Termination Analysis with Types is More Accurate

Vitaly Lagoon[1], Fred Mesnard[2], and Peter J. Stuckey[1]

[1] Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{lagoon,pjs}@cs.mu.oz.au

[2] IREMIA
Université de La Réunion, France
fred@univ-reunion.fr

Abstract. In this paper we show how we can use size and groundness analyses lifted to regular and (polymorphic) Hindley/Milner typed programs to determine more accurate termination of (type correct) programs. Type information for programs may be either inferred automatically or declared by the programmer. The analysis of the typed logic programs is able to completely reuse a framework for termination analysis of untyped logic programs by using abstract compilation of the type abstraction. We show that our typed termination analysis is uniformly more accurate than untyped termination analysis for regularly typed programs, and demonstrate how it is able to prove termination of programs which the untyped analysis can not.

1 Introduction

Logic programming languages are increasingly typed languages. While Mycroft and O'Keefe [22] showed how to include Hindley/Milner types in logic programs, Prolog has never really made use of types. But new logic programming languages such as Gödel [16], Mercury [24] and HAL [13] include strong types as an important part of the language. On the other hand new techniques for inferring types of Prolog programs [10, 12] are increasingly in use. In this paper we investigate the impact of types on universal left termination analysis of logic programs.

The following example shows why termination analysis of typed programs can give better accuracy than untyped analysis. We use Mercury [24] syntax for type definitions and declarations.

```
:- type list(T) ---> [] ; [T|list(T)].
:- type erk ---> a ; b(erk) ; c.
:- pred g(list(erk)).
g(W) :- X = [[a],[R]], Y = [[S,c],[]],
        append(X,Y,Z), Z = [U|V], append(U,U,W).
:- pred append(list(T),list(T),list(T)).
append(A, B, C) :- A = [], B = C.
append(A, B, C) :- A = [D|E], C = [D|F], append(E,B,F).
```

The goal g(W) cannot be proven to universally terminate if we use the term size norm (because the term size of X is unknown), nor will it terminate if we use the list size norm (because the list size of U is unknown). On the other hand, the typed termination analysis we present proves that g(W) is always terminating. Typed termination analysis can determine that the list skeleton of U is fixed, and hence the goal terminates. Note that this relies on polymorphic size analysis.

Our approach maps the original program to a *type separated program* where each subtype is considered separately. Each variable is split into size components for each type. For example X is split into three variables XLL (which corresponds to the type list(list(erk))), XL (list(erk)) and XE (erk).

Each primitive constraint is mapped to its size effect on each subtype separately. The size of a term for a subtype is the number of subterms of that term that match that type. For example the term $t =$[[a],[R]] has 3 subterms, t, [[R]], and [], matching type list(list(erk)), and four matching list(erk), while the number of erk subterms is one more than those in R.

Each call is mapped to calls where a variable is replaced by its (typed) components. Difficulties arise for polymorphic calls where the call has a more specific type than the predicate called. This is overcome by calling the predicate once for each subtype that maps to the type parameter. The code below is the (simplified) type separated program for the program above. Note how call append(X,Y,Z) maps to two calls to the typed append, one where type parameter T is matched to list(erk) and one where it is matched to erk.

```
g(WE,WL) :- XLL = 3, XL = 4, XE = 1 + RE, YLL = 3, YL = 4, YE = SE + 1,
            append(XLL,XL,YLL,YL,ZLL,ZL), append(XLL,XE,XLL,YE,ZLL,ZE),
            ZLL = 1 + VLL, ZL = UL + VL, ZE = UE + VE,
            append(UL,UE,UL,UE,WL,WE).
append(AL,AT,BL,BT,CL,CT) :- AL = 1, AT = 0, BL = CL, BT = CT.
append(AL,AT,BL,BT,CL,CT) :- AL = 1 + EL, AT = D + ET,
            CL = 1 + FL, CT = D + FT, append(EL,ET,BL,BT,FL,FT).
```

We can perform typed size analysis for the original program using this program. Similarly typed rigidity analysis simply interprets $+$ as $\wedge$ and constants as *true*. Surprisingly the *untyped* termination analysis of this program gives the correct termination information for the original program, and more accurate information than same untyped analysis on the original program.

The contributions of this paper are:

- We provide a correct termination analysis for regular and polymorphic Hindley/Milner typed programs.
- We show that for regular typed programs the results are uniformly more accurate than the untyped analysis (assuming no widening is required).
- We give an implementation of typed termination analysis and experiments showing the accuracy benefits.

Due to space considerations the proofs for theorems on typed analysis are omitted, they can be found in [18].

Types have long been advocated as a means to improve logic programming termination analysis, e.g. [4, 19]. De Schreye *et al.*, e.g. [9], were among the first to study the use of inferred *typed norms* in their works on automatic termination analysis. They pointed out that measuring the same term differently with respect to types complicates the analysis. Another solution to this problem is proposed in [14], by copying each parameter for each norm in each predicate, where norms are either given by the user or based on inferred regular types. In contrast to our proposal, these works did not address the issues related to polymorphic types. Ironically, termination analyses for Mercury (e.g. [25]) are currently untyped.

The closest work to ours is that of Bruynooghe *et al.* [7, 26, 5]. The typed termination analysis framework of [26] is similar to the approach defined herein, but the size and rigidity relations for a procedure are considered separated for subtype. This makes the analysis information uniformly less accurate than the approach herein, it also means that the approach is not guaranteed to be more accurate than an untyped analysis even for regular typed programs (the same counterexample as in Example 8). The approach of [26] cannot be applied to all polymorphically typed programs (e.g. the call in Example 7), while the extension in [5] to arbitrary polymorphically typed programs can give incorrect analysis results (e.g. for Example 8) although this can be corrected (see [6]). There are no experiments reported in [7, 26, 5].

The next section recalls types for logic programming. Section 3 presents the typed analysis with correctness and accuracy results. Section 4 describes our experiments and concludes.

2 Preliminaries

In the following we assume a familiarity with the standard definitions and notation for (constraint) logic programs as described in [17], and with the framework of abstract interpretation [8]. In particular, we assume familiarity with semantics $T_P \uparrow \omega$ and $S_P \uparrow \omega$ defined by the least fixpoints of the immediate consequence operator T_P and the non-ground consequence operation S_P. We assume *terms* are made up using tree constructors with arities from a set Σ_{tree} and (Herbrand) variables $\mathcal{V}_{tree}$.

