Formal Aspects of Computing (1994) 3: 1–000

Reasoning about Prolog Programs: from Modes through Types to Assertions

Krzysztof R. Apt[1] and Elena Marchiori[2]

[1]CWI,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Faculty of Mathematics and Computer Science,
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

[2] CWI,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Keywords: Prolog programs, program verification.

Abstract. We provide here a systematic comparative study of the relative strength and expressive power of a number of methods for program analysis of Prolog. Among others we show that these methods can be arranged in the following hierarchy: mode analysis ⇒ type analysis ⇒ monotonic properties ⇒ non-monotonic run-time properties. We also discuss a method allowing us to prove global run-time properties.

1. Introduction

1.1. Motivation

Over the past 9 years a number of proposals were made in the literature for the analysis and verification of Prolog programs, based on the concepts of modes, types and assertions, both monotonic ones and non-monotononic ones, like $var(x)$.

Correspondence and offprint requests to: Elena Marchiori, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

The aim of this paper is to show that these methods can be arranged in a hierarchy in which increasingly stronger program properties can be established and in which each method is a generalization of the preceding ones.

More specifically, we deal here with the following notions: well-moded programs, essentially due to Dembinski and Małuszynski [DM85], well-typed programs, due to Bronsard, Lakshman and Reddy [BLR92], the assertional method of Bossi and Cocco [BC89], the assertional method of Drabent and Małuszyński [DM88]. Moreover we discuss the assertional method of Colussi and Marchiori [CM91], which allows to prove global run-time properties. To render the exposition uniform, the formalisms and the terminology used will sometimes slightly differ from those of the original works.

We believe that the systematic presentation of these methods of program analysis is useful for a number of reasons. First it clarifies the relationship between them. Next, it allows us to justify them by means of simpler correctness proofs than the original ones. Further, it suggests in a natural way some new results about these methods. Finally, it allows us to better understand which program properties can be established by means of which method.

1.2. Preliminaries

We consider logic programs executed by means of the *LD-resolution*, which consists of the SLD-resolution combined with the leftmost selection rule. An SLD-derivation in which the leftmost selection rule is used is called an *LD-derivation*, or simply a *derivation*.

We work here with *queries*, that is sequences of atoms, instead of *goals*, that is constructs of the form $\leftarrow Q$, where Q is a query. Apart from this we use the standard notation of Lloyd [Llo87] and Apt [Apt90]. In particular, given a syntactic construct E (so for example, a term, an atom or a set of equations) we denote by $vars(E)$ the set of the variables appearing in E. Variables are denoted with x, y, z, possibly subscripted, while terms are denoted by r, s, t, possibly subscripted. Moreover, we adopt the Prolog convention to denote variables appearing in a Prolog program by means of strings starting with a capital letter.

Given a substitution $\theta = \{x_1/t_1, ..., x_n/t_n\}$, the set $\{x_1, \ldots, x_n\}$ of variables is denoted by $dom(\theta)$ and $range(\theta)$ denotes the set of variables occurring in $\{t_1, \ldots, t_n\}$. Moreover, $vars(\theta) = dom(\theta) \cup range(\theta)$. Finally, a substitution ρ is called *renaming* if it is a 1-1 and onto mapping from its domain to itself. For two atoms or terms e_1, e_2, we denote by $mgu(e_1, e_2)$ a fixed most general unifier (in short mgu) of e_1, e_2. Recall that mgu's are equivalent up to renaming, i.e., if θ and β are two mgu's of e_1, e_2 then $\theta = \beta\rho$, for some renaming ρ.

2. Well-moded Programs

We start by introducing modes. They were first considered in Mellish [Mel81], and more extensively studied in Reddy [Red84], [Red86] and Dembinski and Małuszynski [DM85].

Definition 2.1. (Mode) Consider an n-ary relation symbol p. By a *mode* for p we mean a function m_p from $\{1, \ldots, n\}$ to the set $\{+, -\}$. If $m_p(i) =$ '$+$', we call i an *input position* of p and if $m_p(i) =$ '$-$', we call i an *output position* of p (both w.r.t. m_p).

We write m_p in a more suggestive form $p(m_p(1), \ldots, m_p(n))$. By a *moding* we mean a collection of modes, each for a different relation symbol. $\square$

Modes indicate how the arguments of a relation should be used. The definition of moding assumes one mode per relation in a program. Multiple modes may be obtained by simply renaming the relations. In the remainder of this section we adopt the following

Assumption 2.2. *Every relation* has a fixed mode associated with it.

This will allow us to talk about input positions and output positions of an atom.

We now introduce the notion of a well-moded program. The concept is due to Dembinski and Małuszynski [DM85]; we use here an elegant formulation due to Rosenblueth [Ros91] (which is equivalent to that of Drabent [Dra87] where well-moded programs are called simple). The definition of a well-moded program constrains the "flow of data" through the clauses of the programs. To simplify the notation, when writing an atom as $p(\mathbf{u}, \mathbf{v})$, we now assume that $\mathbf{u}$ is a sequence of terms filling in the input positions of p and that $\mathbf{v}$ is a sequence of terms filling in the output positions of p.

Definition 2.3. (Well-Moded)
- A query $p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ is called *well-moded* if for $i \in [1, n]$

$$vars(\mathbf{s_i}) \subseteq \bigcup_{j=1}^{i-1} vars(\mathbf{t_j}).$$

- A clause $p_0(\mathbf{t_0}, \mathbf{s_{n+1}}) \leftarrow p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$ is called *well-moded* if for $i \in [1, n+1]$

$$vars(\mathbf{s_i}) \subseteq \bigcup_{j=0}^{i-1} vars(\mathbf{t_j}).$$

- A program is called *well-moded* if every clause of it is. $\square$

Thus, a query is well-moded if
- every variable occurring in an input position of an atom ($i \in [1, n]$) occurs in an output position of an earlier ($j \in [1, i-1]$) atom.

And a clause is well-moded if
- ($i \in [1, n]$) every variable occurring in an input position of a body atom occurs either in an input position of the head ($j = 0$), or in an output position of an earlier ($j \in [1, i-1]$) body atom,
- ($i = n+1$) every variable occurring in an output position of the head occurs in an input position of the head ($j = 0$), or in an output position of a body atom ($j \in [1, n]$).

Note that a query with only one atom is well-moded iff this atom is ground in its input positions. The following notion is due to Dembinski and Małuszynski [DM85].

Definition 2.4. We call an LD-derivation *data driven* if all atoms selected in it are ground in their input positions. $\square$

The following lemma shows the "persistence" of the notion of well-modedness.

Lemma 2.5. An LD-resolvent of a well-moded query and a well-moded clause that is variable disjoint with it, is well-moded.

Proof. An LD-resolvent of a query and a clause is obtained by means of the following three operations:

- instantiation of a query,
- instantiation of a clause,
- replacement of the first atom, say H, of a query by the body of a clause whose head is H.

So we only need to prove the following two claims.

Claim 1. An instance of a well-moded query (resp. clause) is well-moded.

Proof. It suffices to note that for any sequences of terms $\mathbf{s}, \mathbf{t_1}, \ldots, \mathbf{t_n}$ and a substitution σ, $vars(\mathbf{s}) \subseteq \bigcup_{j=1}^{n} vars(\mathbf{t_j})$ implies $vars(\mathbf{s}\sigma) \subseteq \bigcup_{j=1}^{n} vars(\mathbf{t_j}\sigma)$. □

Claim 2. Suppose $H, \mathbf{A}$ is a well-moded query and $H \leftarrow \mathbf{B}$ is a well-moded clause. Then $\mathbf{B}, \mathbf{A}$ is a well-moded query.

Proof. Let $H = p(\mathbf{s}, \mathbf{t})$ and $\mathbf{B} = p_1(\mathbf{s_1}, \mathbf{t_1}), \ldots, p_n(\mathbf{s_n}, \mathbf{t_n})$. We have $vars(\mathbf{s}) = \emptyset$ since H is the first atom of a well-moded query. Thus $\mathbf{B}$ is well-moded. Moreover, $vars(\mathbf{t}) \subseteq \bigcup_{j=1}^{n} vars(\mathbf{t_j})$, since $H \leftarrow \mathbf{B}$ is a well-moded clause and $vars(\mathbf{s}) = \emptyset$. These two observations imply the claim. □

The definition of a well-moded program is designed in such a way that the following theorem, also due to Dembinski and Małuszynski [DM85], holds.

Theorem 2.6. Let P and Q be well-moded. Then all LD-derivations of Q in P are data driven.

Proof. Note that the first atom of a well-moded query is ground in its input positions and a variant of a well-moded clause is well-moded. The conclusion now follows by Lemma 2.5. □

The following is a well-known conclusion of this theorem.

Corollary 2.7. Let P and Q be well-moded. Then for every computed answer substitution σ, $Q\sigma$ is ground.

Proof. Let $\mathbf{x}$ stand for the sequence of all variables that appear in Q. Let p be a new relation of arity equal to the length of $\mathbf{x}$ and with all positions moded as input. Then $Q, p(\mathbf{x})$ is a well-moded query.

Now, σ is a computed answer substitution for Q in P iff $p(\mathbf{x})\sigma$ is a selected atom in an LD-derivation of $Q, p(\mathbf{x})$ in P. The conclusion now follows by Theorem 2.6. □

Let us see now how these results can be applied to specific programs.

Example 2.8. Consider the program `quicksort`:

```
qs([X | Xs], Ys) ←
   part(X, Xs, Littles, Bigs), qs(Littles, Ls),
   qs(Bigs, Bs), app(Ls, [X | Bs], Ys).
qs([], []) ← .

part(X, [Y | Xs], [Y | Ls], Bs) ←
```

```
    X > Y, part(X, Xs, Ls, Bs).
part(X, [Y | Xs], Ls, [Y | Bs]) ←
    X ≤ Y, part(X, Xs, Ls, Bs).
part(X, [], [], []) ← .

app([X|Xs], Ys, [X|Zs]) ← app(Xs, Ys, Zs).
app([], Ys, Ys)← .
```

We mode it as follows: `qs(+,-)`, `part(+,+,-,-)`, `app(+,+,-)`, `>(+,+)`, `≤(+,+)`. It is easy to check that `quicksort` is then well-moded. Assume now that s is a ground term. By Theorem 2.6 all LD-derivations of `qs(s,t)` in `quicksort` are data driven and by Corollary 2.7 we conclude that all the computed answer substitutions σ are such that $t\sigma$ is ground. □

In conclusion, mode analysis is sufficient to derive information on groundness of atom arguments, before or after their selection. Also, as shown in Apt and Pellegrini [AP94] (and on which this section is based), the modes can be used to provide sufficient synctactic conditions that allow the occur-check to be safely omitted from the unification algorithm in Prolog implementations.

