

This definition separates two aspects of correctness. Due to the completeness of SLD-resolution, only the call correctness depends on the computation rule, while the IO correctness can be studied in terms of the declarative semantics. This paper focuses in the first hand on IO correctness, and shows that by abstracting away from the operational aspects, we can obtain simple proofs of IO correctness. Thus, the concept of IO correctness links the view of directional types to the notion of type understood as a restriction of the success set of the program [34, 35].

The problem whether or not a given program is correctly typed under a given computation rule is undecidable. It is then natural to search for sufficient conditions. In the sequel we survey briefly a well-known simple sufficient condition of correct typing for the case of Prolog computation rule. However, for sophisticated computation rules based on coroutining and delays, the behaviour of programs may be rather complex, as shown e.g. by Naish [34]. Therefore it would be rather difficult to provide sufficient conditions for call correctness. On the other hand, as IO correctness does not depend on the computation rule, simple sufficient conditions for IO correctness presented in this paper are applicable even to such programs.

2.2. Well-typing

Definition 2.2. 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 substitutions σ, whenever $\sigma(s_i) \in S_i$ $(1 \leq i \leq n)$, then $\sigma(t) \in T$. If the type judgement is true, we also say that the type of t can be *determined* by the types of $s_1 \ldots s_n$. □

It is undecidable whether a type judgement is true, unless we restrict our type language. However, in all the examples discussed in this paper, it will be obvious whether the judgements are true or not. For a discussion on decidable special cases of type judgements, see e.g. [1, 12].

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

The following is a well-known sufficient condition for a program to be correctly typed under the left-to-right (Prolog) computation rule (cf. [14]):

Definition 2.3. A clause

$$p_0(\mathbf{i_0} : \mathbf{I_0}, \mathbf{o_0} : \mathbf{O_0}) \leftarrow p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$$

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

$$([\,], X) : (\mathsf{list}, \mathsf{list}) \ \Rightarrow \ X : \mathsf{list}$$

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

$$([E|L], R) : (\mathsf{list}, \mathsf{list}) \ \Rightarrow \ (L, R) : (\mathsf{list}, \mathsf{list})$$

and that

$$([E|L], R) : (\mathsf{list}, \mathsf{list}) \quad \wedge \quad LR : \mathsf{list} \ \Rightarrow \ [E|LR] : \mathsf{list}$$

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

The following theorem is stated in [14]. A proof can be found in [7].

Theorem 2.4. Every well-typed program is correctly typed under the Prolog computation rule.

Thus, well-typing is a warranty for correct typing of calls under the Prolog computation rule but not under another computation rule. Consider for example the program

```
parent(john,mary).
parent(mary,ann).
gparent(X,Y) ← parent(X,Z),parent(Z,Y).
```

It is well-typed with the following directional types:

```
parent/2 : (↓ ground, ↑ ground)
gparent/2 : (↓ ground, ↑ ground)
```

Thus, the directional types are call correct under Prolog computation rule, but, for example, they are not call correct for the right-to-left computation rule. On the other hand, whatever is the computation rule, well-typing is a sufficient condition for the IO correctness of the directional types. This is due to the independence of the computed answers of the computation rule used for SLD-resolution.

Later in this paper, we will show that the well-typing criterion is often too weak for proving IO correctness of directional types describing programs that exploit the power of the logical variable.

2.3. Proof trees

We will now summarize a uniform framework for discussing both the operational and the declarative semantics of definite programs. This will allow us to discuss IO correctness without taking into account the computation rule. The framework originates from Deransart and Małuszyński [23], and is based on the notion of proof tree.

In our view, the resolution process can be seen as the stepwise construction of a *skeleton* (by "pasting" together instances of clauses), intertwined with equation solving (unification).

Definition 2.5. A *skeleton* is a finite 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 $A_0 \leftarrow A_1, \ldots, A_k$ 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 $(\sigma(A_1), \bot), ., (\sigma(A_k), \bot)$.

A skeleton with the root label (A, A') will be called a *skeleton for* A. 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. □

The definition may be extended to infinite skeletons. However, the directional types concern finite computations, so that infinite skeletons are not relevant for our purposes.

Definition 2.6. The *set of equations associated to a node* n of a skeleton 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 $E(n) = \{s_1 = t_1, \ldots, s_k = t_k\}$. □

The *set* $E(S)$ *of the equations associated to a skeleton* S consists of all equations associated to the nodes of S. A complete skeleton S is said to be *proper* if $E(S)$ has an mgu. □

The operational semantics of definite programs can be described in terms of skeletons and equations. For example, an LD-resolution[2]step corresponds to choosing the leftmost node n (in preorder of the skeleton), expanding it as described in definition 2.5, and computing a solved form of $E(n)$ (i.e. performing unification). One of the advantages of this view on operational semantics is that we can make

[2]SLD-resolution with the Prolog computation rule.

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.

Definition 2.7. Let S be a skeleton such that $E(S)$ is unifiable with an mgu σ. Then a *proof tree* is obtained from S by replacing every label (A, B) with $\sigma(A)$, for every complete node. □

We give an example to illustrate the introduced concepts.

Example 2.8. Consider the following logic program, that adds and multiplies integers in Peano numeral notation.

$\mathtt{plus(X, 0, X)}.$
$\mathtt{plus(X, s(Y), s(Z))} \leftarrow \mathtt{plus(X, Y, Z)}.$
$\mathtt{mult(X, 0, 0)}.$
$\mathtt{mult(X, s(Y), Z)} \leftarrow \mathtt{plus(W, X, Z), mult(X, Y, W)}.$

Fig. 2.1 shows a complete skeleton constructed from the query

$\mathtt{mult(s(0), s(s(s(0))), X)}$

(for brevity some brackets have been left out). Fig. 2.2 shows the corresponding proof tree. □

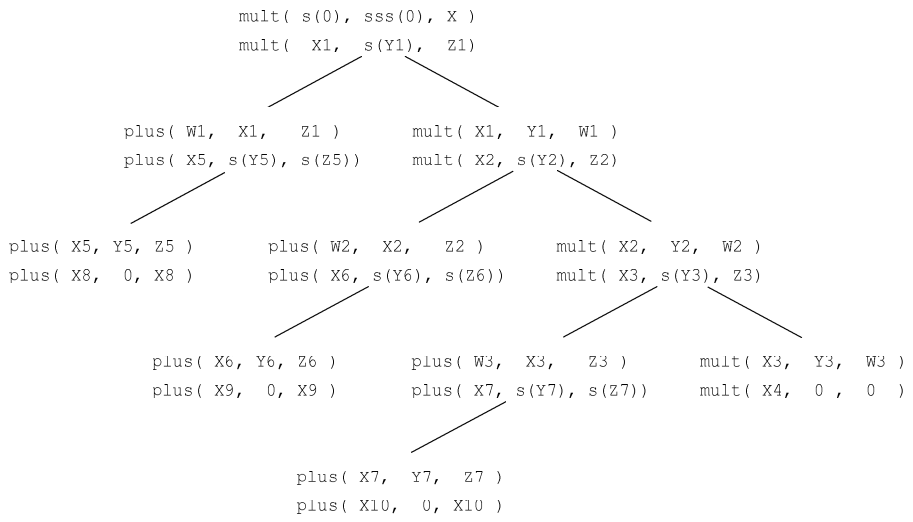


FIGURE 2.1. A complete skeleton

As should be clear from the above example, every SLD-refutation of P (see e.g. Lloyd [30]) starting with an atomic goal $\leftarrow A$ determines a proof tree of P. For a given computation rule there is a one-one correspondence between proof trees and

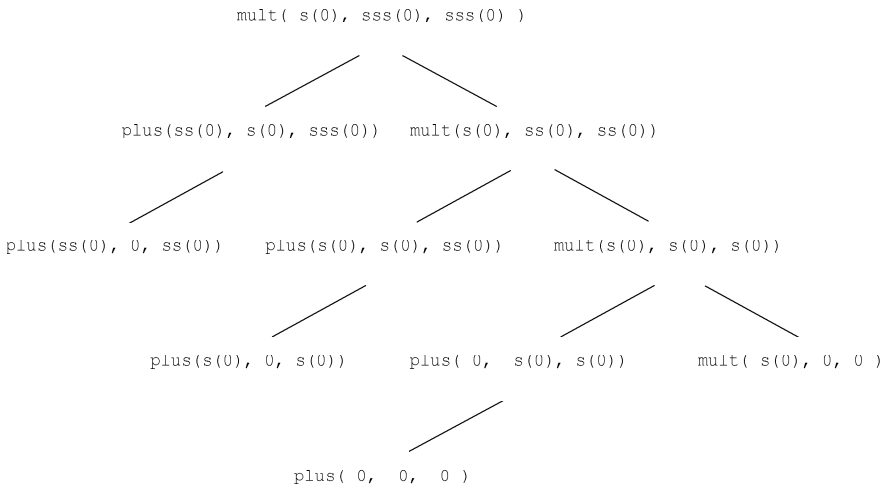


FIGURE 2.2. A proof tree

SLD-refutations starting with atomic goals.