We assume the rules of the program are in *canonical form*. That is, procedure calls and rule heads are normalized such that parameters are distinct variables, and unifications are broken into sequences of simple constraints of the form $x = y$ or $x = f(y_1, \ldots, y_n)$ where $x, y_1, \ldots, y_n$ are distinct variables. In addition, the heads of each rule for predicate p/n are required to be identical atoms $p(x_{p1}, \ldots, x_{pn})$, and every variable not appearing in the head of a rule is required to appear in exactly one rule. It is straightforward to convert any logic program to an equivalent program in this form.

2.1 Typed Logic Programs

The techniques of abstract compilation of typed analyses presented in this work are applied to *typed logic programs*. A typed logic program is a logic program where each program variable x is associated with its respective *type description* $\mathsf{type}(x)$. We shall be interested in two forms of typed programs: *regular typed programs*, where $\mathsf{type}(x)$ is a (monomorphic) deterministic regular type, and *Hindley/Milner typed programs* where $\mathsf{type}(x)$ is a (polymorphic) Hindley/Milner type. The types can be either prescribed by the programmer or inferred by some type inference algorithm (e.g. [21, 11]).

Both kinds of types are defined using the same language of types. We adopt the Mercury syntax of *type definitions* [21].

Type expressions (or *types*) $\tau \in \mathsf{Type}$ are constructed using *type constructors* Σ_{type} and *type parameters* $\mathcal{V}_{type}$. Each type constructor $g/n \in \Sigma_{type}$ must have a unique definition.

Definition 1. *A* type definition *for $g/n \in \Sigma_{type}$ is of the form*

$$\texttt{:- type}\ \ g(\nu_1, \ldots, \nu_n)\ \ \texttt{--->}\ \ f_1(\tau_1^1, \ldots, \tau_{m_1}^1)\ ;\ \cdots\ ;\ f_k(\tau_1^k, \ldots, \tau_{m_k}^k).$$

where $\nu_1, \ldots, \nu_n$ are distinct type parameters (in $\mathcal{V}_{type}$), $\{f_1/m_1, \ldots, f_k/m_k\} \subseteq \Sigma_{tree}$ are distinct tree constructor/arity pairs, and $\tau_1^1, \ldots, \tau_{m_k}^k$ are type expressions in Type involving at most parameters $\nu_1, \ldots, \nu_n$.

Note that we allow the same tree constructor to appear in multiple type definitions (unlike strict Hindley/Milner types).

Example 1. Example type definitions are given below:

```
:- type list(T) ---> [] ; [T|list(T)].³
:- type nest(T) ---> e(T) ; n(list(nest(T))).
:- type list2(T) ---> [] ; [T|list2(T)].
```

Note how `[]` is overloaded as part of `list` and `list2`.

Type definitions define deterministic regular tree grammars in an obvious way. Formally, a grammar $\mathcal{G}(\tau)$ corresponding to the type τ is a tuple $\mathcal{G}(\tau) = \langle \tau, N(\tau), \Delta(\tau) \rangle$, where τ is the start symbol, and $N(\tau)$ and $\Delta(\tau)$ are respectively the sets of *non-terminals* and *productions*.

If $\tau \in \mathcal{V}_{type}$ is a type parameter, then $N(\tau) = \{\tau\}$ and $\Delta(\tau) = \emptyset$. Otherwise $\tau = \sigma(g(\nu_1, \ldots, \nu_n))$ for some type substitution σ on $\{\nu_1 \ldots, \nu_n\}$ and the type definition for g/n is :

$$\texttt{:- type}\ \ g(\nu_1, \ldots, \nu_n)\ \ \texttt{--->}\ \ f_1(\tau_1^1, \ldots, \tau_{m_1}^1)\ ;\ \cdots\ ;\ f_k(\tau_1^k, \ldots, \tau_{m_k}^k).$$

The sets $N(\tau)$ and $\Delta(\tau)$ are defined respectively as the least sets satisfying:

$$N(\tau) = \tau \cup \bigcup_{\substack{i=1\ldots k\\ j=1\ldots m_i}} N(\sigma(\tau_j^i))$$

$$\Delta(\tau) = \left\{ \begin{array}{c} \tau \!\rightarrow \sigma(f_1(\tau_1^1, \ldots, \tau_{m_1}^1)) \\ \vdots \\ \tau \!\rightarrow \sigma(f_k(\tau_1^k, \ldots, \tau_{m_k}^k)) \end{array} \right\} \cup \bigcup_{\substack{i=1\ldots k\\ j=1\ldots m_i}} \Delta(\sigma(\tau_j^i))$$

[3] We write the list constructor `[|]` in the usual Prolog notation.

We assume that type definitions are such that the grammar for any type is *finite*, thus disallowing definitions like `:- type g(T) ---> a ; b(g(list(T)))`. This is a common restriction, although see [23] for the use of such types.

Example 2. The productions for the grammar of `list2(nest(erk))` are

```
list2(nest(erk))  → [] ; [ nest(erk) | list2(nest(erk)) ]
nest(erk)         → e(erk) ; n(list(nest(erk)))
list(nest(erk))   → [] ; [ nest(erk) | list(nest(erk)) ]
erk               → a ; b(erk) ; c
```

while for `list(U)` they are `list(U) → [] ; [ U | list(U) ]`

For a canonical program P we can lift type to act on predicates p/n since the unique head of rules for p/n defines its unique type. If $p(x_{p1}, \ldots, x_{pn})$ is the head of rules for p/n in P, then $\mathsf{type}(p/n) = p(\mathsf{type}(x_{p1}), \ldots, \mathsf{type}(x_{pn}))$. We sometimes use Mercury syntax to illustrate predicate types, for example for `append` in the introduction.

A basic assumption for this paper is that the programs we analyze are well-typed. In practice this means that type-correctness of a program must be verified by a type checker before the typed analysis can be performed. A ground term t is *well-typed* for ground type τ if $t \in \mathcal{L}(\tau)$, that is the term is in the language of the grammar of type τ. A non-ground term t is well-typed for τ if there is a grounding substitution θ such that $\theta(t)$ is well-typed for τ. An atom $p(t_1, \ldots, t_n)$ is *well-typed* if there is a substitution θ such that each $\theta(t_i)$ is well-typed for type $\mathsf{type}(x_{pi})$ for $1 \leq i \leq n$.