3. Well-typed Programs

3.1. Types and Type Judgements

To deal with run-time errors we introduce the notion of a type. We adopt the following general definition.

Definition 3.1. (Type) A *type* is a decidable set of terms closed under substitution. □

Certain types will be of special interest:
List — the set of lists,
Gae — the set of ground arithmetic expressions (gae's in short),
ListGae — the set of lists of gae's.
Ground — the set of ground terms.

Of course, the use of the type *List* assumes the existence of the empty list `[]` and the list constructor `[.|.]` in the language, the use of the type *Gae* assumes the existence of the numeral `0` and the successor function `s(.)` and the use of the type *ListGae* assumes the existence of what the use of the types *List* and *Gae* implies.

Throughout the paper we fix a specific set of types, denoted by *Types*, which includes the above ones. We call a construct of the form $s : S$, where s is a term and S is a type, a *typed term*. Given a sequence $\mathbf{s} : \mathbf{S} = s_1 : S_1, \ldots, s_n : S_n$ of typed terms, we write $\mathbf{s} \in \mathbf{S}$ if for $i \in [1, n]$ we have $s_i \in S_i$, and define $vars(\mathbf{s} : \mathbf{S}) = vars(\mathbf{s})$. Further, we abbreviate the sequence $s_1\theta, ..., s_n\theta$ to $\mathbf{s}\theta$. We say that $\mathbf{s} : \mathbf{S}$ is *realizable* if $\mathbf{s}\eta \in \mathbf{S}$ for some η.

Definition 3.2.
- By a *type judgement* we mean a statement of the form

$$\mathbf{s} : \mathbf{S} \;\Rightarrow\; \mathbf{t} : \mathbf{T}. \tag{1}$$

- We say that a type judgement (1) *is true*, and write

$$\models \mathbf{s} : \mathbf{S} \;\Rightarrow\; \mathbf{t} : \mathbf{T},$$

if for all substitutions θ, $\mathbf{s}\theta \in \mathbf{S}$ implies $\mathbf{t}\theta \in \mathbf{T}$. □

For example, the type judgement $s(s(x)) : \mathit{Gae},\ l : \mathit{ListGae} \;\Rightarrow\; [x \mid l] : \mathit{ListGae}$ is true. How to prove that a type judgement is true is an interesting problem but irrelevant for our considerations. In all considered cases it will be clear how to show that a type judgement is true.

The following simple properties of type judgements hold.

Lemma 3.3. (Type Judgement) Let $\phi, \phi_1, \phi_2, \phi_2', \phi_3$ and ψ be sequences of typed terms.

(i) Suppose that $s \in S$ and $\models s : S, \phi \;\Rightarrow\; \psi$. Then

$$\models \phi \;\Rightarrow\; \psi.$$

(ii) Suppose that $\models \phi_2 \;\Rightarrow\; \phi_2'$ and $\models \phi_1, \phi_2', \phi_3 \;\Rightarrow\; \psi$. Then

$$\models \phi_1, \phi_2, \phi_3 \;\Rightarrow\; \psi.$$

(iii) Suppose that $\models \mathbf{s} : \mathbf{S}, \mathbf{t} : \mathbf{T} \;\Rightarrow\; \mathbf{u} : \mathbf{U}$, $\mathbf{t} : \mathbf{T}$ is realizable, and $\mathit{vars}(\mathbf{t}) \cap \mathit{vars}(\mathbf{s}, \mathbf{u}) = \emptyset$. Then

$$\models \mathbf{s} : \mathbf{S} \;\Rightarrow\; \mathbf{u} : \mathbf{U}.$$

Proof.
(i) By the assumption that all types are closed under substitution.
(ii) Immediate.
(iii) Take θ such that $\mathbf{s}\theta \in \mathbf{S}$ and let η be such that $\mathbf{t}\eta \in \mathbf{T}$. Define $\theta' = \theta_{|\mathit{vars}(\mathbf{s}, \mathbf{u})}$ and $\eta' = \eta_{|\mathit{vars}(\mathbf{t})}$. Then $\sigma = \theta' \cup \eta'$ is well-defined, $\mathbf{s}\sigma \in \mathbf{S}$ and $\mathbf{t}\sigma \in \mathbf{T}$. So $\mathbf{u}\sigma \in \mathbf{U}$, i.e. $\mathbf{u}\theta \in \mathbf{U}$. □

3.2. Well-typed Queries and Programs

The next step is to define types for relations.

Definition 3.4. (Type) Consider an n-ary relation symbol p. By a *type* for p we mean a function t_p from $[1, n]$ to the set *Types*. If $t_p(i) = S$, we call S *the type associated with the position i of p.* □

In the remainder of this section we consider a combination of modes and types and adopt the following

Assumption 3.5. *Every relation* has a fixed mode and a fixed type associated with it.

This assumption will allow us to talk about types of input positions and of output positions of an atom. An n-ary relation p with a mode m_p and type t_p will be denoted by $p(m_p(1) : t_p(1), \ldots, m_p(n) : t_p(n))$. For example, $\mathtt{app}(+ : \mathit{List},\ + : \mathit{List},\ - : \mathit{List})$ denotes a ternary relation `app` with the first two positions moded as input and typed as *List*, and the third position moded as output and typed as *List*.

To simplify the notation, when writing an atom as $p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$ we now assume that $\mathbf{u} : \mathbf{S}$ is a sequence of typed terms filling in the input positions of

p and $\mathbf{v} : \mathbf{T}$ is a sequence of typed terms filling in the output positions of p. We call a construct of the form $p(\mathbf{u} : \mathbf{S}, \mathbf{v} : \mathbf{T})$ a *typed atom.* We say that a typed atom $p(s_1 : S_1, \ldots, s_n : S_n)$ is *correctly typed in position i* if $s_i \in S_i$.

The following notion is due to Bronsard, Lakshman and Reddy [BLR92].

Definition 3.6. (Well-Typed)

- A query $p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$ is called *well-typed* if for $j \in [1, n]$
$$\models \mathbf{o_1} : \mathbf{O_1}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \ \Rightarrow \ \mathbf{i_j} : \mathbf{I_j}.$$
- A clause
$p_0(\mathbf{o_0} : \mathbf{O_0}, \mathbf{i_{n+1}} : \mathbf{I_{n+1}}) \leftarrow p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$ is called *well-typed* if for $j \in [1, n+1]$
$$\models \mathbf{o_0} : \mathbf{O_0}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \ \Rightarrow \ \mathbf{i_j} : \mathbf{I_j}.$$
- A program is called *well-typed* if every clause of it is. □

Thus, a query is well-typed if

- the types of the terms filling in the *input* positions of an atom can be deduced from the types of the terms filling in the *output* positions of the previous atoms.

And a clause is well-typed if

- ($j \in [1, n]$) the types of the terms filling 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 previous body atoms,
- ($j = n + 1$) 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 types of the terms filling in the *output* positions of the body atoms.

Note that a query with only one atom is well-typed iff this atom is correctly typed in its input positions. The following observation clarifies the relation between well-moded and well-typed programs and queries.

Theorem 3.7. The notion of a well-moded program (resp. query) is a special case of the notion of a well-typed program (resp. query).

Proof. Take *Ground* as the only type. Then the notions of a well-moded program (resp. query) and a well-typed program (resp. query) coincide. □

The following lemma stated in Bronsard, Lakshman and Reddy [BLR92] shows persistence of the notion of being well-typed.

Lemma 3.8. An LD-resolvent of a well-typed query and a well-typed clause that is variable disjoint with it, is well-typed.

Proof. We reason as in the proof of Lemma 2.5. So it suffices to prove the following two claims.

Claim 1. An instance of a well-typed query (resp. clause) is well-typed.

Proof. Immediate by definition. □

Claim 2. Suppose $H, \mathbf{A}$ is a well-typed query and $H \leftarrow \mathbf{B}$ is a well-typed clause. Then $\mathbf{B}, \mathbf{A}$ is a well-typed query.

Proof. Let
$H = p(\mathbf{s} : \mathbf{S}, \mathbf{t} : \mathbf{T})$ and $\mathbf{B} = p_1(\mathbf{i_1} : \mathbf{I_1}, \mathbf{o_1} : \mathbf{O_1}), \ldots, p_m(\mathbf{i_m} : \mathbf{I_m}, \mathbf{o_m} : \mathbf{O_m})$. H is the first atom of a well-typed query, so it is correctly typed in its input positions, i.e.

$$\mathbf{s} \in \mathbf{S}. \tag{2}$$

$H \leftarrow \mathbf{B}$ is well-typed, so $\models \mathbf{s} : \mathbf{S}, \mathbf{o_1} : \mathbf{O_1}, \ldots, \mathbf{o_m} : \mathbf{O_m} \;\Rightarrow\; \mathbf{t} : \mathbf{T}$, and for $j \in [1, m]$ $\models \mathbf{s} : \mathbf{S}, \mathbf{o_1} : \mathbf{O_1}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \;\Rightarrow\; \mathbf{i_j} : \mathbf{I_j}$. By the Type Judgement Lemma 3.3(i) we get by virtue of (2)

$$\models \mathbf{o_1} : \mathbf{O_1}, \ldots, \mathbf{o_m} : \mathbf{O_m} \;\Rightarrow\; \mathbf{t} : \mathbf{T}, \tag{3}$$

and for $j \in [1, m]$

$$\models \mathbf{o_1} : \mathbf{O_1}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \;\Rightarrow\; \mathbf{i_j} : \mathbf{I_j}. \tag{4}$$

Now, let $\mathbf{A} = p_{m+1}(\mathbf{i_{m+1}} : \mathbf{I_{m+1}}, \mathbf{o_{m+1}} : \mathbf{O_{m+1}}), \ldots, p_n(\mathbf{i_n} : \mathbf{I_n}, \mathbf{o_n} : \mathbf{O_n})$. $H, \mathbf{A}$ is well-typed, so for $j \in [m+1, n]$

$$\models \mathbf{t} : \mathbf{T}, \mathbf{o_{m+1}} : \mathbf{O_{m+1}}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \;\Rightarrow\; \mathbf{i_j} : \mathbf{I_j},$$

and thus by (3) and the Type Judgement Lemma 3.3(ii) for $j \in [m+1, n]$

$$\models \mathbf{o_1} : \mathbf{O_1}, \ldots, \mathbf{o_{j-1}} : \mathbf{O_{j-1}} \;\Rightarrow\; \mathbf{i_j} : \mathbf{I_j}.$$

This and (4) imply the claim. □

□

This brings us to the following desired conclusions.

