Termination Analysis of Typed Logic Programs†

Vitaly Lagoon (lagoon@cs.mu.oz.au)
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia

Fred Mesnard (fred@univ-reunion.fr)
IREMIA
Université de La Réunion, France

Peter J. Stuckey (pjs@cs.mu.oz.au)
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia

Etienne Payet (epayet@univ-reunion.fr)
IREMIA
Université de La Réunion, France

Fausto Spoto (fausto.spoto@univr.it)
Dipartimento di Informatica
Università di Verona, Italy

Abstract. In this paper we show how we can use size and groundness analyses lifted to regular and (polymorphic) Hindley/Milner typed logic 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 define a methodology for mapping a typed logic program to a *type separated* CLP(N) program that allows us to automatically determine termination characteristics of the original program. We demonstrate how the approach 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 [34] 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 [24], Mercury [37] and HAL [20] include strong types as an important part of the language. On the other hand new techniques for inferring types of Prolog programs [16, 19] are increasingly in use. In this paper we investigate the impact of types on universal left termination analysis of logic programs.

† This is an extended version of the paper [28].

The following example shows why termination analysis of typed programs can give better accuracy than untyped analysis. We use Mercury [37] 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 can it be proved to 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 subtype. For example X of type `list(list(erk))` 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 4 matching `list(erk)`, namely `[a]`, `[R]` and `[]` twice. Moreover, the number of `erk` subterms is one more than those that could appear in instances of 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,YLL,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 = DT + ET,
            CL = 1 + FL, CT = DT + 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 the same untyped analysis on the original program.

Another use of typed termination is that we can rely on automatic (deterministic regular) type inference to improve termination for programs. Consider the program:

```
k(M) :- N = [[0,a],[P]], flat2(N,M).

flat2(D,E) :- D = [], E = [].
flat2(D,E) :- D = [F|G], flat2(G,H), append(F,H,E).
```

Then the goal independent type inference of Gallagher [18] determines the deterministic regular types:

```
:- type lla ---> [] ; [la | lla].
:- type la ---> [] ; [any | la].
:- pred k(any).
:- pred flat2(lla,any).
:- pred append(la,any,any).
```

The resulting type separated program is

```
k(MA) :- NLL = 3, NL = 5, NA = 1 + OA + PA,
      flat2(NLL,NL,NA,MA).

flat2(DLL,DL,DA,EA) :- DLL = 1, DL = 0, DA = 0, EA = 0.
flat2(DLL,DL,DA,EA) :- DLL = 1 + GLL, DL = FL + GL, DA = FA + GA,
      flat2(GLL,GL,GA,HA), append(FL,FA,HA,EA).
```

The termination analysis of this type separated program proves that the goal `k(M)` always terminates, while the untyped analysis cannot (as it requires that the first argument of `flat2` should be ground, which is not the case of the term bound to `N`). This illustrates nicely the use of type inference as an orthogonal preprocessing step to improve the termination of existing untyped programs.

It could be argued that taking into account type information for the purposes of termination is no different than asking the user for specific norms for the program. We believe types are better for four reasons (a) in most cases the programmer will be more familiar with types than norms, (b) in many cases the types will already be part of the program for other reasons, such as type error detection, (c) polymorphism is well understood for types whereas this not so clear for norms, and (d) types can be inferred automatically (although there is related work on the inference of norms [14]).

The contributions of this paper are:

- We provide a correct termination analysis for regular and polymorphic Hindley/Milner typed programs.
- We give an implementation of typed termination analysis and experiments showing the accuracy benefits.

Types have long been advocated as a means to improve logic programming termination analysis, e.g. [4, 29]. De Schreye *et al.*, e.g. [15], 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 [22], 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. [38, 17]) are currently untyped.

The closest work to ours is that of Bruynooghe *et al.* [7, 39, 5]. The typed termination analysis framework of [39] is similar to the approach defined herein, but the size and rigidity relations for a procedure are considered separated for each 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 11). The approach of [39] 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 11) although this can be corrected (see [6]). There are no experiments reported in [7, 39, 5].

The next section recalls types for logic programming. Section 3 defines the approach to untyped termination analysis that we will extend to typed programs. Section 4 presents the typed analysis with correctness and accuracy results. Section 5 describes our experiments and results and Section 6 concludes.

2. Preliminaries

In the following we assume a familiarity with the standard definitions and notation for (constraint) logic programs as described in [25], and with the framework of abstract interpretation [11]. 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 $\mathcal{V}_{tree}$.

A *logic program*, or *program*, is a finite set of rules. A *rule*, or *clause*, is of the form $H \leftarrow B$ where H, the *head*, is an atom and B, the *body*, is a finite sequence of literals. A *literal* is either an atom or a primitive constraint. An *atom* has the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol and the t_i's are terms. A *primitive constraint* is of the form $s = t$ where s and t are terms. In program examples, we use the ISO-Prolog syntax (namely `H :- B.`) for rules.

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 $p(x_{p1}, \ldots, x_{pn})$. Finally every variable can only appear in exactly one rule, except those variables appearing in the head of a rule which must appear only in the heads of one predicate. 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. [33, 18]).

Both kinds of types are defined using the same language of types. We adopt the Mercury syntax of *type definitions* [33].

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

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

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)].[1]
:- type nest(T) ---> e(T) ; n(list(nest(T))).
:- type list2(T) ---> [] ; [T|list2(T)].
:- type erk ---> a ; b(erk) ; c.
:- type pair(A,B) ---> A - B.[2]
```

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 :

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

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

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

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

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

[2] `-` used as the infix pairing operator.

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 [35] for the use of such types.

EXAMPLE 2. *For the type* τ_1 =`list2(nest(erk))`, *we have that* $\Delta(\tau_1)$ *is*

```
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
```

and $N(\tau_1)$ *is the nonterminals appearing on the left hand side of these productions.*

For τ_2 = `list(U)`, *then* $\Delta(\tau_2)$ *is the productions*