We assume the program is well-typed in one of two senses. Either $\mathsf{type}(x)$ is always a regular type, and every atom occurring in a derivation for a well-typed atom is well-typed. Or $\mathsf{type}(x)$ is always a Hindley/Milner type and the program is Hindley/Milner type correct [22].

We will be interested in discovering sizes and rigidities of terms for each possible subtype. Define the multiset of nodes of term t of type τ which match a particular subtype $\tau' \in N(\tau)$, written as $\{\!\{t : \tau\}\!\}_{\tau'}$.

$$\begin{aligned}
\{\!\{f(t_1, \ldots, t_n) : \tau\}\!\}_{\tau} &= \{f\} \uplus \{\!\{t_1 : \tau_1\}\!\}_{\tau} \uplus \cdots \uplus \{\!\{t_n : \tau_n\}\!\}_{\tau} \\
&\quad \text{where } \tau \rightarrow f(\tau_1, \ldots, \tau_n) \in \Delta(\tau) \\
\{\!\{f(t_1, \ldots, t_n) : \tau\}\!\}_{\tau'} &= \{\!\{t_1 : \tau_1\}\!\}_{\tau'} \uplus \cdots \uplus \{\!\{t_n : \tau_n\}\!\}_{\tau'} \\
&\quad \text{where } \tau \neq \tau',\ \tau \rightarrow f(\tau_1, \ldots, \tau_n) \in \Delta(\tau) \\
\{\!\{v : \tau\}\!\}_{\tau'} &= \{v\} \text{ where } v \in \mathcal{V}_{tree} \\
\{\!\{t : \tau\}\!\}_{\tau'} &= \emptyset \text{ otherwise}
\end{aligned}$$

where $\uplus$ denotes multiset union. We can compute size (number of matching non-variable nodes) and rigidity (there exist no matching variable nodes) for subtypes from these multiset expressions.

$$\begin{aligned}
size(t : \tau, \tau') &= |\{\!\{t : \tau\}\!\}_{\tau'} - \mathcal{V}_{tree}| \\
rigid(t : \tau, \tau') &= (\{\!\{t : \tau\}\!\}_{\tau'} \cap \mathcal{V}_{tree} = \emptyset)
\end{aligned}$$

Example 3. For this and later examples we use shorthands l for `list`, n for `nest`, $l2$ for `list2` and e for `erk`. Consider the term $t \equiv [n([X|Y]), Z]$, the table shows the multisets of nodes, and size and rigidities for each subtype when t is of type $l2(n(e))$.

τ	$\{\!\{t : l2(n(e))\}\!\}_\tau$	$size(t : l2(n(e)), \tau)$	$rigid(t : l2(n(e)), \tau)$
$l2(n(e))$	$\{[\|], [\|], []\}$	3	*true*
$l(n(e))$	$\{[\|], Y, Z\}$	1	*false*
$n(e)$	$\{n, Y, Z\}$	1	*false*
e	$\{X, Y, Z\}$	0	*false*

We shall extend our notion of type expressions by introducing two new type construction mechanisms: type intersection, and (named) renaming. These will not be available to the programmer, but used to construct intermediate types for the translation process.

The first is for building the intersections of two types. The type expression $\langle\tau_1 \cap \tau_2\rangle$ defines the intersection of types τ_1 and τ_2, a type with grammar as follows. $\Delta(\langle\tau_1 \cap \tau_2\rangle)$ contains rules

$$\begin{array}{ll}\langle\tau_1 \cap \tau_2\rangle \to f(\langle\tau_{11} \cap \tau_{21}\rangle, \ldots, \langle\tau_{1n} \cap \tau_{2n}\rangle) & \text{where } \tau_1 \to f(\tau_{11}, \ldots, \tau_{1n}) \in \Delta(\tau_1) \\ & \qquad\;\; \tau_2 \to f(\tau_{21}, \ldots, \tau_{2n}) \in \Delta(\tau_2)\end{array}$$

together with rules from $\Delta(\langle\tau_{1i} \cap \tau_{2i}\rangle), 1 \leq i \leq n$. $N(\langle\tau_1 \cap \tau_2\rangle)$ is given by the types $\langle\tau'_1 \cap \tau'_2\rangle$ where $\tau'_1 \in N(\tau_1)$ and $\tau'_2 \in N(\tau_2)$ that occur anywhere in $\Delta(\langle\tau_1 \cap \tau_2\rangle)$ (not just on the left hand side). The start symbol is $\langle\tau_1 \cap \tau_2\rangle$.

The renamed type expression $name.\tau$ where $name \in \mathcal{V}_{tree} \cup \{copy_i \mid i \geq 1\}$ and τ is a type expression, builds a new type which is identical to τ but with different non-terminals. The type $name.\tau$ is defined by the grammar as $\langle name.\tau, \{name.\tau' \mid \tau' \in N(\tau)\}, \Delta\rangle$ where

$$\Delta = \{name.\tau \to f(name.\tau_1, \ldots name.\tau_n) \mid \tau \to f(\tau_1, \ldots \tau_n) \in \Delta(\tau)\}.$$

3 Typed Termination Analysis

We assume the reader is familiar with the untyped termination analysis approach of [20] on how to compute classes of queries for which universal left termination of a pure untyped logic program is guaranteed.

Our approach to typed termination analysis simply maps the typed logic program to an untyped CLP(N) program which represents the size relationships of the original program. We call this the *type separated program*. The *untyped* analysis of this program gives us the termination results for the original typed program.

3.1 The Type Separated Program

The type separated program $\mathsf{type}(P)$ arising from a type program P and a correct typing type is defined as follows. Each rule is mapped to a rule, by mapping each of the literals of the rule to a sequence of literals.

Variables Each variable $v \in \mathcal{V}_{tree}$ with type $\mathsf{type}(v)$ is mapped to a set of typed variables of the form $v.\tau$ where τ is a type. Define $\overline{v : \tau}$ to be the sequence of variables $v.\tau', \tau' \in N(\tau)$ in lexicographic order. The order will be important in order to map calls correctly.

