# Inference of Well-Typings for Logic Programs with Application to Termination Analysis

Bruynooghe, M.; Gallagher, John Patrick; Humbeeck, W. Van

*Published in:*
Static Analysis, 12th International Symposium

*Publication date:*
2005

*Document Version*
Publisher's PDF, also known as Version of record

*Citation for published version (APA):*
Bruynooghe, M., Gallagher, J. P., & Humbeeck, W. V. (2005). Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In *Static Analysis, 12th International Symposium* (pp. 35-51). Kluwer Academic Publishers. Lecture Notes in Computer Science Vol. 3672

# Inference of Well-Typings for Logic Programs with Application to Termination Analysis

Maurice Bruynooghe[1,⋆], John Gallagher[2,⋆⋆], and Wouter Van Humbeeck[1]

[1] Katholieke Universiteit Leuven, Department of Computer Science,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
Maurice.Bruynooghe@cs.kuleuven.ac.be
[2] Roskilde University, Computer Science, Building 42.1
DK-4000 Roskilde, Denmark
jpg@ruc.dk

**Abstract.** A method is developed to infer a polymorphic well-typing for a logic program. Our motivation is to improve the automation of termination analysis by deriving types from which norms can automatically be constructed. Previous work on type-based termination analysis used either types declared by the user, or automatically generated monomorphic types describing the success set of predicates. The latter types are less precise and result in weaker termination conditions than those obtained from declared types. Our type inference procedure involves solving set constraints generated from the program and derives a well-typing in contrast to a success-set approximation. Experiments so far show that our automatically inferred well-typings are close to the declared types and result in termination conditions that are as strong as those obtained with declared types. We describe the method, its implementation and experiments with termination analysis based on the inferred types.

## 1 Introduction and Motivation

For a long time, the selection of the right norm was a barrier to progress towards the full automation of termination analysis of logic programs. Recently, type-based norms have been introduced [23] as well as a technique to perform an analysis based on several norms [8]. There is evidence that the combination of both techniques solves in many cases the problem of norm selection [13,1]. However, most logic programs are untyped. Hence, obtaining type information is a new barrier to full automation. Systems for the automated inference of types do exist [7,24]. They derive monomorphic types that approximate the success-set of the program, and such inferred types are used to generate norms in a system for termination analysis [13]. Success types cannot in general be used directly by methods that require a well-typing [1]. In any case, inferred types obtained by

⋆ Work supported by FWO-Vlaanderen and by GOA/2003/08.
⋆⋆ Work supported in part by European Framework 5 Project ASAP (IST-2001-38059), and the IT-University of Copenhagen.

C. Hankin and I. Silveroni (Eds.): SAS 2005, LNCS 3672, pp. 35–51, 2005.

current methods are often less precise than declared types, which are not necessarily over-approximations of the success set. The derived termination conditions are thus weaker than those obtained with declared types. The type inference described in this paper yields well-typings rather than success-set approximations and in all experiments so far yield types – and hence termination conditions – comparable to user-declared types.

We start by sketching an example of type-based termination analysis [1].

*Example 1.* Consider the `append/3` predicate and its abstraction according to the type signature `append(list(T),list(T),list(T))`. Each argument is abstracted by the type-based `list(T)` norm that abstracts a term by the number of subterms of type `list(T)` and the type-based `T` norm that abstracts a term by the number of subterms of type `T` (subscripts $l$ and $e$ for abstracted variables).

```
append([],L,L).
append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).
append(1,0, 1+Ll,Le, 1+Ll,Le).
append(1+1+Xsl,1+Xe+Xse, 1+Ysl,Yse, 1+1+Zsl,1+Xe+Zse):-
  append(1+Xsl,Xse, 1+Ysl,Yse, 1+Zsl,Zse).
```

This suffices to infer that a call to append/3 terminates if it is `list(T)`-rigid[1] in either the first or the last argument. A goal independent type inference [7,24] infers the type append(list(any),any,any), giving rise to the abstract program:

```
append(1,0, 1+La, 1+La).
append(1+1+Xsl,1+Xa+Xsa, 1+Ysa, 1+1+Xa+1+Zsa):-
  append(1+Xsl,Xsa, 1+Ysa, 1+Zsa).
```

The subscripts $l$ and $a$ of abstracted variables correspond to respectively the `list(any)` and `any`-norm; a term of type `any` has only subterms of type `any`, so the second and third argument have only an `any`-abstraction. The termination condition for the third argument is weaker than with the declared type as it requires `any`-rigidity and this corresponds to groundness.

In this paper, the signature $\texttt{append(a}_1\texttt{(T),a}_2\texttt{(T),a}_2\texttt{(T))}$ is inferred, with the types defined as $\texttt{a}_1\texttt{(T)} \longrightarrow \texttt{[]; [T|a}_1\texttt{(T)]}$ and $\texttt{a}_2\texttt{(T)} \longrightarrow \texttt{[T|a}_2\texttt{(T)]}$. The type $\texttt{a}_1\texttt{(T)}$ is equivalent to `list(T)`; the type $\texttt{a}_2\texttt{(T)}$ may look odd as it lacks a "base case" but it gives a well-typing. Calls such as `append([a],[b|X],Y)` are well-typed, and give rise to well-typed calls in their computations. In short "well-typed programs do not go wrong" even with such peculiar types. Now, the abstracted program is:

```
append(1,0, 1+La2,LT, 1+La2,LT).
append(1+1+Xsa1,1+XT+XsT, 1+Ysa2,YsT, 1+1+Zsa2,1+XT+ZsT):-
  append(1+Xsa1,XsT, 1+Ysa2,YsT, 1+Zsa2,ZsT).
```

Hence calls terminate when $\texttt{a}_1$-rigid in the first or $\texttt{a}_2$-rigid in the third argument.

[1] Rigid: all instances have the same size under the norm.

As the next example shows, a call from outside can extend the type of a predicate.

*Example 2.* The naive reverse procedure is given by the clauses

```
rev([],[]).
rev([X|Xs],Zs) :- rev(Xs,Ys),append(Ys,[X],Zs]).
```

together with the clauses for `append`. The inferred signatures and types are

```
t1(T) --> [T|t1(T)]; []      rev(t2(T),t1(T)).
t2(T) --> [T|t2(T)]; []      app(t1(T),t1(T),t1(T))
```

Note that the two types denote the same set of terms. The analysis derives two distinct types because the cons-functors of both do not interact with each other.