Theorem 3.9. Let P and Q be well-typed and let ξ be an LD-derivation of Q in P. All atoms selected in ξ are correctly typed in their input positions.

Proof. Note that the first atom of a well-typed query is correctly typed in its input positions and that a variant of a well-typed clause is well-typed. The conclusion now follows by Lemma 3.8. □

Corollary 3.10. Let P and Q be well-typed. Then for every computed answer substitution σ, $Q\sigma$ is well-typed in its output positions.

Proof. Let $\mathbf{o} : \mathbf{O}$ stand for the sequence of typed terms filling in the output positions of the atoms of Q. Let p be a new relation of arity equal to the length of $\mathbf{o} : \mathbf{O}$ and with all the positions moded as input and typed as $\mathbf{O}$. Then $Q, p(\mathbf{o} : \mathbf{O})$ is a well-typed query. Now, σ is a computed answer substitution for Q in P iff $p(\mathbf{o})\sigma$ is a selected atom in an LD-derivation of $Q, p(\mathbf{o})$ in P. The conclusion now follows by Theorem 3.9. □

Let us see now how these results can be applied to specific programs.

Example 3.11. Reconsider the program `quicksort`. We type it as follows:

```
qs(+:ListGae, -:ListGae),
part(+:Gae, +:ListGae, -:ListGae, -:ListGae),
```

```
app(+:ListGae, +:ListGae, -:ListGae),
>(+:Gae, +:Gae), ≤(+:Gae, +:Gae).
```

Conforming to Prolog behaviour, we assume that the evaluation of the tests $u > v$ and $u \leq v$ ends in an error if u or v are not gae's. It is easy to check that `quicksort` is then well-typed. Assume now that s is a list of gae's. By Theorem 3.9 we conclude that all atoms selected in the LD-derivations of `qs(s,t)` in `quicksort` are correctly typed in their input positions. In particular, when these atoms are of the form $u > v$ or $u \leq v$, both u and v are gae's. Thus the LD-derivations of `qs(s,t)` do not end in an error. Moreover, by Corollary 3.10 we conclude that all computed answer substitutions σ are such that $\mathtt{t}\sigma$ is a list of gae's. □

Thus, type analysis is sufficient to derive information about the types of atom arguments, before or after their selection. This is sufficient to prove absence of run-time errors in presence of relations involving arithmetic. Also, as shown in Apt and Etalle [AE93] (and on which this section is based), the types can be used to provide sufficient, decidable conditions under which in all program executions unification is equivalent to iterated matching.

4. Well-m-Asserted Programs

In order to prove more complex program properties, one can consider monotonic assertions formed in (an extension of) a first-order language. An assertion ϕ is *monotonic* if, for every substitution σ

$$\models \phi \Rightarrow \phi\sigma. \tag{5}$$

An assertional method to prove run-time properties of a program expressed by means of monotonic assertions was given in Bossi and Cocco [BC89], where the notion of a well-asserted program is introduced, here called a well-monotonically-asserted program, well-m-asserted program for short. A pair $(pre^p, post^p)$ of assertions (called pre- and post-condition), called specification, is associated with every relation p occurring in the program under consideration: pre^p describes properties of the arguments of p before its call, while $post^p$ describes properties of the arguments of p after its call. To denote arguments of a relation, the *assertion language for a program P* contains some special variables, namely, for every relation p defined in P, the variables $x_1^p, \ldots, x_n^p$ are considered, where n is the arity of p. These variables represent the arguments of the relation p, and are called *a-variables*. The set of a-variables occurring in a syntactic construct E is denoted by $a\text{-}vars(E)$.

Definition 4.1. (Specification) A *specification for an n-ary relation p* is a pair $(pre^p, post^p)$ of monotonic assertions, s.t. $a\text{-}vars(pre^p, post^p) \subseteq \{x_1^p, \ldots, x_n^p\}$. □

An *asserted program $\mathcal{A}P$* is obtained by assigning a specification to every relation of P. Sometimes we shall still write P instead of $\mathcal{A}P$. In the remainder of this section we adopt the following

Assumption 4.2. *Every relation* has a fixed specification associated with it.

Definition 4.3. Let $A = p(t_1, \ldots, t_n)$ and $\alpha = \{x_i^p/t_i \mid i \in [1,n]\}$. Define $pre(A) \stackrel{\text{def}}{=} pre^p\alpha$ and $post(A) \stackrel{\text{def}}{=} post^p\alpha$.

- We say that A *satisfies its precondition* if $\models pre(A)$.
- We say that A *satisfies its postcondition* if $\models post(A)$.

We use $post(A_1, \ldots, A_k)$ as a shorthand for $post(A_1) \wedge \ldots \wedge post(A_k)$, where we assume that for $k = 0$ $post(A_1) \wedge \ldots \wedge post(A_k)$ is equal to *true*.

Definition 4.4. (Well-m-Asserted)

- A query $p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n})$ is called *well-m-asserted* if for $j \in [1, n]$

$$\models post(p_1(\mathbf{s_1}), \ldots, p_{j-1}(\mathbf{s_{j-1}})) \Rightarrow pre(p_j(\mathbf{s_j})).$$

- A clause $p(\mathbf{s}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n})$ is called *well-asserted* if for $j \in [1, n+1]$

$$\models pre(p(\mathbf{s})) \wedge post(p_1(\mathbf{s_1}), \ldots, p_{j-1}(\mathbf{s_{j-1}})) \Rightarrow pre(p_j(\mathbf{s_j})),$$

  where $pre(p_{n+1}(\mathbf{s_{n+1}})) \stackrel{\text{def}}{=} post(p(\mathbf{s}))$.
- An asserted program $\mathcal{A}P$ is called *well-m-asserted* if all its clauses are. □

The following observation clarifies the relation between well-m-asserted and well-typed programs and queries.

Theorem 4.5. The notion of a well-typed program (query) is a special case of the notion of a well-m-asserted program (query).

Proof. It suffices to view a typed atom $p(\mathbf{x} : \mathbf{S}, \mathbf{y} : \mathbf{T})$ as a specification for the relation $p(\mathbf{x}, \mathbf{y})$ consisting of $pre^p = \mathbf{x} \in \mathbf{S}$ and $post^p = \mathbf{y} \in \mathbf{T}$. Then a program P is well-typed iff the corresponding asserted program is well-m-asserted. □

The following lemma shows persistence of the notion of being well-m-asserted.

Lemma 4.6. An LD-resolvent of a well-m-asserted query and a well-m-asserted clause that is variable disjoint with it, is well-m-asserted.

Proof. We reason as in the proof of Lemma 2.5. It suffices to prove the following two claims.

Claim 1. An instance of a well-m-asserted query (resp. clause) is well-m-asserted.

Proof. Immediate by the assumption that the assertions are monotonic. □

Claim 2. Suppose $H, \mathbf{A}$ is a well-m-asserted query and $H \leftarrow \mathbf{B}$ is a well-m-asserted clause. Then $\mathbf{B}, \mathbf{A}$ is a well-m-asserted query.

Proof. Let $H = p(\mathbf{s})$ and $\mathbf{B} = p_1(\mathbf{s_1}), \ldots, p_m(\mathbf{s_m})$. H is the first atom of a well-m-asserted query, so it satisfies its precondition, i.e.

$$\models pre(p(\mathbf{s})). \tag{6}$$

Then from the fact that $H \leftarrow \mathbf{B}$ is well-m-asserted and (6) it follows that

$$\models post(p_1(\mathbf{s_1}), \ldots, p_m(\mathbf{s_m})) \Rightarrow post(p(\mathbf{s})), \tag{7}$$

and for $j \in [1, m]$

$$\models post(p_1(\mathbf{s_1}), \ldots, p_{j-1}(\mathbf{s_{j-1}})) \Rightarrow pre(p(\mathbf{s_j})). \tag{8}$$

Now, let $\mathbf{A} = p_{m+1}(\mathbf{s_{m+1}}), \ldots, p_n(\mathbf{s_n})$. Then by $H, \mathbf{A}$ well-m-asserted and by (7) we have, for $j \in [m+1, n]$:

$$\models post(p_1(\mathbf{s_1}),\ldots,p_{j-1}(\mathbf{s_{j-1}})) \Rightarrow pre(p(\mathbf{s_j})). \tag{9}$$

Then by (8) and (9) we obtain that $\mathbf{B},\mathbf{A}$ is well-m-asserted. □
□

This yields the following conclusions.

Theorem 4.7. Let P and Q be well-m-asserted and let ξ be an LD-derivation of Q in P. All atoms selected in ξ satisfy their preconditions.

Proof. Note that the first atom of a well-m-asserted query satisfies its precondition and that a variant of a well-m-asserted clause is well-m-asserted. The conclusion now follows by Lemma 4.6. □

Corollary 4.8. Let P and Q be well-m-asserted. Then for every computed answer substitution σ, $\models post(Q)\sigma$.

Proof. Let $Q = p_1(\mathbf{s_1}),\ldots,p_k(\mathbf{s_k})$. Let p be a new relation of arity equal to the sum of the arities of $p_1, \ldots, p_k$, say n, and with pre_p and $post_p$ both equal to $post_{p_1}\alpha_1 \wedge \ldots \wedge post_{p_k}\alpha_k$, where each α_i renames the p_i-variables to a new set of p-variables. Then $Q, p(\mathbf{s_1},\ldots,\mathbf{s_k})$ is a well-m-asserted query. Now, σ is a computed answer substitution for Q in P iff $p(\mathbf{s_1},\ldots,\mathbf{s_k})\sigma$ is a selected atom in an LD-derivation of $Q, p(\mathbf{s_1},\ldots,\mathbf{s_k})$ in P. The conclusion now follows by Theorem 4.7. □

Again, let us show how these results can be applied to specific programs.