Heads Each head atom $p(x_{p1}, \ldots, x_{pn})$ is directly translated to an atom $p(\overline{x_{p1} : \mathsf{type}(x_{p1})}, \ldots, \overline{x_{pn} : \mathsf{type}(x_{pn})})$. For example, the head atom $q(Z)$ where $\mathsf{type}(Z) = n(e)$ is mapped to the fact $q(Z.e, Z.l(n(e)), Z.n(e))$.

Primitive constraints The primitive constraint $v_1 = v_2$ is mapped to a conjunction of constraints defined as follows. First we build the named instances of the types for v_1 and v_2. Let $\tau_1 = v_1.\mathsf{type}(v_1)$ and $\tau_2 = v_2.\mathsf{type}(v_2)$

Consider $N(\langle \tau_1 \cap \tau_2 \rangle)$ as edges in a bipartite graph with nodes $N(\tau_1)$ and $N(\tau_2)$. Then we can separate these nodes into connected components. For each connected component U we create an equation

$$\Sigma\{x \mid x \in U \cap N(\tau_1)\} = \Sigma\{y \mid y \in U \cap N(\tau_2)\}$$

The intuition is that for the unification to succeed v_1 and v_2 must take a value in the intersection type. And each node in v_1 of type τ_1' can only appear in v_2 as type τ_2' if $\langle \tau_1' \cap \tau_2' \rangle \in N(\langle \tau_1 \cap \tau_2 \rangle)$. Hence the size equation must hold.

Example 4. Consider $X = Y$ where $\mathsf{type}(X) = l(n(e))$ and $\mathsf{type}(Y) = l2(n(e))$. Then $N(\langle X.\mathsf{type}(X) \cap Y.\mathsf{type}(Y) \rangle)$ contains $\{\langle X.l2(n(e)) \cap Y.l(n(e)) \rangle$, $\langle X.n(e) \cap Y.n(e) \rangle$, $\langle X.e \cap Y.e \rangle$, $\langle X.l(n(e)) \cap Y.l(n(e)) \rangle\}$. The connected components are $\{X.l(n(e)), X.l2(n(e)), Y.l(n(e))\}$, $\{X.n(e), Y.n(e)\}$ and $\{X.e, Y.e\}$. The resulting equations are

$$X.l2(n(e) + X.l(n(e)) = Y.l(n(e)), \quad X.n(e) = Y.n(e), \quad X.e = Y.e$$

The sum of sizes of the two kinds of list nodes in X must equal the size of the single kind in Y.

Treatment of the equation $v_0 = f(v_1, \ldots, v_n)$ is similar. First we create the type τ_2 for the right hand side defined by $\langle 1, \{1\} \cup \cup_{i=1}^{n} N(v_i.\mathsf{type}(v_i)), \Delta \rangle$ where $\Delta = \{1 \rightarrow f(v_1.\mathsf{type}(v_1), \ldots, v_n.\mathsf{type}(v_n))\} \cup \cup_{i=1}^{n} \Delta(v_i.\mathsf{type}(v_i))$. The remainder is as for the equation $v_1 = v_2$, where $\tau_1 = v_1.\mathsf{type}(v_1)$. Note the (ab)use of 1 as a type name so that the same equation creation as above holds. The 1 represents the size contribution of the functor f.

Example 5. Consider the equation $A = [D|E]$, where $\mathsf{type}(A) = \mathsf{type}(E) = l(T)$ and $\mathsf{type}(D) = T$. The new type τ_2 consists of rules $1 \rightarrow [D.T|E.l(T)]$, $E.l(T) \rightarrow []$, and $E.l(T) \rightarrow [E.T|E.l(T)]$. The grammar intersection of τ_2 and $A.l(T)$ gives pairs $\{\langle A.l(T) \cap 1 \rangle$, $\langle A.T \cap D.T \rangle$, $\langle A.l(T) \cap E.l(T) \rangle$, $\langle A.T \cap E.T \rangle\}$. The connected components are $\{A.l(T), 1, E.l(T)\}$ and $\{A.T, D.T, E.T\}$. The resulting equations are $A.l(T) = 1 + E.l(T)$ and $A.T = D.T + E.T$.

The equation $A = []$ leads to connected components of the intersection grammar as $\{A.l(T), 1\}$ and $\{A.T\}$. The resulting equations are $A.l(T) = 1$, $A.T = 0$.

Monomorphic calls For a procedure with monomorphic type we translate a body atom $p(v_1, \ldots, v_n)$ as follows. Let $\mathsf{type}(p/n) = p(\tau_1, \ldots, \tau_n)$. Construct new variables $v'_1, \ldots, v'_n$ where $\mathsf{type}(v'_i) = \tau_i$. We translate the call by the translation of each $v_i = v'_i$ followed by the call $p(\overline{v'_1 : \tau_1}, \ldots, \overline{v'_n : \tau_n})$. This maintains the invariant that each p atom has arguments arising from the exact type declared for p/n.

Note if $\mathsf{type}(v_i) = \tau_i$ already we can omit the equations resulting from $v_i = v'_i$ and use $\overline{v_i : \tau_i}$ instead of $\overline{v'_i : \tau_i}$

Example 6. Consider the atom $p(X)$ where $\mathsf{type}(X) = l(n(e))$ and $\mathsf{type}(p/1) = p(l2(n(e)))$, then we create new variable Y of type $l2(n(e))$ and build the equations resulting from $X = Y$ as defined in Example 4 above and the atom $p(Y.e, Y.l(n(e)), Y.l2(n(e)), Y.n(e))$.

Polymorphic calls Handling of polymorphic calls is more complex. The main complexity arises since we must create a new variable whose type *grammar* is an instance of the grammar of the polymorphically typed call. We then will use a call to the polymorphic code for each subtype which matches the type parameter arguments.

Given the program is Hindley/Milner type correct we know that for body atom $p(v_1, \ldots, v_n)$, we have that $\mathsf{type}(p/n) = p(\tau_1, \ldots, \tau_n)$ and there exists type substitution σ such that $\mathsf{type}(v_i) = \sigma(\tau_i), 1 \le i \le n$. Let $\nu_j \in \mathcal{V}_{type}, 1 \le j \le m$ be the type parameters in $\mathsf{type}(p/n)$. For each type parameter ν_j we create a new type $S_j = copy_j.\sigma(\nu_j))$. Define $\sigma' = \{\nu_j \mapsto S_j \mid 1 \le j \le m\}$. We create new variables v'_i where $\mathsf{type}(v'_i) = \sigma'(\tau_i)$. And add the equations resulting from $v_i = v'_i$.