*Example 3.* A program to transpose a matrix represented as a list of rows [1]:

```
transpose(A,B) :- transpose_aux(A,[],B).
transpose_aux([], W, W).
transpose_aux([R|Rs],Z,[C|Cs]) :-
       row2col(R,[C|Cs],Cls1,[],Acc), transpose_aux(Rs,Acc,Cls1).
row2col([],[],[],A,A).
row2col([X|Xs],[[X|Ys]|Cols],[Ys|Cols1],B,C) :-
       row2col(Xs,Cols,Cols1,[[]|B],C).
```

The inferred signature and type definitions are as follows:

```
t1(T) ⟶ []; [t4(T)|t1(T)]      transpose(t1(T),t2(T))
t2(T) ⟶ []; [t3(T)|t2(T)]      transpose_aux(t1(T),t2(T),t2(T))
t3(T) ⟶ []; [T|t3(T)]          row2col(t3(T),t2(T),t2(T),t2(T),t2(T))
t4(T) ⟶ []; [T|t4(T)]
```

The types `t3(T)` and `t4(T)` are equivalent and denote a row of elements `T`. Also `t1(T)` and `t2(T)` are equivalent; they denote a list of rows of `T`. These types are equivalent to what a programmer would declare: the first argument of `row2col/5` is a row and all others are lists of rows. Types inferred by over-approximation of success sets using current techniques, even when using a goal-directed analysis with the goal `transpose(X,Y)` are less accurate.

We define the basic notions of types and set constraints in Section 2; we present the type inference procedure in Section 3. The implementation, complexity and some experiments in both type inference and the use of the types in termination analysis are described in Section 4. An extension for obtaining more polymorphism is given in Section 5. Finally we discuss related work in Section 6 and future research in Section 7.

## 2 Preliminaries

### 2.1 Types

For type definitions, we adopt the syntax of Mercury [19]. *Type expressions (types)*, elements of $\mathcal{T}$, are constructed from an infinite set of type variables (parameters) $\mathcal{V}_\mathcal{T}$ and an alphabet of ranked type symbols $\Sigma_\mathcal{T}$; these are disjoint from the set of variables $V$ and alphabet of functors $\Sigma$ used to construct terms. Variable free types are called monomorphic; the others polymorphic. Type substitutions of the form $\{T_1/\tau_1, \ldots, T_n/\tau_n\}$ with the $T_i$ parameters and the $\tau_i$ types define mappings from types to types by the simultaneous replacement of the parameters $T_i$ by the corresponding types $\tau_i$.

**Definition 1 (Type definition).** *A* type rule *for a type symbol $h/n \in \Sigma_\mathcal{T}$ is of the form $h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k)$; $(k \geq 1)$ where $\bar{T}$ is a $n$-tuple of distinct type variables, $f_1, \ldots, f_k$ are distinct function symbols from $\Sigma$, $\bar{\tau}_i$ $(1 \leq i \leq k)$ are tuples of corresponding arity from $\mathcal{T}$, and type variables in the right hand side, if any, are from $\bar{T}$*[2]*. A* type definition *is a finite set of type rules where no two rules contain the same type symbol on the left hand side, and there is a rule for each type symbol occurring in the type rules.*

A *predicate signature* is of the form $p(\bar{\tau})$ and declares a type $\tau_i$ for each argument of the predicate $p/n$. The mapping $\bar{\tau}_i \rightarrow h(\bar{T})$ can be considered the type signature of the function symbol $f_i$. As in Mercury [19], a function symbol can occur in several type rules, hence can have several type signatures.

A *typed* logic program consists of a logic program, a type definition and a predicate signature for each predicate of the program. Given a typed logic program, a type checker can verify whether the program is well-typed, i.e., that the types of the actual parameters passed to a predicate are an instance of the predicate's type signature. To formalize the well-typing, we first inductively define the well-typing of a term.

**Definition 2.** *A* variable typing *is a mapping from variables to types. A term $t$ has type $h(\bar{\tau})$ (notation $t : h(\bar{\tau})$) under a variable typing $\mu$ iff either $t$ is a variable $X$ and $\mu(X) = h(\bar{\tau})$ or $t$ is of the form $f(t_1, \ldots, t_n)$, the type rule for $h(\bar{T})$ has an alternative $f(\tau_1, \ldots, \tau_n)$ and, for all $i$, $t_i$ has type $\tau_i\{\bar{T}/\bar{\tau}\}$.*

**Definition 3 (Well-typing).** *A typed program $P$ has a well-typing if each clause $p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m \in P$ has a variable typing $\mu$ that satisfies:*

1. *Let $p(\tau_1, \ldots, \tau_n)$ be the predicate signature of $p/n$. Then $t_i$ has the type $\tau_i$ under the variable typing $\mu$ $(1 \leq i \leq n)$.*
2. *For $1 \leq j \leq m$, let $B_j = q(s_1, \ldots, s_l)$ and $q(\tau_1, \ldots, \tau_l)$ be the predicate signature of $q/l$. Then there is a type substitution $\theta$ such that, for all $k$, $s_k$ has type $\tau_k\theta$ under the variable typing $\mu$.*

[2] The last condition is known as *transparency* and is necessary to ensure that well-typed programs cannot go wrong [17,10].

*Example 4.* Given a type definition `list(T)` $\longrightarrow$ `[]; [T | list(T)]` the signature `append(list(T),list(T),list(T))` gives a well-typing of the program of Example 1. The variable typing of the first clause is {`L/list(T)`} and that of the second clause is {`X/T,Xs/list(T),Ys/list(T),Zs/list(T)`}.

To establish the connection with set constraints (Section 2.2) we formalize the denotation of a type. Let $D : \mathcal{V}_\mathcal{T} \to 2^{\mathsf{Term}_\Sigma}$ be a mapping from parameters to sets of ground terms. Let $h(\bar{\tau})$ be defined by the type rule $h(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k)$. Using $e_j$ to denote the $j^{th}$ element in a sequence $\bar{e}$, the denotation of $h(\bar{\tau})$ with respect to $D$, written $\mathsf{Den}_D(h(\bar{\tau}))$ is inductively defined as:

1. For all $T \in \mathcal{V}_\mathcal{T}$, $\mathsf{Den}_D(T) = D(T)$.
2. $\mathsf{Den}_D(h(\bar{\tau})) = \{f_i(\bar{s}) \mid 1 \leq i \leq k, s_j \in \mathsf{Den}_D(\tau_{i_j}\{\bar{T}/\bar{\tau}\})$ for all $j\}$.

