Two Aspects of Directional Types

Johan Boye, Jan Małuszyński [1]
Linköping University
S-581 83 Linköping, Sweden
{ johbo, janma}@ida.liu.se

Abstract

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.

This paper discusses two aspects of directional types: the declarative notion of *input-output correctness* and the operational notion of *call correctness*. By separating these two concepts, we readily obtain better correctness criteria than those existing in the literature. We further show how directional types can be used for controlling execution of logic programs through a delay mechanism.

1 Introduction

Recently there has been a growing interest in the notion of *directional types* for Prolog programs [6, 12, 2, 3, 1, 11]. This kind of prescriptive typing describes the intended ways of calling the program, as well as the user's intuition of how the program behaves when called as prescribed. Together with some methods and tools for type checking, directional types may provide a good support for program validation.

This paper shows that directional types 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. This view allows us to obtain a better correctness criterion than those existing in the literature (e.g. the *well-typing* condition of [6, 2]). 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

[1]Address until July 1995: INRIA-Rocquencourt, B.P. 105, 78153 Le Chesnay cedex, France

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[2]. As an example, consider the `append/3` predicate:

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

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

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

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

A 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 [2, 6] 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 example of Sect. 3. In Sect. 4 we formulate a correctness condition – *S-well-typedness* – which is sufficient to ensure input-output correctness of directional types which are not call correct.

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

[2]Directional types constitute a special case of *inductive assertions* [10].

In section 6, we discuss *type-driven resolution* (or simply T-resolution), which uses directional types as a means for controlling execution of logic programs through a delay mechanism. T-resolution is sound but not complete in general, since the computation may deadlock. We show that S-well-typedness is a sufficient condition for deadlock-free execution under T-resolution.

Proofs of all theorems can be found in [5].

2 Preliminaries

2.1 Types

Adopting to a popular view (e.g. Apt [2]), 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	
Gnd	the set of ground terms	
List	$[] \mid [Any \mid List]$	(lists)
GList	$[] \mid [Gnd \mid GList]$	(ground lists)
BinTree	$void \mid tree(Any, BinTree, BinTree)$	(binary trees)
GBinTree	$void \mid tree(Gnd, GBinTree, GBinTree)$	(ground bin. trees)

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

An atom $p(t_1 : T_1, ..., t_n : T_n)$ is said to be *correctly typed in its i-th position* iff $t_i \in T_i$. The atom `append([],Y,[1,2])` is correctly typed in its first and third positions. Let P be a program and 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 starting from A are correctly typed in their input positions, and if

(2) for every computed answer σ, $A\sigma$ 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*).

2.2 Well-typing

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 $\sigma(s_i) \in S_i$ $(1 \le i \le n)$, then $\sigma(t) \in T$.

To simplify the notation, we will throughout this section write an atom as $p(\mathbf{u} : \mathbf{U}, \mathbf{t} : \mathbf{T})$, where $\mathbf{u{:}U}$ is a sequence of typed terms filling in the input positions of p_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 Prolog computation rule (cf. [6]):

Definition 2.1

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

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

  and if

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

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

Thus a clause is well-typed if

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

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

$$([], X) : (\mathit{List}, \mathit{List}) \ \Rightarrow \ X : \mathit{List}$$

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

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

and that

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

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

2.3 Proof trees

We now summarize a uniform framework for discussing both the operational and the declarative semantics of definite programs. This will allow us to discuss IO correctness without taking the operational model into account. The framework originates from Deransart and Małuszyński [9], and is based on the notion of proof tree.

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

Definition 2.2 A *skeleton* is a tree defined as follows:

- if G is an (atomic) initial query, then the node labeled $(G, \perp)$ 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 $(A, \perp)$;
  2. choose a clause $A_0 :- A_1, \ldots, A_k$ in P, such that A and A_0 have the same predicate symbol and the same arity;
  3. change n's label into $(A, \sigma(A_0))$, (where σ is a renaming to fresh variables), and add k children to n, labeled $(\sigma(A_1), \perp), ., (\sigma(A_k), \perp)$.