```
list(U) → [] ; [ U | list(U) ].
```

and $N(\tau_2) = \{$`list(U)`$, U\}$.

For a canonical program P we can lift type to act on predicates p/n since the rules for p/n have the same head. 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 a ground type τ if t is a term in the language of the grammar for the 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 $\mathsf{type}(x_{pi})$ 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 [34].

We will be interested in defining two notions that we will call size and rigidity 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 $\{\!\{t : \tau\}\!\}_{\tau'}$. We have $\{\!\{t : \tau\}\!\}_{\tau'} = \emptyset$ if $\tau' \notin N(\tau)$. Otherwise

$$\begin{aligned}
\{\!\{f(t_1,\ldots,t_n) : \tau\}\!\}_{\tau} &= \{f\} \uplus \{\!\{t_1 : \tau_1\}\!\}_{\tau} \uplus \cdots \uplus \{\!\{t_n : \tau_n\}\!\}_{\tau} \\
&\quad \text{where } \tau \to f(\tau_1,\ldots,\tau_n) \in \Delta(\tau) \\
\{\!\{f(t_1,\ldots,t_n) : \tau\}\!\}_{\tau'} &= \{\!\{t_1 : \tau_1\}\!\}_{\tau'} \uplus \cdots \uplus \{\!\{t_n : \tau_n\}\!\}_{\tau'} \\
&\quad \text{where } \tau \neq \tau',\ \tau \to f(\tau_1,\ldots,\tau_n) \in \Delta(\tau) \\
\{\!\{v : \tau\}\!\}_{\tau'} &= \{v\} \text{ where } v \in \mathcal{V}_{tree},
\end{aligned}$$

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

$$\langle \tau_1 \cap \tau_2 \rangle \to f(\langle \tau_{11} \cap \tau_{21} \rangle, \ldots, \langle \tau_{1n} \cap \tau_{2n} \rangle) \quad \text{where}$$

$$\tau_1 \to f(\tau_{11},\ldots,\tau_{1n}) \in \Delta(\tau_1) \quad \text{and} \quad \tau_2 \to f(\tau_{21},\ldots,\tau_{2n}) \in \Delta(\tau_2)$$

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 Δ is

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

3. Untyped Termination Analysis

In this section we summarize (cf. [30, 21, 31, 32]) the *untyped* approach to computing classes of queries for which universal left termination of a pure untyped logic program is guaranteed. We call such classes *termination conditions*. More precisely, let P be a logic program and q a predicate symbol of P. A termination condition for q is a set TC_q of goals of the form $\leftarrow C, q(\tilde{x})$ where C is a Herbrand constraint such that, for any goal $G \in TC_q$, each derivation of P and G using the left-to-right selection rule is finite. This is the approach implemented in the cTI termination analyser [12].

This analysis uses three main constraint structures: Herbrand terms for the initial program P (seen as a CLP(H) program), non-negative integers, and Booleans (P is abstracted into both a CLP(N) and a CLP(B) program) and consists of six distinct steps, which will be illustrated on the following definition for the predicates `append/3`, `nrev/2` and `app3/4`.

```
append([], B, B).
append([D|E], B, [D|F]) :- append(E, B, F).

nrev([], []).
nrev([E|X], Y) :- nrev(X, Z), append(Z, [E], Y).