The composition of all mgu's of an SLD-refutation is an mgu of the set of equations associated with the corresponding skeleton. This mgu applied to the labels of the skeleton gives rise to a proof tree. The mgu of the set of equations of a proper skeleton restricted to the variables of the initial query is also the computed answer substitution of the corresponding SLD-refutation.

The declarative semantics of a definite program P is traditionally defined as the set of all ground atomic logical consequences of P, or equivalently as the least Herbrand model of P (see e.g. [30]). A completeness result for SLD-resolution states that this is the set of all ground atomic queries which have SLD-refutations (called also the *success set* of P). It follows by Def. 2.7 that the set of all ground instances of the root labels of all proof trees of P is the success set of P hence the least Herbrand model of P.

On the other hand, some authors consider as declarative any semantics not referring to a notion of computation (or state transition). In this sense the *S-semantics* [27] provides a declarative reconstruction of the operational behaviour of logic programs. The S-semantics of a program P is a set of not-necessarily ground atoms. More precisely, since non-ground atoms are considered equivalent under variable renaming, the elements of the S-semantics are the equivalence classes of such atoms. The S-semantics of a program has a simple characterization in terms of the computed answer substitutions of SLD-refutations. The equivalence class of an atom $A = p(t_1, ..., t_n)$ belongs to the S-semantics of P iff $A = \sigma(B)$, where B is an atomic query of the form $p(V_1, ..., V_n)$ with $V_1, ..., V_n$ being distinct variables, and σ is a computed answer substitution for B. Consequently, since every SLD-refutation gives rise to a proof tree, the S-semantics can be characterized in terms of proof trees. The equivalence class of an atom $A = p(t_1, ..., t_n)$ belongs to the S-semantics of P iff A is the root label of a proof tree obtained from a complete skeleton for an atom of the form $p(V_1, ..., V_n)$ where $V_1, ..., V_n$ are distinct variables.

Yet another declarative semantics of a program P can be defined as the set of

root labels of all proof trees of P. In [23] it is called the *proof-theoretic semantics* of P and it is denoted $\mathcal{PT}_P$.

Notice that this set may include non-ground atoms, and that it is closed under substitution. The proof-theoretic semantics has a straightforward relation to the S-semantics. Intuitively, $\mathcal{PT}_P$ is a closure of the S-semantics of P under arbitrary substitutions. More precisely, an atom A is in $\mathcal{PT}_P$ iff there exists a substitution σ and an atom B such that $A = \sigma(B)$ and the equivalence class of B is in the S-semantics of P.

The proof theoretic semantics describes all atomic logical consequences of the program P: the universal closure of each atom in $\mathcal{PT}_P$ is a logical consequence of P. A more comprehensive discussion including proofs can be found in Chapter 2 of [23].

2.4. Dependencies

Intuitively, a directional type describes the data flow from the inputs to the outputs of a predicate. Viewing a computation as the construction of a (proper) skeleton, it is possible to discuss the data flow over the positions of skeletons. This section presents abstract notions which can be used for formalization of this intuition, and which can provide a basis for deriving our new well-typing condition.

For the rest of this section, we assume that we have some unambiguous way of referring to the atoms in the program. 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 2.9. 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, when proving well-typing, the terms occurring in the consequents of the type judgements always occur at output positions in the head, or at input positions of the body. For convenience, we introduce a name for these positions:

Definition 2.10. $A(i)$ 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 extend this terminology for the positions of the initial atomic queries: a query G is seen as a clause $\leftarrow G$ with the empty head.

Definition 2.11. Let C be a clause (possibly the initial query). A binary relation,

relating some importing clause positions of C to some exporting clause positions of C, will be called a *local dependency relation* for C. □

In the sequel, we will assume that each clause C has a fixed local dependency relation $\triangleright_C$.

A skeleton is obtained by pasting together instances of clauses. To model the dataflow in a complete skeleton S, we construct a *compound dependency graph* $\triangleright_S$ by pasting together the local dependency graphs for the clauses used in S. More precisely:

Definition 2.12. Let S be a complete skeleton, and let n be a node in S, labeled with $(p(s_1,\ldots,s_k), p(t_1,\ldots,t_k))$. 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

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

if one of the following cases apply:

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

□

Depending on the intended use, the local dependency relation may show the possible flow of data between importing positions and exporting positions, or some other dependence between these positions. The intuition of the $\triangleright_S$ relation is to extend this idea to complete skeletons. Figure 2.3 shows a possible choice of local dependency relations on the clauses of the **append** program, and the global dependency relation on a skeleton of this program.

The equations of a node can be seen as connections transmitting values from exporting positions of one clause to importing positions of the other clause. If the transitive closure of $\triangleright_S$ is an ordering relation, the instantiation of its maximal elements can be expressed as a function of the instantiation of its minimal elements. This idea is a basis of *type-driven* resolution introduced in Sect. 6 where the equations of the skeleton are solved in accordance with the relation $\triangleright_S$. This motivates us to introduce the following concept.

Definition 2.13. Let P be a definite program including an initial query. If for every complete skeleton S of P the transitive closure of the relation $\triangleright_S$ is a partial ordering, then the program is said to be *non-circular*. □

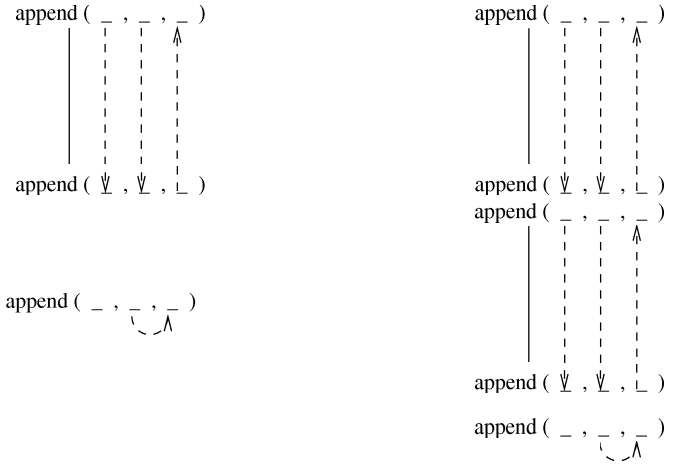


FIGURE 2.3. The local dependency relations and a global dependency relation

The non-circularity concept stems originally from the field of attribute grammars [29]. It is well-known that this property is decidable (see e.g. [23, Sect. 4.3]). For the sake of completeness we give a brief justification of this result, using the terminology of this paper (instead of attribute grammar terminology).

Let G be a query with an empty local dependency relation. For any complete skeleton S for G denote by $\triangleright_{S,p}$ the relation on the positions of p defined as follows: $p(i) \triangleright_{S,p} p(j)$ iff $r(i) \triangleright_S^* r(j)$, where r is the root of S. Thus we "project" the global dependency relation of the skeleton S on the positions of the root predicate p. Generally, there are infinitely many distinct skeletons S, but the family of distinct relations $\triangleright_{S,p}$ is finite, since p has a finite number of positions. Denote this family $\mathcal{D}_p$. Consider a clause C of the form

$$p(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), \ldots, p_m(\mathbf{t_m})$$

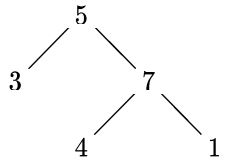
where $\mathbf{t}, \mathbf{t_1}, \ldots, \mathbf{t_n}$ are sequences of terms. For $k = 1, ..., m$ select an arbitrary relation r_k in $\mathcal{D}_{p_k}$. Let R_C be the transitive closure of the relation obtained by the composition of $\triangleright_C$ with the relations r_k. Then the restriction of R_C to the positions of the head predicate is a relation in $\mathcal{D}_p$. This shows that the family $\{\mathcal{D}_p : p$ is a predicate of $P\}$ has a fixpoint characterization, and due to its finiteness can be effectively computed.

The program is non-circular iff no relation R_C has a cycle. This is due to the fact that the relations in $\mathcal{D}_p$ give the exact characterization of the global dependencies between the positions of p in any skeleton. The uppermost segment of a loop in a skeleton would be determined by the local dependency relation of a clause C. Thus for checking non-circularity we have to construct all relations R_C for every clause C. This may lead to combinatorial explosion since each family $\mathcal{D}_p$ may have many elements. It has been shown (see e.g. [28]) that the circularity problem is of exponential complexity. In practice non-circularity can usually be established by sufficient tests with polynomial (or even linear) complexity (see [22] for a survey).