Example 4.9. Reconsider the program `quicksort`. We associate with its relations the following specifications:

$$\begin{array}{ll}
pre^{qs} = ListGae(x_1^{qs}); & post^{qs} = perm(x_1^{qs},x_2^{qs}), sorted(x_2^{qs});\\
pre^{app} = ListGae(x_1^{app},x_2^{app}); & post^{app} = conc(x_1^{app},x_2^{app},x_3^{app});\\
pre^{part} = ListGae(x_2^{part}), Gae(x_1^{part}); & post^{part} = \phi^{part};\\
pre^{>} = Gae(x_1^{>},x_2^{>}); & post^{>} = x_1^{>} > x_2^{>};\\
pre^{\leq} = Gae(x_1^{\leq},x_2^{\leq}); & post^{\leq} = x_1^{\leq} \leq x_2^{\leq};
\end{array}$$

where $perm(x,y)$ states that x, y are lists and y is a permutation of x, $sorted(x)$ states that x is a sorted list of gae's, $conc(x,y,z)$ states that x, y, z are lists and z is a concatenation of x and y, and

$$\begin{array}{rl}\phi^{part} = & ListGae(x_3^{part},x_4^{part}) \wedge (el(x_2^{part}) = el(x_3^{part}) \cup el(x_4^{part})) \wedge\\
& \forall x(x \in el(x_3^{part}) \Rightarrow x < x_1^{part}) \wedge\ \forall x(x \in el(x_4^{part}) \Rightarrow x \geq x_1^{part}),\end{array}$$

where for a list x, $el(x)$ denotes the set of its elements. It is easy to check that `quicksort` is then well-m-asserted. Assume now that s is a list of gae's. By Theorem 4.7 we conclude that the LD-derivations of `qs(s,t)` do not end in an error. Moreover, by Corollary 4.8 we conclude that all computed answer substitutions σ are such that $t\sigma$ is a sorted permutation of s. □

Thus, static analysis based on monotonic assertions is sufficient to prove monotonic run-time properties and partial correctness of programs. Also, as shown in Bossi, Cocco and Fabris [BCF91], monotonic assertions can be used in a method for proving program termination.

5. Well-*dot*-Asserted Programs

Certain properties are not expressible by means of monotonic assertions: for instance, some structural properties of a term t, like t being a variable, or t not being a ground term, or t sharing some variable with another term. The use of such run-time properties is relevant for e.g. program optimization; to determine for which class of queries the program terminates; or to describe the behaviour of a program containing some built-in predicates. In order to deal with these run-time properties, one can consider an assertion language containing also non-monotonic assertions.

In this section, an assertional method for proving run-time properties which employs non-monotonic assertions is described. This method was introduced in Drabent and Małuszyński [DM88]. The approach is analogous to that presented in the previous section, with the exception that here, due to the presence of non-monotonic assertions, the *assertion language for a program P* contains for every relation p occurring in P, the variables ${}^\bullet p_i$, called *input variables*, and $p_i^\bullet$, called *output variables*, for $i \in [1,n]$, where n is the arity of p. We call these variables a-variables: input variables represent the arguments of p at the moment of its call, while output variables represent the arguments of p after its call. The set of a-variables appearing in a syntactic construct E is denoted by *a-vars*(E). The assertion language also contains variables representing terms (meta variables), and terms of the object language.

Definition 5.1. (Specification) A *specification for an n-ary relation p* is a pair $(pre_p, post_p)$ of assertions, s.t. $a\text{-}vars(pre_p) \subseteq \{{}^\bullet p_1, \ldots, {}^\bullet p_n\}$ and $a\text{-}vars(post_p) \subseteq \{{}^\bullet p_1, \ldots, {}^\bullet p_n, p_1^\bullet, \ldots, p_n^\bullet\}$. □

An *asserted program $\mathcal{A}P$* is obtained by assigning a specification to every relation defined in P. Sometimes we shall still write P instead of $\mathcal{A}P$. In the remainder of this section we adopt the following

Assumption 5.2. *Every relation* has a fixed specification associated with it.

Before we define semantics of pre- and postconditions, we introduce the following notation.
For an atom $A = p(t_1, \ldots, t_n)$ let $pre(A)$ denote the pair (pre_p, α), where $\alpha = \{{}^\bullet p_i/t_i \mid i \in [1,n]\}$, and let $post(A, A\sigma)$ denote the pair $(post_p, \beta)$, where $\beta = \{{}^\bullet p_1/t_1, \ldots, {}^\bullet p_n/t_n,\ p_1^\bullet/(t_1\sigma), \ldots, p_n^\bullet/(t_n\sigma)\}$. We say that $pre(A)$ *is true*, and write $\models pre(A)$, if pre_p is true in any interpretation where the value of ${}^\bullet p_i$ is ${}^\bullet p_i\alpha$, for $i \in [1,n]$. Analogously we say that $post(A, A\sigma)$ *is true*, and write $\models post(A, A\sigma)$, if $post_p$ is true in any interpretation where the values of ${}^\bullet p_i$ and $p_i^\bullet$ are ${}^\bullet p_i\beta$ and $p_i^\bullet\beta$, respectively, for $i \in [1,n]$. We will often write (A,σ) instead of $(A, A\sigma)$.

Definition 5.3.
- We say that A *satisfies its precondition* if $\models pre(A)$.
- We say that (A,σ) *satisfies its postcondition* if $\models post(A,\sigma)$. □

The notation $post((A_1,\sigma_1), \ldots, (A_k,\sigma_k))$ is used as a shorthand for $post(A_1,\sigma_1) \wedge \ldots \wedge post(A_k,\sigma_k)$, where we assume that for $k = 0$ $post(A_1,\sigma_1) \wedge \ldots \wedge post(A_k,\sigma_k)$ is equal to *true*.

The following notion is central in the definition of a well-*dot*-asserted program.

Definition 5.4. (Valuation Sequence) We say that a sequence $\rho_0, \ldots, \rho_n$ of substitutions is a *valuation sequence* for a clause $p(\mathbf{s_0}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n})$ and an atom $p(\mathbf{t})$ if the following conditions are satisfied:

1. $vars(\mathbf{t}) \cap vars(\mathbf{s_0}, \ldots, \mathbf{s_n}) = \emptyset$;
2. $\rho_0 = mgu(p(\mathbf{t}), p(\mathbf{s_0}))$;
3. there exist $\sigma_1, \ldots, \sigma_n$ s.t. for all $i \in [1, n]$:

$\rho_i = \rho_{i-1}\sigma_i$,

$dom(\sigma_i) \subseteq vars(\mathbf{s_i}\rho_{i-1})$,

$range(\sigma_i) \cap vars((\mathbf{s_0}, \ldots, \mathbf{s_n})\rho_{i-1}) \subseteq vars(\mathbf{s_i}\rho_{i-1})$. □

The above definition describes a derivation for the atomic query $p(\mathbf{t})$, when the clause $p(\mathbf{s_0}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n})$ is chosen as first input clause. Notice that condition 1. expresses the requirement that the input clause and the query are standardized apart, while intuitively condition 3. defines σ_i to be an *abstraction* of a computed answer substitution for $p_{i-1}(\mathbf{s_{i-1}}\rho_{i-1})$. As in [DM88], we denote a query Q by the clause $goal \leftarrow Q$, where $goal$ is a new relation symbol, which is assumed to have both precondition and postcondition equal to *true*.

Definition 5.5. (Well-*dot*-Asserted)

- A clause $c : p(\mathbf{s_0}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n})$ is called *well-dot-asserted* if, for every atom $p(\mathbf{t})$ that satisfies its precondition and for every valuation sequence $\rho_0, \ldots, \rho_n$ for c and $p(\mathbf{t})$, for $j \in [1, n+1]$

$$\models post((p_1(\mathbf{s_1}\rho_0), \sigma_1), \ldots, (p_{j-1}(\mathbf{s_{j-1}}\rho_{j-2}), \sigma_{j-1})) \Rightarrow pre(p_j(\mathbf{s_j}\rho_{j-1})),$$

where $pre(p_{n+1}(\mathbf{s_{n+1}}\rho_n)) \stackrel{\text{def}}{=} post(p(\mathbf{t}), \rho_n)$.
- An asserted program $\mathcal{A}P$ is called *well-dot-asserted* if all its clauses are. □

Now we show that the notion of a well-m-asserted program is a special case of the notion of a well-*dot*-asserted program. To this end, we introduce a preliminary notion and a lemma.

Definition 5.6. (Simplified Form) A specification $(pre_p, post_p)$ is in *simplified form* if $vars(post_p) \cap \{{}^\bullet p_1, \ldots, {}^\bullet p_n\} = \emptyset$, where n is the arity of p. An asserted program is in *simplified form* if all its specifications are. □

In other words, a specification is in simplified form if its postcondition does not contain input variables. So for an atom $A = p(t_1, \ldots, t_n)$, we have that (A, σ) satisfies its postcondition if $\models (post_p, \beta)$, with $\beta = \{\, p_i^\bullet/(t_i\sigma) \mid i \in [1, n]\}$. Then we use the simpler notation $\models post(A\sigma)$.

The following expected property of monotonic assertions will be used.

Lemma 5.7. The truth of a monotonic assertion is invariant under renaming, i.e. if σ is a renaming then $\models \phi \Leftrightarrow \phi\sigma$.

Assume now that specifications are monotonic and in simplified form. Consider the map u which transforms a specification $(pre_p, post_p)$ into the specification $(pre^p, post^p)$ obtained replacing ${}^\bullet p_i$ and $p_i^\bullet$ with x_i^p, for $i \in [1, n]$. Notice that u is a bijection from specifications $(pre_p, post_p)$ in simplified form with monotonic assertions to specifications $(pre^p, post^p)$ used to define well-m-asserted programs.

Theorem 5.8. The notion of a well-m-asserted program is a special case of the notion of a well-*dot*-asserted program.