**Proposition 1.** *Let $t[\bar{X}]$ denote a term with variables $\bar{X}$; $\mu$ a variable typing and $D : \mathcal{V}_\mathcal{T} \to 2^{\mathsf{Term}_\Sigma}$ a mapping from type variables to sets of ground terms. Then $t[\bar{X}]$ has type $h(\bar{\tau})$ under $\mu$ iff $\mathsf{Den}_D(h(\bar{\tau})) \supseteq \{t[\bar{X}]\{\bar{X}/\bar{s}\} \mid s_i \in \mathsf{Den}_D(\mu(X_i))\}$.*

### 2.2 Set Constraints for Well-Typings

**Set Constraints and Their Solutions.** Set expressions are terms constructed from an infinite set of set variables $\mathcal{V}_\mathcal{S}$ and the same alphabet of functors $\Sigma$ as used for constructing terms. Given a mapping $V : \mathcal{V}_\mathcal{S} \to 2^{\mathsf{Term}_\Sigma}$ from set variables to sets of ground terms, one can inductively define the denotation for set expressions $e$ with respect to $V$, written $\mathsf{Den}_V(e)$, as follows:

1. For all $s \in \mathcal{V}_\mathcal{S}$, $\mathsf{Den}_V(s) = V(s)$.
2. $\mathsf{Den}_V(f(e_1, \ldots, e_n)) = \{f(s_1, \ldots, s_n) \mid s_i \in \mathsf{Den}_V(e_i), 1 \leq i \leq n\}$.

The set constraints that we consider are of two kinds, namely $t_1 = t_2$ where $t_1, t_2 \in \mathcal{V}_\mathcal{S}$ and $t_1 \supseteq f(e_1, \ldots, e_n)$ where $t_1 \in \mathcal{V}_\mathcal{S}$ and $f(e_1, \ldots, e_n)$ is a set expression. We call set constraints of the first kind *equality* constraints and those of the second kind *containment* constraints.

Let $\mathcal{S}$ be a set of set constraints (or *constraint system*). A *solution* for $\mathcal{S}$ is any mapping $S : \mathcal{V}_\mathcal{S} \to 2^{\mathsf{Term}_\Sigma}$ such that for each constraint the following holds.

1. For all $t_1 = t_2 \in S$, $\mathsf{Den}_S(t_1) = \mathsf{Den}_S(t_2)$.
2. For all $t_1 \supseteq f(e_1, \ldots, e_n) \in S$, $\mathsf{Den}_S(t_1) \supseteq \mathsf{Den}_S(f(e_1, \ldots, e_n))$.

**Solved Form and Normal Form.** A constraint system $\mathcal{S}$ is in *solved form* if, for each equality constraint $t_1 = t_2$, $t_1$ has no other occurrences in $\mathcal{S}$.

Given a constraint system, one can derive an equivalent solved form by repeatedly taking an equality constraint $t_1 = t_2$ where $t_1$ has other occurrences and substituting $t_1$ by $t_2$ (or alternatively, replacing the equation by $t_2 = t_1$ and substituting $t_2$ by $t_1$) throughout the other constraints. Any resulting equalities $t = t$ are removed. As each such step reduces the number of set variables on the

left hand side of an equality with other occurrences, and no new variables are introduced, the process terminates and yields a solved form.

Let $\mathcal{S}$ be a constraint system in solved form, and let $t \in \mathcal{V}_\mathcal{S}$ be a set variable. Then $t$ is *constrained* in $\mathcal{S}$ if $t$ appears on the left hand side of a constraint, otherwise $t$ is *unconstrained* in $\mathcal{S}$. Note that a constrained set variable in a solved form occurs either in the left hand side of one equality constraint or in the left hand side of one or more containment constraints. In constructing a solution for a constraint system $\mathcal{S}$ in solved form, one can freely choose a denotation for its unconstrained variables. We denote a solution for the unconstrained variables in $\mathcal{S}$ by $U$. Denote by $S[U]$ any solution of $\mathcal{S}$ that extends $U$.

**Definition 4 (Minimal solution).** *A solution $S[U]$ of $\mathcal{S}$ is* minimal *with respect to $U$ iff for each solution $S'[U]$, it holds that for all set variables $s$, $S[U](s) \subseteq S'[U](s)$.*

We often omit $U$ when it is not relevant and denote a solution by $S$.

**Proposition 2.** *Let $S$ be a minimal solution of a constraint system $\mathcal{S}$ in solved form and $t$ a set variable constrained by containment constraints. $f(\bar{s}) \in \mathsf{Den}_S(t)$ iff there is a containment constraint $t \supseteq f(\bar{e})$ such that $f(\bar{s}) \in \mathsf{Den}_S(f(\bar{e}))$.*

**Definition 5 (Normal form).** *A constraint system is in* normal form *if it is in solved form and additionally the following conditions are satisfied.*

1. *It does not contain two distinct containment constraints $t \supseteq f(e_1, \ldots, e_n)$ and $t \supseteq f(e'_1, \ldots, e'_n)$.*
2. *All $e_i$ in containment constraints $t \supseteq f(e_1, \ldots, e_n)$ are set variables.*

Note that 1 corresponds to the requirement that functions symbols are distinct in the right hand side of a type rule. A constraint system $\mathcal{S}$ can be normalised by applying the following operations until a fixpoint is reached.

1. If $\mathcal{S}$ contains $t \supseteq f(e_1, \ldots, e_n)$ and $t \supseteq f(e'_1, \ldots, e'_n)$ with $e_1, \ldots, e_n, e'_1, \ldots, e'_n$ set variables then replace the latter by the constraints $e_1 = e'_1, \ldots, e_n = e'_n$.
2. If $\mathcal{S}$ contains $t \supseteq f(e_1, \ldots, e_j, \ldots, e_n)$ where $e_j$ is not a set variable, then replace it by $t \supseteq f(e_1, \ldots, s, \ldots, e_n)$ and $s \supseteq e_j$ with $s$ a fresh set variable.
3. Apply the rules for deriving a solved form.

**Proposition 3.** *The reduction to normal form terminates. Moreover, if $\mathcal{S}$ is a constraint system and $\mathcal{S}'$ is the normal form obtained by applying the procedure above, then every solution of $\mathcal{S}'$ is also a solution of $\mathcal{S}$.*

Given a normalised constraint system with a set variable $t$, we define $type(t)$ as a type that has the same denotation as the minimal solution of $t$ as follows.

First, define a directed graph with the set variables as nodes. For each constraint $t \supseteq f(s_1, \ldots s_n)$ add, for all $i$, the arc $(t, s_i)$; for each constraint $t = s$

add the arc $(t, s)$. Note that unconstrained variables have no out-arcs. For each constrained set variable $t$, define $params(t)$ to be the set of unconstrained variables reachable from $t$ in the graph. For each unconstrained variable $s$ define a unique type parameter $T_s$. For each variable $t$ constrained by containment constraints, define a unique type symbol $\tau_t/n$ where $n = |params(t)|$.