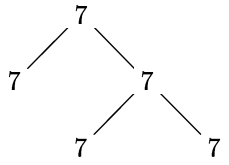
3. AN INFORMAL EXAMPLE

In this section, we give an example of a program which is correctly IO typed but not well-typed. We claim that such directional types often are of practical interest, especially for programs using incomplete data structures. 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 expect 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 program in Fig. 3.1 solves the problem in one pass.

```
maxtree(Tree, NewTree) ←
    maxtree(Tree, Max, Max, NewTree).

maxtree(void, _, 0, void).
maxtree(tree(Lbl, Lft, Rgt), Max, MaxSoFar, tree(Max, NewLft, NewRgt)) ←
    maxtree(Lft, Max, MaxLft, NewLft),
    maxtree(Rgt, Max, MaxRgt, NewRgt),
    max(Lbl, MaxLft, MaxRgt, MaxSoFar).

max(A, B, C, A) ← A ≥ B, A ≥ C.
max(A, B, C, B) ← B ≥ A, B ≥ C.
max(A, B, C, C) ← C ≥ A, C ≥ B.
```

FIGURE 3.1. A program solving the maximum-labeling problem

The predicate `max/4` computes the maximum of its three first arguments. The predicate `maxtree/4` traverses the input tree, and finds the maximal label in the tree. It also constructs the output tree, in which all nodes are labeled with the same logical variable. Upon success of `maxtree/4`, this logical variable is unified with the maximal label.

Note that upon success of the intermediate calls to `maxtree/4`, the fourth argument is bound to a non-ground binary tree. Thus the most precise correct direc-

tional type for this program (using the types in Example 2.1) is:

$\texttt{maxtree}/2 : (\downarrow \mathsf{intbintree}, \uparrow \mathsf{intbintree})$
$\texttt{maxtree}/4 : (\downarrow \mathsf{intbintree}, \downarrow \mathsf{any}, \uparrow \mathsf{int}, \uparrow \mathsf{bintree})$
$\texttt{max}/4 : \quad (\downarrow \mathsf{int}, \downarrow \mathsf{int}, \downarrow \mathsf{int}, \uparrow \mathsf{int})$
$\geq: \quad (\downarrow \mathsf{int}, \downarrow \mathsf{int})$

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

$$\begin{array}{r} \texttt{Tree} : \mathsf{intbintree} \ \wedge \\ (\texttt{Max} : \mathsf{int}, \ \texttt{NewTree} : \mathsf{bintree}) \Rightarrow \\ \texttt{NewTree} : \mathsf{intbintree} \end{array}$$

is not true. The problem is caused by the variable `NewTree`: one cannot conclude that `NewTree` is an integer binary tree just from the fact that it is a binary tree. Thus we cannot use the well-typing condition to conclude that the directional type is IO correct.

Now consider changing the directional type for `maxtree/4` as follows (the other predicates are typed as before):

$\texttt{maxtree}/4 : (\downarrow \mathsf{intbintree}, \downarrow \mathsf{int}, \uparrow \mathsf{int}, \uparrow \mathsf{intbintree})$

The idea is that if `maxtree/4` is called with its second argument bound to an integer, then the last argument will be bound to an integer binary tree upon success. Now the directional type for the program as a whole is not call correct under the Prolog computation rule; the `maxtree/4` predicate is called with the second argument being a variable, not an integer. However, the directional type remains IO correct, as will be shown by the method presented in the next section.

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 [18] and presented more recently in [21, 23], 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 for an n-ary predicate p will be alternatively represented as a pair of formulae

$$\begin{array}{l} \langle \ \psi_1(p_1) \wedge \ldots \wedge \psi_n(p_n), \\ \ \ \psi'_1(p_1) \wedge \ldots \wedge \psi'_n(p_n) \ \ \rangle \end{array}$$

where $p_1, \ldots, p_n$ are the only free variables of the formulae, referring to the argument positions of the predicate. The formulae ψ and ψ' are defined as follows: If the i-th argument of p is an input argument of type T, then $\psi_i = \psi'_i = T$. If the i-th argument of p is an output argument of type T, then $\psi_i = \mathsf{any}$ and $\psi'_i = T$. For example the directional type

append/3: ($\uparrow$ list, $\uparrow$ list, $\downarrow$ list)

will be represented as

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

Here any and list are unary predicates of the specification language, interpreted on the domain of terms. Thus, a directional type $D = \langle \psi, \psi' \rangle$ for the predicate p specifies two sets of atoms, $S_1(D)$ and $S_2(D)$, consisting of the elements of the form $p(t_1, \ldots, t_n)$, which satisfy, respectively, ψ and ψ' in the interpretation considered. For example, for the directional type D above, $S_1(D)$ 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.
- for each i the i-th assertion of the first formula is either $\mathsf{any}(p_i)$, 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 $\{p_1, \ldots, p_n\}$. 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 $\mathcal{I}$ is an interpretation on a term domain and we consider an annotation Δ for p to be a specification of two sets of atoms, denoted $S_1(\Delta)$ and $S_2(\Delta)$. These sets are defined as follows.

Let ψ be an assertion in Δ, and let $t_1, ..., t_n$ be terms. Denote by $\psi[p_1/t_1, ..., p_n/t_n]$ the formula obtained from ψ by replacement of all occurrences of each variable p_i by the term t_i, for $i = 1, ..., n$. The set $S_1(\Delta)$ consists of all atoms $p(t_1, ..., t_n)$ such that for every $\psi \in Inh$ the formula $\psi(p_1/t_1, ..., p_n/t_n)$ is true in $\mathcal{I}$. Similarly, the set $S_2(\Delta)$ consists of all atoms $p(t_1, ..., t_n)$ such that for every $\psi \in Syn$ the formula $\psi(p_1/t_1, ..., p_n/t_n)$ is true in $\mathcal{I}$.

An annotation Δ is said to be *closed under substitution* iff for every term t in $S_i(\Delta)$ $(i = 1, 2)$ and for every substitution σ, $\sigma(t)$ is in $S_i(\Delta)$. For example, consider the following annotation Δ of append/3:

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

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:

- $S_1(\Delta)$ 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,
- $S_2(\Delta)$ 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.

This annotation is closed under substitution, but it is not a directional type.

4.2. Program annotations

We will now discuss the use of annotations for specification of logic programs. An annotation Δ for a program P is any pair $\langle Inh, Syn \rangle$ where Inh and Syn are finite sets of assertions for the predicates of the program.

For a given interpretation $\mathcal{I}$ a given program annotation specifies, as described above, two sets of atoms $S_1(\Delta)$ and $S_2(\Delta)$. To establish correctness of a program P with respect to such a specification we will compare some semantics of P with these sets of atoms. We will consider two kinds of semantics: the input-output semantics defined below, and the proof-theoretic semantics $\mathcal{PT}_P$. The former will be used to extend to arbitrary annotations the notion of input-output correctness, defined for directional types in Sect. 2.1. The latter is essentially the semantics used in the annotation method. We will show that both kinds of semantics coincide in the case of the annotations closed under substitution.

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 $\sigma(g)$ such that g is in I, and σ is a computed answer substitution for the goal $\leftarrow g$ under the SLD-resolution.

An annotation Δ is *input-output correct* for P iff $\mathcal{IO}_P(S_1(\Delta)) \subseteq S_2(\Delta)$. □

For example the annotation Δ of Sect. 4.1 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 Sect. 2.1.

We will now relate the annotations to the proof-theoretic semantics. Recall that the proof-theoretic semantics $\mathcal{PT}_P$ of a program P is the set of the root labels of all proof trees of P. An annotation may be used to state a property of a subset of $\mathcal{PT}_P$. In that case, the inherited assertions of the annotation specify the subset of $\mathcal{PT}_P$, 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

$$(\mathcal{PT}_P \cap S_1(\Delta)) \subseteq S_2(\Delta)$$

□

For example, the annotation Δ of Sect. 4.1 is not only input-output correct but also success correct and describes an interesting property of the proof-theoretic semantics of the `append` program.

On the other hand, consider the annotation

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

where var is a unary predicate of the metalanguage L, such that $\mathsf{var}(t)$ is true in $\mathcal{I}$ whenever t is a variable. The annotation 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. Assume now that nat is the set of terms representing natural numbers: $\{zero, s(zero), \ldots\}$. 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 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 $S_1(\Delta)$ which is the root label of a proof tree and which does not belong to $S_2(\Delta)$. 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 $S_1(\Delta)$ such that a substitution σ is a computed answer substitution for $\leftarrow g$, but $\sigma(g)$ is not in $S_2(\Delta)$. The SLD-refutation producing σ corresponds to a proper skeleton. Thus $\sigma(g)$ is the root label of a proof tree. As $S_1(\Delta)$ is closed under substitution it includes $\sigma(g)$. 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 $S_1(\Delta)$.

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 input-output correctness of directional types.

4.3. The Annotation Method

We will now briefly survey a method for proving success correctness of annotations introduced in [20] for attribute grammars and adapted for the case of logic programs in [18]. The method is called *the annotation method.* More recent presentations can be found in [21] and in [23].

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 $p(t_1, \ldots, t_n)$ is in $S_1(\Delta)$ then it is also in $S_2(\Delta)$. This corresponds to checking validity of an implication of the form

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

where the ψ_is (ψ'_is) are the inherited (synthesized) assertions of p in Δ, instantiated by the substitution $\{p_1/t_1, \ldots, p_n/t_n\}$.