The final step is to add calls to the type separated predicate p. For each combination of $S'_j \in N(S_j)$ for all $1 \le j \le m$ we add the call

$$p(\overline{v'_1 : \tau_1}, \ldots, \overline{v'_n : \tau_n})$$

except that the type variable ν_j is replaced by $copy_j.\sigma(\nu_j)$ in all variable names except where it appears as ν_j (e.g. $v'_i.\nu_j$) where it is replaced by S'_j. Note the construction of the vectors of variables is completed before the type name substitution, to ensure that the lexicographic order agrees with the definition of p in the type separated program.

Example 7. Consider a call $p(X)$ where $\mathsf{type}(X) = l(n(e))$ and $\mathsf{type}(p/1) = p(l(T))$. Then $\sigma = \{T \mapsto n(e)\}$. We create a type $copy_1.n(e)$. Let Y be a new variable where $\mathsf{type}(Y) = l(copy_1.n(e))$. We then create the equations resulting from $X = Y$ (which are analogous to those from Example 4). The unsubstituted call is $p(Y.l(T), Y.T)$. We create a copy for each $\tau' \in N(copy_1.n(e))$. The resulting translation is the last 4 lines of the program shown in Figure 1 (where $copy_1$ is written as c for brevity).

The copying of types avoids confusing $Y.l(copy_1.n(e))$ which represents the list nodes in the outer skeleton of the list Y, with $Y.copy_1.l(n(e))$ which represents the list nodes appearing inside nests in Y.

```
q(Z.e,Z.l(n(e)),Z.n(e)) :- Z.n(e)=1+X.n(e), Z.l(n(e))=X.l(n(e)), Z.e=X.e,
      X.l(n(e)=Y.l(c.n(e))+Y.c.l(n(e)), X.n(e)=Y.c.n(e), X.e=Y.c.e,
      p(Y.l(c.n(e)), Y.c.l(n(e))),
      p(Y.l(c.n(e)), Y.c.n(e)),
      p(Y.l(c.n(e)), Y.c.e).
```

Fig. 1. Type separated rule for `q(Z) :- Z = n(X), p(X).`

The type separated program just translates each rule in P into the corresponding rule in $\mathsf{type}(P)$. The translation of the rule `q(Z) :- Z = n(X), p(X)` where $\mathsf{type}(Z) = n(e)$ and $\mathsf{type}(X) = l(n(e))$ and $\mathsf{type}(p/1) = l(T)$ is shown in Figure 1.

Note that the type separated program can be significantly larger than the original program, but only by a factor equal to the largest number of non-terminals in a type multiplied by the largest number of type parameters appearing in a single predicate type.

3.2 Typed Analysis

The key theorems of this paper are that the untyped analyses of the type separated program are correct with respect to the typed original program. The size analysis computes an approximation of the answer semantics of $\mathsf{type}(P)$ using the abstract domain Size [15].

Theorem 1. *([18] Theorem 23) Analysis using* Size *of the type separated program* $\mathsf{type}(P)$ *is correct with respect to type correct program* P.

Let $post_p^{\mathcal{N}}(\overline{x_{p1}}_{\mathsf{type}}, \ldots, \overline{x_{pn}}_{\mathsf{type}}) \iff C$ *be the result of the size analysis of* $\mathsf{type}(P)$*, then for any well-typed atom* $p(t_1, \ldots, t_n) \in T_P \uparrow \omega$ *we have that* $\{x_{pi}.\tau \mapsto size(t_i : \mathsf{type}(x_{pi}), \tau) \mid 1 \leq i \leq n, \tau \in N(\mathsf{type}(x_{pi}))\}$ *is a solution of* C.

Rigidity analysis translates the $+$ of the type separated program as $\wedge$ and replaces all the numbers by *true*, it then computes an approximation of this CLP(B) program using the abstract domain Pos [2].

Theorem 2. *([18] Theorem 17) Rigidity analysis using* Pos *of the type separated program* $\mathsf{type}(P)$ *is correct with respect to type correct program* P.

Let $post_p^{\mathcal{B}}(\overline{x_{p1}}_{\mathsf{type}}, \ldots, \overline{x_{pn}}_{\mathsf{type}}) \iff C$ *be the result of the rigidity analysis of* $\mathsf{type}(P)$*, then for any well-typed atom* $p(t_1, \ldots, t_n) \in S_P \uparrow \omega$ *we have that* $\{x_{pi}.\tau \mapsto rigid(t_i : \mathsf{type}(x_{pi}), \tau) \mid 1 \leq i \leq n, \tau \in N(\mathsf{type}(x_{pi}))\}$ *is a solution of* C.

3.3 Accuracy of Typed Analysis

The typed analysis is generally much more accurate than the untyped analysis. We can show that for regular typed programs the typed analysis is uniformly more accurate than the untyped analysis.

Theorem 3. *([18] Theorem 18)*
For any regular typed program P, let $post_p^{\mathcal{B}}(\overline{x_{p1}}_{\mathsf{type}}, \ldots, \overline{x_{pn}}_{\mathsf{type}}) \Longleftrightarrow C_{\mathsf{type}}$ be the result of the Pos *rigidity analysis of* $\mathsf{type}(P)$*, and $post_p^{\mathcal{B}}(x_{p1}, \ldots x_{pn}) \Longleftrightarrow C$ be the result of the untyped* Pos *rigidity (groundness) analysis of P. Then*

$$(C_{\mathsf{type}} \wedge \bigwedge_{i=1}^{n} x_{pi} \leftrightarrow (\wedge_{\tau \in N(\mathsf{type}(x_{pi}))} x_{pi}.\tau)) \quad \rightarrow \quad C$$

The same result holds for typed term size analysis, provided no widening operation is used. Since the widening operation for Size is *non-monotonic* more accurate information from the typed analysis is not guaranteed to lead to more accurate results after widening.