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

Definition 2.3 The *set of equations associated to the node* n is denoted by $E(n)$, and is defined as follows:

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

For example, an LD-resolution[3] step corresponds to choosing the leftmost node n (in preorder of the skeleton), expanding it as described in definition 2.2, and computing a solved form of $E(n)$ (i.e. performing unification). A *proof tree* is a complete skeleton, together with a solution of all the associated equations. Every successful LD-derivation corresponds to a proof tree, and the obtained mgu of the set of equations restricted to the variables of the root label of the skeleton is the computed answer substitution.

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

[3]SLD-resolution with Prolog's computation rule.

2.4 Dependencies

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

Definition 2.4 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-typedness, 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 2.5 $A(i)$ 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. □

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

Definition 2.6 For each clause C, its *local dependency relation* $\rightsquigarrow_C$ is 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. □

A skeleton is obtained by pasting together instances of clauses. To model the dataflow in a complete skeleton T, we construct a *compound dependency graph* $\rightsquigarrow_T$ by pasting together the local dependency graphs for the clauses used in T (for a formal definition, see the full version of the paper [5]).

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

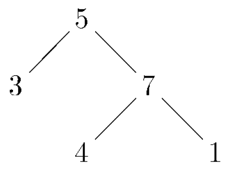
Definition 2.7 If the relation $\rightsquigarrow_T$ is a partial ordering for every skeleton T, then the program is said to be *non-circular*. □

The non-circularity concept stems originally from the field of attribute grammars. It is well-known that this property is decidable (see e.g. [9]).

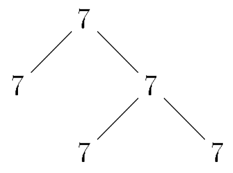
3 An informal example

In this section, we give an example of a program which is IO correct but not well-typed. We claim that such directional types often are of practical interest, especially for programs using incomplete data structures. The interested reader can find more examples in the full version of the paper [5].

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 following program solves the problem in one pass.

