Pos($\mathcal{T}$): Analyzing Dependencies in Typed Logic Programs

Maurice Bruynooghe¹, Wim Vanhoof¹, and Michael Codish²

¹ Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: {maurice,wimvh}@cs.kuleuven.ac.be

² Ben-Gurion University, Department of Computer Science,
P.O.B. 653, 84105 Beer-Sheva, Israel; e-mail: mcodish@cs.bgu.ac.il

Abstract. Dependencies play a major role in the analysis of program properties. The analysis of groundness dependencies for logic programs using the class of *positive* Boolean functions is a main applications area. The precision of such an analysis can be improved through the integration of either pattern information or type information. This paper develops an approach in which type information is exploited. Different from previous work, a separate simple analysis is performed for each subtype of the types. If the types are polymorphic, then dependencies for specific type instances are derived from the analysis of the polymorphic typed program.

1 Introduction

Dependencies play an important role in program analysis. A statement “program variable X has property p ” can be represented by the propositional variable x^p and dependencies between properties of program variables can be captured as Boolean functions. For example, the function denoted by $x^p \rightarrow y^p$ specifies that whenever x has property p then so does y . In many cases, the precision of a dataflow analysis for a property p is improved if the underlying analysis domain captures dependencies with respect to that given property. The analysis of groundness dependencies for logic programs using the class of *positive* Boolean functions is a main applications area. It aims at identifying which program variables are ground i.e. cannot be further instantiated. The class of *positive* Boolean functions, *Pos* consists of the Boolean functions f for which $f(\text{true}, \dots, \text{true}) = \text{true}$.

One of the key steps in a groundness dependency analysis is to characterise the dependencies imposed by the unifications that could occur during execution. If the program specifies a unification of the form $\text{term}_1 = \text{term}_2$ and the variables in term_1 and term_2 are $\{X_1, \dots, X_m\}$ and $\{Y_1, \dots, Y_n\}$ respectively, then the corresponding groundness dependency imposed is $(x_1 \wedge \dots \wedge x_m) \leftrightarrow (y_1 \wedge \dots \wedge y_n)$ where the groundness of a program variable X is represented by the propositional variable x . This formula specifies that variables in term_1 are (or will become) ground if and only if the variables in term_2 are (or do). It is possible to improve

the precision of an analysis if additional information about the structure (or patterns) of terms is available. For example, if we know that $term_1$ and $term_2$ are both difference lists of the form $H_1 - T_1$ and $H_2 - T_2$, respectively, then the unification $term_1 = term_2$ imposes the dependency $(h_1 \leftrightarrow h_2) \wedge (t_1 \leftrightarrow t_2)$ which is more precise than $(h_1 \wedge t_1) \leftrightarrow (h_2 \wedge t_2)$ which would be derived without the additional information. This has been the approach in previous works such as [20, 24, 8, 11, 2] where pattern analysis can be used to enhance the precision of other analyses.

Introducing pattern information does not allow to distinguish between e.g. bounded lists such as $[1, X, 3]$ and open ended lists such as $[1, 2|Z]$ because the open end can be situated at an arbitrary depth. Making such distinction requires to consider type information. The domain *Pos* has been adapted to do so in [6] where each type was associated with an incarnation of *Pos*. However, that analysis was for untyped programs and each incarnation was developed in a somewhat ad-hoc fashion. The model based analysis of [14] is also related. Here, the models of the program, based on different pre-interpretations express different kinds of program properties. But again, the choice of a pre-interpretation is on a case by case basis. Others in one or another way annotate the types with information about the positions where a variable can occur. This is the case in: [26, 27]; the binding time analysis of [30]; and also in [21] which uses types and applies linear refinement to enrich the type domain with *Pos*-like dependencies. Most close to our approach is the work of [19] which associates properties with what we call in this paper the constituents of the type of the variable (the types possible for subterms of its value). In that work, unifications are abstracted by Boolean formulas expressing the groundness dependencies between different constituents. The main difference is that they construct a single compiled clause covering all constituents while we construct one clause for each constituent. A feature distinguishing our work from the other type based approaches is that we have an analysis for polymorphic types which eliminates the need to analyze a polymorphic predicate for each distinct type instance under which it is called.

Type information can be derived by analysis, as e.g. in [18, 15]; specified by the user and verified by analysis as possible in Prolog systems such as Ciao [16]; or declared and considered part of the semantics of the program as with strongly typed languages such as Gödel [17], Mercury [28] and HAL [13]. The analysis as worked out in this paper is for strongly typed programs.

The next section recalls the essentials of *Pos*. Section 3 introduces types and defines the constituents of a type. Section 4 develops the analysis for monomorphic types, it defines the abstraction for different kinds of unification and proves that they are correct. Section 5 deals with the polymorphic case. It describes how to abstract a call with types that are an instance of the polymorphic types in the called predicate's definition. It proves that the results of the polymorphic analysis approximate the results of a monomorphic analysis and points out the (frequent) cases where both analyses are equivalent. Section 6 discusses applications and related work.

2 Analyzing Groundness Dependencies with *Pos*

Program analysis aims at computing finite approximations of the possibly infinite number of program states that could arise at runtime. Using abstract interpretation [12], approximations are expressed using elements of an abstract domain and are computed by abstracting a concrete semantics; the algebraic properties of the abstract domain guarantee that the analysis is terminating and correct.

The formal definition of *Pos* states that a *Pos* function φ describes a substitution θ (a program state) if any set of variables that might become ground by further instantiating θ is a model of φ . For example, the models of $\varphi = x \wedge (y \rightarrow z)$ are $\{\{X\}, \{X, Z\}, \{X, Y, Z\}\}$. We can see that φ describes $\theta = \{X/a, Y/f(U, V), Z/g(U)\}$ because under further instantiation X is in all of the models and if Y is in a model (becomes ground) then so is Z . Notice that $\theta = \{X/a\}$ is not described by φ as $\{X/a, Y/a\}$ is a further instantiation of θ and $\{X, Y\}$ is not a model of φ . *Pos* satisfies the algebraic properties required of an abstract domain [9]. See also [23] for more details.

A simple way of implementing a *Pos* based groundness analysis is described in [7] and is illustrated in Figure 1. For the purpose of this paper it is sufficient to understand that the problem of analyzing the concrete program (on the left part of Figure 1) is reduced to the problem of computing the concrete semantics of the abstract program (in the middle and on the right). The result is given at the bottom of the figure. For additional details of why this is so, refer to [7, 10, 23]. The least model of the abstract program (e.g. computed using meta-interpreters such as those described in [5, 7]) is interpreted as representing the propositional formula $x_1 \leftrightarrow x_2$ and $(x_1 \wedge x_2) \leftrightarrow x_3$ for the atoms `rotate`(X_1, X_2) and `append`(X_1, X_2, X_3) respectively. This illustrates a goal-independent analysis. Goal-dependent analyses are supported by applying Magic sets or similar techniques (see e.g. [7]).