Theorem 4. *([18] Theorem 24)*
For any regular typed program P, let $post_p^{\mathcal{N}}(\overline{x_{p1}}_{\mathsf{type}}, \ldots, \overline{x_{pn}}_{\mathsf{type}}) \Longleftrightarrow C_{\mathsf{type}}$ be the result of the Size *analysis of* $\mathsf{type}(P)$ *assuming no widening operations were used, and $post_p^{\mathcal{N}}(x_{p1}, \ldots x_{pn}) \Longleftrightarrow C$ be the result of the untyped* Size *analysis of P. Then*

$$(C_{\mathsf{type}} \wedge \bigwedge_{i=1}^{n} x_{pi} = (\Sigma_{\tau \in N(\mathsf{type}(x_{pi}))} x_{pi}.\tau)) \quad \rightarrow \quad C$$

The accuracy results do not extend to polymorphic analysis, even for rigidity analysis.

Example 8. Consider the following simple program with two possible type declarations for `p`, the first polymorphic and the second monomorphic.

```
:- type pair(U,V) ---> U-V.
:- type foo ---> d ; e ; f.
:- pred p(list(T),list(T)).
:- pred p(list(pair(erk,foo)),list(pair(erk,foo)).
p(A,B) :- A = [].
p(A,B) :- B = [].
```

The rigidity analysis of the predicate `p/2` obtains the respective answers (for brevity we use *lpef* for *list*(*pair*(*erk*, *foo*)), *pef* for *pair*(*erk*, *foo*))

$$(A.list(T) \wedge A.T) \vee (B.list(T) \wedge B.T)$$
$$(A.lpef \wedge A.pef \wedge A.erk \wedge A.foo) \vee (B.lpef \wedge B.pef \wedge B.erk \wedge B.foo)$$

The analysis of a call to polymorphically typed `p` where $T = pef$ gives 3 copies of the answer for `p` conjoined. This is the answer

$$((A.lpef \wedge A.pef) \vee (B.lpef \wedge B.pef)) \wedge ((A.lpef \wedge A.erk) \vee (B.lpef \wedge B.erk)) \wedge ((A.lpef \wedge A.foo) \vee (B.lpef \wedge B.foo)$$

This is less accurate than the answer using the monomorphic type, and in fact less accurate than the untyped groundness analysis result $A \vee B$ when mapped back to these variables.

If the original program P does not have any (type) polymorphic recursive procedures, we can monomorphise the program and analyze this program. Of course then we make no reuse of the analysis of polymorphic code, and the monomorphisation could cause an exponential increase in code size.

In practice the widening operation and the possible inaccuracy resulting from the handling of polymorphic calls do not seem to occur for real programs. In the empirical results in Section 4 the (polymorphic) typed analysis is never less accurate than the untyped analysis.

We conjecture that the typed size analysis is also uniformly more accurate than (untyped) list size analysis for regular typed programs (when widening is not used). The proof techniques of [18] do not apply to this case, but the type separated program contains all the constraints in the CLP(N) program used for list size analysis.

3.4 Level Mappings

We need to show that the type separated program is also correct for the computation of level mappings. This is obvious for monomorphic programs since the size analysis is correct. For polymorphic programs we need to justify the level mappings for each instance used. The result follows from the following theorem that shows that the level mapping never depends on arguments which have parameter types.

Theorem 5. *If there exists a level mapping for a polymorphically typed predicate p, then there exists a level mapping with zero coefficients for the arguments which have type $\tau \in \mathcal{V}_{type}$.*

Proof. (Sketch) It is easy to show that the size constraints C arising in the type separated program for an argument of parameter type $\nu \in \mathcal{V}_{type}$ are such, if θ is a solution of C then $\theta' = \{v.\nu \mapsto 0 \mid \nu \in \mathcal{V}_{type}\} \cup \{v.\tau \mapsto \theta(v.\tau) \mid \tau \notin \mathcal{V}_{type}\}$ is also a solution of C'. This is because no function symbols can appear in positions relating to the parameter type. Suppose $\bar{a}$ is a level mapping for some binary clause $p(x_1, \ldots, x_k)$:-$C, p(y_1, \ldots, y_k)$, Then $\bar{a} \cdot \theta(\bar{x}) > \bar{a} \cdot \theta(\bar{y})$ for every solution θ of C. But since θ' is also a solution we have that $\bar{a}'$ is also a correct level mapping, where $a'_i = a_i$ except for arguments i of parameter type where $a'_i = 0$.

Bruynooghe et al [5] prove a weaker version of this theorem that shows that if there is a level mapping for a monomorphic instance of a polymorphic typed procedure, then there is a level mapping for the polymorphic typed procedure.

The above result also means that the direct termination analysis on the type separated program will not remove possible termination proofs. In the set of binary unfoldings corresponding to the program the first instance of a recursive predicate will compute the appropriate level mapping, and later instances can safely use the same level mapping since they do not differ on arguments whose type is not a type parameter.

3.5 Termination Conditions

The final result is that the termination conditions computed from the type separated program are correct for the original typed program.

Theorem 6. *The typed termination conditions computed from* $\mathsf{type}(P)$ *are correct.*

Let ϕ_p *be a termination condition for* $p(\overline{x_{p1}}_{\mathsf{type}}, \ldots, \overline{x_{pn}}_{\mathsf{type}})$ *in* $\mathsf{type}(P)$*. Then if* $p(t_1, \ldots, t_n)$ *is an atom such that* $\{x_{pi}.\tau \mapsto rigid(t_i : \mathsf{type} x_{pi}, \tau) \mid 1 \leq i \leq n, \tau \in N(\mathsf{type}(x_{pi}))\}$ *is a solution of* ϕ_p*, then this goal terminates.*

Proof. (Sketch) Since the analysis results are correct, and the duplication of calls for polymorphic calls does not change the termination problem, the termination conditions computed are correct.

4 Experiments and Conclusion

We implemented the translation to the type separated program as a source to source transformation for Prolog programs with Mercury style type definitions and declarations. We then passed the resulting CLP(N) programs to the *untyped* termination analyzer of [20]. The typed termination results are simply read from the results of the untyped analysis of $\mathsf{type}(P)$.

In the first experiment we considered (pure Prolog + arithmetic) programs from the first 10 chapters of the book [1], avoiding some almost repeats. We rewrote `question` in order to avoid the 90 anonymous variables in one clause (which made the untyped and typed analysis run out of memory). We manually added polymorphic Hindley/Milner types to these programs and checked well-typedness. We analyzed left termination of the resulting type separated programs for the first predicate in each program, except `query`, where we used the query `?- map(L), color_map(L,[r,g,b])` from page 138 of the text.