Proof. Let $\mathcal{A}P$ be an asserted program in simplified form and with monotonic assertions. Let $\mathcal{A}'P$ be the asserted program obtained by replacing every specification $(pre_p, post_p)$ of $\mathcal{A}P$ with $u(pre_p, post_p)$. We show that $\mathcal{A}P$ is well-*dot*-asserted iff $\mathcal{A}'P$ is well-m-asserted. Let $c : p(\mathbf{s_0}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n})$ be a clause of P.
Suppose that $\mathcal{A}P$ is well-*dot*-asserted. We prove that c is well-m-asserted.
Fix an arbitrary $i \in [1, n+1]$. Let α be s.t.

$$\models (pre(p(\mathbf{s_0})) \wedge post(p_1(\mathbf{s_1}), \ldots, p_{i-1}(\mathbf{s_{i-1}})))\alpha. \tag{10}$$

We show that $pre(p_i(\mathbf{s_i}))\alpha$ is true. Let $A \stackrel{\text{def}}{=} p(\mathbf{s_0}\alpha)$.
Consider the sequence $\rho_0, \ldots, \rho_n$, where $\rho_0 = \alpha_{|vars(\mathbf{s_0})}$ and $\rho_i = \alpha_{|vars(\mathbf{s_i})}$, for $i \in [1, n]$. By Lemma 5.7 we can assume $vars(\mathbf{s_0}\alpha) \cap vars(\mathbf{s_0}, \ldots, \mathbf{s_n}) = \emptyset$ without loss of generality. It is easy to check that $\rho_0, \ldots, \rho_n$ is a valuation sequence for A and c. Moreover, by (10) A satisfies its precondition. Since $\mathcal{A}P$ is well-*dot*-asserted and in simplified form, then by (10) we have that $pre(p_i(\mathbf{s_i}))\alpha$ is true.
Conversely, suppose that $\mathcal{A}'P$ is well-m-asserted.
We prove that c is well-*dot*-asserted. Let $p(\mathbf{t})$ be s.t.

$$\models pre(p(\mathbf{t})), \tag{11}$$

and let $\rho_0, \ldots, \rho_n$ be a valuation sequence for $p(\mathbf{t})$ and c. Then

$$\rho_0 = mgu(p(\mathbf{t}), p(\mathbf{s_0})). \tag{12}$$

Fix an arbitrary j in $[1, n+1]$. Let α be s.t.

$$\models \ post(p_1(\mathbf{s_1}\rho_1), \ldots, p_{j-1}(\mathbf{s_{j-1}}\rho_{j-1}))\alpha. \tag{13}$$

We show that $\models \ pre(p_j(\mathbf{s_j}\rho_{j-1}))\alpha$.
By the definition of valuation sequence, we have $\rho_{j-1} = \rho_k\sigma_{k+1}\ldots\sigma_{j-1}$, for $k \in [0, j-1]$. Then by (11), (12) and by (5) (i.e., by the definition of monotonic assertion) we have $\models \ pre(p(\mathbf{s_0})\rho_{j-1})\alpha$, hence by (13)
$\models \ (pre(p(\mathbf{s_0})\rho_{j-1}) \wedge post(p_1(\mathbf{s_1}\rho_{j-1}), \ldots, p_{j-1}(\mathbf{s_{j-1}}\rho_{j-1})))\alpha$.
Then the result follows from the fact that $\mathcal{A}'P$ is well-m-asserted. □

The following lemma shows persistence of the notion of being well-*dot*-asserted.

Lemma 5.9. An LD-resolvent of a well-*dot*-asserted query and a well-*dot*-asserted clause that is variable disjoint with it, is well-*dot*-asserted.

Proof. Let $c : p(\mathbf{s}) \leftarrow p_1(\mathbf{s_1}), \ldots, p_m(\mathbf{s_m})$ be a well-*dot*-asserted clause and let $Q = p(\mathbf{t}), p_{m+1}(\mathbf{s_{m+1}}), \ldots, p_n(\mathbf{s_n})$ be a well-*dot*-asserted query s.t. Q is variable disjoint with c. Let $\theta = mgu(p(\mathbf{t}), p(\mathbf{s}))$. Let $R = (p_1(\mathbf{s_1}), \ldots, p_n(\mathbf{s_n}))\theta$ be the resolvent of Q and c. Consider a valuation sequence $\rho_0, \ldots, \rho_n$ for R. Notice that ρ_0 is equal to the identity substitution ϵ. Fix a $j \in [1, n]$. Let α be s.t.

$$\models post((p_1(\mathbf{s_1})\theta\rho_0, \sigma_1), \ldots, (p_j(\mathbf{s_j})\theta\rho_{j-1}, \sigma_j))\alpha. \tag{14}$$

We show that $\models pre(p_{j+1}(\mathbf{s_{j+1}})\theta\rho_j)\alpha$. We distinguish the following two cases.
- $j \leq m-1$. By Q well-*dot*-asserted we have that $\models pre(p(\mathbf{t}))$. Then the sequence $\tau_1 = \rho_0^1, \ldots, \rho_m^1$ of substitutions is a valuation sequence for $p(\mathbf{t})$ and c, where $\rho_k^1 = \theta\rho_k$, for $k \in [0,m]$. Then the result follows from c well-*dot*-asserted. Moreover, from τ_1 valuation sequence for $p(\mathbf{t})$ and c it follows that

$$\models post((p_1(\mathbf{s_1})\theta\rho_0, \sigma_1), \ldots, (p_m(\mathbf{s_m})\theta\rho_{m-1}, \sigma_m)) \Rightarrow post(p(\mathbf{t}), \theta\rho_m). \quad (15)$$

- $m \leq j \leq n$. The sequence $\tau_2 = \rho_0^2, \ldots, \rho_n^2$ of substitutions is a valuation sequence for Q, where $\rho_0^2 = \epsilon$ and $\rho_k^2 = \theta\rho_{k+m-1}$ for $k \in [1,n]$. Then

$$\models post(\mathbf{t}, \theta\rho_m) \wedge post((p_{m+1}(\mathbf{s_{m+1}})\theta\rho_m, \sigma_m), \ldots, (p_j(\mathbf{s_j})\theta\rho_{j-1}, \sigma_j)) \Rightarrow pre(p_{j+1}(\mathbf{s_{j+1}})\theta\rho_j).$$

So the result follows from (15), and from (14). □

Theorem 5.10. Let P be and Q be well-*dot*-asserted and let ξ be an LD-derivation of Q in P. Then all atoms selected in ξ satisfy their preconditions.

Proof. Note that the first atom of a well-*dot*-asserted query satisfies its precondition and that a variant of a well-*dot*-asserted clause is correct. The conclusion now follows by Lemma 5.9. □

Corollary 5.11. Let P be a well-*dot*-asserted program in simplified form. If $p(\mathbf{s})$ satisfies its precondition then $\models post(p(\mathbf{s}), \sigma)$, for every computed answer substitution σ.

Proof. Consider the query $Q = p(\mathbf{s}), p'(\mathbf{s})$, where p' is a new relation of arity equal to the arity, say n, of p, and with $pre_{p'}$ equal to $post_p\alpha$, where α renames the output variables of p to a new set of input variables, and with $post_{p'}$ equal to $true$. It is easy to check that Q is a well-*dot*-asserted query. The result now follows from Theorem 5.10. □

Notice that in Corollary 5.11 the specifications are assumed to be in simplified form: this hypothesis allows us to give a simple proof of that result, as a corollary of Theorem 5.10. The proof of Corollary 5.11 in the general case (where specifications are not supposed to be in simplified form) is more technical and can be found in [DM88]. These results extend to programs containing some built-in relations. For instance the built-in relation var can be characterized by the specification $pre_{var} = true$ and $post_{var} = (var(var^\bullet) \wedge {}^\bullet var = var^\bullet)$, where the assertion $var(t)$ is true iff t is an object variable. We conclude this section illustrating how these results can be applied to our running example `quicksort`.

Example 5.12. Consider the following specifications which are in simplified form:

$$\begin{array}{ll}
pre_{qs} = ListGae({}^\bullet qs_1), var({}^\bullet qs_2); & post_{qs} = perm(qs_1^\bullet, qs_2^\bullet), sorted(qs_2^\bullet); \\
pre_{app} = ListGae({}^\bullet app_1, {}^\bullet app_2); & post_{app} = conc(app_1^\bullet, app_2^\bullet, app_3^\bullet); \\
pre_{part} = ListGae({}^\bullet part_2), Gae({}^\bullet part_1); & post_{part} = \phi_{part}; \\
pre_{>} = Gae({}^\bullet >_1, {}^\bullet >_2); & post_{>} = >_1^\bullet > >_2^\bullet; \\
pre_{\leq} = Gae({}^\bullet \leq_1, {}^\bullet \leq_2); & post_{\leq} = \leq_1^\bullet \leq \leq_2^\bullet;
\end{array}$$

where

$$\phi_{part} = ListGae(part_3^\bullet, part_4^\bullet) \wedge (el(part_2^\bullet) = el(part_3^\bullet) \cup el(part_4^\bullet)) \wedge$$

$$\forall x(x \in el(part_3^\bullet) \Rightarrow x < part_1^\bullet) \wedge\ \forall x(x \in el(part_4^\bullet) \Rightarrow x \geq part_1^\bullet),$$

where $perm(x, y)$, $sorted(x)$ and $conc(x, y, z)$, and $el(x)$ are defined as in Example 4.9. It is not difficult to check that `quicksort` is well-*dot*-asserted. Assume now that `s` is a list of gae's and that `x` is a variable. By Theorem 4.7 we conclude that in all LD-derivations of `qs(s,x)` whenever qs is called, its second argument is a variable. Moreover, by Corollary 5.11 we conclude that all computed answer substitutions σ are such that $x\sigma$ is a sorted permutation of s. □

Thus, static analysis based on non-monotonic assertions associated with relations is sufficient to derive information about the form of individual atom arguments (like being a variable), before or after their execution. This type of run-time properties can be used for program optimization.

6. Proving Global Properties of Prolog Programs

In the methods presented in the two previous sections, specifications are associated with the relations occurring in the program. As a consequence one cannot express global run-time properties, describing relationships among the atoms of a query during its execution. For instance, consider the query $Q = p(x), p(y)$, and assume that during the execution of Q, x and y are always bound to terms which do not have variables in common. This property cannot be expressed in the previous approaches, because it is a property of the query, and not of the individual atoms of the query. To allow global analysis of programs, one can associate assertions with program points. An assertion describes then the values of the variables of the program when the computation reaches the relative program point. This can be done by annotating the clauses with assertions, as in $H \leftarrow \{I_0\}B_1\{I_1\}\dots B_n\{I_n\}$. This method has been proposed by Colussi and Marchiori in [CM91]. Since a special substitution ρ is used in the verification condition of this method, we call here ρ-well-asserted programs those asserted programs which satisfy the method. We show by means of an example that the notion of ρ-well-asserted program allows also to prove some non-monotonic local properties which one cannot prove by means of the notion of well-*dot*-asserted program. Moreover, we introduce a simple method to prove global run-time properties of programs, based on the notion of *well-asserted program*, and prove results analogous to those given in the previous sections. We show that the notion of well-asserted program is simpler, yet less expressive, than the notion of ρ-well-asserted program. However, the question if the notion of well-*dot*-asserted program is a special case of a ρ-well-asserted program remains to be investigated.