<i>Concrete rotate</i>	<i>Abstract rotate</i>	<i>Auxiliary predicates</i>
<code>rotate(<i>Xs</i>, <i>Ys</i>) :- append(<i>As</i>, <i>Bs</i>, <i>Xs</i>), append(<i>Bs</i>, <i>As</i>, <i>Ys</i>).</code>	<code>rotate(<i>Xs</i>, <i>Ys</i>) :- append(<i>As</i>, <i>Bs</i>, <i>Xs</i>), append(<i>Bs</i>, <i>As</i>, <i>Ys</i>).</code>	<code>iff(true, []). iff(true, [true <i>Xs</i>]) :- iff(true, <i>Xs</i>). iff(false, <i>Xs</i>) :- member(false, <i>Xs</i>).</code>
<code>append(<i>Xs</i>, <i>Ys</i>, <i>Zs</i>) :- <i>Xs</i> = [], <i>Ys</i> = <i>Zs</i>.</code>	<code>append(<i>Xs</i>, <i>Ys</i>, <i>Zs</i>) :- iff(<i>Xs</i>, []), iff(<i>Ys</i>, [<i>Zs</i>]).</code>	
<code>append(<i>Xs</i>, <i>Ys</i>, <i>Zs</i>) :- <i>Xs</i> = [<i>X</i> <i>Xs1</i>], <i>Zs</i> = [<i>X</i> <i>Zs1</i>], append(<i>Xs1</i>, <i>Ys</i>, <i>Zs1</i>).</code>	<code>append(<i>Xs</i>, <i>Ys</i>, <i>Zs</i>) :- iff(<i>Xs</i>, [<i>X</i>, <i>Xs1</i>]), iff(<i>Zs</i>, [<i>X</i>, <i>Zs1</i>]), append(<i>Xs1</i>, <i>Ys</i>, <i>Zs1</i>).</code>	
<code>rotate(<i>X</i>, <i>X</i>).</code>		<code>append(true, true, true). append(false, <i>Y</i>, false). append(<i>X</i>, false, false).</code>
<i>Least model (abstract rotate):</i>		

Fig. 1. Concrete and abstract programs; least model of the abstract program.

3 About Terms and Types

We assume familiarity with logic programming concepts [22, 1]. We let $Term$ denote the set $T(\Sigma, V)$ of terms constructed from the sets of function symbols Σ and variables V . Types, like terms are constructed from (type) variables and (type) symbols. We denote by $\mathcal{T}$ the set of terms $T(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ (or types in this case) constructed from type symbols $\Sigma_{\mathcal{T}}$ and type variables $V_{\mathcal{T}}$. We assume that the sets of symbols, variables, type symbols and type variables are fixed and disjoint. We write $t:\tau$ to specify that term t has type τ and f/n to specify that (function or type) symbol f has arity n . Besides the usual substitutions which are mappings from variables to terms, we also have type substitutions which are mappings from type variables to types.

We assume a standard notion of strong typing as for example in Mercury [28]. This means that each predicate is associated with a single type declaration and that programs are well typed. Hence, one can associate a unique type with every term and variable in a program clause. A type containing type variables is said to be *polymorphic*, otherwise it is *monomorphic*. The application of a type substitution to a polymorphic type gives a new type which is an *instance* of the original type. For each type symbol, a unique type rule associates that symbol with a finite set of function symbols.

Definition 1. *The rule for a type symbol $h/n \in \Sigma_{\mathcal{T}}$ is a definition of the form $h(\bar{V}) \longrightarrow f_1(\bar{\tau}_1); \dots; f_k(\bar{\tau}_k)$. where: $\bar{V}$ is an n -tuple from $V_{\mathcal{T}}$; for $1 \leq i \leq k$, $f_i/m \in \Sigma$ with $\bar{\tau}_i$ an m -tuple from $\mathcal{T}$; and type variables occurring in the right hand side occur in $\bar{V}$. The function symbols $\{f_1, \dots, f_k\}$ are said to be associated with the type symbol h . A finite set of type rules is called a type definition.*

Example 1. The keyword **type** introduces a type rule:

type `list(T) ---> [] ; [T | list(T)]`.

The function symbols $[\cdot]$ (`nil`) and $[\cdot|\cdot]$ (`cons`) are associated with the type symbol `list`. The type definition specifies also the denotation of each type (the set of terms belonging to the type). For this example, terms of polymorphic type `list(T)` are variables (typed `list(T)`), terms of the form `[]`, or terms of the form `[t1|t2]` with t_1 of type T and t_2 of type `list(T)`. As T is a type variable we cannot determine or commit to its structure under instantiation. Hence only a variable can be of type T . Applying the type substitution $\{T/int\}$ on `list(T)` gives the type `list(int)`. Terms of the form `[t1|t2]` are of type `list(int)` if t_1 is of type `int` and t_2 is of type `list(int)`. Type instances can also be polymorphic, e.g. `list(list(T))`.

The next definition specifies the *constituents* of a type τ . These are the possible types for subterms of terms of type τ .

Definition 2. *Let $h(\bar{\tau}) \longrightarrow f_1(\bar{\tau}_1); \dots; f_k(\bar{\tau}_k)$ be an instance of a rule in ρ and τ an argument of one of the $f_i(\bar{\tau}_i)$. We say that τ is a ρ -constituent of $h(\bar{\tau})$. The constituents relation is the minimal pre-order (reflexive and transitive) $\preceq_{\rho}: \mathcal{T} \times \mathcal{T}$ including all pairs $\tau' \preceq_{\rho} \tau$ such that τ' is a ρ -constituent of τ . The set of ρ -constituents of τ is $Constituents_{\rho}(\tau) = \{ \tau' \in \mathcal{T} \mid \tau' \preceq_{\rho} \tau \}$. When clear from the context we omit ρ and write $\preceq$ and *Constituents*.*

Example 2. With $list/1$ as in Example 1 and the atomic type int , we have:

- $T \preceq list(T)$, $list(T) \preceq list(T)$, $int \preceq list(int)$, $int \preceq int$, $list(T) \preceq list(list(T))$,
 $T \preceq list(list(T))$, $list(list(T)) \preceq list(list(T))$, $T \preceq T$, ...
- $Constituents(int) = \{int\}$, $Constituents(T) = \{T\}$, $Constituents(list(T)) = \{T, list(T)\}$,
 $Constituents(list(int)) = \{int, list(int)\}$.