Table 1 compares the typed analysis, and untyped analysis using term size and list size. For each procedure we show the termination condition for each analysis, and give a ✓ or = to show when the typed termination is more or equally accurate compared to the untyped analysis. The termination conditions use notation $||\_||_{ls}$, $||\_||_{ts}$, and $||\_||_{bt}$ to represent list size, term size, and binary tree size (number of nodes) metrics respectively. For the predicate `color_map(X,Y)`, `Y` is a list of colors and `X` is a list of regions. A region is a tuple of three elements: a country, a color, and a list of neighbors. The metric $||X||_{24}$ requires that X is a finite list of regions, each having a finite list of neighbors.

The total analysis time for all programs in Table 1 for the two untyped analyses took 0.5 seconds each. The total typed analysis took 5.0 seconds. We ran the cTI termination analyzer, written in SICStus Prolog 3.10.1 using the PPL library [3], a timeout of 3 seconds for each strongly connected component, allowing 4 iterations before widening on an Intel 686, 2.4 GHz, 512 Mb, Linux 2.4.

Table 1. Some programs from Apt's book.

Page#	Query	type-size	term-size		list-size	
114	`sunny`	*true*	*true*	=	*true*	=
115	`neighbour(X,Y)`	*true*	*true*	=	*true*	=
118	`num(X)`	$\|\|X\|\|_{ts}$	$\|\|X\|\|_{ts}$	=	*false*	✓
120	`sum(X,Y,Z)`	$\|\|Y\|\|_{ts} \vee \|\|Z\|\|_{ts}$	$\|\|Y\|\|_{ts} \vee \|\|Z\|\|_{ts}$	=	*false*	✓
122	`mult(X,Y,Z)`	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts}$	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts}$	=	*false*	✓
122	`less(X,Y)`	$\|\|X\|\|_{ts} \vee \|\|Y\|\|_{ts}$	$\|\|X\|\|_{ts} \vee \|\|Y\|\|_{ts}$	=	*false*	✓
124	`list(X)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=
125	`len(X,Y)`	$\|\|X\|\|_{ls} \vee \|\|Y\|\|_{ts}$	$\|\|X\|\|_{ts} \vee \|\|Y\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	✓
126	`member(X,Y)`	$\|\|Y\|\|_{ls}$	$\|\|Y\|\|_{ts}$	✓	$\|\|Y\|\|_{ls}$	=
127	`subset(X,Y)`	$\|\|X\|\|_{ls} \wedge \|\|Y\|\|_{ls}$	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts}$	✓	$\|\|X\|\|_{ls} \wedge \|\|Y\|\|_{ls}$	=
127	`app(X,Y,Z)`	$\|\|X\|\|_{ls} \vee \|\|Z\|\|_{ls}$	$\|\|X\|\|_{ts} \vee \|\|Z\|\|_{ts}$	✓	$\|\|X\|\|_{ls} \vee \|\|Z\|\|_{ls}$	=
129	`select(X,Y,Z)`	$\|\|X\|\|_{ls} \vee \|\|Z\|\|_{ls}$	$\|\|X\|\|_{ts} \vee \|\|Z\|\|_{ts}$	✓	$\|\|X\|\|_{ls} \vee \|\|Z\|\|_{ls}$	=
130	`perm(X,Y)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=
131	`perm1(X,Y)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=
131	`prefix(X,Y)`	$\|\|X\|\|_{ls} \vee \|\|Y\|\|_{ls}$	$\|\|X\|\|_{ts} \vee \|\|Y\|\|_{ts}$	✓	$\|\|X\|\|_{ls} \vee \|\|Y\|\|_{ls}$	=
132	`suffix(X,Y)`	$\|\|Y\|\|_{ls}$	$\|\|Y\|\|_{ts}$	✓	$\|\|Y\|\|_{ls}$	=
132	`sublist(X,Y)`	$\|\|Y\|\|_{ls}$	$\|\|Y\|\|_{ts}$	✓	$\|\|Y\|\|_{ls}$	=
133	`reverse(X,Y)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=
133	`reverse1(X,Y)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=
135	`palindrome(X)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=
136	`question(X)`	*true*	$\|\|X\|\|_{ts}$	✓	*true*	=
137	`color_map(X,Y)`	$\|\|X\|\|_{24} \wedge \|\|Y\|\|_{ls}$	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts}$	✓	*false*	✓
138	`query`	*true*	*false*	✓	*false*	✓
139	`bin_tree(X)`	$\|\|X\|\|_{bt}$	$\|\|X\|\|_{ts}$	✓	*false*	✓
139	`tree_member(X,Y)`	$\|\|Y\|\|_{bt}$	$\|\|Y\|\|_{ts}$	✓	*false*	✓
140	`in_order(X,Y)`	$\|\|X\|\|_{bt}$	$\|\|X\|\|_{ts}$	✓	*false*	✓
140	`front(X,Y)`	$\|\|X\|\|_{bt}$	$\|\|X\|\|_{ts}$	✓	*false*	✓
238	`max(X,Y,Z)`	*true*	*true*	=	*true*	=
240	`ordered(X)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=
241	`ss(X,Y)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=
241	`qs(X,Y)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=
242	`ms(X,Y)`	$\|\|X\|\|_{ls}$	*false*	✓	$\|\|X\|\|_{ls}$	=
247	`search_tree(X)`	$\|\|X\|\|_{bt}$	$\|\|X\|\|_{ts}$	✓	*false*	✓
247	`in(X,Y)`	$\|\|Y\|\|_{bt}$	$\|\|Y\|\|_{ts}$	✓	*false*	✓
248	`minimum(X,Y)`	$\|\|X\|\|_{bt}$	$\|\|X\|\|_{ts}$	✓	*false*	✓
249	`insert(X,Y,Z)`	$\|\|Y\|\|_{bt} \vee \|\|Z\|\|_{bt}$	$\|\|Y\|\|_{ts} \vee \|\|Z\|\|_{ts}$	✓	*false*	✓
249	`delete(X,Y,Z)`	$\|\|Y\|\|_{bt} \vee \|\|Z\|\|_{bt}$	$\|\|Y\|\|_{ts} \vee \|\|Z\|\|_{ts}$	✓	*false*	✓
253	`len1(X,Y)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	✓	$\|\|X\|\|_{ls}$	=

Table 2. Some logic programs from the literature.