app3(X, Y, Z, U) :- append(X, Y, V), append(V, Z, U).
```

Step 1: From Prolog to CLP(N). From the Prolog program P, a CLP(N) program $P^{\mathcal{N}}$ is obtained by applying a symbolic norm. Usually, we use the *term-size* norm, but termination inference for logic program can be based on any linear norm. The symbolic term-size norm is inductively defined as follows:

$$\|t\| \stackrel{\text{def}}{=} \begin{cases} 1 + \sum_{i=1}^{n} \|t_i\| & \text{if } t = f(t_1, \ldots, t_n) \text{ with } n \geq 0; \\ t, & \text{if } t \text{ is a variable.} \end{cases}$$

For example, $\|f(0,0)\| = 3$.

For our running example, we obtain the following CLP(N) clauses:

$append_{\mathcal{N}}(1, b, b).$
$append_{\mathcal{N}}(1 + d + e, b, 1 + d + f) \leftarrow append_{\mathcal{N}}(e, b, f).$

$nrev_{\mathcal{N}}(1, 1).$
$nrev_{\mathcal{N}}(1 + e + x, y) \leftarrow nrev_{\mathcal{N}}(x, z), append_{\mathcal{N}}(z, 2 + e, y).$

$app3_{\mathcal{N}}(x, y, z, u) \leftarrow append_{\mathcal{N}}(x, y, v), append_{\mathcal{N}}(v, z, u).$

Step 2: Computing a numeric model. A model of the CLP(N) program is now computed in the domain Size using the standard widening described in [23]. For each predicate p, the model describes, with a finite conjunction of linear equalities and inequalities denoted by $post_p^{\mathcal{N}}$, the linear inter-argument relations that hold for every solution of p. In our example we obtain the following model:

$$\begin{array}{lll} post^{\mathcal{N}}_{append}(a,b,c) & \iff & a \geq 1 \wedge a + b = c + 1, \\ post^{\mathcal{N}}_{nrev}(x,y) & \iff & x \geq 1 \wedge x = y, \\ post^{\mathcal{N}}_{app3}(x,y,z,u) & \iff & x \geq 1 \wedge x + y \geq 2 \wedge x + y + z = u + 2. \end{array}$$

Step 3: Computing a numeric level mapping. The information provided by the numerical model is crucial to compute a *level mapping* $|\cdot|^{\mathcal{N}}$. Let p be an n-ary predicate symbol in the CLP(N) program. The level mapping associates to each p a function $f_p\colon \mathbb{N}^n \to \mathbb{N}$ that is guaranteed to decrease when going from the head of the clause to each *recursive* call(s), if any, for each clause defining p.

For example, a level mapping $|\cdot|^{\mathcal{N}}$ such that $|nrev_{\mathcal{N}}(x, y)|^{\mathcal{N}} = x$ intuitively means: for each ground instance[3] of each recursive clause defining $nrev_{\mathcal{N}}$, the first argument decreases when going from the head of the clause to the recursive call (since $1+e+x > x$ for each $e, x \in \mathbb{N}$). Since no clause defining $app3_{\mathcal{N}}$ is recursive, the level mapping can be defined so that $|app3(x, y, z, t)|^{\mathcal{N}} = 0$.

The techniques proposed in [21] and [31] reduce its computation to CLP(N) *binary* recursive clauses $p(x_1, \ldots, x_n) \leftarrow C, p(y_1, \ldots, y_n)$, where C is a conjunction of linear constraints, and look for a *linear* level mapping of the form $|p(x_1, \ldots, x_n)|^{\mathcal{N}} = \sum_{k=1}^{n} a_i x_i$, where the coefficients a_i are non-negative integers. Let us briefly explain these two points.

Given a CLP(N) program P, the binarization described in [21] is done by abstracting the binary clause semantics of [8]. An upper approximation of a T_P-like fixpoint is computed with the help of the

[3] That is, where natural numbers have replaced variable symbols.

standard widening and a model of P, and consists of a finite set of *directly* recursive binary clauses. The binarization described in [31] is a direct application of the definition of partial acceptability [32]. With the help of a model of P, a clause with n mutually recursive calls in its body gives rise to a set of n *mutually* recursive binary clauses.

Back to our running example, in both of the above approaches, the only binary clauses to consider are:

$append_{\mathcal{N}}(a,b,c) \leftarrow a+d \geq c, c > d, a+d = c+e, append_{\mathcal{N}}(e,b,d).$
$nrev_{\mathcal{N}}(a,b) \leftarrow a \geq 1+x, nrev_{\mathcal{N}}(x,z).$

So, for the case of directly recursive binary clauses, it remains to find the a_i's verifying $C \rightarrow \sum_{k=1}^{n} a_i x_i > \sum_{k=1}^{n} a_i y_i$. Such coefficients can be obtained by applying an algorithm [36] which is based on linear programming, and is complete (for rational numbers) as it will always provide a linear level mapping if one exists.

Our extension, which is described in [31], consists in first computing a constraint over the coefficients of a generic linear level mapping. Then we generate a concrete level mapping. Notice that for a multi-directional predicate (such as `append/3`) we may get multiple linear level mappings ($|append(a,b,c)|^{\mathcal{N}} = a$ and $|append(a,b,c)|^{\mathcal{N}} = c$). These are combined, with the min operator, into one non-linear level mapping:

$$\begin{array}{ll} |append(a,b,c)|^{\mathcal{N}} & = min(a,c), \\ |nrev(x,y)|^{\mathcal{N}} & = x, \\ |app3(x,y,z,u)|^{\mathcal{N}} & = 0. \end{array}$$

Step 4: From CLP(N) to CLP(B). From the CLP(N) program $P^{\mathcal{N}}$ a CLP(B) program, $P^{\mathcal{B}}$, is obtained by mapping each natural number to *true*, each variable symbol to itself, and addition to logical conjunction.

$append_{\mathcal{B}}(true, b, b).$
$append_{\mathcal{B}}(true \wedge d \wedge e, b, true \wedge d \wedge f) \leftarrow append_{\mathcal{B}}(e,b,f).$

$nrev_{\mathcal{B}}(true, true).$
$nrev_{\mathcal{B}}(true \wedge e \wedge x, y) \leftarrow nrev_{\mathcal{B}}(x,z), append_{\mathcal{B}}(z, true \wedge e, y).$

$app3_{\mathcal{B}}(x,y,z,u) \leftarrow append_{\mathcal{B}}(x,y,v), append_{\mathcal{B}}(v,z,u).$

The purpose of $P^{\mathcal{B}}$ is the one of capturing boundedness dependencies within $P^{\mathcal{N}}$ or, equivalently, rigidity dependencies within the original program. In the domain Pos of positive Boolean formulæ[2], a model for $P^{\mathcal{B}}$ is computed and a Boolean level mapping $|\cdot|^{\mathcal{B}}$ is obtained from the numerical level mapping computed in Step 3. In order to do that, the

$$
\begin{array}{ll}
pre_{\text{append}} & = \nu T \,.\, \lambda(a,b,c)\,. \\
& \left\{ \begin{array}{l} |\text{append}(a,b,c)|^{\mathcal{B}} \\ \forall d,e,f : [(a \leftrightarrow (true \wedge d \wedge e)) \wedge (c \leftrightarrow (true \wedge d \wedge f))] \\ \qquad\qquad\qquad\qquad\qquad\qquad \rightarrow T(d,b,f) \end{array} \right. \\
pre_{\text{nrev}} & = \nu T \,.\, \lambda(x,y)\,. \\
& \left\{ \begin{array}{ll} |\text{nrev}(x,y)|^{\mathcal{B}} & (1) \\ \forall e,x',z : [(x \leftrightarrow (1 \wedge e \wedge x'))] \rightarrow T(x',z) & (2) \\ \forall e,x',z : [(x \leftrightarrow (1 \wedge e \wedge x')) \wedge post^{\mathcal{B}}_{\text{nrev}}(x',z)] & \\ \qquad\qquad\qquad\qquad \rightarrow pre_{\text{append}}(z, 1 \wedge e, y) & (3) \end{array} \right. \\
pre_{\text{app3}} & = \nu T \,.\, \lambda(x,y,z,u)\,. \\
& \left\{ \begin{array}{l} |\text{app3}(x,y,z,u)|^{\mathcal{B}} \\ \forall v : true \rightarrow pre_{\text{append}}(x,y,v) \\ \forall v : post^{\mathcal{B}}_{\text{append}}(x,y,v) \rightarrow pre_{\text{append}}(v,z,u) \end{array} \right.
\end{array}
$$

Figure 1. Calculating Boolean termination conditions for the running example.

translation scheme outlined above is augmented with the association of the logical disjunction $x \vee y$ to $\min(x,y)$: this means that $\min(x,y)$ is a bounded quantity if x or y or both are bounded. Here is what we obtain for the example program:

$$
\begin{array}{ll}
post^{\mathcal{B}}_{append}(a,b,c) & \iff (a \wedge b) \leftrightarrow c, \\
post^{\mathcal{B}}_{app3}(x,y,z,u) & \iff (x \wedge y \wedge z) \leftrightarrow u, \\
post^{\mathcal{B}}_{nrev}(x,y) & \iff x \leftrightarrow y, \\
|append(a,b,c)|^{\mathcal{B}} & = a \vee c, \\
|nrev(x,y)|^{\mathcal{B}} & = x, \\
|app3(x,y,z,u)|^{\mathcal{B}} & = true.
\end{array}
$$

For instance, as we use the term-size norm, this model tells us that for any computed answer θ to a call $\texttt{nrev}(x,y)$, $\theta(x)$ is ground if and only if $\theta(y)$ is ground. In the logic programming case, this is precisely a groundness analysis.[4] The computation of the Boolean model is subject to a timeout on each scc of the call graph.

Step 5: Computing Boolean termination conditions. The information obtained from $P^{\mathcal{B}}$ for each program point is combined with the level mapping by means of the following Boolean μ-calculus formulæ, whose solution gives the desired Boolean termination conditions. The formulæ are shown in Figure 3.

[4] For constraint logic programming, it is a boundedness analysis where the additional expressivity is sensible (see, for instance, Example 6.5.4. in [32]).

For computing the Boolean termination condition pre_{nrev}, we consider the clause