To ensure that analyses are finite, we restrict types to have finite sets of constituents. This is common as a type with an infinite set of constituents is not well defined. For example, consider the definition: **type** $\mathbf{t}(T) \dashrightarrow \mathbf{f}(\mathbf{t}(\mathbf{t}(T)))$. The type $t(T)$ is defined in terms of its constituent type $t(t(T))$, which in turn is defined in terms of its constituent type $t(t(t(T)))$, and so on.

The following defines the instantiation of terms with respect to a given type.

Definition 3. Let ρ define the types τ and τ' . We say that the term $t:\tau'$ is τ -instantiated (under ρ) if: (1) τ is a constituent of τ' ; and (2) there does not exist a well-typed instance $\sigma:\tau'$ containing a variable of type τ . The predicate $\mu_\tau^\rho(t:\tau')$ is true if and only if $t:\tau'$ is τ -instantiated. For a set of typed variables of interest V and a well typed substitution θ , we denote by $\mu_\tau(\theta, V) = \{ X:\tau' \in V \mid \mu_\tau(X\theta:\tau') \}$. This is the subset of the variables of interest which are τ -instantiated by θ .

If τ' has no constituents of type τ then we do not consider $t:\tau'$ as τ -instantiated because no instance of any term of type τ' can have a subterm of type τ . In the case $t:T$ where T is a polymorphic parameter (and T has no constituents of type τ), the intuition is that for some instance τ' of T , it could be the case that $t:\tau'$ should

$t: list(int)$	$\mu_{list(int)}(t)$	$\mu_{int}(t)$
$[1, 2]$	<i>true</i>	<i>true</i>
$[1, X]$	<i>true</i>	<i>false</i>
$[1 X]$	<i>false</i>	<i>false</i>
$[]$	<i>true</i>	<i>true</i>

Fig. 2. The τ -instantiation of some $list(int)$ terms.

not be considered τ -instantiated. Figure 2 illustrates the $list(int)$ -instantiation and int -instantiation for several terms of type $list(int)$. First note that both $list(int)$ and int are constituents of $list(int)$. Hence, $\mu_{list(int)}([1, X])$ is *true* because all $list(int)$ -subterms of well-typed instances of $[1, X]$ are instantiated; and $\mu_{list(int)}([1|X])$ is *false* because the subterm $X: list(int)$ is a variable. Also $\mu_{int}([1|X])$ is *false* as e.g. $[1, Y|Z]$ is an instance with the variable Y of type int . Note that $\mu_{int}(s) \rightarrow \mu_{list(int)}(s)$. In general, the following property holds:

Proposition 1. For constituent types $\tau_1 \preceq \tau_2$ and any term t , $\mu_{\tau_1}(t) \rightarrow \mu_{\tau_2}(t)$.

We observe that classical groundness for a typed term $t:\tau$ can be expressed with respect to types instantiation by requiring t to be instantiated with respect to all constituents of τ .

$$ground(t) \leftrightarrow \bigwedge \{ \mu_{\tau_i}^\rho(t:\tau) \mid \tau_i \in Constituents(\tau) \}. \quad (1)$$

4 $Pos(\mathcal{T})$ in a monomorphic setting

We let the propositional variable x^τ denote the predicate $\mu_\tau^\rho(X:\tau')$ meaning that typed program variable $X:\tau'$ is instantiated with respect to type τ . Instantiation analysis using $Pos(\mathcal{T})$ generalises groundness analysis with Pos . Informally, a positive Boolean formula φ , representing τ -instantiation dependencies, describes a substitution θ if any set of variables that might become τ -instantiated by further instantiating θ is a model of φ . The abstraction and concretization functions are formalised as:

Definition 4. Let V be a set of (typed) variables of interest and τ a type. The τ -abstraction (in terms of models) of the set Θ of well-typed substitutions and the τ -concretization for the positive Boolean function ψ are:

$$\begin{aligned}\alpha_\tau(\Theta) &= \{ \mu_\tau(\theta\sigma) \mid \theta \in \Theta, \sigma \text{ is a well typed substitution} \} \\ \gamma_\tau(\psi) &= \{ \theta \mid \alpha_\tau(\{\theta\}) \subseteq \psi \}\end{aligned}$$

Example 3. Let $V = \{X:\text{list}(\text{int}), Y:\text{int}, Z:\text{list}(\text{int})\}$ and $\theta = \{X \mapsto [Y|Z]\}$. Then, $\alpha_{\text{list}(\text{int})}(\{\theta\})$ has models $\{\emptyset, \{x, z\}\}$ (observe that $Y:\text{int}$ is not $\text{list}(\text{int})$ -instantiated because $\text{list}(\text{int})$ is not a constituent of int) and $\alpha_{\text{int}}(\{\theta\})$ has models $\{\emptyset, \{y\}, \{z\}, \{x, y, z\}\}$. These sets of models correspond respectively to the formulae $x \leftrightarrow z$ and $x \leftrightarrow (y \wedge z)$.

In a classic groundness analysis, the unification $A = [X|Xs]$ is abstracted as $a \leftrightarrow (x \wedge xs)$. Indeed, we assume that any subterm of the term that A is bound to at runtime could unify with any subterm of the terms bound to X or Xs . In the presence of types we know that A and $[X|Xs]$ are both of the same type (otherwise the program is not well-typed). In addition, we know that all unifications between subterms of (the terms bound to) A and $[X|Xs]$ are between terms corresponding to the same types. So in this example (assuming both terms to be of type $\text{list}(\text{int})$), we can specify $a \leftrightarrow xs$ for type $\text{list}(\text{int})$ and $a \leftrightarrow (x \wedge xs)$ for type int . It is important to note that the interpretations of the variables in $a \leftrightarrow xs$ and in $a \leftrightarrow (x \wedge xs)$ are different. The former refers to subterms of type $\text{list}(\text{int})$ whereas the latter refers to subterms of type int . These intuitions are formalised in the following definitions and theorems.

Definition 5. The τ abstraction of a normalised typed logic program P is obtained by abstracting the equations in P . An equation of the form $X = Y$ with X and Y of type τ' is abstracted by $\text{iff}(X, [Y])$ (implementing $x \leftrightarrow y$) if $\tau \preceq \tau'$ and otherwise it is abstracted by **true**. An equation of the form $X = f(Y_1, \dots, Y_n)$ with typed variables $X:\tau', Y_1:\tau_1, \dots, Y_n:\tau_n$ is abstracted by $\text{iff}(X, [W_1, \dots, W_k])$ (implementing $x \leftrightarrow w_1 \wedge \dots \wedge w_k$) if $\tau \preceq \tau'$ and otherwise it is abstracted by **true** where $\{W_1, \dots, W_k\} = \{Y_i \mid 1 \leq i \leq n, \tau \preceq \tau_i\}$.