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

maxtree(void, _, L, L, void).
maxtree(tree(Lbl,Lft,Rgt),Max,In,[Lbl|Out],tree(Max,NLft,NRgt)):-
        maxtree(Lft, Max, In, IO, NLft),
        maxtree(Rgt, Max, IO, Out, NRgt).
```

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

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

$$\begin{array}{l}
\texttt{maxtree/2} : (\downarrow \mathit{GBinTree},\ \uparrow \mathit{GBinTree}) \\
\texttt{maxtree/5} : (\downarrow \mathit{GBinTree},\ \downarrow \mathit{Any},\ \downarrow \mathit{GList},\ \uparrow \mathit{GList},\ \uparrow \mathit{BinTree}) \\
\texttt{max/2} : (\downarrow \mathit{GList},\ \uparrow \mathit{Gnd})
\end{array}$$

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

$$\begin{array}{rl}
Tree : \mathit{GBinTree} & \wedge \\
(Labels, NewTree) : (\mathit{GList}, \mathit{BinTree}) & \wedge \\
Max : \mathit{Gnd} & \Rightarrow \\
NewTree : \mathit{GBinTree} &
\end{array}$$

is not true. The problem is caused by the variable $NewTree$: one can not conclude that $NewTree$ is a ground binary tree just from the fact that it is a binary tree. Thus we cannot use the well-typing condition to conclude that the program is correctly typed.

Now consider changing the directional type for `maxtree/5` as follows (the other predicates are typed as before):

$$\texttt{maxtree/5} : (\downarrow GBinTree,\ \downarrow Gnd,\ \downarrow GList,\ \uparrow GList,\ \uparrow GBinTree)$$

The idea is that if `maxtree/5` is called with its second argument bound to a ground term, then the last argument will be bound to a ground binary tree upon success. Now the directional type for the program as a whole is not call correct under LD-resolution; the `maxtree/5` predicate is called with the second argument being a variable, not a ground term. However, the directional type remains IO-correct, as will be shown by the method presented in Sect. 4.

4 Proving IO correctness

The problem whether a given directional type is IO correct or not is independent of a particular computation rule. Thus the problem can be discussed in terms of proof trees of a program rather than in terms of computations. The method for proving IO correctness of directional types presented in this section is a special case of the *annotation* method for proving properties of proof trees, introduced in [8] (see also [9]). For brevity, we introduce our method directly, instead of deriving it from the annotation method, as done in the full version of the paper [5].

The well-typing condition of section 2 requires (among other things), that the types of terms at exporting clause positions in the body can be inferred from the types at importing clause positions in *preceding* literals. The reason for only looking at preceding literals is that Prolog's computation rule is assumed. If we abstract away from the computation rule, this restriction is no longer necessary; Given an exporting clause position e, we may use *all* importing positions in the clause to infer the type of e (to be more exact, we will only regard those importing positions which share a variable with e). This line of reasoning motivates the definition of *sharing-based well-typing* (*S-well-typing*):

Definition 4.1 Let $\mathcal{T}$ be a directional type for a program P, and let C be a clause of P. Denote by $\leadsto_C$ the dependency relation determined by $\mathcal{T}$ on the positions of C.

- For a given exporting position e in C:
  - let t be the term occurring in C on e, and let T be the type associated to e by $\mathcal{T}$.

– let $i_1, ..., i_k$ be all importing positions of C such that $i_j \leadsto_C e$, and let $t_1, ..., t_n$ be the terms on these positions of C typed, respectively, $T_1, ..., T_n$ by $\mathcal{T}$.

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

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

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

Theorem 4.2 Every S-well-typed program is correctly IO-typed.

Consider the `maxtree` program of Section 3 with the second directional type, i.e. :

`maxtree/2` : $(\downarrow GBinTree, \ \uparrow GBinTree)$
`maxtree/5` : $(\downarrow GBinTree, \ \downarrow Gnd, \ \downarrow GList, \ \uparrow GList, \ \uparrow GBinTree)$
`max/2` : $(\downarrow GList, \ \uparrow Gnd)$

It is easy to verify that each clause is S-well-typed. By methods used in the field of attribute grammars, the program can automatically be proved non-circular [9]. Hence the program is IO correct, and we may conclude that the second argument of `maxtree/2` will be a ground binary tree upon success.

5 Call correctness under LD-resolution

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

Reconsider the `maxtree` program in Sect. 3 with the directional type above. When executed with LD-resolution, in every call to the recursive clause for `maxtree/5`, the first and third position (but not the second) are correctly typed. Upon success, the fourth position (but not the fifth) is correctly typed. Intuitively, the reason is that the dataflow to these positions follows the execution order of LD-resolution. We say that these positions are *well-typed*.

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

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

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

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

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

It is easily realized that there exists a *largest* set of well-typed clause positions (see [5] for a proof). A clause position of P will be called *well-typed* if it belongs to this set.

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

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

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

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

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

maxtree(void, _, L, L, void).
maxtree(tree(Lbl,Lft,Rgt),Max,In,[Lbl|Out],tree(Max,NLft,NRgt)):-
      maxtree(Lft, Max, In, Out1, NLft),
      maxtree(Rgt, Max, Out1, Out, NRgt).
```

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

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

- if e is well-typed under $\mathcal{T}$, 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`: $(\downarrow GBinTree, \uparrow Any)$

`maxtree/5`: $(\downarrow GBinTree, \downarrow Any, \downarrow GList, \uparrow GList, \uparrow BinTree)$

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

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

Corollary 5.5 Let P be a program, and let G be an atom which is correctly typed in its input positions. Let $H :- A_1, \ldots, A_n$ be a clause in P. Then in every LD-derivation starting from G: if the query $\sigma(A_j, \ldots, A_n, B_1, \ldots, B_m)$ is reached, then the well-typed input positions in $\sigma(A_j)$ are correctly typed.