The idea of the annotation method is to consider the structure of the proof trees. Each proof tree is constructed from instances of the clauses of the program, where the body atoms of a clause 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 each of the body atoms is an implication constructed as discussed above. The interesting case is when the antecedent of the implication holds, since otherwise the implication holds trivially.

We now introduce the following terminology: the antecedent of the head implication and the consequents of body implications are called the *premise* 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 each conclusion assertion follows from some premise assertions of the clause.

As an example consider the following annotation for the `append` program:

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

Then for the clause

$$\texttt{append}([\texttt{A}|\texttt{X}], \texttt{Y}, [\texttt{A}|\texttt{Z}]) \leftarrow \texttt{append}(\texttt{X}, \texttt{Y}, \texttt{Z}).$$

there will be two instances of the annotation:

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

and

$$\langle\ \{\mathsf{list}(X), \mathsf{list}(Y)\},\ \{\mathsf{list}(Z)\}\qquad\rangle$$

The premise assertions are:

$$\text{(a)}\ \mathsf{list}([A|X]),\ \text{(b)}\ \mathsf{list}(Y),\ \text{(c)}\ \mathsf{list}(Z)$$

The remaining assertions are the conclusion assertions:

$$(1)\ \mathsf{list}([A|Z]),\ (2)\ \mathsf{list}(X),\ (3)\ \mathsf{list}(Y)$$

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

In our example

(1) follows from (c) $\mathcal{I} \models \forall A, Z(\mathsf{list}(Z) \Rightarrow \mathsf{list}([A|Z])$
(2) follows from (a) $\mathcal{I} \models \forall A, X(\mathsf{list}([A|X]) \Rightarrow \mathsf{list}(X))$
(3) follows from (b) $\mathcal{I} \models \forall Y(\mathsf{list}(Y) \Rightarrow \mathsf{list}(Y))$

If the verification conditions hold for a clause, they hold also for all instances of the clause. One could expect that if the root label of a proof tree satisfies its inherited assertions, then every node of the tree satisfies all its assertions. For example, consider the following proof tree of the `append` program:

```
append([A],[Y],[A,Y])
                |
append([],[Y], [Y])
```

The root label satisfies the inherited assertions of append: list($[A]$) and list($[Y]$. The constituent clause instances in the tree are:

append([A|[ ]], [Y], [A|[Y]]) ← append([ ], [Y], [Y]).
append([ ], [Y], [Y]).

They give rise to the following instances of the verification conditions:

$\mathcal{I} \models \forall A, Y\ \mathsf{list}([Y]) \Rightarrow \mathsf{list}([A|Y])$
$\mathcal{I} \models \forall A\ \mathsf{list}([A|[\,]]) \Rightarrow \mathsf{list}([\,])$
$\mathcal{I} \models \forall Y\ \mathsf{list}([Y]) \Rightarrow \mathsf{list}([Y])$

There is, however, a danger that a 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:

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

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:

$$\mathcal{I} \models \forall X(\mathsf{ground}(X) \Rightarrow \mathsf{ground}(X))$$

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 premise assertions. However, in the only proof tree of this program, the combination of the verification conditions of its clauses gives the statement

$$\forall X(\mathsf{ground}(X) \Leftrightarrow \mathsf{ground}(X))$$

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 premise assertions of this clause. We say that the conclusion assertion *depends* on these premise 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 premises contains the assertions list($[A|X]$) and list($[B|X]$), then the conclusion list(X) may be obtained by each of the premises.

Generally we may consider an arbitrary relation between the premises and the conclusions of a clause, which may or may not properly indicate the premises 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. Figure 4.1 shows an LDS for the example annotation of Sect. 4.1.

Any proof tree is obtained by pasting together instances of the clauses. Hence,

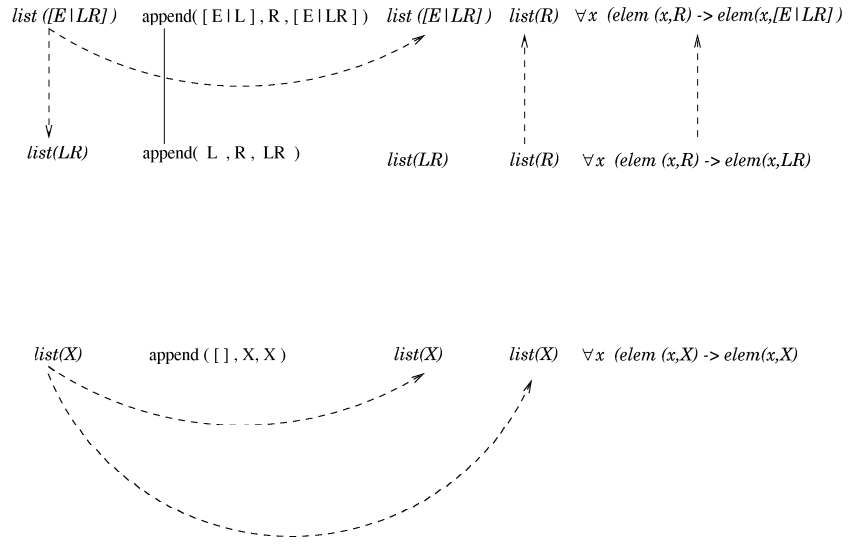


FIGURE 4.1. A logical dependency scheme

by pasting together the copies of the dependency graphs of the clauses we obtain the dependency graph of the skeleton. The mechanism is identical to that described by the definition of the relation $\triangleright_T$ of Sect. 2.4, 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. As discussed in Sect. 2.4, this property is decidable and has been studied in the context of attribute grammars. The well-known techniques for checking non-circularity of attribute grammars apply directly to LDS's.

Intuitively, an LDS can be seen as a plan for proving success correctness of an annotation. For each clause the LDS shows the premise 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* with respect to a given LDS iff the verification condition induced by the LDS hold in every clause. In other words, each conclusion assertion of a clause should be implied by the premise assertions on which it depends in the LDS.

The following theorem gives a sufficient condition for success correctness of an annotation. It captures the essence of the annotation method: for proving success correctness of an annotation Δ one has to find an annotation Δ', possibly "strengthening" Δ, and an LDS which allows to verify Δ. For a more comprehensive discussion of the annotation method the reader is referred to [23], where Theorem 7.2 p.353 states not only the soundness but also the relative completeness of the method. In this paper we focus on sufficient conditions for correctness of directional types. Therefore we do not discuss the completeness issue.

Theorem 4.5. Let $\Delta = (Inh, Syn)$ be an annotation for a program P. If there exist

an annotation $\Delta' = (Inh', Syn')$ for P and a non-circular LDS such that

- Δ' is sound wrt the LDS,
- $S_1(\Delta) = S_1(\Delta')$ and $S_2(\Delta') \subseteq S_2(\Delta)$

then Δ is success-correct for P.

Proof : Assume that there exist Δ' and an LDS for P and Δ' with the required properties. Let T be a proof tree of P, such that its root label r is in $S_1(\Delta)$. We have to show that it is also in $S_2(\Delta)$. For this, it suffices to show that r is in $S_2(\Delta')$.

Since $S_1(\Delta) = S_1(\Delta')$, then r is in $S_1(\Delta')$. Consider now the dependency graph on T determined by the LDS. The nodes of the graph correspond to the instances of the assertions of Δ', called in the sequel the *assertions* of T. Due to the non-circularity assumption the dependency relation on the assertions of T is a partial ordering. The minimal elements of this ordering are those which do not depend on any other assertions of the tree, in particular the inherited assertions of the root. The inherited assertions of the root hold since r is in $S_1(\Delta')$. The remaining minimal assertions hold by the assumption that the LDS is sound. Consequently, by soundness and non-circularity of the LDS all other assertions of T must also hold. Thus r is in $S_2(\Delta')$, hence also in $S_2(\Delta)$. □

The annotation Δ' plays the role of a 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 strengthening the conditions of the annotation method. The first step in that direction is to restrict the attention to the cases when a given annotation Δ is sufficient to achieve the proof, so that no additional Δ' is needed. A further simplification consists in assigning a particular LDS to each program and annotation. This LDS may or may not be sound. Thus, the application of theorem 4.5 reduces to checking whether for a 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 verification of directional types.

In the particular case of directional types, there is a one-one correspondence between the assertions of the annotation and the positions of the predicates. Thus, the premise 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 [5, 14] (see also Sect. 2.1) requires that for every clause the 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. [22]) 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 will 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.

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 premise 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 premises whose corresponding positions share variables with p. This suggestion is justified by the observation that for every valuation for which all these premise assertions are satisfied, the logical values of the remaining premises are irrelevant for the satisfaction of ψ.

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. For each clause C, including the initial query, its *S-dependency relation* $\rightsquigarrow_C$ is a binary relation on the positions of C, defined as follows:

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

iff $A(i)$ is an importing clause position in C, $B(j)$ is an exporting clause position in C, and $A(i)$ and $B(j)$ have at least one common variable. □

The relation $\rightsquigarrow_C$ is obviously a special case of the local dependency relation $\triangleright_C$ (Def. 2.11). Thus for any skeleton T, the S-dependency relations induce a compound dependency relation, as explained in Def. 2.12. This compound dependency relation will be denoted $\rightsquigarrow_T$.