Theorem 1. Correctness. Let E be of the form $X = Y$ or $X = f(Y_1, \dots, Y_n)$, φ a positive Boolean function on a set of variables V including the variables in E of a type having τ as constituent, and φ' the τ -abstraction of E . If $\theta \in \gamma_\tau(\varphi)$ and $\sigma = \text{mgu}(E)$ then $\theta\sigma|_V \in \gamma_\tau(\varphi \wedge \varphi')$.

Otherwise, observe that $\gamma_\tau(\varphi \wedge \varphi') = \gamma_\tau(\varphi) \cap \gamma_\tau(\varphi')$ hence it suffices to show that $\theta\sigma|_V \in \gamma_\tau(\varphi)$ and $\theta\sigma|_V \in \gamma_\tau(\varphi')$. The former holds because the set $\gamma_\tau(\varphi)$ is closed under instantiation. For the latter, we distinguish:

-

The adaptation of the implementation technique of Section 2 for an analysis of `append(list(int), list(int), list(int))` is as follows:

The least model of `append_int/3` expresses the $Pos(int)$ -formula $z \leftrightarrow x \wedge y$, i.e. that all subterms of Z of the type int are instantiated iff those of X and Y are. The least model of `append_list_int/3` expresses the $Pos(list(int))$ -formula $x \wedge (y \leftrightarrow z)$, i.e. that all subterms of X of type $list(int)$ are instantiated (in other words the backbone of the list is instantiated when `append/3` succeeds) and that those of Y are instantiated iff those of Z are instantiated. Classical groundness, is obtained by composing the two models, using Equation (1) in Section 3:

```
append(X,Y,Z) :- append_list_int(Xl,Yl,Zl), append_int(Xe,Ye,Ze),
                  iff(X,[Xl,Xe]), iff(Y,[Yl,Ye]), iff(Z,[Zl,Ze]).
```

Proposition 2. *Let p/n be a predicate in a typed logic program and τ and τ' types such that $\tau' \preceq \tau$. Let $\bar{Y}$ denote the arguments $Y_i : \tau_i$ of p/n such that $\tau' \preceq \tau_i$*

but $\tau \not\leq \tau_i$. Denote the results of the τ - and τ' - analyses of p/n by φ and φ' and let ψ be the conjunction of the variables in $\bar{Y}$. Then $\varphi \wedge \psi \rightarrow \varphi'$.

5 Polymorphism in $Pos(\mathcal{T})$

Type polymorphism is an important abstraction tool: a predicate defined with arguments of a polymorphic type can be called with actual arguments of any type that is an instance of the defined type. For example, the `append/3` predicate from Section 4 can be defined with respect to the polymorphic type definition `append(list(T), list(T), list(T))`, stating that its arguments are lists of the same type T . Abstracting `append/3` for this definition results in the same abstractions as in Section 4 but with `list(T)` and T replacing `list(int)` and `int`.

The results of the analysis with type variables is correct for any choice of types instantiating them (e.g., `list(int)` and `int`, or `list(list(int))` and `list(int)` when `append/3` is called with actual types `list(int)` or `list(list(int))` respectively). It is also possible to obtain the result of the analysis for each type-instance by which it is called and sometimes this gives a more precise result. However, it is more efficient to analyze the definition once for its given polymorphic types, and then derive the abstractions of a particular call from that result.

The need for such an approach is even more crucial when analyzing large programs distributed over many modules. It is preferable that an analysis does not need the actual code of the predicates it imports (and of the predicates called directly or indirectly by the imported predicates) but only the result of the call independent analysis. See [25, 4, 29] for discussions about module based analysis.

Example 4. Consider a predicate `p/2` with both arguments of type `list(list(int))`. The program and its abstractions for the `int`, `list(int)` and `list(list(int))` are as follows:

Concrete definition	Abstractions
<code>p(X,Y) :- append(X,X,Y).</code>	<code>p_list_list_int(X,Y) :- append_list_T(X,X,Y).</code> <code>p_list_int(X,Y) :- append_T(X,X,Y).</code> <code>p_int(X,Y) :- append_T(X,X,Y).</code>

Intuitively, it is clear that the constituent `list(list(int))` from the call to `append/3` corresponds to the constituent `list(T)` in `append/3`'s definition. Hence, instead of computing the `list(list(int))`-abstraction of `append/3`, its `list(T)`-abstraction can be used. Likewise, the constituents `list(int)` and `int` from the call both correspond to the constituent T in the definition, hence also their abstractions.

To formalize the relationship between the constituents of the actual types and those of the formal types, we introduce the notion of a *variable typing*.

Definition 6. A variable typing ζ is a mapping from program variables to types; $\text{dom}(\zeta)$ is the set of variables for which the mapping is defined; $X\zeta$ denotes the type of X under the variable typing ζ . Given variable typings ζ and ζ' over the same domain, ζ' is an instance of ζ if, for every $X \in \text{dom}(\zeta)$ there exists a type substitution ξ such that $X\zeta' = X\zeta\xi$.

The notion of *Constituents* extends in a straightforward way to variable typings. In a well-typed program, the types of the arguments of a call are always at least as instantiated as the types of the corresponding arguments in the predicate definition. Hence the constituents occurring in a variable typing ζ' of a call $p(X_1, \dots, X_n)$ can be related to the constituents of the variable typing ζ of the head $p(Y_1, \dots, Y_n)$ as follows:

Definition 7. Consider variable typings ζ' and ζ with ζ' an instance of ζ . The type mapping between ζ and ζ' is the minimal relation $R_{\zeta'}^{\zeta} : \mathcal{T} \times \mathcal{T}$ such that:

- If $X/\tau_1 \in \zeta'$ and $X/\tau_2 \in \zeta$ then $(\tau_1, \tau_2) \in R_{\zeta'}^{\zeta}$.
- If $(\tau_1, \tau_2) \in R_{\zeta'}^{\zeta}$ and $\tau_2 \in V_{\mathcal{T}}$ then $\forall \tau \in \text{Constituents}(\tau_1) : (\tau, \tau_2) \in R_{\zeta'}^{\zeta}$.
- If $(\tau_1, \tau_2) \in R_{\zeta'}^{\zeta}$ and $\tau_2 = h(V_1, \dots, V_n)\xi_2$ and $\tau_1 = h(V_1, \dots, V_n)\xi_1$ and $h(V_1, \dots, V_n) \longrightarrow f_1(\tau_{1_1}, \dots, \tau_{1_{m_1}}) ; \dots ; f_l(\tau_{1_1}, \dots, \tau_{1_{m_l}})$ is the type rule for h/n then, for all $\tau_{i_j}, (\tau_{i_j}\xi_1, \tau_{i_j}\xi_2) \in R_{\zeta'}^{\zeta}$.