Now, for each set variable $t$ define $type(t)$ as $T_t$ if $t$ is unconstrained, as $type(s)$ if $t$ is constrained by an equality constraint $t = s$, and as $\tau_t(T_1, \ldots, T_n)$ if $t$ is constrained by containment constraints where $T_1, \ldots, T_n$ are the type parameters corresponding to $params(t)$ (enumerated in some order). To construct the type rules, let $t$ be a constrained variable, and $t \supseteq f_1(\bar{t}_1), \ldots, t \supseteq f_m(\bar{t}_m)$ the containment constraints having $t$ on the left. Then construct a type rule $type(t) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_m(\bar{\tau}_m)$ where $\bar{\tau}_1, \ldots, \bar{\tau}_m$ are obtained from $\bar{t}_1, \ldots, \bar{t}_m$ by substituting each set variable $t_{i,j}$ by $type(t_{i,j})$.

*Example 5.* Consider the set variables $\mathtt{a}_1$, $\mathtt{a}_2$, $\mathtt{a}_3$ and the solved form $\mathtt{a}_1 \supseteq$ `[]`, $\mathtt{a}_1 \supseteq$ `[x|`$\mathtt{a}_1$`]`, $\mathtt{a}_3 \supseteq$ `[x|`$\mathtt{a}_3$`]`, $\mathtt{a}_2 = \mathtt{a}_3$.

The associated directed graph is $\{(\mathtt{a}_1, \mathtt{x}), (\mathtt{a}_1, \mathtt{a}_1), (\mathtt{a}_2, \mathtt{a}_3), (\mathtt{a}_3, \mathtt{x}), (\mathtt{a}_3, \mathtt{a}_3)\}$ The set variable `x` is unconstrained; let $type$`(x) = X`. We have $params(\mathtt{a}_1)$ `= {x}` and $params(\mathtt{a}_3)$ `= {x}`. We use $\tau_{\mathtt{a}_i} = \mathtt{a}_i$, so $type(\mathtt{a}_1)$ `=` $\mathtt{a}_1$`(X)` and $type(\mathtt{a}_3)$ `=` $\mathtt{a}_3$`(X)`. Hence the derived type rules are $\mathtt{a}_3$`(X)` $\longrightarrow$ `[X|` $\mathtt{a}_3$`(X)]` and $\mathtt{a}_1$`(X)` $\longrightarrow$ `[]; [X |` $\mathtt{a}_1$`(X)]`. Finally, $type(\mathtt{a}_2)$ `=` $\mathtt{a}_3$`(X)` because $\mathtt{a}_2 = \mathtt{a}_3$.

From Proposition 2 and the way the types are derived the following proposition follows immediately.

**Proposition 4.** *Let $\mathcal{S}$ be a constraint system in normal form and let $S[U]$ be a minimal solution. Let $type(s)$ be as defined above and let $\rho$ denote the type definition derived from $\mathcal{S}$. For all $u \in domain(U)$ define $D(type(u)) = U(u)$. Then, for each set variable $s$ it holds that $\mathsf{Den}_D(type(s)) = \mathsf{Den}_S[U](s)$.*

## 3 Inference of a Well-Typing

The purpose of type inferencing is to derive a typed program, that is, a type definition and a set of predicate signatures such that the program is well-typed. Whereas well-typing allows the type of a call to be an instance of the declared type, we will derive types such that they are equal. In Section 5 we outline a method for deriving truly polymorphic well-typings. Here, the approach is to associate a set variable with each type in the signatures of the predicates and one with each variable in the program code and to formulate a constraint system whose solution denotes a well typing. Then the constraint system is reduced to normal form. According to Proposition 3, its solutions are solutions of the original system, hence well-typings. From the normal form, the type definition is extracted as described in Section 2.2 and the predicate signatures are obtained by taking the types $type(s)$ of the corresponding set variables.

### 3.1 Generation of Constraints

Let $P$ be a program. We introduce fresh set variables $p_1, \ldots, p_n$ for each predicate $p/n$ of $P$ and a fresh set variable $t_x$ for each variable $x$ of $P$[3]. In concrete examples we reuse the program variables as set variables in the constraints (that is, $t_x = x$), since there can be no confusion between them. The constraint system for a program is the union of the constraint systems generated for each atom in the program. The constraints generated from an atom $p(u_1, \ldots, u_n)$ are:

$$\{p_j \supseteq u_j \mid \text{if } u_j \text{ is not a variable}\} \cup \{p_j = u_j \mid \text{if } u_j \text{ is a variable}\}$$

*Example 6.* Consider the `append/3` program of Example 1. Using the set variables `ap1, ap2` and `ap3` for the `append/3` predicate, we obtain:

- From `append([],L,L)`: `ap1` $\supseteq$ `[]`, `ap2 = L`, `ap3 = L`.
- From `append([X|Xs],Ys,[X|Zs])`: `ap1`$\supseteq$`[X|Xs]`, `ap2 = Ys`, `ap3`$\supseteq$`[X|Zs]`.
- From `append(Xs,Ys,Zs)`: `ap1 = Xs`, `ap2 = Ys`, `ap3 = Zs`.

A normal form of this system consists of the constraints

```
ap1 ⊇ []    ap1 ⊇ [X|ap1]   ap3 ⊇ [X|ap3]
Ys = ap3    L = ap3         Xs = ap1      ap2 = ap3   Zs = ap3
```

As shown in Example 5, we obtain the following types and signature:

```
ap1(X) ⟶ []; [X | ap1(X)]          append(ap1(X),ap3(X),ap3(X))
ap3(X) ⟶ [X | ap3(X)]
```

While the type of the first argument is isomorphic to the `list(T)` type, that of the second and third argument is not as the `[]` alternative is not included. Interestingly, this type is accepted by Mercury [19]. It is only when `append/3` is called from elsewhere in the program as e.g. in the `rev` program of Example 2 that our type inference extends the type `ap3(X)` with a base case. The type inference on the `rev` program still results in two distinct (although equivalent) types. Although we are used to a signature `append(list(T),list(T),list(T))`, nothing in the code of `append/3` imposes this; `append(list(T),mylist(T),mylist(T))` where `mylist(T)` ⟶ `mynil; [X | mylist(X)]` is an equally good signature. In fact, unless there is a call that imposes a base case, the choice of the base case is open, so one can argue that `ap3(X)`, a type without a base case is the most general and the most natural one.