Definition 4.7. Let P be a directionally typed program, and let C be a clause of P.

- For a given exporting position e in C:
  - — let t be the term filling in e, and let T be the type associated to e.
  - — 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 at these positions, typed $T_1, ..., T_n$, respectively.

  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 and all its clauses are S-well-typed.

□

From this definition we obtain at once the following result.

Theorem 4.8. Every S-well-typed program is correctly IO typed.

Proof : As already discussed, the directional types of the program predicates can be seen as an annotation of the program. The relations $\rightsquigarrow_C$ provide an LDS for this annotation. The S-well-typedness condition is a rephrasing of the soundness and non-circularity requirement for this LDS. Hence, by theorem 4.5 the annotation is success correct. As the directional types are closed under substitution, by theorem 4.4 the annotation is also IO correct. □

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

`maxtree/2` : ($\downarrow$ intbintree, $\uparrow$ intbintree)
`maxtree/4` : ($\downarrow$ intbintree, $\downarrow$ int, $\uparrow$ int, $\uparrow$ intbintree)
`max/4` : ($\downarrow$ int, $\downarrow$ int, $\downarrow$ int, $\uparrow$ int)
$\geq$: ($\downarrow$ int, $\downarrow$ int)

As discussed in Sect. 3 the program is not well-typed with this directional type. We now show that it is S-well-typed, and 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.

We will now show that the program is S-well-typed with this directional type. First consider the clause defining `maxtree/2`. There are three exporting positions in this clause; consequently we have to prove three type judgements, each of which is trivial:

`Tree` : intbintree $\Rightarrow$ `Tree` : intbintree
`Max` : int $\Rightarrow$ `Max` : int
`NewTree` : intbintree $\Rightarrow$ `NewTree` : intbintree

In the recursive clause for `maxtree/4`, we have to prove nine type judgements. Four of these are non-trivial, namely:

`tree(Lbl,Lft,Rgt)` : intbintree $\Rightarrow$ `Lft` : intbintree
`tree(Lbl,Lft,Rgt)` : intbintree $\Rightarrow$ `Rgt` : intbintree
`tree(Lbl,Lft,Rgt)` : intbintree $\Rightarrow$ `Lbl` : int
`Max` : int $\wedge$ `NewLft` : intbintree $\wedge$ `NewRgt` : intbintree $\Rightarrow$
`tree(Max,NewLft,NewRgt)` : intbintree

It is easy to see that these four type judgements are all true. In the unit clause for `maxtree/4` there are no type judgements to prove, and in the clauses for `max` there are only trivial type judgements to prove.

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. [22].

5. CALL CORRECTNESS UNDER THE PROLOG COMPUTATION RULE

We will 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:

`maxtree/2` : ($\downarrow$ intbintree, $\uparrow$ intbintree)
`maxtree/4` : ($\downarrow$ intbintree, $\downarrow$ int, $\uparrow$ int, $\uparrow$ intbintree)
`max/4` : ($\downarrow$ int, $\downarrow$ int, $\downarrow$ int, $\uparrow$ int)
$\geq$: ($\downarrow$ int, $\downarrow$ int)

When executed with the Prolog computation rule, in every call to the recursive clause for `maxtree/4`, the first position (but not the second) is 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 the Prolog computation rule. 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 $H \leftarrow A_1, \ldots, A_n$. If $A_j(i) \in \mathcal{W}$, then its type can be determined 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 $H \leftarrow A_1, \ldots, A_n$. If $H(i) \in \mathcal{W}$, then its type can be determined 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 $\mathcal{W}$. □

□

Lemma 5.2. Given a program with a directional type, there exists a largest set of well-typed clause positions.

Proof : 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. □

Definition 5.3. A clause position of P is called *well-typed* if it belongs to the largest set of well-typed positions. 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 5.3 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 determined from the types of the importing clause positions in the head.

Now consider the recursive clause for `maxtree/4`. 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 in Fig. 5.1.

```
maxtree(Tree, NewTree) ←
    maxtree(Tree, Max, Max, NewTree).

maxtree(void, _, 0, void).
maxtree(tree(Lbl, Lft, Rgt), Max, MaxSoFar, tree(Max, NewLft, NewRgt)) ←
    maxtree(Lft, Max, MaxLft, NewLft),
    maxtree(Rgt, Max, MaxRgt, NewRgt),
    max(Lbl, MaxLft, MaxRgt, MaxSoFar).

max(A, B, C, A) ← A ≥ B, A ≥ C.
max(A, B, C, B) ← B ≥ A, B ≥ C.
max(A, B, C, C) ← C ≥ A, C ≥ B.
```

FIGURE 5.1. The well-typed positions of the `maxtree` program

We conclude that the first argument of `maxtree/2`, the first and third arguments of `maxtree/4`, and the first argument of `max/2` are well-typed.

Definition 5.4. Given a directional type $\mathcal{T}$ of a program P, we obtain $\mathcal{T}_W$, the *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` : ($\downarrow$ intbintree, $\uparrow$ any)
`maxtree/4` : ($\downarrow$ intbintree, $\downarrow$ any, $\uparrow$ int, $\uparrow$ any)
`max/4` : ($\downarrow$ int, $\downarrow$ int, $\downarrow$ int, $\uparrow$ int)
$\geq$: ($\downarrow$ int, $\downarrow$ int)

Theorem 5.5. 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 $H \leftarrow A_1, \ldots A_n$, 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 2.3 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 5.6. Let P be a directionally typed 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 5.7. Let P be a directionally typed program, and let G be an atom which is correctly typed in its input positions. Let $H \leftarrow A_1, \ldots, A_n$ be a clause in P. Then in every Prolog 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. This raises the question which input arguments of a predicate fall

in this category for a given computation rule. The concept of well-typed position gives a sufficient condition identifying such arguments for the Prolog computation rule.

6. TYPE-DRIVEN RESOLUTION

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 idea is to suspend unification when the arguments are not correctly typed. In contrast to some Prolog systems (e.g. SICStus [15]), the suspension is argument-wise rather than atom-wise.

An interesting question is whether the computation may reach the deadlock situation where no resolution can be performed, even though the set of the suspended unifications is not empty. We show that S-well-typedness is a sufficient condition for a program to be deadlock-free under T-resolution.

The argument-wise suspension can be simulated by the atom-wise suspensions, by rewriting the program into a "flattened" form, where unification is explicitly expressed by equation atoms. Such a transformation can be seen as a way of implementing T-resolution in a Prolog system with atom-wise delays. However, we believe that T-resolution is a natural concept as concerns enforcing declared directional types during the execution of the program. This is confirmed by the fact that the notion of S-well-typedness has a direct application also in this case.

6.1. T-resolution

We first introduce some auxiliary concepts. In what follows, we assume that the i-th argument position of a predicate p is always classified as an input or as an output, and has the associated type T_i.

Definition 6.1. A *query* is 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. A *directionally typed term* is a term tagged with a direction and a type. A directionally typed equation is an equation of two terms with the same tag. □

For instance, $\mathtt{X} : \downarrow \mathsf{int}$ and $[1, 2] : \uparrow \mathsf{intlist}$ are two examples of directionally typed terms.

Definition 6.3. A directionally typed equation $s = t$ is *eligible* iff either both s and t are tagged $\downarrow T$ and $s \in T$, or both s and t are tagged $\uparrow T$ and $t \in T$. □

Definition 6.4. 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 an eligible equation $s = t$, and s and t unify with mgu σ, then $Q' \equiv (\sigma(G), \sigma(E))$. If s and t do not unify then $Q' \equiv$ **fail**.

3. Otherwise, if E contains an equation $p(s_1, \ldots, s_n) = p(t_1, \ldots, t_n)$, where p is a predicate symbol, then

$$Q' \equiv (G; (E - \{p(s_1, \ldots, s_n) = p(t_1, \ldots, t_n)\} \cup \{s_1 = t_1, \ldots, s_n = t_n\})$$

4. Otherwise, if $G \equiv A_1, \ldots, A_k$, where $A_1 \equiv p(s_1, \ldots, s_n)$, and $H \leftarrow B_1, \ldots B_n$ is a (renamed) clause, where $H \equiv p(t_1, \ldots, t_n)$,then

$$Q' \equiv (B_1, \ldots, B_m, A_2, \ldots A_k \ ; \ E \cup \{p(s_1, \ldots, p_n) = p(t_1, \ldots, t_n)\})$$

5. Otherwise, Q has no T-derivative.

□

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

Definition 6.5. 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 with 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 SLD-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 6.6 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.