The type function $\phi_{\zeta'}^{\zeta} : \mathcal{T} \mapsto 2^{\mathcal{T}}$ is defined for constituents of the most instantiated type $\zeta' : \phi_{\zeta'}^{\zeta}(\tau) = \{\tau' \mid (\tau, \tau') \in R_{\zeta'}^{\zeta}\}$.

When ζ and ζ' are obvious from the context, we will write R and $\phi(\tau)$.

Example 5. The type mapping and type function associated to the variable typings $\zeta = \{X/\text{list}(T)\}$ and $\zeta' = \{X/\text{list}(\text{list}(\text{int}))\}$ are respectively:

- $R = \{(\text{list}(\text{list}(\text{int})), \text{list}(T)), (\text{list}(\text{int}), T), (\text{int}, T)\}$ and
- $\phi = \{(\text{list}(\text{list}(\text{int})), \{\text{list}(T)\}), (\text{list}(\text{int}), \{T\}), (\text{int}, \{T\})\}$.

If ζ' denotes the variable typing associated to the variables of the call, and ζ denotes the variable typing associated to the called predicate's head variables, the mapping R expresses that the τ -abstraction of a call corresponds to the $\phi(\tau)$ -abstraction of the called predicate. As an illustration, ϕ from Example 5 gives the mappings used in Example 4.

However, in general $\phi(\tau)$ is not a singleton, as a constituent of the call can be mapped to itself *and* to one or more type variables in the polymorphic type definition. Consider for example a predicate $q/4$ and its abstractions:

Concrete	Abstraction w.r.t. int	Abstraction w.r.t. T
$\text{pred } q(\text{int}, \text{int}, T, T).$	$q_int(X, Y, U, V) :-$	$q_T(X, Y, U, V) :-$
$q(X, Y, U, V) :- X=Y, U=V.$	$\text{iff}(X, [Y]).$	$\text{iff}(U, [V]).$
$q(X, Y, U, V) :- X=0.$	$q_int(X, Y, U, V) :-$	$q_T(X, Y, U, V).$
	$\text{iff}(X, []).$	

Suppose $q(A, B, C, D)$ is a call with A, B, C and D of type int . The type substitution on the defined types is $\{T/\text{int}\}$, and hence $\phi(\text{int}) = \{\text{int}, T\}$. A correct int -abstraction of the *call* should include both the int -abstraction and the T -abstraction from the definition, since in the polymorphic analysis, it was assumed that T does not have int as constituent. Hence, the int -abstraction of the call becomes $q_int(A, B, C, D) \wedge q_T(A, B, C, D)$. The formal definition of the τ -abstraction of a predicate call is then as follows:

Definition 8. Let ζ' and ζ be the variable typings associated respectively to a call $p(X_1, \dots, X_n)$ and the head of p 's definition; let $\tau' \in \text{Constituents}(\zeta')$; The τ' -abstraction of $p(X_1, \dots, X_n)$ is defined as $\wedge\{p\text{-}\tau(X_1, \dots, X_n) \mid \tau \in \phi(\tau')\}$.

Handling a predicate call in such a way, however, possibly causes some precision loss. Indeed, one can verify that a *monomorphic* analysis of $q(X, Y, U, V)$ for the type $p(\text{int}, \text{int}, \text{int}, \text{int})$ gives the formula $((x \leftrightarrow y) \wedge (u \leftrightarrow v)) \vee x$ while the conjunction of calls $p\text{-int}(X, Y, U, V), p\text{-T}(X, Y, U, V)$ gives the less precise formula $((x \leftrightarrow y) \vee x) \wedge ((u \leftrightarrow v) \vee \text{true})$. However, the result is always safe:

Theorem 2. Let p be a predicate definition and ζ and ζ' variable typings for the variables of p such that ζ' is an instance of ζ . For $\tau \in \text{Constituents}(\zeta)$, let φ^τ denote the τ -abstraction of p for the variable typing ζ , and similarly for $\tau' \in \text{Constituents}(\zeta')$, let $\psi^{\tau'}$ denote the τ' -abstraction of p for the variable typing ζ' . Then it holds that $\psi^{\tau'} \rightarrow \wedge\{\varphi^\tau \mid \tau \in \phi(\tau')\}$.

Theorem 2 states that the use of Definition 8 safely approximates the result obtained by an analysis of the called predicate for the type instances used in the call. As the above example illustrates, the equivalence does not hold and some precision loss is possible. However, such precision loss occurs only when constituents of parametric type instances interfere with each other or with other types. In case there is no such interference (i.e. $\phi(\tau')$ is a singleton), the results of call-independent analysis equal those of call-dependent analysis. We believe that interference is more the exception than the rule. Indeed, it only happens when type instances of two different type variables have a common constituent or when the instance of a type variable has a constituent common with the type of another argument. Even if the types do interfere, a loss of precision is avoided if the lub of the $\text{Pos}(\mathcal{T})$ -formulas of each individual clause body is equal to the $\text{Pos}(\mathcal{T})$ -formula of one of the clause bodies (the same for all types τ). This could well be often the case in practice because different clauses of a program tend to establish the same degree of instantiatedness (or in case of recursion, the $\text{Pos}(\mathcal{T})$ -formula of the base case implies the $\text{Pos}(\mathcal{T})$ -formula of the recursive clause(s)):

Corollary 1. With the same notations as in Theorem 2, let φ_j^τ denote the τ -abstraction of the body of clause j of predicate p 's definition. If $\phi(\tau')$ has only one element or there exists a clause k such that for all $\tau \in \phi(\tau')$: $\varphi_k^\tau = \vee\{\varphi_j^\tau \mid j \text{ is a clause}\}$, i.e. the τ -abstraction of the body of clause k is the upper-bound of the τ -abstractions of all clause bodies, then $\psi^{\tau'} = \wedge\{\varphi^\tau \mid \tau \in \phi(\tau')\}$.

Intuitively, one can say that some dependencies are ignored by analysing the predicate separately for each element in $\phi(\tau')$, and that they are only partly recovered by taking the conjunction of the obtained formulas. We refer to Appendix A.1 for the proofs of Theorem 2 and Corollary 1.