Ref.	Query	type-size	term-size		list-size		reg-size
[19]	`flatten_length(X,Y,Z)`	$\lvert\lvert X\rvert\rvert_{lls} \vee (\lvert\lvert X\rvert\rvert_{ls} \wedge \lvert\lvert Y\rvert\rvert_{ls}) \vee (\lvert\lvert Y\rvert\rvert_{ls} \wedge \lvert\lvert Z\rvert\rvert_{ts})$	$\lvert\lvert X\rvert\rvert_{ts} \vee (\lvert\lvert Y\rvert\rvert_{ts} \wedge \lvert\lvert Z\rvert\rvert_{ts})$	✓	$\lvert\lvert X\rvert\rvert_{ls} \wedge \lvert\lvert Y\rvert\rvert_{ls}$	✓	$\lvert\lvert X\rvert\rvert_{lls} \vee (\lvert\lvert X\rvert\rvert_{ls} \wedge \lvert\lvert Y\rvert\rvert_{ts}) \vee (\lvert\lvert Y\rvert\rvert_{ls} \wedge \lvert\lvert Z\rvert\rvert_{ts})$
[26]	`flatten(X,Y)`	$\lvert\lvert X\rvert\rvert_{lls} \vee (\lvert\lvert X\rvert\rvert_{ls} \wedge \lvert\lvert Y\rvert\rvert_{ls})$	$\lvert\lvert X\rvert\rvert_{ts}$	✓	$\lvert\lvert X\rvert\rvert_{ls} \wedge \lvert\lvert Y\rvert\rvert_{ls}$	✓	$\lvert\lvert X\rvert\rvert_{lls} \vee (\lvert\lvert X\rvert\rvert_{ls} \wedge \lvert\lvert Y\rvert\rvert_{ls})$
[14]	`p(X)`	$\lvert\lvert X\rvert\rvert_{ts}$	*false*	✓	*false*	✓	*false*
[14]	`factor(X,Y)`	$\lvert\lvert X\rvert\rvert_{ts}$	*false*	✓	*false*	✓	*false*
[14]	`t(X)`	$\lvert\lvert X\rvert\rvert_{ts}$	*false*	✓	*false*	✓	*false*
Sect. 1	`g(X)`	*true*	*false*	✓	$\lvert\lvert X\rvert\rvert_{ls}$	✓	$\lvert\lvert X\rvert\rvert_{ts}$

The current implementation is fairly naive, as the type separated program introduces many new variables which perform no useful role, and could be eliminated. We believe a slightly specialized termination analyzer for typed programs could remove a significant part of this overhead.

In Table 2, our second experiment, we selected six programs from the literature, including our first example of Section 1. First, we manually added polymorphic Hindley/Milner types and checked well-typedness. The termination condition $||X||_{lls}$ is true when X denotes a finite list of elements, each of which is also a finite list. The accuracy of the typed analysis is striking, although it should be noted that the last four examples were synthesized for this purpose.

Next, we applied bottom-up inference of regular types [11] to the *untyped* programs. To construct the type separated program we used the inferred regular types. The results are shown in the last column of Table 2. In some cases we significantly improve the termination conditions for an *untyped* program without needing any user-added type information. Total analysis times for this set of examples are 0.5 seconds for each of the typed analyses, and 0.1 seconds for each of the untyped analyses.

In summary, typed termination analysis seems in practice to be uniformly more accurate than the untyped analysis, and even a naive implementation is reasonably efficient. There are many further questions to pursue. One promising direction is a hybrid analyzer that first uses untyped analysis and refines to typed analysis where untyped analysis is not able to prove termination.

References

1. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
2. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
3. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. *SAS*, LNCS 2477:213–229, 2002.

4. A. Bossi, N. Cocco, and M. Fabris. Typed norms. *ESOP*, LNCS 582:73–92, 1992.
5. M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. Reuse of results in termination analysis of typed logic programs. *SAS*, LNCS 2477:477–492, 2002.
6. M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. A note on the reuse of results in termination analysis of typed logic programs. Forthcoming TR, Dpt CS, K.U. Leuven, 2003.
7. M. Bruynooghe, W. Vanhoof, and M. Codish. Pos(T) : Analyzing dependencies in typed logic programs. *PSI*, LNCS 2244:406–420, 2001.
8. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *POPL*, 238–252, 1977.
9. S. Decorte, D. De Schreye, and M. Fabris. Exploiting the power of typed norms in automatic inference of interargument relations. TR 246, Dpt CS, , K.U.Leuven, 1997.
10. F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6):751–777, 2001.
11. J. Gallagher and A. de Waal. Fast and precise regular approximations of logic programs. *ICLP*, pages 599–613. MIT Press, 1994.
12. J. Gallagher and G. Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. *SAS*, LNCS 2257:243–261, 2002.
13. M. García de la Banda, B. Demoen, K. Marriott, and P.J. Stuckey. To the Gates of HAL: A HAL tutorial. LNCS, 2441:47–66, 2002.
14. S. Genaim, M. Codish, J. Gallagher, and V. Lagoon. Combining norms to prove termination. *VMCAI*, LNCS 2294:126–138, 2002.
15. N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. PhD thesis, USM de Grenoble, France, 1979.
16. P. Hill and J. Lloyd. *The Gödel Language*. MIT Press, 1994.
17. J. Jaffar, M. Maher, K. Marriott, and P.J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1–3):1–46, 1998.
18. V. Lagoon and P. J. Stuckey. Polymorphic analysis of typed logic programs. TR, Dpt CSSE, University of Melbourne, Australia, 2003. `www.cs.mu.oz.au/~pjs/papers/poly-tr.ps`
19. J. Martin, A. King, and P. Soper. Typed norms for typed logic programs. *LOPSTR*, LNCS 1207:224–238, 1996.
20. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. *SAS*, LNCS 2126:93–110, 2001.
21. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
22. A. Mycroft and R. A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
23. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
24. 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.
25. C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. *SAS*, LNCS 1302:160–171, 1997.
26. W. Vanhoof and M. Bruynooghe. When size does matter. *LOPSTR*, LNCS 2372:129–147, 2002.