We illustrate this resolution process on an example. Recall the `maxtree` program of Sect. 3, with the directional type of Sect. 4.4. Consider the initial T-query

$$(\texttt{maxtree}(\texttt{tree}(5, \texttt{void}, \texttt{void}), \texttt{NewTree}); \{\})$$

It has only one T-derivative, which through the steps (4), (3), (2) and (1) of Definition 6.5 reduces to the query:

$$(\texttt{maxtree}(\texttt{tree}(5, \texttt{void}, \texttt{void}), \texttt{Max1}, \texttt{Max1}, \texttt{NewTree1});\ \{\texttt{NewTree} = \texttt{NewTree1}\})$$

the equation $\texttt{NewTree} = \texttt{NewTree1}$ is left unsolved since `NewTree1` is not yet correctly typed. We can depict this situation with the incomplete proof tree shown in figure 6.1. (Two terms stacked upon each other indicate an unsolved equation.)

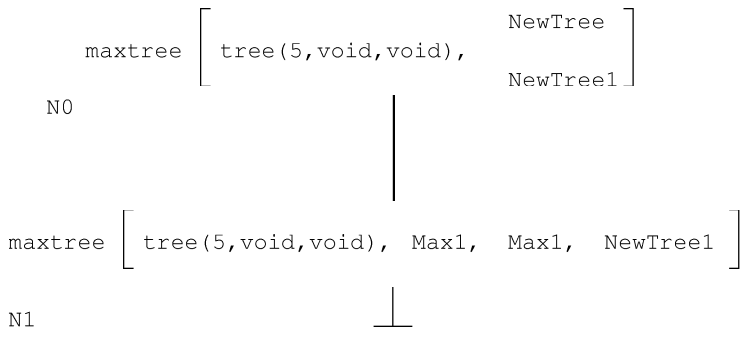


FIGURE 6.1. An incomplete proof tree

We proceed by resolving the only atom in the query (thus expanding the node N_1). Some T-derivation steps later we obtain the incomplete proof tree depicted in figure 6.2.

When we now resolve the atom $\texttt{max}(5, 0, 0, \texttt{WidSoFar2})$ at node N_2, the variable `WidSoFar2` gets instantiated to 5. We can now solve the equation

$$\texttt{Max1} = \texttt{WidSoFar2}$$

at node N_1, since `WidSoFar2` is intantiated to a correctly typed term (of type int). We can then solve (in the following order) the equations

$$\texttt{Max1} = \texttt{Max2},\ \texttt{tree}(\texttt{Max2}, \texttt{void}, \texttt{void}) = \texttt{NewTree1}$$

(at node N_1), and finally the equation

$$\texttt{NewTree} = \texttt{NewTree1}$$

at node N_0. The complete proof tree is depicted in Fig. 6.3.

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.

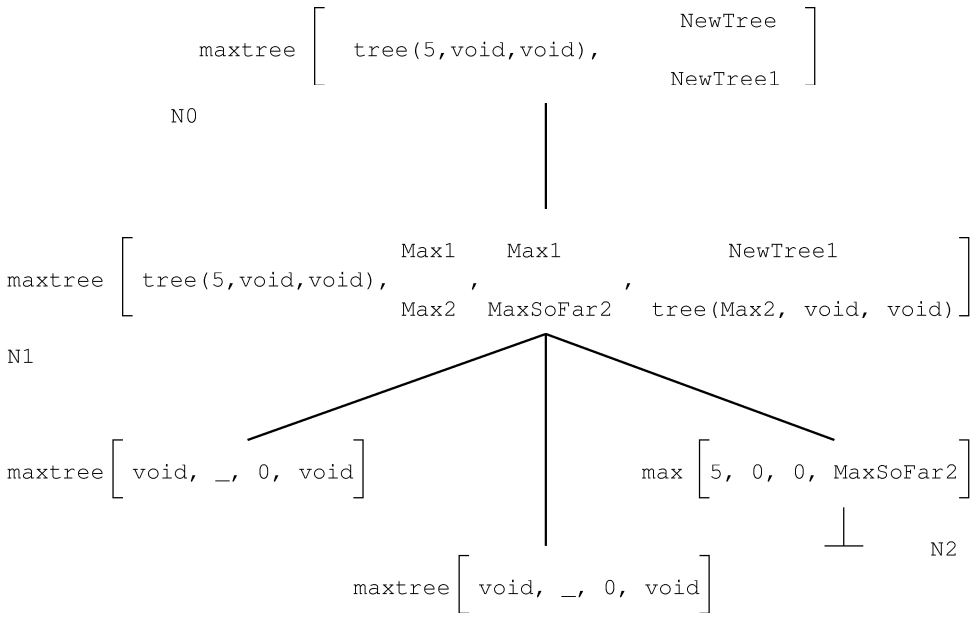


FIGURE 6.2. Another incomplete proof tree

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 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 c does not depend on any importing position of D. In case (1), by assumption the input positions of the initial query are correctly typed, so these equations are eligible. In case (2), the type judgement for c has an empty premise; hence c is correctly typed, and the equation $c = t$ is eligible.

(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. Assume that at some step of the computation all not yet solved equations at the positions in $dep(n_i)$ are eligible. If any of them has no solution, we have reached **fail**. Otherwise all of these equations can be solved. Let σ be their mgu. Since the program is S-well-typed, $\sigma(s)$ is correctly typed, and the equation $\sigma(s) = \sigma(t)$ is eligible. □

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

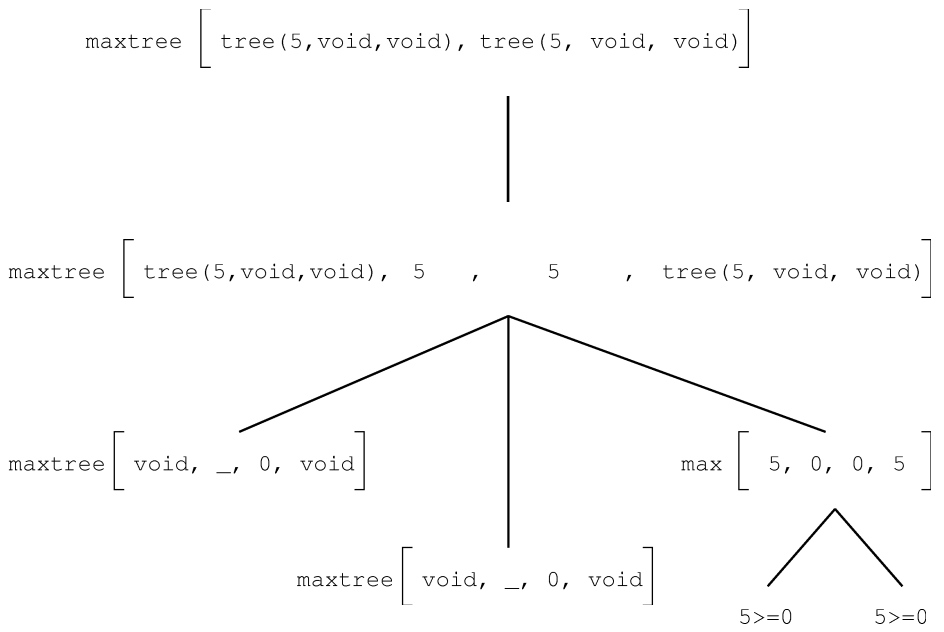


FIGURE 6.3. A complete proof tree

Theorem 6.7. 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, $\sigma(g)$ is correctly typed in its output positions.

Proof : As T-resolution is sound, the result follows by Theorem 4.8. □

The use of directional types for control provides a synchronization method for concurrent logic programming. Here T-resolution seems to be better suited than an atom-wise delaying strategy. Consider for instance the program shown in Fig. 6.4, modeling a producer-consumer process. A producer freely produces some items i which are to be consumed by a consumer. The consumer writes a confirmation c for every consumed item, and 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.

The producer and the consumer are binary predicates, whose arguments reflect their information about the state of production and the state of consumption. To control the computation we use the following directional types:

producer : ($\uparrow$ item, $\downarrow$ conf)
consumer : ($\downarrow$ item, $\uparrow$ conf)
read : ($\downarrow \{c\}$)
write : ($\downarrow \{i\}$, $\uparrow \{c\}$)

where

- The type item consists of all terms of the form $[i\,|\,t\,]$ where i is a constant (representing one produced item) and t is arbitrary term.

```
pc ← producer(X, X1), consumer(X, X1).

producer([i, i|T], [X1, X2|T1]) ←
   read(X1),
   producer([i|T], [X2|T1]).
producer([i], [X]) ←
   read(X).

consumer([X1, X2|T], [Y1, Y2|T1]) ←
   write(X1, Y1),
   consumer([X2|T], [Y2|T1]).
consumer([X], [Y]), ←
   write(X, Y).

write(i, c).
read(c).
```

FIGURE 6.4. A producer-consumer program

- The type conf consists of all terms of the form $[c\,|t\,]$ where c is a constant (representing a confirmation of consumption) and t is arbitrary term.

If atom-wise delays are used, the query pc would lead to deadlock. However, the program is S-well-typed, and hence every T-derivation starting from pc is deadlock-free.

A T-derivation simulates the interaction of the producer and the consumer described above. If the actual goal contains a call to 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 skeleton. Each produced item causes further instantiation of the list structure that X is bound to, and gives a possibility for one consumption step (that is, one delayed equation corresponding to the first argument of consumer in some node, becomes eligible). This gives 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 gives the producer 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.