**Theorem 1.** *The type signatures and the type rules derived from the normal form of the constraints generated from a program $P$ are a well-typing for $P$.*

The proof follows immediately from Propositions 1, 3 and 4 (see [3]).

[3] We assume program clauses do not share variables and predicates $p/n$ and $p/m$ with $n \neq m$ do not occur.

## 4 Implementation and Experiments

The algorithm for type inference system consists of four main stages: (i) generation of the constraints from the program text, (ii) realisation of a solved form, (iii) normalisation and (iv) generation of the parameterised type definitions. Of these, normalisation is the only stage whose implementation requires careful consideration in order to be able to apply the system to larger programs.

*Constraint Generation.* One constraint is generated for each argument of each atom (see Section 3.1). This is achieved in a single pass over the program.

*Solved Form.* Constraint generation implies that the number of constraints is linear in the size of the program. Collecting the set of all the equalities, we compute the set of equivalence classes such that all members of an equivalence class are equal to each other. This can be done in time linear in the number of equality constraints. An element of each class is selected; denote by $rep(s)$ the selected element of $s$'s class. The constraints of the solved form then consist of (i) the set of equalities $\{s = rep(s) \mid s \text{ is different from } rep(s)\}$ and (ii) the containment constraints with each variable $s$ replaced by $rep(s)$. Given a suitable representation of the equivalence classes (see the discussion on *union-find* below) the substitution can be done in time proportional to the number of containment constraints. The resulting system is in solved form. Thus reduction to solved form can be achieved in linear time (with respect to the size of the program).

*Normal Form.* Normalisation is achieved starting from the containment constraints of the solved form. As described in Section 2.2, normalisation causes new constraints to be added, which can destroy solved form.

We focus on the removal of non-normal constraints $t \supseteq f(\bar{s}_1)$, $t \supseteq f(\bar{s}_2)$; the other case of non-normal constraints is trivial and can be removed in one pass. The algorithm for producing normal form is as follows, in outline.

```
Initialise equivalence classes, one class per variable;
while (not in normal form) {
  Pick a pair of constraints t1 ⊇ f(...) and t2 ⊇ f(...),
    where t1 and t2 are in the same equivalence class;
  Generate the appropriate constraints to remove the violation;
  Adjust equivalence classes using the generated equalities;
endwhile
```

The adjustment of the equivalence classes is essentially merging; when a constraint $s = t$ is generated we merge the equivalence classes of which $s$ and $t$ respectively are members. All the containment constraints whose left-hand-sides are in the same equivalence class are stored with the representative element of that class. The management of the equivalence classes uses the well-known *union-find* algorithms [21], so that the adjustment of the equivalence classes, and the location of the representative for a given class, can be done in close to constant time.

Thus the time taken to normalise is roughly linear in the number of constraints generated during normalisation. This is not directly determined by the

size of the program, since it depends on the distribution of variables in the program, the number of clauses for each predicate, and so on. However for typical programs the number of generated constraints is roughly proportional to the size of the program.

*Conversion to parametrised type definitions.* The procedure for finding the parameters involves constructing the dependency graph and finding the reachable unconstrained variables from each constrained variable, as described in Section 3. The time required for reachability computation is proportional to the number of normalised constraints, for each constrained variable.

In summary, each stage can be achieved efficiently in time roughly proportional to the size of the program. In our Prolog implementation, the elements of the equivalence classes in the union-find algorithm are stored in a balanced tree, thus giving logarithmic-time rather than constant-time execution of the *find* operation. Our experiments confirm that the running time of the type inference is roughly $O(n.log(n))$ where $n$ is the size of the program.

### 4.1 Inference Experiments

We applied the procedure to a range of programs from the termination analysis literature as well as many other programs (including the implementation of the procedure itself). The procedure shows reasonable scalability: space does not permit a detailed table of statistics so we quote a few timings to give an impression. The largest program we attempted is the Aquarius compiler benchmark (4,192 clauses, 19,976 generated constraints, 18,168 normalisation constraints) for which type inference takes approximately 100 seconds on a Macintosh Powerbook G4. The Chat parser (515 clauses, 2,010 generated constraints, 1,613 normalisation constraints) requires 4.5 seconds. Programs of 100 clauses or less are analysed in fractions of a second. The software runs in Ciao or SICStus Prolog and can be downloaded from `http://www.ruc.dk/~jpg/Software/`. A sample of derived types can be found in [3].

### 4.2 Termination Analysis Experiments

We took a set of 45 small programs from [1] (most of them in turn are from the experiments in [13]) which included declared types. We compared the termination conditions obtained from the inferred types with those obtained from the declared types. We did so using the TerminWeb analyser [20]. On all examples, the termination conditions were equivalent.

In a second experiment, we inferred regular types [6] that approximate the success set of the program and used them for type-based termination analysis. Regular types are not always well-typings. As TerminWeb expects well-typings, we used the cTI termination analyser [13] for this experiment. The system is weaker than TerminWeb and cannot prove termination for 4 of the programs. For 3 programs, termination conditions are obtained with the well-typing but not with the regular types. For 14 programs, the termination conditions are

equivalent. For the remaining 24 programs, the well-typing results in stronger termination conditions. Typically, using the regular types, some argument is required to be ground while rigidity of some type constituent suffices when using the well-typing.

It is interesting to compare the inferred types with the declared types. For 27 of the 45 programs the inferred type is equivalent to the declared types in the sense that there is a simple renaming of type symbols that maps the inferred types to the declared types. The reverse mapping is not always possible, because sometimes distinct types are inferred that are a renaming of each other (and hence of a single declared type). Moreover, in most remaining cases one can say that the inferred type is more precise in the sense that the type allows fewer cases. Typically, a base case is missing as in the type `ap3(X)` ⟶ `[X | ap3(X)]` of the third argument of `append`. For two programs, `der` and `parse`, the analysis distinguishes somewhere two types whereas the declared type has a single type that is the union of both. For the program `minimum` shown in Example 9 of Section 5 there is a more substantial difference. The declared type signature is `minimum(t(X),X)` with type rule `t(X)` ⟶ `void; tree(X,t(X),t(X))`. The code in question does not access the right branch of the tree, hence there is no reason to infer it is a tree; the type inference derives the signature `minimum(t1(X,Y),X)` with `t1(X,Y)` ⟶ `void; tree(X,t1(X,Y),Y)`. This difference is irrelevant when analysing termination. In this case one can observe that the declared type is an instance of the inferred type, since the denotations of `t(X)` and `t1(X,t(X))` are the same.

This experiment suggests that the types we infer are comparable to those one would declare. Often they are identical, and in the remaining cases, the most frequent situation is that the solved form that corresponds to the declared types is an extension of the solved form derived by our analysis.