$$nrev_{\mathcal{B}}(x, y) \leftarrow [x \leftrightarrow (1 \wedge e \wedge x')], nrev_{\mathcal{B}}(x', z), append_{\mathcal{B}}(z, 1 \wedge e, y).$$

We are looking for a Boolean relation $T(x, y)$ satisfying the following conditions:

- for each (x, y) in T, the level mapping has to be bounded, which leads to condition (1) in Figure 3.
- the recursive call to `nrev/2` has to terminate, hence condition (2);
- for any state resulting from the evaluation of the first call, the subsequent call to `append/3` has to terminate, giving condition (3);
- finally, we are interested in the weakest solution for T, hence the Boolean termination condition is defined as a greatest fixpoint: $pre_{\text{nrev}} = \nu T \,.\, \lambda(x, y) \,.\, \{(1) \wedge (2) \wedge (3)\}$.

The greatest fixpoint is evaluated with a Boolean μ-solver [9] (inspired from Toupie [10]) and computes in the Pos abstract domain. Another approach, based on backward analysis [26], is proposed in [21]. Solving the equations for our example gives:

$$\begin{array}{ll} pre_{\text{append}}(a, b, c) & = a \vee c, \\ pre_{\text{nrev}}(x, y) & = x, \\ pre_{\text{app3}}(x, y, z, u) & = (x \wedge y) \vee (x \wedge u). \end{array}$$

Step 6: Back to termination conditions. In the final step of the analysis, the Boolean termination conditions are lifted to termination conditions with the following interpretation, where the C's are CLP(H) constraints:

- each goal '`?-` C, `append(A,B,C).`' left-terminates if `A` or `C` are ground in C;
- each goal '`?-` c, `nrev(X,Y).`' left-terminates if `X` is ground in c;
- each goal '`?-` C, `app3(X,Y,Z,U).`' left-terminates if either `X` and `Y` are ground in C or `X` and `U` are ground in C.

4. Typed Termination Analysis

In this section we extend the untyped termination analysis approach recounted in the previous section to compute termination conditions of typed logic programs.

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.

4.1. The Type Separated Program

The type separated program $\mathsf{type}(P)$ arising from a typed 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.

4.1.1. *Translating 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 to map calls correctly.

4.1.2. *Translating 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 $q(Z.e, Z.l(n(e)), Z.n(e))$.

4.1.3. *Translating 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.

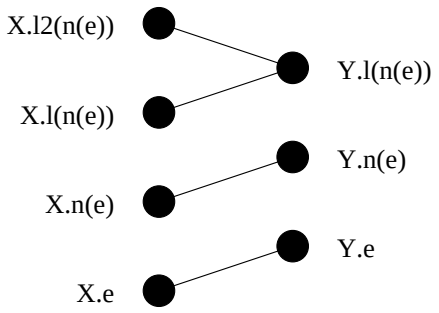


Figure 2. The bipartite graph corresponding to the nodes in $N(\langle X.l2(n(e)) \cap Y.l(n(e))\rangle)$.

EXAMPLE 4. *Consider* $X = Y$ *with* $\mathsf{type}(X) = l2(n(e))$ *and* $\mathsf{type}(Y) = l(n(e))$. *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 bipartite graph is shown in Figure 2. The connected components are

$$\{X.l(n(e)), X.l2(n(e)), Y.l(n(e))\},\ \{X.n(e), Y.n(e)\},\ \{X.e, Y.e\}.$$

The resulting equations are

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

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 $\Delta = \{1 \to f(v_1.\mathsf{type}(v_1), \ldots, v_n.\mathsf{type}(v_n))\} \cup \cup_{i=1}^{n} \Delta(v_i.\mathsf{type}(v_i))$. The remainder is as for the equation $v_1 = v_2$ above, where $\tau_1 = v_0.\mathsf{type}(v_0)$ and $\tau_2 = 1$. 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* $1 \to [D.T|E.l(T)]$, $E.l(T) \to []$, *and* $E.l(T) \to [E.T|E.l(T)]$. *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*

$$\{A.l(T), 1, E.l(T)\} \quad \text{and} \quad \{A.T, D.T, E.T\}.$$

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* $A.l(T) = 1$, $A.T = 0$.

4.1.4. *Translating 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 that 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* $\mathsf{type}(X) = l2(n(e))$ *and* $\mathsf{type}(p/1) = p(l(n(e)))$, *then we create new variable* Y *of type* $l(n(e))$ *and build the equations resulting from* $X = Y$ *as defined in Example 4 above together with the call* $p(Y.e, Y.l(n(e)), Y.n(e))$.

4.1.5. *Translating 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. Note the type must be an instance, but there are many grammars corresponding to the same Hindley/Milner type. 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 a type substitution σ such that $\mathsf{type}(v_i) = \sigma(\tau_i), 1 \leq i \leq n$. Let $\nu_j \in \mathcal{V}_{type}, 1 \leq j \leq m$ 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 $\sigma' = \{\nu_j \mapsto S_j \mid 1 \leq j \leq m\}$. 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 $1 \leq j \leq m$ 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 that 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* $p(Y)$ *where* $\mathsf{type}(Y) = l(n(e))$ *and* $\mathsf{type}(p/1) = p(l(T))$. *Then we have* $\sigma = \{T \mapsto n(e)\}$. *We create a type* $copy_1.n(e)$. *Let* X *be a new variable where* $\mathsf{type}(X) = l(copy_1.n(e))$. *We then create the equations resulting from* $X = Y$ *(which are analogous to those from Example 4). The unsubstituted call is* $p(X.l(T), X.T)$. *We create a copy for each* $\tau' \in N(copy_1.n(e))$. *The resulting translation is:*

$$X.l(copy_1.n(e)) + X.copy_1.l(n(e)) = Y.l(n(e)),$$
$$X.copy_1.n(e) = Y.n(e), X.copy_1.e = Y.e,$$
$$p(X.l(copy_1.n(e)), X.copy_1.l(n(e))),$$
$$p(X.l(copy_1.n(e)), X.copy_1.n(e)),$$
$$p(X.l(copy_1.n(e)), X.copy_1.e),$$

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

EXAMPLE 8. *Consider a call* $p(X)$ *where* $\mathsf{type}(X) = pair(e, l(e))$ *and* $\mathsf{type}(p/1) = p(pair(U, V))$. *Then* $\sigma = \{U \mapsto e, V \mapsto l(e)\}$. *We create new copies of the types* **`erk`** *and* **`list(erk)`**, *and build* $\sigma' = \{U \mapsto copy_1.e, V \mapsto copy_2.l(e)\}$. *The new variable* X' *has type* $pair(copy_1.e, copy_2.l(e))$ *and the equation* $X = X'$ *builds*

$$X.pair(e, l(e)) = X'.pair(copy_1.e, copy_2, l(e)),$$
$$X.l(e) = X'.copy_2.l(e),$$
$$X.e = X'.copy_1.e \oplus X'.copy_2.e.$$

The original call is $p(X'.pair(U, V), X'.U, X'.V)$, *and we obtain two copies,*

$$p(X'.pair(copy_1.e, copy_2, l(e)), X'.copy_1.e, X'.copy_2.e),$$
$$p(X'.pair(copy_1.e, copy_2, l(e)), X'.copy_1.e, X'.copy_2.l(e)).$$

4.1.6. *Translating Programs*

The type separated program simply translates each rule in P into the corresponding rule in $\mathsf{type}(P)$, by translating each literal appropriately.

EXAMPLE 9. *The translation of the rule*

$$q(Z) \leftarrow Z = n(X),\ p(X)$$

where $\mathsf{type}(Z) = n(e)$, $\mathsf{type}(X) = l(n(e))$ *and* $\mathsf{type}(p/1) = l(T)$ *is shown below.*

$$
\begin{array}{l}
q(Z.e, Z.l(n(e)), Z.n(e)) \leftarrow \\
\qquad Z.n(e) = 1 + Z.n(e),\ Z.l(n(e)) = X.l(n(e)),\ Z.e = X.e, \\
\qquad X.l(n(e) = Y.l(copy_1.n(e)) + Y.copy_1.l(n(e)), \\
\qquad X.n(e) = Y.copy_1.n(e),\ X.e = Y.copy_1.e, \\
\qquad p(Y.l(copy_1.n(e)),\ Y.copy_1.l(n(e))), \\
\qquad p(Y.l(copy_1.n(e)), Y.copy_1.n(e)), \\
\qquad p(Y.l(copy_1.n(e)), Y.copy_1.e).
\end{array}
$$

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.

4.2. Typed Analysis

The theoretical results supporting the approach described in this paper are proven in [27]. They show 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 [23].

THEOREM 1. *([27] Theorems 6.15, 7.1) Analysis using* Size *of the type separated program* $\mathsf{type}(P)$ *is correct with respect to the type correct program* P.

Namely, 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. *([27] Theorems 6.15, 7.2) Rigidity analysis using* Pos *of the type separated program* $\mathsf{type}(P)$ *is correct with respect to the type correct program* P.

Namely, 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.

EXAMPLE 10. *The analysis results for the program given in the introduction are*

$$g(W.list(erk), W.erk) \leftarrow_{size} 1 \leq W.list(erk) \leq 19$$
$$g(W.list(erk), W.erk) \leftarrow_{rigid} W.list(erk)$$
$$append(A.list(T), A.T, B.list(T).B.T, C.list(T), C.T) \leftarrow_{size}$$
$$A.list(T) + B.list(T) = 1 + C.list(T) \wedge A.T + B.T = C.T$$
$$append(A.list(T), A.T, B.list(T).B.T, C.list(T), C.T) \leftarrow_{rigid}$$
$$A.list(T) \wedge (B.list(T) \leftrightarrow C.list(T)) \wedge ((A.T \wedge B.T) \leftrightarrow C.T)$$

4.3. Accuracy of Typed Analysis

The typed analysis is generally much more accurate than the untyped analysis. It can be shown that for regular typed programs the typed analysis is uniformly more accurate than the untyped analysis.

THEOREM 3. *([27] Theorems 5.6, 5.18)*
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. *([27] Theorems 5.9, 5.18)*
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 term* 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 11. *Consider the following simple program with two possible type declarations for* `p`*, the first polymorphic and the second monomorphic.*