7.1. The Dutch Flag problem

A program solving Dijkstra's "Dutch flag problem" [24, Chapter 14] is given in Sterling and Shapiro [43, p. 246]. In this section we will describe a version of this

program. The problem reads:

> ' "Given a list of elements colored *red*, *white*, and *blue*, reorder the list so that the red elements appear first, then all the white elements, followed by the blue elements. This reordering should preserve the original relative order of elements of the same color." For example, the list [*red(1), white(2), blue(3), red(4), white(5)*] should be reordered to [*red(1), red(4), white(2), white(5), blue(3)*].' [43, p. 245]

A straightforward solution is to build three separate output lists, one for each colour, as follows:

```
distribute/4 : (↓list, ↑list, ↑list, ↑list)

distribute([red(X)|Xs], [red(X)|Rs], Ws, Bs) ←
     distribute(Xs, Rs, Ws, Bs).
distribute([white(X)|Xs], Rs, [white(X)|Ws], Bs) ←
     distribute(Xs, Rs, Ws, Bs).
distribute([blue(X)|Xs], Rs, Ws, [blue(X)|Bs]) ←
     distribute(Xs, Rs, Ws, Bs).
distribute([ ], [ ], [ ], [ ]).
```

The top predicate becomes:

```
dutch/2 : (↓list, ↑list)

dutch(L, RWB) ←
     distribute(L, R, W, B),
     append(R, W, RW),
     append(RW, B, RWB).
```

Note that when the whole input list has been processed (the base case of `distribute` is applicable), the tails of the output lists are unified with the empty list. A more efficient solution, without the two calls to `append`, is obtained if we use open lists. We will now modify `distribute` in the following way: For every output list, we add an extra argument. When we reach the end of the input list, the tail of each output list will be unified with its extra argument In this way, we may unify the tail of an output list with something else than the empty list.

Now, to solve the problem, we only have to call `dist` appropriately: the tail of the "red" list structure should be the "white" list structure, the tail of the "white" list structure should be the "blue" list structure, and the tail of the "blue" list structure should be [ ]. The program is listed in Fig. 7.1.

The reader may verify that the program is S-well-typed; hence whenever `dutch/2` is called with the first argument bound to a list, the second argument will be a list upon success.

7.2. A small typesetting program

In this section, we will illustrate how directional types and the S-well-typing condition can be used to reason about programs of the language Gaplog [31]. This is an extension of Prolog which allows for connection of function symbols with external functional procedures. A term having such a symbol as its main functor will be

```
dist/7 : (↓list, ↑list, ↓list, ↑list, ↓list, ↑list, ↓list)

dist([red(X)|Xs], [red(X)|Rs], RsTail, Ws, WsTail, Bs, BsTail) ←
     dist(Xs, Rs, RsTail, Ws, WsTail, Bs, BsTail).
dist([white(X)|Xs], Rs, RsTail, [white(X)|Ws], WsTail, Bs, BsTail) ←
     dist(Xs, Rs, RsTail, Ws, WsTail, Bs, BsTail).
dist([blue(X)|Xs], Rs, RsTail, Ws, WsTail, [blue(X)|Bs], BsTail) ←
     dist(Xs, Rs, RsTail, Ws, WsTail, Bs, BsTail).
dist([ ], R, R, W, W, B, B).

dutch/2 : (↓list, ↑list)

dutch(L, RWB) ←
     dist(L, RWB, WB, WB, B, B, [ ]).
```

FIGURE 7.1. A program solving the Dutch Flag problem

called an *interpreted term.* For example, in the program discussed below, we will assume that the functor $+$ is interpreted as a function computing the sum of two integers. The language of definite clauses with interpreted function symbols has a clean declarative semantics, as explained in [31]. Operationally, the interpreted terms are handled as follows: Whenever such a term (e.g. $s + t$) is unified with another term u, it is checked that s and t are both 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. For a detailed account of this kind of operational semantics see Małuszyński et al. [31] (similar suggestions can be found in [2, 3, 33]). We note that the standard Prolog arithmetic is also a mechanism for evaluation of interpreted terms but it does not allow for delaying of insufficiently instantiated arguments of the arithmetic operations.

The operational semantics described above can be seen as a restricted form of T-resolution (using only the type ground), combined with evaluation of interpreted terms. The restriction is that only arguments including interpreted function symbols are delayed, while the others are unified without delay. In [10], a condition that guarantees deadlock-free execution was given. The condition is a special case of S-well-typedness and can be summarized as follows. Let P be a directionally typed program, where the only type used is ground. Assume that P is (1) S-well-typed and (2) all its interpreted terms appear only at exporting positions. Then any execution of P starting with a goal having the properties (1) and (2) will not deadlock.

The restriction (2) reflects the limitations of the operational semantics. For example, the equation $x + 1 = 2$ cannot be solved since the interpreted term $x + 1$ is not ground and cannot be evaluated. Thus, in contrast to the usual unification, groundness of one side of the equation does not guarantee that the other side will also become ground. On the other hand, S-well-typedness gives a sufficient codition for eventual groundness of all exporting positions.

Consider a simple 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:

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

typesetrow(Line, Ind, [Text|Ts], [ColWid|Cs], [NChars|Ns],
           [put(Line, Ind, Text)|Insts]) ←
       no_of_chars(Text, NChars),
       typesetrow(Line, Ind + ColWid, Ts, Cs, Ns, Insts).

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

typesettab(Line, Ind, [Row|Rows], ColWidths, WidSoFar, [InstRow|Insts]) ←
       typesetrow(Line, Ind, Row, ColWidths, Widths, InstRow),
       compute_max(Widths, WidSoFar, NewWidSoFar),
       typesettab(Line + 1, Ind, Rows, ColWidths, NewWidSoFar, Insts).
```

FIGURE 7.2. A typesetting program

```
[[this, is, a, 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,a), put(1,14,text)],
 [put(2,1,another), put(2,8,line), put(2,12,of), put(2,14,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 a text
another line of text

Note that every column is supposed to have the width of the longest word in the column. Obviously, this information is not available until we have processed the whole input list. Thus, the typesetting problem is intuitively a "two-pass" problem, just like the maximum-labeling problem of Sect. 3. However, by using logical variables as pointers, we can solve the problem in one pass. The solution is reminiscent of (though more complicated than) that of the `maxtree` program.

The program is given in Fig. 7.2.

The predicate `typesetrow/6` typesets one row of the table. We assume that `no_of_chars` is a predicate that, given an atom a in the first argument, returns the number of characters of the textual representation of a in the second argument. The arguments of `typesetrow/6` represent (from left to right) the current line, the current indentation on the line, the description of one row of the table (e.g. [`this`, `is`, `a`, `text`]), a list containing the widths of each column, a list containing 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:

`typesetrow/6` : ($\downarrow$ ground, $\downarrow$ ground, $\downarrow$ ground, $\downarrow$ ground, $\uparrow$ ground, $\uparrow$ ground)
`no_of_chars/2` : ($\downarrow$ ground, $\uparrow$ ground)

`typesetrow/6` is called from `typesettab/6`. The arguments of `typesettab/6` rep-

resent (from left to right) the current line, the current indentation on the line, the description of the whole table, a list containing the widths of each column, a list containing the widths of the widest element in each column in the rows processed so far, and the output list of typesetting instructions (for simplicity the output list is not flattened).

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:

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

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:

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

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

$$\texttt{typesettab}(1, 1, [[\texttt{this}, \texttt{is}, \texttt{a}, \texttt{text}], [\texttt{another}, \texttt{line}, \texttt{of}, \texttt{text}]], \_, [0, 0, 0, 0], \texttt{I})$$

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

Note that the widths of each column are not computed until the whole table has been processed. In the second clause of `typesetrow`, the variable `ColWid` will be unbound in the addition `Ind` + `ColWid`, when execution reaches this point. As explained, the computation of `Ind` + `ColWid` will suspend until `Ind` and `ColWid` are bound to ground values. However, since both the program and the goal are S-well-typed, and since interpreted terms appear at exporting positions only, no computations will suspend indefinitely.

8. DISCUSSION AND 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 [9, 17, 26]. In the approach of [26], 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 [9] can be seen as a special case of [26], where the properties described by the assertions are closed under substitution. The input assertions of [26] 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 tuples of types closed under substitution,
- for every argument one can only specify either its input assertion or its output assertion, but not both.

The second aforementioned paper [17] 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.

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

Most of the papers on directional types consider types closed under substitution. We have shown that under this assumption IO correctness coincides with success correctness which can be proved by the annotation method. Thus, specializations of the annotation method can be used for proving correctness of directional types. We have shown in Sect. 4.3 that the well-typing criterion presented in the literature can be seen as such specializations. 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 [14]. As long as the assertions are closed under substitution, their IO correctness can be proved by the annotation method, regardless of whether they are unary or not. Soundness of any new sufficient condition obtained by a new specialization of the annotation method would automatically follow from the soundness of the annotation method.