## 5 Towards Inference of a Polymorphic Well-Typing

So far we derive a single signature for a predicate `p` that is valid for all its occurrences. While we do derive parametric types, our types are not truly polymorphic, because we insist that the type of a call is identical to the signature of the predicate rather than being an instance of it. When using the types for type-based termination analysis, polymorphic types are potentially more useful since the norms are more simple and more reuse of results is feasible [2,13]. We develop an extension where the type of calls can be different instances of the predicates signatures. First we illustrate the difficulty of achieving this.

*Example 7.* Consider the artificial program $P$ consisting of the clause `p :- append([a],[b],M), append([M],[M],R).` together with $P_{\mathtt{app}}$, the definition of `append`. The relevant part of the normal form of the constraint system generated from $P_{\mathtt{app}}$ and the extracted well-typing are respectively

```
ap1 ⊇ []      ap1 ⊇ [X|ap1]          ap3 ⊇ [X|ap3]      ap2 = ap3
ap1(X) ⟶ []; [X | ap1(X)]            append(ap1(X),ap3(X),ap3(X))
ap3(X) ⟶ [X | ap3(X)]
```

The extra constraints on `append` coming from the `p` clause are `ap1` $\supseteq$ `[a]`, `ap2` $\supseteq$ `[b]`, `ap3` = `M`, `ap1` $\supseteq$ `[M]`, `ap2` $\supseteq$ `[M]`, and `ap3` = `R`. The constraints on `ap1` and `ap2` give rise to `M = X`, `X` $\supseteq$ `a`, `X` $\supseteq$ `b` and `ap3` $\supseteq$ `[]`. Finally, `ap3` = `M` enforces the same type for `X` and `ap3`; hence we obtain the signature `append(`$ap_1$`,`$ap_3$`,`$ap_3$`)` with the types $ap_1 \longrightarrow$ `[]; [`$ap_3$ | $ap_1$`]` and $ap_3 \longrightarrow$ `[]; a; b; [`$ap_3$ | $ap_3$`]`.

Note that one cannot obtain types equivalent to the latter signature by instantiating the type parameter of the former. Moreover, we obtain an imprecise type for $ap_3$ that includes `a` and `b` as alternatives because the constraints imply that all calls to `append` have the same type.

**Procedure for Deriving Polymorphic Types.** We first introduce some concepts and notations. A predicate $p$ *depends directly* on a predicate $q$ when $q$ occurs in the right hand side of a clause with $p$ in the head. A set variable $s$ *depends directly* on a set variable $t$ when $t$ occurs in the right hand side of a constraint with $s$ in the left hand side. In both cases, the *depends* relation is the transitive closure of the directly depends relation. With $P_p$, we denote the part of a program defining predicate $p$ and the predicates $p$ depends on. With $\mathcal{S}_P$, we denote the constraint system generated by program $P$. With $\mathcal{S}^{\bar{p}}$, we denote the part of the normal form of $\mathcal{S}$ that contains all constraints with on the left hand side either one of the $p_i$ or a set variable on which one of the $p_i$ depends, i.e., the part of the normal form needed to construct the complete type definitions of the types $type(p_i)$. With $\rho_i(\mathcal{S})$ we denote a renaming of $\mathcal{S}$ where each set variable $s$ is replaced by $s^i$. Finally, when using $s_{\equiv}$ in the context of $\mathcal{S}$, we mean either $s$ itself or a $t$ such that $s = t$ belongs to the normal form of $\mathcal{S}$.

Now consider the partitioning of a program in two parts $P$ and $Q$ such that if $P$ has a clause with head $p$, then it has all clauses with as head either $p$ or predicates on which $p$ depends[4]. Our goal is to derive a well-typing for all predicates such that the variable typing in $Q$ of calls to $P$ are instances of the (polymorphic) signatures of the predicates in $P$. As shown in Example 7, this is not straightforward to achieve. For each call $p(\bar{t})$ in $Q$ to a predicate in $P$, we assume that the function $id(p(\bar{t}))$ returns an index that is unique for the call. From $P$ we generate the constraint system $\mathcal{S}_P$ as described in Section 3.1. When generating $\mathcal{S}_Q$, calls $p(\bar{t})$ to predicates in $P$ are treated differently. Instead of the constraints $p_j\ rel\ t_j$ (with $rel \in \{=, \supseteq\}$), we generate $\rho_{id(p(\bar{t}))}(p_j)\ rel\ t_j$ (the left hand side is renamed); moreover we add to $\mathcal{S}_Q$ the constraint system $\rho_{id(p(\bar{t}))}(\mathcal{S}_P^{\bar{p}})$, a renaming of the constraints relevant for $type(p_j)$ (for all $j$). Creating a different instance for each call ensures that each call can have a distinct well-typing. Note that $\mathcal{S}_P$ and $\mathcal{S}_Q$ do not share any set variables.

Next, the following operations are exhaustively applied on (the normal form of) $\mathcal{S}_P$ and $\mathcal{S}_Q$.

1. Let $q$ be a set variable from $S_P$ with $type(q)$ not a type parameter. If, for some $i$, ${q^i}_{\equiv} \supseteq f(\bar{t}) \in S_Q$ and there is no $\bar{s}$ such that $q_{\equiv} \supseteq f(\bar{s}) \in S_P$ (i.e., $q$ contributes to the type signature of one or more predicates in $P$ and $type(q)$ has no case for functor $f$ while $type(q^i)$ of the signature of the call with

[4] More generally, one could consider a partition of strongly connected components.

identifier $i$ does) then add $q \supseteq f(\bar{r})$ to $\mathcal{S}_P$ with $\bar{r}$ new set variables[5] and, for all $j$ such that $q^j$ exists in $Q$, add $\rho_j(q \supseteq f(\bar{r}))$ to $\mathcal{S}_Q$ (all copies in $Q$ are updated).

2. Let $s$ and $t$ be different set variables in $S_P$ such that $s$ depends on $t$ or $t$ on $s$. If, for some $i$, $s^i_{\equiv} = t^i_{\equiv} \in \mathcal{S}_Q$ and $s_{\equiv} = t_{\equiv} \notin \mathcal{S}_P$, then add $s = t$ to $\mathcal{S}_P$ and, for all $j \neq i$ such that $s^j$ exists in $Q$, add $\rho_j(s = t)$ to $\mathcal{S}_Q$. This rule is needed because, if $type(s)$ is different from $type(t)$, then there is no way —because of the dependency— that their instances can be equal.