6 Discussion

The main goal of this research is to improve the precision of groundness dependencies by taking into consideration type information. Precision is improved for

two reasons: (a) when unifying terms (or variables) of the same type it is possible to refine the resulting dependencies (e.g. list elements with list elements, and list backbone with list backbone); and (b) computation paths which unify terms of different types can be pruned.

While $Pos(\mathcal{T})$ can improve the precision of groundness dependencies it is important also to note that the $Pos(\mathcal{T})$ formulas provide valuable information on their own. In future work we intend to use it to improve automated termination analysis which in turn will be used in a binding-time analysis for the off-line specialisation of logic programs (improving the work in [3].).

Some preliminary experiments, using the implementation technique of [7] indicate that our approach can be done more efficiently than the approach in [19] where a single model is computed, and where a program variable corresponds to several Boolean variables, one for each type constituent. The reason is that the computation of the least fixpoint of the abstracted program slows down more than linearly with the number of variables. Moreover, proposition 2 allows for extra savings by reusing results from one type constituent in the computation for other constituents.

References

1. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers B.V., 1990.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. Efficient structural information analysis for real CLP languages. In *Proc. LPAR2000*, volume 1955 of *LNCS*. Springer, 2000.
3. M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In *Proc. ESOP'98*, volume 1381 of *LNCS*, pages 27–41. Springer, 1998.
4. F. Bueno, M. de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A model for inter-module analysis and optimizing compilation. In *Preproceedings LOPSTR2000*, 2000.
5. M. Codish. Efficient goal directed bottom-up evaluation of logic programs. *J. Logic Programming*, 38(3):354–370, 1999.
6. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proc. SAS'94*, volume 864 of *LNCS*, pages 281–296. Springer, 1994.
7. M. Codish and B. Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *J. Logic Programming*, 25(3):249–274, 1995.
8. M. Codish, K. Marriott, and C. Taboch. Improving program analyses by structure untupling,. *Journal Logic Programming*, 43(3), June 2000.
9. A. Cortesi, G. Filé, and W. Winsborough. Prop revisited: propositional formula as abstract domain for groundness analysis. In *Proc. LICS'91*, pages 322–327. IEEE Press, 1991.
10. A. Cortesi, G. Filé, and W. H. Winsborough. Optimal groundness analysis using propositional logic. *J. Logic Programming*, 27(2):137–167, 1996.

11. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: open product and generic pattern construction. *Science of Computer Programming*, 38(1-3):27–71, 2000.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL'97*, pages 238–252, 1977.
13. B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In *Proc. CP'99*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.
14. J. Gallagher, D. Boulanger, and H. Saglam. Practical model-based static analysis for definite logic programs. In *Proc. ILPS'95*, pages 351–365. MIT Press, 1995.
15. J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In *Proc. ICLP'94*, pages 599–613. MIT Press, 1994.
16. M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Debugging, and optimization using the Ciao system preprocessor. In *Proc. ICLP'99*, pages 52–66, 1999.
17. P. Hill and J. Lloyd. *The Gödel Language*. MIT Press, 1994.
18. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic programming*, 13(2&3):205–258, 1992.
19. V. Lagoen and P. J. Stuckey. A framework for analysis of typed logic programs. In *Proc. FLOPS 2001*, volume 2024 of *LNCS*, pages 296–310. Springer, 2001.
20. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM TOPLAS*, 16(1):35–101, 1994.
21. G. Levi and F. Spoto. An experiment in domain refinement: Type domains and type representations for logic programs. In *Proc. PLILP/ALP'98*, volume 1490 of *LNCS*, pages 152–169. Springer, 1998.
22. J.W. Lloyd. *Foundation of Logic Programming*. Springer, second edition, 1987.
23. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1-4):181–196, 1993.
24. A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the practicality of abstract equation systems. In *Proc. ICLP'95*, pages 781–795. MIT Press, 1995.
25. G. Puebla and M. Hermenegildo. Some issues in analysis and specialization of modular Ciao-Prolog programs. In *Proc. ICLP Workshop on Optimization and Implementation of Declarative Languages*, 1999.
26. O. Ridoux, P. Boizumault, and F. Malésieux. Typed static analysis: Application to groundness analysis of Prolog and lambda-Prolog. In *Proc. FLOPS'99*, volume 1722 of *LNCS*, pages 267–283. Springer, 1999.
27. J.G. Smaus, P. M. Hill, and A. King. Mode analysis domains for typed logic programs. In *Proc. LOPSTR'99*, volume 1817 of *LNCS*, pages 82–101. Springer, 2000.
28. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Logic Programming*, 29(1-3):17–64, October–November 1996.
29. W. Vanhoof. Binding-time analysis by constraint solving: a modular and higher-order approach for Mercury. In *Proc. LPAR2000*, volume 1955 of *LNCS*, pages 399 – 416, Springer, 2000.
30. W. Vanhoof and M. Bruynooghe. Binding-time analysis for Mercury. In *Proc. ICLP'99*, pages 500 – 514. MIT Press, 1999.

A Proofs

A.1 Proof of Theorem 2 and Corollary 1

In the formulation and proofs that follow, we consider a program construct under two variable typings ζ and ζ' such that ζ' is an instance of ζ . With X a program variable, we denote with τ_X and τ'_X the type of X in respectively ζ and ζ' . We relate the τ' -abstraction of the construct for a type $\tau' \in \text{Constituents}(\zeta')$ with its $\phi(\tau')$ -abstractions. First, we consider a single unification.

Lemma 1. *With E an equality, let φ^τ denote its τ -abstraction for the variable typing ζ and and $\psi^{\tau'}$ its τ' -abstraction for the variable typing ζ' . Then it holds that $\psi^{\tau'} = \bigwedge \{\varphi^\tau \mid \tau \in \phi(\tau')\}$.*

Proof. E is of the form $X = \dots$. Assume $Y_1, \dots, Y_n$ are the variables in the rhs. such that $\tau' \in \text{Constituents}(\tau'_{Y_i})$. Then $\psi^{\tau'} = x \leftrightarrow y_1 \wedge \dots \wedge y_n$. Let $\phi(\tau') = \{\tau_1, \dots, \tau_n\}$. The variables $Y_{i_1}, \dots, Y_{i_{n_i}}$ such that $\tau_i \in \text{Constituents}(\tau_{Y_{i_j}})$ are a subset of $\{Y_1, \dots, Y_n\}$ and $\varphi^{\tau_i} = x \leftrightarrow y_{i_1} \wedge \dots \wedge y_{i_{n_i}}$. Moreover, by construction of ϕ it holds that $\tau' \in \text{Constituents}(\tau'_{Y_i})$ if and only if $\exists \tau_j \in \text{Constituents}(\tau_{Y_i})$ and hence $\bigcup_i \{Y_{i_1}, \dots, Y_{i_{n_i}}\} = \{Y_1, \dots, Y_n\}$. Consequently,

$$\begin{aligned} \psi^{\tau'} &= x \leftrightarrow y_1 \wedge \dots \wedge y_n \\ &= (x \leftrightarrow y_{1_1} \wedge \dots \wedge y_{1_{n_1}}) \wedge \dots \wedge (x \leftrightarrow y_{k_1} \wedge \dots \wedge y_{k_{n_k}}) \\ &= \varphi^{\tau_1} \wedge \dots \wedge \varphi^{\tau_k} \end{aligned} \quad \square$$