The call correctness for Prolog computation rule and the IO correctness of assertions not closed under substitution can still be proved by the method of [26]. Actually the method of [26] is complete [25], so that its verification condition can be seen as the best possible well-typing for Prolog execution rule.

8.2. Well-typing vs. S-well-typing

As already discussed, a directional type can be seen as an annotation. To prove its IO correctness it suffices to find a logical dependency scheme which is sound and non-circular. The *well-typing* criterion implicitly uses an LDS which is always non-circular, since the local dependencies between positions of the body atoms are always directed from left to right. This was shown in Sect. 4.3.

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:

$\mathtt{q(X)} \leftarrow \mathtt{p(X,X)}.$
$\mathtt{p(X,X)}.$

with the directional type:

$\mathtt{q} : (\downarrow \mathsf{ground})$
$\mathtt{p} : (\downarrow \mathsf{ground}, \uparrow \mathsf{ground})$

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 : (↓ int)
q : (↓ list)

The program is well-typed since:

⊨ [ ] : int ⇒ X : list

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

true ⇒ X : list

is not true.

A more comprehensive discussion on the relation between well-typing and S-well-typing can be found in [12].

8.3. Related work on directional types

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. We are also able to prove some interesting properties of programs using incomplete data structures. For such programs it is usually rather difficult to provide non-trivial well-typings.

The concepts of S-well-typing and of well-typed position generalize conditions proposed in [23] for groundness analysis of definite programs. The "data-driven" programs of [23] are well-typed programs, such that the only type used is the type ground of all ground terms. Similarly, the "simple" programs of [23] 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 [31] 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 [8], termination, e.g. in [38], AND-parallelism [19], and the question whether a program can be executed with pattern matching instead of full unification, e.g. in [6].

The first available implementation of the language Mercury [42] uses a condition akin to well-typing. All clauses must be well-typed (with the type ground), possibly after reordering the literals of the clause. However, the S-well-typing condition can also handle clauses whose body literals cannot be reordered to obtain well-typing, as shown by the example programs of Sections 3 and 7.

Existence of some systems that perform directional type checking is reported in the literature.

In the context of our work, Nixon [36] implemented a system that allows for defining new *regular* types and for associating a directional type with user-defined predicates. The predefined predicates have standard directional types, which can be changed by the user. The system can check the correctness of a directional type

with respect to either the well-typing condition or the S-well-typing condition, and issues a warning for every type judgement it is unable to prove. The system handles full Prolog.

The system of Rouzaud and Nguyen-Phoung [40] checks well-typing of programs, using a rather complicated type system. The system by Vétillard [44] checks well-typing of constraint logic programs (over the reals). The problem of directional type checking has also been addressed by Aiken and Lakshman [1], but there is no reference to an existing implementation.

9. 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 correctness of a directional type can be proved by the annotation method of Deransart [21, 23], 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. The S-well-typedness condition introduced in this paper is an example of such a specialisation.

We also considered directional types as a tool for controlling execution of logic programs. The idea of such execution is based on argument-wise delaying of unification of the arguments that are not correctly typed. This mechanism, called T-resolution, is more fine-grained than the atom-wise delaying used 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 [41].

T-resolution is sound but not complete, 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 [11], 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 [37]. 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.

It may be interesting to extend the presented techniques for the case of polymorphic directional types and also to address the problem of type inference. Some

preliminary results were reported in [12]. In this context relation to abstract in terpretation techniques is a natural question.

Acknowledgements. The authors would like to thank many persons who substantially contributed to this work. 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 gratefully acknowledge the comments given by Włodek Drabent, Gérard Ferrand, Eric Vétillard and the anonymous referees.

REFERENCES

1. A. Aiken and T. K. Lakshman. Directional type checking of logic programs. In Le Charlier (editor), *Static analysis — Proceedings of the first international static analysis symposium (SAS'94)*, pp. 43–60. LNCS 864, Springer-Verlag, 1994.
2. H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of logic programming*, vol. **16**, no. 3–4, pp. 195–234, 1993.
3. H. Aït-Kaci, P. Lincoln and R. Nasr. LeFun: Logic, Equations and FUNctions. In *Proceedings of the 4th IEEE international symposium on logic programming*, pp. 17–23, San Francisco, IEEE Computer Society Press, 1987.
4. K. Apt (editor). *Logic Programming — Proceedings of the Joint international symposium and conference*, Washington D.C. (USA). The MIT Press, 1992.
5. K. Apt. Declarative programming in Prolog. In *Proceedings of the International symposium on logic programming*, Vancouver (Canada), pp. 12–35. The MIT Press, 1993.
6. K. Apt and S. Etalle. On the unification-free Prolog programs. In *Proceedings of the conference on Mathematical foundations of computer science*, Berlin, pp. 1–19. Springer-Verlag, 1993.
7. K. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal aspects of computing* 3, pp. 743–765, 1994.
8. K. Apt and A. Pellegrini. Why the occur-check is not a problem. In *Proceedings of the 4th international symposium on Programming language implementation and logic programming*, Leuven (Belgium), pp. 69–86. LNCS 631, Springer-Verlag, 1992.
9. A. Bossi and N. Cocco. Verifying correctness of logic programs, in: *Proc. of TAPSOFT'89*, pp. 96-110. LNCS 352, Springer-Verlag, 1989.
10. J. Boye. S-SLD-resolution – an operational semantics for logic programs with external procedures. In Małuszyński and Wirsing [32], pp. 383–393.
11. J. Boye. Avoiding dynamic delays in functional logic programs. In *Proceedings of the 5th international symposium on Programming language implementation and logic programming*, pp. 12–27. LNCS 714, Springer-Verlag, 1993.
12. J. Boye. *Directional Types in Logic Programming.* Ph.D. thesis no. 437, Linköping studies in science and technology, 1996.
13. J. Boye and J. Małuszyński. Two aspects of directional types. In Sterling (editor). *Logic programming — Proceedings of the 12th international conference*, pp. 747–761. The MIT Press, 1995.
14. F. Bronsard, T. K. Lakshman and U. Reddy. A framework of directionality for proving termination of logic programs. In Apt [4], pp. 321–335.

15. 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.

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

17. L. Colussi and E. Marchiori. Proving correctness of logic programs using axiomatic semantics. In K. Furokawa (editor). *Logic Programming — Proceedings of the 8th international conference*, pp. 629–642. The MIT Press, 1991.

18. 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.

19. P. Dembiński and J. Małuszyński. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the 1985 IEEE symposium on logic programming*, Boston (USA), pp. 29–38. IEEE Computer Society Press, 1985.

20. P. Deransart. Logical Attribute Grammars. In *Proc. of IFIP 83*, pp. 463-469 North-Holland, 1983.

21. P. Deransart. Proof methods of declarative properties of definite programs. *Theroretical Computer Science*, 118, pp. 99-166, 1993.

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

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

24. E. Dijkstra. *A discipline of programming*. Englewood Cliffs NJ: Prentice-Hall, 1976.

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

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

27. M. Falaschi, G. Levi, M. Martelli and C. Palamidessi. Declarative modelling of the operational behaviour of logic languages. *Theoretical Computer Science* 69(3), pp. 289–318, 1989.

28. M. Jazayeri, W. Ogden and W. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Journal of the ACM*, vol. **18**, pp. 679–706, 1975.

29. D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, vol. **2**, pp. 127-145, 1968. Correction in *ibid.*, vol **5**, pp. 95–96, 1971.

30. J.W. Lloyd. *Foundations of Logic Programming*. Second Edition, Springer Verlag, Berlin, 1987.

31. 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.

32. J. Małuszyński and M. Wirsing (editors). *Programming language implementation and logic programming — Proceedings of the 3rd international symposium*, Passau (Germany). LNCS 528, Springer-Verlag, 1991.

33. L. Naish. Adding equations to NU-Prolog. In Małuszyński and Wirsing [32], pp. 15–26.

34. L. Naish. Coroutining and the construction of terminating logic programs. Report 92/5, Dept. of Computer Science, Univ. of Melbourne, 1992.

35. L. Naish. A declarative view of modes. Report 96/7, Dept. of Computer Science, Univ. of Melbourne, 1996.

36. L. Nixon. *A directional type checker for Prolog*. Master's thesis LiTH-IDA-Ex-9602, Linköping university, 1996.

37. J. Paakki. Multi-pass execution of functional logic programs. In *Conference record of the 21st ACM symposium on Principles of programming languages (POPL)*, Portland (USA), pp. 361–374. The ACM Press, 1994.

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

39. U.S. Reddy. A typed foundation for directional logic programming. In E. Lamma and P. Mello (eds.) *Extensions of logic programming*. LNAI 660, pp. 282–318. Springer Verlag, Berlin, 1992.

40. Y. Rouzaud and L. Nguyen-Phoung. Integrating modes and types into a Prolog type checker. In Apt [4] , pp. 85–97.

41. V.A. Saraswat. *Concurrent Constraint Programming Languages*. The MIT Press, 1990.

42. Z. Somogyi, F. Henderson and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. To appear in *Journal of logic programming*, 1996.

43. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.

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