Thus we may conclude (for instance) that the types of the first and third arguments of `maxtree/2` are call invariants.

Corollary 5.6 Let P be a program, and let G be an atom which is correctly typed in its input positions. Then for every computed answer substitution σ, $\sigma(G)$ is correctly typed in its well-typed output positions.

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 [7]), 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.

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

Definition 6.2 [Eligible equation] Let p/n be a predicate with an associated directional type. We denote the type of the i:th position with T_i. Let $p(s_1, \ldots, s_n) = p(t_1, \ldots, t_n)$ be an equation. The equation $s_j = t_j$ is *eligible* if either

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

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

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

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

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

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

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

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

$$\begin{array}{l}\texttt{maxtree/2} : (\downarrow \mathit{GBinTree}, \uparrow \mathit{GBinTree}) \\ \texttt{maxtree/5} : (\downarrow \mathit{GBinTree}, \downarrow \mathit{Gnd}, \downarrow \mathit{GList}, \uparrow \mathit{GList}, \uparrow \mathit{GBinTree}) \\ \texttt{max/2} : (\downarrow \mathit{GList}, \uparrow \mathit{Gnd})\end{array}$$

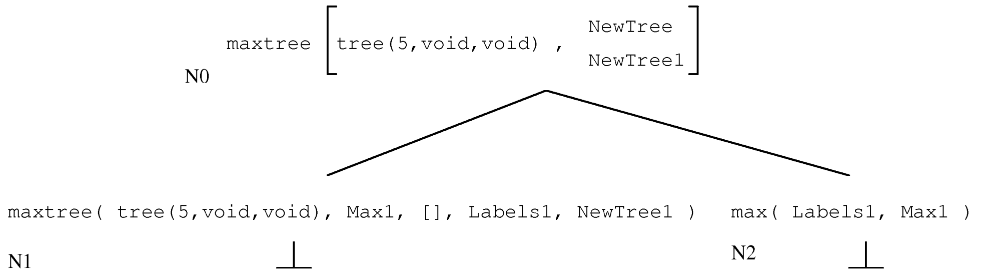
Consider the initial query

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

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

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

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



maxtree [tree(5,void,void) , NewTree / NewTree1]
N0
maxtree(tree(5,void,void), Max1, [], Labels1, NewTree1)
N1
max(Labels1, Max1)
N2

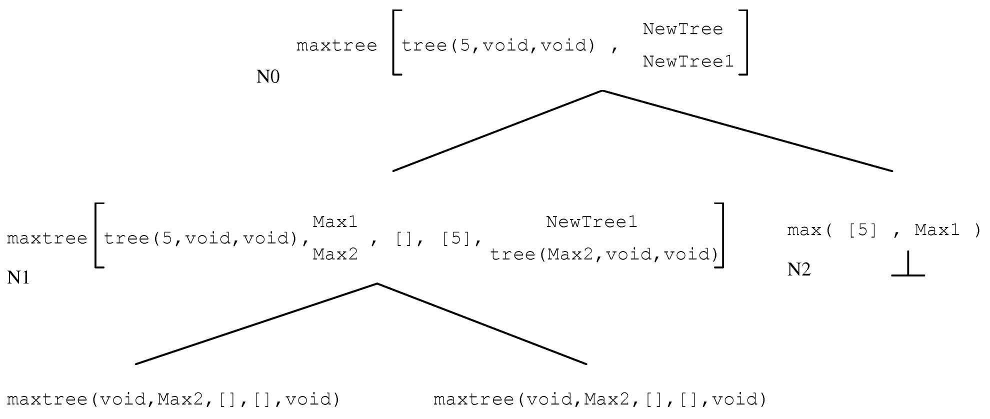
Figure 1: An incomplete proof tree

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

When we now resolve the atom `max([5], Max1)`, the variable `Max1` becomes instantiated to 5. We can now solve the equation `Max1=Max2` at node N_1, since `Max1` now is instantiated to the right type (Gnd). We can then solve the equation `NewTree1=tree(Max2,void,void)` at node N_2, and finally the equation `NewTree=NewTree1` at node N_0.