```
:- 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 the polymorphically typed predicate `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 it. 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 5 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 [27] 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. Our experiments (Section 5) confirm this claim.

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 lemma and the theorem that show that the level mapping never depends on arguments which have parameter types.

LEMMA 5. *Assume an element* $post_p^{\mathcal{N}}(\overline{x_{p1}}_{\mathsf{type}}, \ldots, \overline{x_{pn}}_{\mathsf{type}}) \iff C$ *in the (nonempty) CLP(*$\mathbb{N}$*) semantics of* $\mathsf{type}(P)$*. Then* C *admints a solution assigning zero to all variables corresponding to type parameters i.e., variables of the form* $v.\eta$*, where* $v \in \mathcal{V}_{tree}$ *and* $\eta \in \mathcal{V}_{type}$*.*

Proof. (Sketch) By construction variables based on a parameter η can appear in $\mathsf{type}(P)$ either in constraints of the form $x.\eta = 0$ or $x_1.\eta + \cdots + x_n.\eta = y_1.\eta + \cdots + y_m.\eta$. Such constraints clearly admit $\widetilde{0}$ as a solution. Similarly, any conjunction, convex hull (abstract disjunction), and widening of such constraints also admit $\widetilde{0}$ as a solution.

THEOREM 6.
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) Suppose $\widetilde{a}$ is a level mapping for some binary clause

$$p(x_1, \ldots, x_k) \ \texttt{:-} \ C, \ p(y_1, \ldots, y_k)$$

Then $\Sigma_{i=1}^{k}(a_i \cdot \theta(x_i)) > \Sigma_{i=1}^{k}(a_i \cdot \theta(y_i))$ for every solution θ of C. By construction C is a conjunction of constraints found in the semantics of CLP($\mathbb{N}$) program obtained from $\mathsf{type}(P)$. By Lemma 5 if θ is a solution of C then $\theta' = \{\, v.\eta \mapsto 0 \mid \eta \in \mathcal{V}_{type} \,\} \cup \{\, v.\tau \mapsto \theta(c.\tau) \mid \tau \notin \mathcal{V}_{tree} \,\}$ is also a solution of C. Thus, given a level mapping $\widetilde{a}$ we can construct another correct level mapping $\widetilde{a'}$, where $a'_i = a_i$ except for arguments i of parameter type where $a'_i = 0$.

Bruynooghe *et al* [5] prove a slightly 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. (i.e., it will not weaken the termination conditions). 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.