Some properties of propositional logic (b is a propositional formula):

$$\text{If } \forall i : b'_i \rightarrow b_i \text{ then } \bigwedge_i b'_i \rightarrow \bigwedge_i b_i \quad (1)$$

$$\text{If } \forall i : b'_i \rightarrow b_i \text{ then } \bigvee_i b'_i \rightarrow \bigvee_i b_i \quad (2)$$

$$\forall i, j : \bigvee_j (\bigwedge_i b_{i,j}) \rightarrow \bigwedge_i (\bigvee_j b_{i,j}) \quad (3)$$

The following lemma concerns the τ' - and τ -abstractions of a predicate definition.

Lemma 2. *Let p be a predicate definition; let φ^τ and $\varphi_{i,j}^\tau$ denote respectively the τ -abstraction of p and of $B_{i,j}$, the i^{th} body atom in the j^{th} clause, for the type assignment ζ , and similarly, $\psi^{\tau'}$ and $\psi_{i,j}^{\tau'}$ the τ' -abstraction of p and $B_{i,j}$ for the variable typing ζ' . If for each body atom $B_{i,j}$: $\psi_{i,j}^{\tau'} \rightarrow \bigwedge \{\varphi_{i,j}^\tau \mid \tau \in \phi(\tau')\}$ then $\psi^{\tau'} \rightarrow \bigwedge \{\varphi^\tau \mid \tau \in \phi(\tau')\}$.*

Proof. Assuming $\psi_{i,j}^{\tau'} \rightarrow \bigwedge \{\varphi_{i,j}^\tau \mid \tau \in \phi(\tau')\}$, we prove $\psi^{\tau'} \rightarrow \bigwedge \{\varphi^\tau \mid \tau \in \phi(\tau')\}$. Let $B_{1,j}, \dots, B_{n_j,j}$ be the body of the j^{th} clause; let $\varphi_j^\tau = \bigwedge \{\varphi_{i,j}^\tau \mid i \in \{1, \dots, n_j\}\}$ and $\psi_j^{\tau'} = \bigwedge \{\psi_{i,j}^{\tau'} \mid i \in \{1, \dots, n_j\}\}$ be their respectively τ - and τ' -abstraction. Using the assumption, (1) and commutativity of $\bigwedge$, we obtain: $\psi_j^{\tau'} \rightarrow \bigwedge \{\varphi_j^\tau \mid \tau \in \phi(\tau')\}$.

Using (2) we obtain: $\bigvee_j \psi_j^{\tau'} \rightarrow \bigvee_j (\bigwedge_{\tau \in \phi(\tau')} \varphi_j^\tau)$.

Using (3) we obtain: $\bigvee_j \psi_j^{\tau'} = \psi^{\tau'} \rightarrow \bigwedge_{\tau \in \phi(\tau')} (\bigvee_j \varphi_j^\tau = \bigwedge_{\tau \in \phi(\tau')} \varphi^\tau)$. $\square$

From the above lemma, we derive two corollaries that we can use to prove Corollary 1 (p. 10). A first one handles the first option in the condition of Corollary 1, namely the case that $\phi(\tau') = \{\tau\}$, that is, when there is a one-to-one correspondence between τ' in ζ' and τ in ζ .

Corollary 2. *With the same notations as in Lemma 2, if $\phi(\tau')$ has only one element and for each body atom: $\psi_{i,j}^{\tau'} = \wedge\{\varphi_{i,j}^\tau | \tau \in \phi(\tau')\}$ then $\psi^{\tau'} = \wedge\{\varphi^\tau | \tau \in \phi(\tau')\}$.*

Proof. Immediate from the proof of Lemma 2. $\square$

The second corollary handles the case that the least upperbound of the τ -abstractions of the individual clauses equals the τ -abstraction of one of the clauses.

Corollary 3. *With the same notations as in Lemma 2, if $\psi_{i,j}^{\tau'} = \wedge\{\varphi_{i,j}^\tau | \tau \in \phi(\tau')\}$ and there exists a clause k such that for all $\tau \in \phi(\tau')$: $\varphi_k^\tau = \vee\{\varphi_j^\tau | j \text{ is a clause}\}$ then $\psi^{\tau'} = \wedge\{\varphi^\tau | \tau \in \phi(\tau')\}$.*

Proof. With the notations of in Lemma 2 we now obtain $\psi_j^{\tau'} = \wedge\{\varphi_j^\tau | \tau \in \phi(\tau')\}$.

Hence $\vee_j \psi_j^{\tau'} = \psi^{\tau'} = \vee_j \left(\bigwedge_{\tau \in \phi(\tau')} \varphi_j^\tau \right)$.

Using the assumption, $\vee_j \left(\bigwedge_{\tau \in \phi(\tau')} \varphi_j^\tau \right) = \bigwedge_{\tau \in \phi(\tau')} \varphi_k^\tau = \bigwedge_{\tau \in \phi(\tau')} \varphi^\tau$. $\square$

Now, we can prove Theorem 2 by using Lemmas 1 and 2 from above.

Theorem 2. *Let p be a predicate definition; let φ^τ denote the τ -abstraction of p for the variable typing ζ ; let $\psi^{\tau'}$ denote the τ' -abstraction of p for the variable typing ζ' . Then it holds that $\psi^{\tau'} \rightarrow \wedge\{\varphi^\tau | \tau \in \phi(\tau')\}$.*

Proof. The formula $\psi^{\tau'}$ is computed by a bottom up fixpoint operator that, starting from the formula *false* for all predicates, recomputes the $Pos(\mathcal{T})$ -formula of each predicate until a fixpoint is reached. The proof is by induction on the number of iterations.

The predicate p is defined by the clauses $C_1, \dots, C_m$, and a clause C_i is a conjunction of body atoms $B_{i,1}, \dots, B_{i,n_i}$.