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

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.



maxtree tree(5,void,void) , NewTree NewTree1
N0
maxtree tree(5,void,void), Max1 Max2 , [], [5], NewTree1 tree(Max2,void,void)
N1
max([5] , Max1)
N2
maxtree(void,Max2,[],[],void)
maxtree(void,Max2,[],[],void)

Figure 2: Another incomplete proof tree

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

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

7 Discussion and Conclusions

We have separated two aspects of directional types: the input-output characterisation of the program, which is independent of the computation model, and the characterisation of the call patterns, which strongly depends on the execution rule. By abstracting away from the computation rule, we have obtained a relatively simple sufficient condition (S-well-typedness) for IO correctness. This condition enables us to prove IO properties of interesting programs, for which the well-typedness criterion does not apply.

We also considered directional types as a means for controlling execution of logic programs. The idea of such execution is to enforce the types specified by a given declaration, by argument-wise delaying the unification of the arguments which are not correctly typed. 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 [13]. We formalized the model of execution by the concept of T-resolution. T-resolution is sound but not complete in general, since the computation may deadlock. We have shown that S-well-typedness is a sufficient condition for deadlock-free execution under T-resolution.

The notion of S-well-typedness 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, and 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 [4], where the resulting logic program is executed without delays under the Prolog computation rule.

The usefulness of T-resolution is an open question. In this paper we use it more as an illustration of the thesis that the methods for directional types apply not only to Prolog. It is an interesting question whether a variant of the technique used here for deriving a sufficient condition for deadlock-freeness of T-resolution, may be of interest for concurrent constraint programming.

Our future work aims at automated checking of S-well-typedness of programs, as well as extending the conditions for well-typing and S-well-typing to types which are not closed under substitution.

References

1. A. Aiken and T. K. Lakshman. Directional type checking of logic programs. *Proc. of SAS'94*, pp. 43–60. Springer-Verlag, 1994.
2. K.R. Apt. Declarative programming in Prolog. In *Proc. of ILPS'93*, pp. 12–35. The MIT Press, 1993.
3. K.R. Apt and E. Marchiori. *Reasoning about Prolog programs: from modes through types to assertions*. Technical report CS-R9358, CWI Amsterdam, 1993.
4. J. Boye. Avoiding dynamic delays in functional logic programs. In *Proc. PLILP'93*, pp. 12-27. Springer-Verlag, 1993.
5. J. Boye and J. Małuszyński. *Directional types and the annotation method.* Report RR2471, INRIA Rocquencourt, 1995.
6. F. Bronsard, T. K. Lakshman and U. Reddy. A framework of directionality for proving termination of logic programs. In *Proc. of JICSLP'92*, pp. 321–335. The MIT Press, 1992.
7. 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.
8. 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).
9. P. Deransart and J. Małuszyński. *A grammatical view on logic programming*. The MIT Press, 1993.
10. W. Drabent and J. Małuszyński. Induction assertion method for logic programs. *Theoretical Computer Science* 59, pp. 133–155, 1988.
11. D. Pedreschi. A proof method for run-time properties of Prolog programs. In *Proc. of ICLP'94*, pp. 584–598. The MIT Press, 1994.
12. Y. Rouzaud and L. Nguyen-Phoung. Integrating modes and subtypes into a Prolog type checker. In *Proc. of JISCLP'92*, pp. 85–97. The MIT Press, 1992.
13. V.A. Saraswat. *Concurrent constraint programming languages*. The MIT Press, 1990.