4.5. Termination Conditions

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

THEOREM 7. *The typed termination conditions that are computed from* $\mathsf{type}(P)$ *are correct.*

Namely, let pre_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* pre_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.

5. Experiments

5.1. Extending the untyped termination system

For Hindley/Milner typed programs, 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 [31]. The typed termination results are simply read from the results of the untyped analysis of $\mathsf{type}(P)$.

For the regular typed termination analysis, we first removed any type definition and declaration, inferred the regular types bottom-up with the tool provided by John Gallagher [18], and rewrote the inferred types into the common syntax we used for the Hindley/Milner typed termination analysis. Finally, the same steps as described above were performed.

5.2. Experiments

We ran the cTI termination analyzer, written in SICStus Prolog 3.11.0 using the PPL library [3], a timeout of 4 seconds for each strongly connected component, allowing 4 iterations before widening on an Intel 686, 2.4 GHz, 512 Mb, Linux 2.4.

In the first experiment we considered pure Prolog versions of the 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 analyses run out of memory). We also selected six programs from the termination literature, including our first example of Section 1, that were constructed to be more difficult for termination analyses.

Table I reports the results for the programs in the book [1] while Table II gives the results for six programs from the literature.

Each table reports the results for the typed analysis using the Hindley/Milner types (*type-size*), the typed analysis using inferred regular types (*reg-size*) and untyped analysis using term size (*term-size*) and list size (*list-size*) metrics. For each procedure we show the termination condition for each 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. The termination condition $||X||_{lls}$ is true when X denotes a finite list of elements, each of which is also a finite list. 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 relative precision of the results of these four analyses is shown in Table III, where we use darker colors to represent which analysis scored better results compared to the others. While the colors in Table III allows one to determine which analysis gives better results than another, they can hide when an analysis is actually too imprecise. For instance, when an analysis yields *false* very often as a termination condition for a predicate. Hence in Table III we crossed those boxes corresponding to *false* in Tables I and II. Hence you can see that the analysis using the list size norm yields *false* very often, much more than the other analyses.

Sometimes the results of the analyses do not form a total order. Hence for these programs there is some freedom in the choice of the colors in Table III. For instance, we considered that the termination condition for `flatten_length` by using list size in Table II is less precise than that computed for the same predicate through regular type inference, although there is no logical implication between them. This problem arises for `flatten_length` and `flatten` only.

In summary the results show that the Hindley/Milner typed analysis is always more precise than the untyped analyses, and the regular type analysis is always more precise than the term size untyped analysis.

Table I. Some programs from Apt's book.

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

Table II. Some logic programs from the literature.

Ref.	Query	type-size	term-size	list-size	reg-size
[29]	`flatten_ length(X,Y,Z)`	$\|\|X\|\|_{lls} \vee$ $(\|\|X\|\|_{ls} \wedge \|\|Y\|\|_{ls}) \vee$ $(\|\|Y\|\|_{ls} \wedge \|\|Z\|\|_{ts}) \vee$	$\|\|X\|\|_{ts} \vee$ $(\|\|Y\|\|_{ts} \wedge$ $\|\|Z\|\|_{ts})$	$\|\|X\|\|_{ls} \wedge$ $\|\|Y\|\|_{ls}$	$\|\|X\|\|_{lls} \vee$ $(\|\|X\|\|_{ls} \wedge \|\|Y\|\|_{ts}) \vee$ $(\|\|Y\|\|_{ts} \wedge \|\|Z\|\|_{ts}) \vee$
[39]	`flatten(X,Y)`	$\|\|X\|\|_{lls} \wedge$ $(\|\|X\|\|_{ls} \vee \|\|Y\|\|_{ls})$	$\|\|X\|\|_{ts}$	$\|\|X\|\|_{ls} \wedge$ $\|\|Y\|\|_{ls}$	$\|\|X\|\|_{ts}$
[22]	`p(X)`	$\|\|X\|\|_{ts}$	*false*	*false*	*false*
[22]	`factor(X,Y)`	$\|\|X\|\|_{ts}$	*false*	*false*	*false*
[22]	`t(X)`	$\|\|X\|\|_{ts}$	*false*	*false*	*false*
Sect. 1	`g(X)`	*true*	*false*	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$

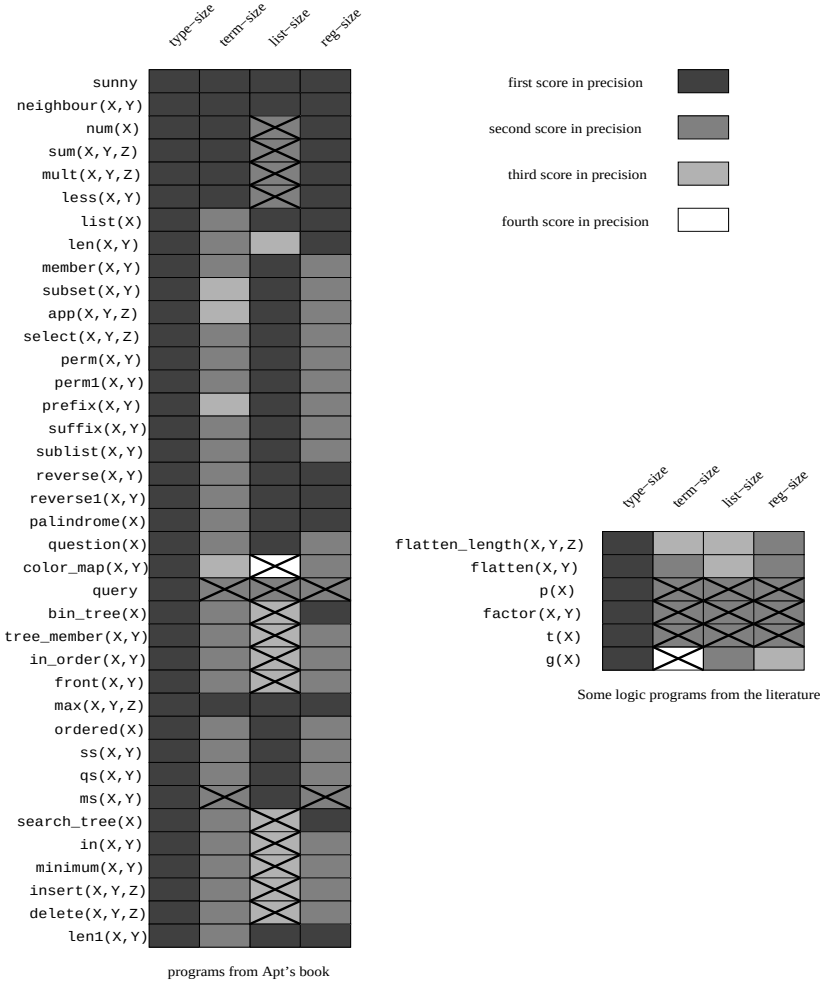
Our experiments with the regular type inference show that in some cases we significantly improve the termination conditions for an *untyped* program without needing any user-added type information. Moreover, the analysis based on regular type inference is able to synthesize non-trivial term norms that induce termination for the programs. For instance norms for lists, binary trees and the unusual $||X||_{24}$ norms are *automatically* synthesized for the programs in Tables I and II. This shows that we can, in some sense, get the benefits of typed analysis *for free*, that is without having to (manually) add types to our programs.