The *assertion language for a program P* contains the variables of the program and all their renamings. It is assumed that assertions are semantically invariant w.r.t. renaming, i.e. $\models \phi \Leftrightarrow \phi\sigma$, for every assertion ϕ and renaming σ. As in the previous section, we denote a query Q by the clause $goal \leftarrow Q$, where $goal$ is a new relation symbol.

Definition 6.1. (Asserted Program)

- An *asserted clause* $\mathcal{A}c$ is defined as $H \leftarrow \{I_0\}B_1\{I_1\}\dots B_n\{I_n\}$, where $c = H \leftarrow B_1, \dots, B_n$, and $I_0, \dots, I_n$ are assertions. A formula $\{I_{i-1}\}B_i\{I_i\}$ is called a *specification*.
- An *asserted program* $\mathcal{A}P$ is a set of asserted clauses, one for each clause of P. □

Sometimes we shall still write P instead of $\mathcal{A}P$. In the remaining of this section we adopt the following

Assumption 6.2. *Every program* is annotated by means of a fixed set of assertions.

Informally, an asserted program is correctly asserted if for every clause c, the assertions I_0, I_1, ..., I_n associated with c are proven to be global invariants. In order to prove global invariance, unification is described by means of a predicate relation as follows.

Definition 6.3. (The Relation $\{\phi\}\,\mathcal{U}\,\{\psi\}$) Let $\mathcal{U}$ be a a set of pairs of terms or of pairs of atoms and let ϕ and ψ be assertions. Then $\{\phi\}\,\mathcal{U}\,\{\psi\}$ holds iff for all substitutions α such that $\phi\alpha$ is true, whenever there exists $\mu = mgu(\mathcal{U}\alpha)$ then $\psi\alpha\mu$ is true. □

A unifier for $\mathcal{U}$ is a substitution which unifies every pair of $\mathcal{U}$, while β is an mgu of $\mathcal{U}$ if it is a unifier and for every other unifier α of $\mathcal{U}$, we have that $\alpha = \beta\gamma$, for some substitution γ.

In [CM91], a sound proof-system for deriving $\{\phi\}\,\mathcal{U}\,\{\psi\}$ is given. In [CM93], a specific assertion language is considered and a sound and complete calculus is introduced, which computes a strongest (w.r.t. implication) assertion ψ such that $\{\phi\}\,\mathcal{U}\,\{\psi\}$ holds, i.e., a strongest postcondition of $\mathcal{U}$ w.r.t. the precondition ϕ.

The following definition of *matches* is central in the concept of ρ-well-asserted program. First some useful notions are introduced. Let α be a substitution. Then:

- $free(X;Y)\alpha$ iff $\forall x \in X, \forall y \in Y(var(x\alpha) \wedge (x \neq y \Rightarrow x\alpha \notin vars(y\alpha)))$.

So, $free(X;Y)\alpha$ is true when $X\alpha$ is a set of distinct variables, which do not occur in the terms of $(Y \setminus X)\alpha$.

For an assertion ϕ let $\phi^{x_1,\ldots,x_n}_{s_1,\ldots,s_n}$ denote the assertion obtained from ϕ by simultaneously replacing every occurrence of the x_i's with the s_i's.

With a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, we associate the set of pairs of terms $\mathcal{E}_\theta = \{(x_1,t_1),\ldots,(x_n,t_n)\}$. For a specification $spec = \{pre\}A\{post\}$, those variables which occur free in *spec* but do not occur in A are called *auxiliary variables*, denoted $aux(spec)$. More generally we write $aux_V(E)$ to denote the set of variables that occur free in V but do not occur in E.

Definition 6.4. (Matches) Let $spec = \{pre\}A\{post\}$ be a specification and let $\mathcal{A}c = H \leftarrow \{I_0\}B_1\ \{I_1\}\ldots B_n\{I_n\}$ an asserted clause. We say that *spec matches* $\mathcal{A}c$ if there exist: a variant $\mathcal{A}c'$ of $\mathcal{A}c$, two disjoint sets of variables X, Y, and a substitution ρ such that

1. $X \supseteq vars(spec)$,
2. $Y \supseteq vars(\mathcal{A}c')$,
3. $dom(\rho) \subseteq aux_Y(c')$ and
4. $range(\rho) \supseteq aux_X(A)$

and such that

$$\{pre \wedge\ free(Y; X \cup Y)\}\ \mathcal{U}\ \{I'_0\} \qquad DOWN$$

$$\{I'_n \wedge\ free(X; X \cup Y)\}\ \mathcal{U}\ \{post\} \qquad UP$$

where $\mathcal{U} = \{(A,H')\} \cup \mathcal{E}_\rho$. □

We say that a specification *spec* matches the asserted program $\mathcal{A}P$ if *spec* matches every asserted clause of $\mathcal{A}P$.

Definition 6.5. (ρ-well-asserted program) Let $\mathcal{A}P$ be an asserted program. We say that $\mathcal{A}P$ is *ρ-well-asserted* if every its specification matches $\mathcal{A}P$. □

In [CM91] the soundness of this method with respect to LD-resolution is proven. More in particular, the authors show that if a program is ρ-well-asserted, then the assertions which decorate the program are global invariants when LD-resolution is considered as computational mechanism.

The following simple example shows that the above notion allows also to prove *local* non-monotonic properties which cannot be proven using the notion of well-*dot*-asserted program.

Example 6.6. Consider the following asserted program $\mathcal{A}P$:

$$\begin{array}{ll} & \texttt{goal} \leftarrow \{share(X,Y)\}\ \texttt{p(X,Y)}\ \{share(X,Y)\}. \\ \mathcal{A}c: & \texttt{p(V,W)} \leftarrow \{\neg ground(V)\}\ \texttt{q(V)}\ \{share(V,W)\}. \\ & \texttt{q(Z)} \leftarrow \{share(Z,Wc)\}. \end{array}$$

Then it is easy to verify that $\mathcal{A}P$ is ρ-well-asserted, where the substitution $\rho = \{Wc/W\}$ can be used when proving that $\{\neg ground(V)\}q(V)\{share(V,W)\}$ matches $\mathcal{A}P$.

Now, a local property of P which is implied by the ρ-well-assertedness of $\mathcal{A}P$ can be expressed by means of the following specifications for p and q:

$$pre_p = share({}^\bullet p_1, {}^\bullet p_2),\ post_p = share(p_1^\bullet, p_2^\bullet),$$
$$pre_q = \neg ground({}^\bullet q),\ post_q = \neg ground(q^\bullet).$$

One can prove that P with the above specifications is not well-*dot*-asserted. In fact, consider the sequence:

$$\rho_0 = \{V/f(W1,W2), W/f(W1,W3)\},$$
$$\rho_1 = \rho_0\sigma_1,$$

where $\sigma_1 = \{W1/a\}$ and a is a constant. Then ρ_0, ρ_1 is a valuation sequence for c and $p(f(W1,W2), f(W1,W3))$. Moreover:

- $pre(p(f(W1,W2), f(W1,W3)))$ is true;
- $pre(q(V)\rho_0)$ is true;
- $post(q(V)\rho_0, \sigma_1)$ is true but $post(p(f(W1,W2), f(W1,W3)), \rho_1)$ is false.

□

We introduce now a simpler method to prove global run-time properties, based on the notion of *well-asserted program*. The definition of a well-asserted program uses the following concept of *agreement*. First some useful assertions are introduced. Let α be a substitution. Then

- $share(r,s)\alpha$ iff $vars(r\alpha) \cap vars(s\alpha) \neq \emptyset$;
- $(inst_A(r,s))\alpha$ iff $r\alpha = s\alpha\beta$, for some β s.t. $dom(\beta) \subseteq vars(A\alpha)$.

Definition 6.7. (Agreement) Let $spec = \{pre\}A\{post\}$ be a specification and let $\mathcal{A}c$ be an asserted clause. We say that *spec agrees with* $\mathcal{A}c$ if there exists a variant $\mathcal{A}c' = H \leftarrow \{I_0\}B_1\ \{I_1\} \ldots B_n\{I_n\}$ of $\mathcal{A}c$, which is variable disjoint with *spec*, s.t. the following conditions are satisfied:

$$\{pre \wedge\ free(Y; X \cup Y)\}\ (A,H)\ \{I_0\}, \qquad \textbf{CALL}$$

$$(I_n \wedge A = H \wedge \exists \mathbf{x}'(inst_{A'}(\mathbf{x}, \mathbf{x}') \wedge pre^{\mathbf{x}}_{\mathbf{x}'}) \;\Rightarrow\; post, \qquad \textbf{EXIT}$$

where $Y = vars(\mathcal{A}c')$, $X = vars(spec)$,
$\mathbf{x} = X \setminus \{x \in X \mid pre \Rightarrow \neg share(y, x),\;$ for all y occurring in $A\}$,
$\mathbf{x}'$ is a variant of $\mathbf{x}$ consisting of fresh variables, and A' denotes $A^{\mathbf{x}}_{\mathbf{x}'}$.

We say that a specification *spec* agrees with an asserted program $\mathcal{A}P$ if *spec* agrees with every asserted clause of $\mathcal{A}P$. □

- CALL says that if the precondition *pre* of A is satisfied when A calls $\mathcal{A}c'$ then I_0 is satisfied;
- EXIT says that if I_n is satisfied when the execution of $\mathcal{A}c'$ reaches its exit, then *post* is satisfied. However, since the variables of *spec* do not occur in $\mathcal{A}c'$, the equation $A = H$ is used to recover information on the variables of A. Moreover, the precondition *pre* is used to recover information on variables of *spec* which are not in A: due to the non-monotonicity of the assertions, information given by *pre* about the variables in $\mathbf{x}$ (i.e., the variables of A and those variables which can share with some variable of A) is not anymore valid. Therefore, $\mathbf{x}$ is replaced by $\mathbf{x}'$, and $inst_{A'}(\mathbf{x}, \mathbf{x}')$ is used to specify their relationship.