Finally the (polymorphic) type signatures for the predicates defined in $P$ are extracted from $S_P$. The extraction of the types from $\mathcal{S}_Q$ needs a small adjustment. For a predicate $p$ defined in $P$, the type of its $j^{th}$ argument $type(p^i_j)$ is $type(p_j)\{s_1/type(s^i_1), \ldots, s_k/type(s^i_k)\}$ with $\{s_1, \ldots, s_k\} = params(type(p_j))$.

*Example 8.* We reconsider Example 7. $P$ consists of the `append` clauses. $\mathcal{S}_P^{\overline{app}}$; the relevant part of the solved form is as follows:

$$\texttt{ap}_1 \supseteq \texttt{[]} \quad \texttt{ap}_1 \supseteq \texttt{[X|ap}_1\texttt{]} \quad \texttt{ap}_2 = \texttt{ap}_3 \quad \texttt{ap}_3 \supseteq \texttt{[X|ap}_3\texttt{]}$$

$\mathcal{S}_Q$ consists of

$$\begin{array}{llll}
\texttt{ap}^1_1 \supseteq \texttt{[]} & \texttt{ap}^1_1 \supseteq \texttt{[X}^1\texttt{|ap}^1_1\texttt{]} & \texttt{ap}^1_2 = \texttt{ap}^1_3 & \texttt{ap}^1_3 \supseteq \texttt{[X}^1\texttt{|ap}^1_3\texttt{]} \\
\texttt{ap}^1_1 \supseteq \texttt{[a]} & & \texttt{ap}^1_2 \supseteq \texttt{[b]} & \texttt{ap}^1_3 = \texttt{M} \\
\texttt{ap}^2_1 \supseteq \texttt{[]} & \texttt{ap}^2_1 \supseteq \texttt{[X}^2\texttt{|ap}^2_1\texttt{]} & \texttt{ap}^2_2 = \texttt{ap}^2_3 & \texttt{ap}^2_3 \supseteq \texttt{[X}^2\texttt{|ap}^2_3\texttt{]} \\
\texttt{ap}^2_1 \supseteq \texttt{[M]} & & \texttt{ap}^2_2 \supseteq \texttt{[M]} & \texttt{ap}^2_3 = \texttt{R}
\end{array}$$

The normal form is:

$$\begin{array}{llll}
\texttt{ap}^1_1 \supseteq \texttt{[]} & \texttt{ap}^1_2 = \texttt{ap3}^1 & \texttt{ap}^1_3 \supseteq \texttt{[]} & \texttt{X}^1 \supseteq \texttt{a} \quad \texttt{M} = \texttt{ap}^1_3 \\
\texttt{ap}^1_1 \supseteq \texttt{[X}^1\texttt{|ap}^1_1\texttt{]} & & \texttt{ap}^1_3 \supseteq \texttt{[X}^1\texttt{|ap}^1_3\texttt{]} & \texttt{X}^1 \supseteq \texttt{b} \\
\texttt{ap}^2_1 \supseteq \texttt{[]} & \texttt{ap}^2_2 = \texttt{ap}^2_3 & \texttt{ap}^2_3 \supseteq \texttt{[]} & \texttt{X}^2 = \texttt{ap}^1_3 \quad \texttt{R} = \texttt{ap}^2_3 \\
\texttt{ap}^2_1 \supseteq \texttt{[X}^2\texttt{|ap}^2_1\texttt{]} & & \texttt{ap}^2_1 \supseteq \texttt{[X}^2\texttt{|ap}^2_3\texttt{]} &
\end{array}$$

Rule 1 applies on $\texttt{ap}_3$, the constraint $\texttt{ap}_3 \supseteq \texttt{[]}$ is added to $\mathcal{S}_P$ ($\texttt{ap}^1_3 \supseteq \texttt{[]}$ and $\texttt{ap}^2_3 \supseteq \texttt{[]}$ are already in $\mathcal{S}_Q$) and the extracted types are:

```
ap1(X) ⟶ []; [X | ap1(X)]          append(ap1(X),ap3(X),ap3(X))
ap3(X) ⟶ []; [X | ap3(X)]
```

The signature of the first call is `append(type(`$\texttt{ap}^1_1$`),type(`$\texttt{ap}^1_2$`),type(`$\texttt{ap}^1_3$`))` which is an instance of the above; the instance of the type parameter `X` is given by `type(`$\texttt{X}^1$`)` which is `t1` $\longrightarrow$ `a; b`. Similarly, in the second call, the type parameter is instantiated into `type(`$\texttt{X}^2$`)` = `type(`$\texttt{ap}^1_3$`)` which is the type `ap3(t1)`.

*Example 9.* This example illustrates the need for the second rule.

```
minimum(tree(X, void, Y), X).
minimum(tree(U, Left, V), W) :- minimum(Left, W).
p(S,M) :- minimum(tree(a,S,S),M).
```

[5] They are unconstrained, hence $type(r_k)$ are new type parameters in the type signature of $p$.

Let $P$ consist of the first two clauses; the solved form of $\mathcal{S}_P$ is:

$$\begin{array}{llll} \texttt{min}_1 \supseteq \texttt{tree(X,min}_1\texttt{,Y)} & \texttt{min}_2 = \texttt{X} & \texttt{U = X} & \texttt{W = X} \\ \texttt{min}_1 \supseteq \texttt{void} & \texttt{Left = min}_1 & \texttt{V = Y} & \end{array}$$

This gives a signature with two parameters, namely `minimum(tr(X,Y),X)` with $\texttt{tr(X,Y)} \longrightarrow \texttt{void; tree(X,tr(X,Y),Y)}$. The solved form of $\mathcal{S}_Q$ is

$$\begin{array}{llllll} \texttt{min}_1^1 \supseteq \texttt{tree(X}^1\texttt{,min}_1^1\texttt{,min}_1^1\texttt{)} & \texttt{min}_2^1 = \texttt{X}^1 & \texttt{Y}^1 = \texttt{X}^1 & \texttt{S} = \texttt{X}^1 & & \\ \texttt{min}_1^1 \supseteq \texttt{void} & \texttt{M} = \texttt{X}^1 & \texttt{p}_1 = \texttt{min}_1^1 & \texttt{p}_2 = \texttt{X}^1 & \texttt{X}^1 \supseteq \texttt{a} & \end{array}$$

This system implies the constraint $\texttt{Y}^1 = \texttt{min}_1^1$ while $\texttt{min}_1$ depends on `Y` in $\mathcal{S}_P$. Hence $\texttt{Y} = \texttt{min}_1$ has to be added to $\mathcal{S}_P$. For $\texttt{min}_1$, this gives the constraints $\texttt{min}_1 \supseteq \texttt{tree(X,min}_1\texttt{,min}_1\texttt{)}$ and $\texttt{min}_1 \supseteq \texttt{void}$ hence we obtain the signature `minimum(tr(X),X)` with $\texttt{tr(X)} \longrightarrow \texttt{void; tree(X,tr(X),tr(X))}$. For $p/2$ the signature is `p(tr(t),t)` with $\texttt{t} \longrightarrow \texttt{a}$.