The total analysis time for all programs in Table I for the two untyped analyses took 0.7 seconds each. The Hindley/Milner typed analysis took 8.6 seconds. The regular type analysis (including type inference) took 3.6 seconds. The total analysis times for all programs in Table II was 0.3 seconds for each of the untyped analyses, 2.9 seconds for the Hindley/Milner typed analysis, and 1.0 second for the regular type analysis with type inference. Clearly there is a significant overhead for typed analyses. Type separation is a non-trivial transformation which can take up to one third of the total analysis time. Moreover, it increases the arity for logic procedures, and the core termination analysis is more expensive. This is true in particular for the Hindley/Milner typed analysis, while for the regular typed analysis with bottom-up type inference, the inferred types are often weaker, leading to a less precise but faster analysis.

Our second experiment is based on some typed Prolog programs, in this case from the HAL [13] benchmark suite.[5] The results of the analysis are reported in Table IV. The metric $||X||_d$ requires that X

[5] The reader might wonder why we did not investigate Mercury programs, which are typed logic programs. Mode information for Mercury programs makes the calculation of universal termination conditions uninteresting, since we are only interested in termination for given modes.

Table III. A comparison of the precision of the analyses in Tables I and II.



is a finite formula built from integers, the variable x, the operators +, -, *, /, ^ and the function symbols log and exp. The metric $||X||_{nat}$ requires that X is of form $s^n(0)$ where n is a non-negative integer. Concerning the analysis of `tak` and `warplan`, time-out events happened inside cTI, which gave the *false* termination condition. For `deriv`, a time-out event happened while inferring regular types, which explains the *false* termination condition.

For `deriv` and `hanoi_difflist` the typed termination condition is in effect no more accurate than the untyped, since the $||\cdot||_d$ and $||\cdot||_{nat}$ norms are just term size restricted to the type. For `boyer`, `qsortapp` and `qsort_difflist` the typed termination condition allows us to ig-

nore the size of the elements, making them somewhat stronger. For `hanoi_difflist`, `mmatrix` and `serialize` the typed analysis provides substantially stronger conditions. The list size analysis is only useful in the `qsort` cases, where the computation exclusively uses lists.

Table IV. Some HAL programs.

Program	Query	type-size	term-size	list-size	reg-size
`boyer`	`rewrite(X,Y)`	*false*	*false*	*false*	*false*
	`tautology(X,Y,Z)`	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ls} \wedge \|\|Z\|\|_{ls}$	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts} \wedge \|\|Z\|\|_{ts}$	*false*	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts} \wedge \|\|Z\|\|_{ts}$
`deriv`	`deriv(X,Y,Z)`	$\|\|X\|\|_{d} \vee \|\|Z\|\|_{d}$	$\|\|X\|\|_{ts} \vee \|\|Z\|\|_{ts}$	*false*	*false*
`hanoiapp`	`shanoi(X,Y,Z,T,U)`	$\|\|X\|\|_{nat}$	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts} \wedge \|\|Z\|\|_{ts} \wedge \|\|T\|\|_{ts}$	*false*	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts} \wedge \|\|Z\|\|_{ts} \wedge \|\|T\|\|_{ts}$
`hanoi_difflist`	`shanoi(X,Y,Z,T,U)`	$\|\|X\|\|_{nat}$	$\|\|X\|\|_{ts}$	*false*	$\|\|X\|\|_{nat}$
`mmatrix`	`mmultiply(X,Y,Z)`	$(\|\|Y\|\|_{lls} \wedge \|\|Z\|\|_{lls}) \vee (\|\|X\|\|_{lls} \wedge \|\|Z\|\|_{lls}) \vee (\|\|X\|\|_{ls} \wedge \|\|Y\|\|_{lls}) \vee (\|\|X\|\|_{lls} \wedge \|\|Y\|\|_{ls})$	$(\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts}) \vee (\|\|X\|\|_{ts} \wedge \|\|Z\|\|_{ts}) \vee (\|\|Y\|\|_{ts} \wedge \|\|Z\|\|_{ts})$	*false*	$(\|\|X\|\|_{ls} \wedge \|\|Y\|\|_{ts}) \vee (\|\|X\|\|_{ts} \wedge \|\|Z\|\|_{lls}) \vee (\|\|Y\|\|_{ts} \wedge \|\|Z\|\|_{ls}) \vee$
`qsortapp`	`qsort(X,Y)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$
`qsort_difflist`	`qsort(X,Y)`	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$	$\|\|X\|\|_{ls}$	$\|\|X\|\|_{ts}$
`serialize`	`serialize0(X,Y)`	$\|\|X\|\|_{ls} \vee \|\|Y\|\|_{ls}$	$\|\|X\|\|_{ts} \wedge \|\|Y\|\|_{ts}$	*false*	$\|\|X\|\|_{ls} \vee \|\|Y\|\|_{ls}$
`tak`	`tak(X,Y,Z,T)`	*false*	*false*	*false*	*false*
`warplan`	`plans(X,Y,Z)`	*false*	*false*	*false*	*false*

Table V gives the analysis times for each of the programs. Interestingly, for these more realistic programs the overhead of the typed termination analysis appears to be not as great as for the smaller programs in the first experiment.

Table V. Time in seconds for analysing some HAL programs.