Definition 6.8. (Well-Asserted) We say that $\mathcal{A}P$ is *well-asserted* if every specification of it agrees with $\mathcal{A}P$. □

As already mentioned at the beginning of this section, the notion of well-asserted program is simpler, though less expressive, than the notion of ρ-well-asserted program. In fact, it is easy to check that there is no assertion I s.t.

$$\texttt{goal} \leftarrow \{share(X,Y)\}\ \texttt{p(X,Y)}\ \{share(X,Y)\}.$$
$$\texttt{p(V,W)} \leftarrow \{\neg ground(V)\}\ \texttt{q(V)}\ \{share(V,W)\}.$$
$$\texttt{q(Z)} \leftarrow \{I\}.$$

is well-asserted.

Notice that, while in the definition of well-assertedness of the two previous methods the clauses of the program are examined independently, here all the clauses of the program are examined. In order to show the persistence of the notion of being well-asserted, due to the global character of the method, we need to reason in the context of a specific well-asserted program. Therefore we introduce the notion of asserted compound query. We give first some preliminary terminology. For a variant $\mathcal{A}c = H \leftarrow \{I_0\}A_1\{I_1\}\ldots A_m\{I_m\}$ of an asserted clause we call *a suffix of* $\mathcal{A}c$ any asserted query $\mathcal{A}Q = \{I_{i-1}\}A_i\{I_i\}\ldots A_m\{I_m\}$, with $i \in [1, m+1]$, and we refer to $\mathcal{A}c$ as *the asserted clause of* $\mathcal{A}Q$. Moreover, we denote $\{I_{i-1}\}$ by $pre(\mathcal{A}Q)$ and $\{I_m\}$ by $post(\mathcal{A}Q)$. Notice that $pre(\mathcal{A}Q) = post(\mathcal{A}Q)$ if $\mathcal{A}Q = \{I_m\}$.

Definition 6.9. (Asserted Compound Query) An *asserted compound query (for $\mathcal{A}P$)* is a pair $\mathcal{A}S = (\alpha, \Gamma)$ consisting of a substitution α and a sequence

$$\Gamma = (\langle \mathcal{A}Q_1, B_1, \phi_1\rangle, \ldots, \langle \mathcal{A}Q_{n-1}, B_{n-1}, \phi_{n-1}\rangle, \langle \mathcal{A}Q_n, goal, true\rangle),$$

s.t. $n \geq 1$, and for $i \in [1, n-1]$, $\{\phi_i\}B_i\mathcal{A}Q_{i+1}$ is a suffix of (a variant of) an asserted clause of $\mathcal{A}P$, and $\mathcal{A}Q_n$ is a suffix of the asserted goal-clause of $\mathcal{A}P$. □

The intuition behind the above definition is illustrated by an example after Definition 6.11. We now introduce the notion of an asserted resolvent and derivation.

Definition 6.10. (Asserted Resolvent) Let $\mathcal{A}S = (\alpha, \Gamma)$ be an asserted compound query, with $\Gamma = (\gamma_1, \ldots, \gamma_n)$, $n \geq 1$, where $\gamma_i = \langle \mathcal{A}Q_i, B_i, \phi_i \rangle$, for $i \in [1, n]$. Let $\mathcal{A}c = H \leftarrow \{I_0\}A_1 \ldots A_m\{I_m\}$ be a variant of an asserted clause.
- If $\mathcal{A}Q_1 = \{pre\}A\{post\}A_{m+1}\{I_{m+1}\} \ldots A_n\{I_n\}$ then

$$(\alpha\theta, (\langle \{I_0\}A_1 \ldots A_m\{I_m\}, A, pre\rangle, \langle \{post\}A_{m+1} \ldots A_n\{I_n\}, B_1, \phi_1\rangle, \gamma_2, \ldots, \gamma_n))$$

is called an *asserted resolvent of* $\mathcal{A}S$ *and* $\mathcal{A}c$, with $\theta = mgu(H, A\alpha)$;

- If $\mathcal{A}Q_1$ is an assertion and $n > 1$ then $(\alpha, (\gamma_2, \ldots, \gamma_n))$ is called an *asserted resolvent of* $\mathcal{A}S$. □

From now on we denote by $goal \leftarrow \mathcal{A}Q$ the asserted goal-clause of $\mathcal{A}P$.

Definition 6.11. (Asserted Derivation) An *asserted LD-derivation* $\mathcal{A}\xi$ *of* $goal \leftarrow \mathcal{A}Q$ *w.r.t.* α is a maximal sequence $\mathcal{A}R_0, \mathcal{A}R_1, \ldots$ of asserted compound queries, s.t. $\mathcal{A}R_0 = (\alpha, (\langle \mathcal{A}Q, goal, true\rangle))$, $\mathcal{A}R_{i+1}$ is an asserted resolvent of $\mathcal{A}R_i$, for $i \geq 0$, and all the variants of asserted clauses used are standardized apart, i.e., they are variable disjoint between each other and with $\mathcal{A}Q$, as well as with $vars(\alpha)$. □

We give now a simple example of asserted derivation. Let

$\mathcal{A}g =$ `goal` $\leftarrow \{I_0\}$ `p(X)` $\{I_1\}$ `q(X)` $\{I_2\}$.
$\mathcal{A}c_1 =$ `p(f(Y))` $\leftarrow \{I_0^1\}$ `r(Y)` $\{I_1^1\}$.
$\mathcal{A}c_2 =$ `r(a)` $\leftarrow \{I_0^2\}$.
$\mathcal{A}c_3 =$ `q(Z)` $\leftarrow \{I_0^3\}$.

The following is an asserted derivation for $\mathcal{A}g$ w.r.t. $\alpha = \{x/f(a)\}$.
$\mathcal{A}R_0 = (\alpha, (\langle \{I_0\}p(x)\{I_1\}q(x)\{I_2\},\ goal,\ true\rangle))$,
$\mathcal{A}R_1 = (\beta_1, (\langle \{I_0^1\}r(y)\{I_1^1\},\ p(x),\ \{I_0\}\rangle,\ \langle \{I_1\}q(x)\{I_2\},\ goal,\ true\rangle))$,
$\mathcal{A}R_2 = (\beta_1, (\langle \{I_0^2\},\ r(y),\ \{I_0^1\}\rangle,\ \langle \{I_1^1\},\ p(x),\ \{I_0\}\rangle,\ \langle \{I_1\}q(x)\{I_2\},\ goal,\ true\rangle))$,
$\mathcal{A}R_3 = (\beta_1, (\langle \{I_1^1\},\ p(x),\ \{I_0\}\rangle,\ \langle \{I_1\}q(x)\{I_2\},\ goal,\ true\rangle))$,
$\mathcal{A}R_4 = (\beta_1, (\langle \{I_1\}q(x)\{I_2\},\ goal,\ true\rangle))$.
$\mathcal{A}R_5 = (\beta_2, (\langle \{I_0^3\},\ q(x),\ \{I_1\}\rangle,\ \langle \{I_2\},\ goal,\ true\rangle))$,
$\mathcal{A}R_6 = (\beta_2, (\langle \{I_2\},\ goal,\ true\rangle))$,

where $\beta_1 = \{x/f(a), y/a\}$ and $\beta_2 = \{x/f(a), y/a, z/f(a)\}$. The following notion of well-asserted compound query clarifies the role of the B_i's together with their preconditions ϕ_i's: they are used in an asserted compound query for relating the $post(\mathcal{A}Q_i)$'s with the $pre(\mathcal{A}Q_{i+1})$'s.

Definition 6.12. Let $\mathcal{A}S$ be as in Definition 6.9.

- We say that $\mathcal{A}S$ *is well-asserted* if:

  1. each specification occurring in Γ agrees with $\mathcal{A}P$;
  2. for $i \in [1, n-1]$,

$$(post(\mathcal{A}Q_i) \wedge B_i = H_i \wedge \exists \mathbf{x_i'}(inst_{B_i'}(\mathbf{x_i}, \mathbf{x_i'}) \wedge \phi_i{}^{\mathbf{x_i}}_{\mathbf{x_i'}})) \Rightarrow pre(\mathcal{A}Q_{i+1}),$$

  where H_i is the head of the asserted clause of $\mathcal{A}Q_i$, $\mathbf{x_i} = vars(\phi_i) \setminus \{x \mid \phi_i \Rightarrow \neg share(x, y)\ \textit{for all } y \textit{ occurring in } B_i\}$, $\mathbf{x_i'}$ is a variant of $\mathbf{x_i}$ consisting of fresh variables, and B_i' denotes $B_i{}^{\mathbf{x_i}}_{\mathbf{x_i'}}$.

- We say that $\mathcal{A}S$ *satisfies its precondition* if

$$(pre(\mathcal{A}Q_1) \wedge (\wedge_{i\in[1,n-1]}(B_i = H_i \wedge \exists \mathbf{x_i'}(inst_{B_i'}(\mathbf{x_i}, \mathbf{x_i'}) \wedge \phi_i{}^{\mathbf{x_i}}_{\mathbf{x_i'}})))\alpha$$

is true, with H_i, $\mathbf{x_i}$ and $\mathbf{x_i'}$ defined as above, and where $\wedge_{i\in[1,0]}\phi_i \stackrel{\text{def}}{=} true$. □

From now on we assume that $\mathcal{A}P$ is a well-asserted program. The following result is a counterpart of Lemmata 4.6 and 5.9.

Lemma 6.13. Let $\mathcal{A}\xi$ be an asserted LD-derivation of $goal \leftarrow \mathcal{A}Q$ w.r.t. α. Let $\mathcal{A}R$ be an asserted compound query of $\mathcal{A}\xi$. Suppose that $\mathcal{A}R$ is well-asserted and that $\mathcal{A}R$ satisfies its precondition. Let $\mathcal{A}R'$ be the asserted resolvent of $\mathcal{A}R$ in $\mathcal{A}\xi$. Then $\mathcal{A}R'$ is well-asserted and it satisfies its precondition.

Proof. By the definition of a well-asserted compound query and the fact that a variant of a well-asserted program is well-asserted, it follows that $\mathcal{A}R'$ is well-asserted.

Now, let $\mathcal{A}R = (\alpha, (\gamma_1, \ldots, \gamma_k))$, $k > 1$, where $\gamma_i = \langle \mathcal{A}Q_i, B_i, \phi_i\rangle$, for $i \in [1, k]$, and let $\mathcal{A}R' = (\beta, \Gamma')$. We distinguish the two cases of Definition 6.10.