## 6 Related Work

We can contrast this work to previous work on inferring types for logic programs in which a regular approximation of the success set (minimal Herbrand model) of a program is computed [16,25,5,11,6,22]. We derive a well-typing, which may or may not be a safe approximation of the success set. As a result our approach is not based directly on abstract interpretation, and the inference algorithm has a different structure, based on solving constraints rather than computing a fixpoint.

Our procedure resembles in some ways the set constraint approximations of logic programs developed by Heintze and Jaffar [9], as well as earlier work on deriving regular types from functional programs [18,12]. We also generate set constraints and solve them, but again, our constraints do not represent an over-approximation of the success set in contrast to the cited works. Because we aim at well-typing instead of approximating the success set, our set constraints are much simpler than those of Heintze and Jaffar. In particular there are no intersections in our set expressions, and this allows an efficient solution procedure. Marlow and Wadler [15] describe the automatic derivation of types for Erlang using a similar approach, namely the generation of set constraints capturing the well-typing requirements followed by a constraint solving procedure. Their type system is somewhat more expressive than ours, including a limited form of type complement, and the constraints generated require a more complex solution procedure. However their approach yields truly polymorphic types such that the calls are subtypes of the type signature, and thus their constraint solutions methods could be applicable in our future work in extending Section 5. Christiansen [4] also describes a method of generating type declarations that give a well-typing, using a constraint-solving approach, but his method requires some given types.

Finally we note that unlike classical type inference for ML and other typed languages, we assume no basic types. In part of [14], the authors describe (polymorphic) type reconstruction for logic programs: given a set of types and a type for each functor, they derive types for the predicates and the variables of the

program. It is noteworthy that they point out that it has been shown that the problem is undecidable unless the type of body occurrences of a recursive polymorphic predicate is *identical* to the type of the predicate (we impose this too). The main difference with our approach is that we do not provide any type definitions in advance but construct new definitions during the analysis. We share the latter property with the work in [25]; however, to our understanding, the authors do not infer parametric types - type variables are merely names for types that are defined by their own type rules - and their types are less precise than ours, since they are success set approximations.

## 7 Conclusion

We have presented a method for automatically deriving polymorphic well-typings for logic programs, along with its implementation and the results of some experiments. Distinguishing features of our approach are: (1) No types are assumed, the analysis constructs its own types; (2) recursive calls to a predicate are assumed to have the same type as the original call to the predicate; (3) set constraints impose only conditions for well-typing, not conditions for approximating the success set; (4) the same function symbol can be used in different type rules, i.e., a function symbol can have several type signatures. The experiments show that the inferred types are useful for termination analysis; indeed we may claim to have solved the problem of type inference for deriving norms, since we could not find any example where a user-declared type gave better termination conditions than our automatically derived types.

Future work will focus on two aspects. Firstly, we will develop the approach to polymorphism described in Section 5. Secondly we will investigate to what extent the inferred types could be used for error detection. As the procedure derives a well-typing for every program, it may seem that the possibilities are limited, but there are clear cases when the call constraints for a predicate are not consistent with any intended solution of the constraints derived from the predicate definition, and in such cases an error is indicated. For example, any call to `append` in which the first argument contains a function other than `[]` or `[.|.]` is erroneous. The exact conditions for such errors are the subject of future research.

## Acknowledgements

We wish to thank Tom Schrijvers for finding errors in an earlier draft, and for useful discussions. He also verified that the generated types were accepted by the Mercury type checker.

## References

1. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. Draft, 2004.

2. M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. Reuse of results in termination analysis of typed logic programs. In *Static Analysis, SAS 2002*, volume 2477 of *LNCS*, pages 477–492, 2002.
3. M. Bruynooghe, J. Gallagher, and W. Van Humbeeck. Inference of well-typings for logic programs with application to termination analysis. Technical Report CW 409, Dept. Comp. Sc., Katholieke Universiteit Leuven, 2005.
4. H. Christiansen. Deriving declarations from programs (extended abstract). In *CPP'97, Workshop on Constraint Programming for Reasoning about Programming, Leeds*, 1997.
5. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Logic in Computer Science, LICS'91*, pages 300–309, 1991.
6. J. P. Gallagher and D. de Waal. Fast and precise regular approximation of logic programs. In *Logic Programming, ICLP'94*, pages 599–61, 1994.
7. J. P. Gallagher and G. Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In *Practical Aspects of Declarative Languages, PADL 2002*, volume 2257 of *LNCS*, pages 243–261, 2002.
8. S. Genaim, M. Codish, J. P. Gallagher, and V. Lagoon. Combining norms to prove termination. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation, VMCAI 2002*, volume 2294 of *LNCS*, pages 126–138, 2002.
9. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Principles of Programming Languages, POPL'90*, pages 197–209, 1990.
10. P. M. Hill and R. W. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.
11. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.
12. N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Principles of Programming Languages, POPL'82*, pages 66–74, 1982.
13. V. Lagoon, F. Mesnard, and P. J. Stuckey. Termination analysis with types is more accurate. In *Logic Programming, ICLP 2003*, volume 2916 of *LNCS*, pages 254–268, 2003.
14. T. L. Lakshman and U. S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Logic Programming, ISLP 1991*, pages 202–217, 1991.
15. S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *ICFP*, pages 136–149, 1997.
16. P. Mishra. Towards a theory of types in Prolog. In *Logic Programming, ISLP 1984*, pages 289–298, 1984.
17. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.
18. J. C. Reynolds. Automatic construction of data set definitions. In *Information Processing 68*, pages 456–461, 1996.
19. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
20. C. Taboch, S. Genaim, and M. Codish. TerminWeb: Semantic based termination analyser for logic programs, 2002. http://www.cs.bgu.ac.il/ mcodish/TerminWeb.
21. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

22. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–210, 1994.
23. W. Vanhoof and M. Bruynooghe. When size does matter. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation, LOPSTR 2001*, volume 2372 of *LNCS*, pages 129–147, 2002.
24. C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *Static Analysis, SAS 2002*, pages 102–116, 2002.
25. J. Zobel. Derivation of polymorphic types for Prolog programs. In *Logic Programming, ICLP 1987*, pages 817–838, 1987.