Program	type-size	term-size	list-size	reg-size
`boyer`	1.40	0.77	0.51	3.04
`deriv`	0.49	0.24	0.18	time-out
`hanoiapp`	0.67	0.16	0.15	0.69
`hanoi_difflist`	0.86	0.16	0.16	1.15
`mmatrix`	1.16	0.18	0.19	1.14
`qsortapp`	0.54	0.14	0.17	0.85
`qsort_difflist`	0.48	0.14	0.18	0.80
`serialize`	1.65	0.24	0.22	1.60
`tak`	time-out	time-out	0.14	time-out
`warplan`	time-out	time-out	0.62	time-out

6. Conclusion

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. Moreover, the use of regular type inference leads to a fully-automated push-button termination analyzer, whose precision is comparable to that of a termination analyser that uses human-provided type information.

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.

There should be little difficulty in reusing our type separation translation in other termination analysers for untyped programs. To this end, we have made the type separation preprocessor, together with the programs analyzed for benchmarking, are available under the GNU General Public License from `http://www.univ-reunion.fr/~gcc`. The cTI analyzer is available under the GNU General Public License from the cTI's web site: `http://www.cs.unipr.it/cTI` via anonymous CVS.

Acknowledgments

We would like to acknowledge many fruitful discussions about typed analysis and termination with Michael Codish and Maurice Bruynooghe that have helped improve this work.

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. In M. V. Hermenegildo and G. Puebla, editors, *SAS*, volume 2477 of *LNCS*, pages 213–229, Madrid, Spain, 2002. Springer-Verlag, Berlin.
4. A. Bossi, N. Cocco, and M.Fabris. Typed norms. In Bernd Krieg-Brückner, editor, *ESOP'92*, volume 582 of *LNCS*, pages 73–92. Springer, 1992.
5. Maurice Bruynooghe, Michael Codish, Samir Genaim, and Wim Vanhoof. Reuse of results in termination analysis of typed logic programs. In *Static Analysis, 9th International Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 477–492. Springer-Verlag, 2002.
6. Maurice Bruynooghe, Michael Codish, Samir Genaim, and Wim Vanhoof. A note on the reuse of results in termination analysis of typed logic programs. 2003.

7. Maurice Bruynooghe, Wim Vanhoof, and Michael Codish. Pos(T) : Analyzing dependencies in typed logic programs. In *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Revised Papers*, volume 2244 of *LNCS*, pages 406–420. Springer-Verlag, 2001.
8. M. Codish and C. Taboch. A semantics basis for termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
9. S. Colin, F. Mesnard, and A. Rauzy. Constraint logic programming and mu-calculus. *ERCIM/COMPULOG Workshop on Constraints*, 1997.
10. M-M. Corsini and A. Rauzy. Toupie: the μ-calculus over finite domains as a constraint language. *Journal of Automated Reasonning*, 19:143–171, 1997.
11. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.
12. http://www.complang.tuwien.ac.at/cti/.
13. M. García de la Banda, B. Demoen, K. Marriott, and P.J. Stuckey. To the gates of HAL: a HAL tutorial. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in LNCS, pages 47–66. Springer-Verlag, 2002.
14. S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms: a missing link in automation termination analysis. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming*, pages 420–436. MIT Press, 1993.
15. S. Decorte, D. De Schreye, and M. Fabris. Exploiting the power of typed norms in automatic inference of interargument relations. Technical Report 246, Department of Computer Science, K.U.Leuven, 1997.
16. F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6):751–777, 2001.
17. J. Fischer. Termination analysis for Mercury using convex constraints. Honours report, Department of Computer Science and Software Engineering, The University of Melbourne, 2002. http://www.cs.mu.oz.au/research/mercury/information/papers/juliensf-hons.ps.gz.
18. J. Gallagher and A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *ICLP*, pages 599–613, Santa Margherita Ligure, Italy, 1994. MIT Press.
19. J. Gallagher and G. Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *Proceedings of the 4th Int. Symp. on Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 243–261. Springer-Verlag, Berlin, 2002.
20. M. García de la Banda, B. Demoen, K. Marriott, and P.J. Stuckey. To the Gates of HAL: A HAL tutorial. *Lecture Notes in Computer Science*, 2441:47–66, 2002.
21. S. Genaim and M. Codish. Inferring termination conditions for logic programs using backwards analysis. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the Eighth International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 681–690. Springer-Verlag, December 2001.
22. S. Genaim, M. Codish, J. Gallagher, and V. Lagoon. Combining norms to prove termination. In Agostino Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation, Third International Workshop, VMCAI 2002, Venice,*

Italy, January 21-22, 2002, volume 2294 of *LNCS*, pages 126–138. Springer, 2002.
23. N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. PhD thesis, Université scientifique et médicale de Grenoble, Grenoble, France, March 1979.
24. P. Hill and J. Lloyd. *The Gödel Language*. The MIT Press, Cambridge, Mass., 1994.
25. 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.
26. A. King and L. Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming*, 2(4–5):517–547, 2002.
27. V. Lagoon. *Analysis of Typed Logic Programs*. PhD thesis, Department of Computer Science, University of Melbourne, Australia, 2003. (forthcoming). Draft available at `http://www.cs.mu.oz.au/~pjs/papers/vltd.ps`.
28. V. Lagoon, F. Mesnard, and P.J. Stuckey. Termination analysis with types is more accurate. In Catuscia Palamidessi, editor, *Proceedings of the 19th International Conference on Logic Programming*, LNCS, page to appear. Springer-Verlag, 2003.
29. J. Martin, A. King, and P. Soper. Typed norms for typed logic programs. In J. Gallagher, editor, *LNCS*, volume 1207, pages 224–238. Springer-Verlag, 1996. Proc. of LOPSTR'96.
30. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs by means of approximations. In M. J. Maher, editor, *Proc. of the 1996 Joint Intl. Conf. and Symp. on Logic Programming*, pages 7–21. MIT Press, 1996.
31. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In P. Cousot, editor, *SAS*, volume 2126 of *LNCS*, pages 93–110. Springer-Verlag, Berlin, 2001.
32. F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. *ACM ToCL*, 4(2), 2003.
33. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
34. A. Mycroft and R. A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
35. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
36. K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Proc. of the 1991 Intl. Symp. on Principles of Database Systems*, pages 216–226. ACM, 1991.
37. 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.
38. C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. van Hentenrick, editor, *SAS*, volume 1302 of *LNCS*. Springer-Verlag, 1997.
39. W. Vanhoof and M. Bruynooghe. When size does matter - Termination analysis for typed logic programs. In A. Pettorossi, editor, *Logic-based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001, Selected Papers*, volume 2372 of *LNCS*, pages 129–147. Springer-Verlag, 2002.