- $\mathcal{A}Q_1 = \{pre\}A\{post\}A_{m+1}\{I_{m+1}\}\ldots A_n\{I_n\}$. Then for some asserted input clause $\mathcal{A}c = H \leftarrow \{I_0\}A_1 \ldots A_m\{I_m\}$, β and Γ' are defined as specified by the first case of Definition 6.10. Since $\mathcal{A}R$ satisfies its precondition, then $pre\alpha$ is true. Moreover, by definition of asserted derivation it follows that $free(Y; X\alpha \cup Y)$ is true, with $Y = vars(\mathcal{A}c)$ and $X = vars(\{pre\}A\{post\})$. Then by CALL we have that $I_0\alpha\theta$ is true. Moreover, $H\alpha\theta = H\theta$, $H\theta = A\alpha\theta$, and $pre^{\mathbf{x}}_{\mathbf{x}'}\alpha\theta = pre^{\mathbf{x}}_{\mathbf{x}'}\alpha$. Then $\exists \mathbf{x}'(inst_{A'}(\mathbf{x}, \mathbf{x}') \wedge pre^{\mathbf{x}}_{\mathbf{x}'})\alpha\theta$ is true by choosing $\mathbf{x}'$ equal to $\mathbf{x}\alpha$. Moreover, for $i \in [1, k-1]$, from $H_i\alpha = B_i\alpha$ it follows that $H_i\alpha\theta = B_i\alpha\theta$ and from the standardization apart it follows that $\phi_i{}^{\mathbf{x_i}}_{\mathbf{x_i'}}\alpha\theta = \phi_i{}^{\mathbf{x_i}}_{\mathbf{x_i'}}\alpha$. Let $\phi = (A = H \wedge \exists \mathbf{x}'(inst_{A'}(\mathbf{x}, \mathbf{x}') \wedge pre^{\mathbf{x}}_{\mathbf{x}'}))$. Then

$$(I_0 \wedge \phi \wedge (\wedge_{i\in[1,k-1]}(B_i = H_i \wedge \exists \mathbf{x_i'}(inst_{A'}(\mathbf{x_i}, \mathbf{x_i'}) \wedge \phi_i{}^{\mathbf{x_i}}_{\mathbf{x_i'}})))\alpha\theta$$

is true, hence $\mathcal{A}R'$ satisfies its precondition.

- $\mathcal{A}Q_1$ is an assertion. Then $\beta = \alpha$ and $\Gamma' = (\gamma_2, \ldots, \gamma_k)$ and the conclusion follows by the hypothesis that $\mathcal{A}R$ is well-asserted and it satisfies its precondition. □

This yields the following conclusions.

Theorem 6.14. Let $\mathcal{A}Q = \{I_0\}A_1 \ldots A_n\{I_n\}$. Let α be s.t. $I_0\alpha$ is true. Let $\mathcal{A}\xi$ be an asserted derivation of $goal \leftarrow \mathcal{A}Q$ w.r.t. α. Then all the asserted compound queries of $\mathcal{A}\xi$ satisfy their preconditions.

Proof. Note that $(\alpha, (\langle \mathcal{A}Q, goal, true\rangle))$ is well-asserted and satisfies its precondition and that a variant of a well-asserted program is well-asserted. The conclusion now follows by Lemma 6.13. □

Corollary 6.15. Let $\mathcal{A}Q = \{I_0\}A_1 \ldots A_n\{I_n\}$ and let α be s.t. $I_0\alpha$ is true. Then for every computed answer substitution σ of $Q\alpha$ in P we have that $I_n\alpha\sigma'$ is true, where $\sigma' = (\sigma\rho)_{|dom(\sigma)}$, for some renaming ρ.

Proof. Let p be a new relation symbol and consider the program $\mathcal{A}P'$ obtained replacing $goal \leftarrow \mathcal{A}Q$ with $goal \leftarrow \mathcal{A}Q'$, where $\mathcal{A}Q' = \{I_0\}A_1 \ldots A_n\{I_n\}p\{true\}$.

Let $\mathcal{AS} = (\alpha, (\langle \mathcal{A}Q', goal, true\rangle))$. Then $\mathcal{AP}'$ is well-asserted and $\mathcal{AS}$ is well-asserted and it satisfies its precondition. Now, if σ is a computed answer of $Q\alpha$ in P then there exists an asserted derivation $\mathcal{A}\xi$ of $goal \leftarrow \mathcal{A}Q'$ w.r.t. α such that $(\alpha\theta_1 \ldots \theta_k, (\langle \{I_n\}p\{true\}, goal, true\rangle))$ is an asserted resolvent of $\mathcal{A}\xi$, where $(\theta_1 \ldots \theta_k)_{|vars(Q\alpha)} = \sigma'$, with $\sigma' = (\sigma\rho)_{|dom(\sigma)}$, for some renaming ρ. Then by Theorem 6.14 we have that $I_n\alpha\theta_1 \ldots \theta_k$ is true. So the conclusion follows because $I_n\alpha\theta_1 \ldots \theta_k = I_n\alpha\sigma'$. □

We conclude this last section by illustrating how these results can be applied to our running example `quicksort`.

Example 6.16. Reconsider the program `quicksort` augmented with the goal-clause `goal` $\leftarrow$ `qs(X,Y)`, asserted as follows.

```
goal ← {ListGae(X), free(Y)} qs(X,Y) {ListGae(X,Y)}.
qs([X|Xs],Ys) ←
    {I^1 ∧ free(Ys, Littles, Bigs, Ls, Bs)} part(X,Xs,Littles,Bigs)
    {I^1 ∧ ListGae(Littles, Bigs) ∧ free(Ys, Ls, Bs)} qs(Littles,Ls)
    {I^1 ∧ ListGae(Littles, Bigs, Ls) ∧ free(Ys, Bs)} qs(Bigs,Bs)
    {I^1 ∧ ListGae(Littles, Bigs, Ls, Bs) ∧ free(Ys)} app(Ls,[X|Bs],Ys)
    {I^1 ∧ ListGae(Littles, Bigs, Ls, Ys, Bs)}.
qs([],[]) ← {true}.

part(X,[Y|Xs],[Y|Ls],Bs) ←
    {Gae(X,Y) ∧ ListGae(Xs) ∧ free(Ls, Bs)} X > Y
    {Gae(X,Y) ∧ ListGae(Xs) ∧ free(Ls, Bs)} part(X,Xs,Ls,Bs)
    {Gae(X,Y) ∧ ListGae(Xs, Ls, Bs)}.
part(X,[Y|Xs],Ls,[Y|Bs]) ←
    {Gae(X,Y) ∧ ListGae(Xs) ∧ free(Ls, Bs)} X ≤ Y
    {Gae(X,Y) ∧ ListGae(Xs) ∧ free(Ls, Bs)} part(X,Xs,Ls,Bs)
    {Gae(X,Y) ∧ ListGae(Xs, Ls, Bs)}.
part(X,[],[],[]) ← {Gae(X)}.

app([X|Xs],Ys,[X|Zs]) ←
    {I^6 ∧ free(Zs)} app(Xs,Ys,Zs)
    {I^6 ∧ ListGae(Zs)}.
app([],Ys,Ys) ← {ListGae(Ys)}.
```

where $I^1 = Gae(X) \wedge ListGae(Xs)$ and $I^6 = Gae(X) \wedge ListGae(Xs, Ys)$. Here $ListGae(x_1, \ldots, x_n)$ is an abbreviation for $ListGae(x_1) \wedge \ldots \wedge ListGae(x_1)$ and $free(x_1, \ldots, x_n)$ is an abbreviation $free(x_1; X) \wedge \ldots \wedge free(x_n; X)$, where $X = \{x_1, \ldots, x_n\}$. It is not difficult to check that `quicksort` is well-asserted. Then by Theorem 6.14 we have that `qs(Littles,Ls)` and `qs(Bigs,Bs)` do not share variables during the execution of `goal`. Hence they can be executed in parallel. □

Thus, global analysis based on non-monotonic assertions is sufficient to prove global run-time properties of programs. This could be used, for example, for identifying which parts of a program can be executed in parallel. Also, as shown in Colussi and Marchiori [CM91], this method can be extended to prove total correctness, i.e., partial correctness and termination, of Prolog programs in presence of various built-in's.

Acknowledgements We thank Annalisa Bossi, Nicoletta Cocco, Livio Colussi, Włodek Drabent, Jan Rutten and Frank Teusink for useful discussions. This research was partly supported by the ESPRIT BRA 6810 (Compulog 2).

References

[AE93] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, Lecture Notes in Computer Science, pages 1–19, Berlin, 1993. Springer-Verlag.

[AP94] K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Toplas*, 1994. In print.

[Apt90] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.

[BC89] A. Bossi and N. Cocco. Verifying correctness of logic programs. In *Proceedings of Tapsoft '89*, pages 96–110, 1989.

[BCF91] A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In *Proceedings of Tapsoft '91*, pages 153–180, 1991.

[BLR92] F. Bronsard, T.K. Lakshman, and U.S. Reddy. A framework of directionality for proving termination of logic programs. In K.R. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 321–335. MIT Press, 1992.

[CM91] L. Colussi and E. Marchiori. Proving correctness of logic programs using axiomatic semantics. In *Proceedings of the Eight International Conference on Logic Programming*, pages 629–644. The MIT Press, 1991.

[CM93] L. Colussi and E. Marchiori. Unification as predicate transformer. *Submitted*, 1993. Preliminary version in Proceedings JICSLP' 92, 67-85.

[DM85] P. Dembinski and J. Maluszynski. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the International Symposium on Logic Programming*, pages 29–38, Boston, 1985.

[DM88] W. Drabent and J. Maluszynski. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

[Dra87] W. Drabent. Do logic programs resemble programs in conventional languages? In *Proc. of the Joint International Symposium on Logic Programming*, pages 389–396. IEEE Computer Society, 1987.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[Mel81] C.S. Mellish. The automatic generation of mode declarations for prolog programs. Technical report, Department of Artificial Intelligence, Univ. of Edinburgh, 1981. DAI Research Paper 163.

[Red84] U. S. Reddy. Transformation of logic programs into functional programs. In *International Symposium on Logic Programming*, pages 187–198. IEEE Computer Society, 1984.

[Red86] U.S. Reddy. On the relationship between logic and functional languages. In D. DeGroot and G. Lindstrom, editors, *Functional and Logic Programming*, pages 3–36. Prentice-Hall, 1986.

[Ros91] D.A. Rosenblueth. Using program transformation to obtain methods for eliminating backtracking in fixed-mode logic programs. Technical Report 7, Universidad Nacional Autonoma de Mexico, Instituto de Investigaciones en Matematicas Aplicadas y en Sistemas, 1991.