- First iteration. Since Lemma 1 holds for the unifications, and the predicate calls in the body only introduce the formula *false*, we have for each body atom $B_{i,j}$ that $\psi_{i,j}^{\tau'} = \wedge\{\varphi_{i,j}^\tau | \tau \in \phi(\tau')\}$ where $\psi_{i,j}^{\tau'}$ denotes the τ' -abstraction of $B_{i,j}$ for variable typing ζ' and $\varphi_{i,j}^\tau$ denotes the τ -abstraction of $B_{i,j}$ for variable typing ζ . Applying Lemma 2 results in $\psi^{\tau'} \rightarrow \wedge\{\varphi^\tau | \tau \in \phi(\tau')\}$.
- Induction step. Assume the formula holds after k iterations. For the abstraction of p as constructed in iteration $k+1$, we have either by Lemma 1 (for the unifications) or by the induction hypothesis (for the predicate calls) that for each body atom $B_{i,j}$ holds that $\psi_{i,j}^{\tau'} \rightarrow \wedge\{\varphi_{i,j}^\tau | \tau \in \phi(\tau')\}$.

Applying Lemma 2 on obtains: $\psi^{\tau'} \rightarrow \wedge\{\varphi^\tau | \tau \in \phi(\tau')\}$. $\square$

Corollary 1 on page 10 can be proven analogously to the proof of Theorem 2, using Corollaries 2 and 3 instead of Lemma 2.

A.2 Proof of Proposition 2

Proposition 2. *Let p/n be a predicate in a typed logic program and τ and τ' types such that $\tau' \preceq \tau$. Let $\bar{Y}$ denote the arguments $Y_i : \tau_i$ of p/n such that $\tau' \preceq \tau_i$ but $\tau \not\preceq \tau_i$. Denote the results of the τ - and τ' -analyses of p/n by φ and φ' and let ψ be the conjunction of the variables in $\bar{Y}$. Then $\varphi \wedge \psi \rightarrow \varphi'$.*

Proof. Let us denote by $\bar{X}$ the arguments of p/n not in $\bar{Y}$ and write $\varphi(\bar{X}, \bar{Y})$, $\varphi'(\bar{X}, \bar{Y})$ and $\psi(\bar{Y})$ instead of φ , φ' and ψ . Moreover we assume without loss of generality that the arguments of p/n are $p(\bar{X}, \bar{Y})$.

The proof is based on the observation is that there is a one-one correspondence between value assignments $\{\bar{x}/\bar{v}\}$ that are models of the τ -abstraction $\varphi(\bar{x})$ of a predicate q/m and the atoms $q(\bar{v})$ that are computed during the computation of the least fixpoint of the abstracted program. With P_τ and $P_{\tau'}$ respectively the τ - and τ' -abstractions of the program, we proof by induction that $p(\bar{u}, \bar{v}) \in T_{P_\tau} \uparrow k$ implies $p(\bar{u}, \overline{true}) \in T_{P_{\tau'}} \uparrow k$. (We assume the least fixpoint of the iff/2 predicate is computed in advance.)

Iteration 1. If $p(\bar{u}, \bar{v}) \in T_{P_\tau} \uparrow 1$, it means that P_τ contains a clause $c_\tau = p(\bar{X}, \bar{Y}) \leftarrow B_1, \dots, B_m$ with all B_i calls to iff/2 (τ -abstractions of equalities) and that there is a substitution $\theta = \{\bar{X}/\bar{u}, \bar{Y}/\bar{v}, \bar{Z}_1/\bar{w}_1\}$ such that $\forall i : B_i\theta$ is true in the least model of iff/2 (note that the variables of $\bar{Y}$ do not occur in the body). Hence $P_{\tau'}$ contains a clause $c_{\tau'} = p(\bar{X}, \bar{Y}) \leftarrow B'_1, \dots, B'_m, C_1, \dots, C_n$ with the B'_i τ' -abstractions of the equalities having B_i as their τ -abstraction and the C_i τ' -abstractions of equalities having $true$ as τ -abstraction. To prove that $p(\bar{u}, \overline{true}) \in T_{P_{\tau'}} \uparrow 1$ it suffices to prove that $\sigma = \{\bar{X}/\bar{u}, \bar{Y}/\overline{true}, \bar{Z}_1/\bar{w}_1, \bar{Z}_2/\overline{true}\}$ makes the body of $c_{\tau'}$ $true$ (the variables of $\bar{Z}_2$ occur in the body of $c_{\tau'}$ but not in the body of c_τ). To prove the latter, we show that the three kinds of atoms in the body of $c_{\tau'}$ are $true$ in the least model of iff/2 under the substitution σ :

- B_i is the τ -abstraction of an equation $U = V$. Obviously, $B'_i = B_i$, and $B_i\sigma = B_i\theta$ is true in the least model of iff/2.
- B_i is the τ -abstraction of an equation $U = f(\bar{V})$. B_i is of the form $iff(U, [\bar{V}_\tau])$ with $\bar{V}_\tau$ the variables of $\bar{V}$ having τ as a constituent of their type. Hence B'_i is of the form $iff(U, [\bar{V}_\tau, \bar{V}_{\tau'}])$ with $\bar{V}_{\tau'}$ the variables of $\bar{V}$ having τ' but not τ as a constituent of their type. Note that σ binds the variables of $\bar{V}_{\tau'}$ to $true$. We know that $B_i\theta$ is $true$, that means either U and $\bar{V}_\tau$ are bound to $true$ and thus also $B_i\sigma$ is $true$ or U and at least one of $\bar{V}_\tau$ are bound to $false$, also in this case, $B_i\sigma$ is $true$.
- C_i is of the form $iff(\dots)$ with all variables not occurring in the body of c_τ .

All these variables are bound to $true$ under σ hence $C_i\sigma$ is $true$.

Iteration $k + 1$. Assume $p(\bar{u}, \bar{v}) \in T_{P_\tau} \uparrow k + 1$. The difference with the first iteration is the form of the clauses c_τ and $c_{\tau'}$. The body of c_τ now contains also calls of the form $q(\bar{X}', \bar{Y}')$ (with $Y'_i : \tau_i$ the arguments of q/m such that $\tau' \preceq \tau_i$ but $\tau \not\preceq \tau_i$). The body of $c_{\tau'}$ contains corresponding calls $q(\bar{X}', \bar{Y}')$. We know that $q(\bar{X}', \bar{Y}')\theta \in T_{P_\tau} \uparrow k$, hence, by induction hypothesis, $q(\bar{X}', \bar{Y}')\sigma \in T_{P_{\tau'}} \uparrow k$. The other atoms in the body of $c_{\tau'}$ are true for the same reasons as in the first iteration, hence the whole body is true under σ and $p(\bar{u}, \overline{true}) \in T_{P_{\tau'}} \uparrow k + 1$.