

**Integrating types in abstract
interpretation based automatic
termination analysis of logic programs**

Stefaan Decorte, Danny de Schreye, Massimo Fabris

Report CW 222, January 1996



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Integrating types in abstract interpretation based automatic termination analysis of logic programs

Stefaan Decorte, Danny de Schreye, Massimo Fabris

Report CW222, January 1996

Department of Computer Science, K.U.Leuven

Abstract

Types are receiving increasing attention in logic programming languages. Several new languages, such as Gdel and Mercury, are typed languages. Also, the more recent Prolog systems either support and exploit optional type declarations, or perform type inference at compile time, with the purpose of verification and code-optimisation. Research in program verification, especially on termination analysis, has followed this development. In recent termination analysis works, the added power of including type information in the analysis has been demonstrated.

In this paper, we further clarify the role of types in termination analysis, with an emphasis on automation. Our main contribution is on the automatic generation of appropriate well-founded mappings (based on norms) for proving termination of a given program and (class of) queries. We propose two different approaches, which both exploit type information. A first approach infers one semi-linear norm from the given types. The second approach is more refined. It infers a set of generalised norms, referred to as typed norms. The generalisation involved in typed norms, is that the way in which a term is measured under such norms is parametrised with respect to the specific type to which the term is considered to belong.

The first approach has the advantage that it is compatible with several existing techniques for termination analysis and therefore, can be combined with them without difficulty. The second approach is strictly more refined and achieves higher precision, but it requires that part of the termination analysis – specifically: inference of inter-argument or size relations – needs to be redesigned. We develop an appropriately extended technique for inferring such relations, based on bottom-up abstract interpretation.

All parts of the proposed analysis are fully automatable and a prototype system has been implemented. A report on extensive testing performed with this system is included.

Integrating Types in Abstract Interpretation Based Automatic Termination Analysis of Logic Programs

Stefaan Decorte* Danny De Schreye* Massimo Fabris[†]

Abstract

Types are receiving increasing attention in logic programming languages. Several new languages, such as Gödel and Mercury, are typed languages. Also, the more recent Prolog systems either support and exploit optional type declarations, or perform type inference at compile time, with the purpose of verification and code-optimisation. Research in program verification, especially on termination analysis, has followed this development. In recent termination analysis works, the added power of including type information in the analysis has been demonstrated.

In this paper, we further clarify the role of types in termination analysis, with an emphasis on *automation*. Our main contribution is on the automatic generation of appropriate well-founded mappings (based on *norms*) for proving termination of a given program and (class of) queries. We propose two different approaches, which both exploit type information. A first approach infers *one* semi-linear norm from the given types. The second approach is more refined. It infers a set of generalised norms, referred to as *typed norms*. The generalisation involved in typed norms, is that the way in which a term is measured under such norms is parametrised with respect to the specific type to which the term is considered to belong.

The first approach has the advantage that it is compatible with several existing techniques for termination analysis and therefore, can be combined with them without difficulty. The second approach is strictly more refined and achieves higher precision, but it requires that part of the termination analysis – specifically: inference of *interargument* or *size relations* – needs to be redesigned. We develop an appropriately extended technique for inferring such relations, based on bottom-up abstract interpretation.

All parts of the proposed analysis are fully automatable and a prototype system has been implemented. A report on extensive testing performed with this system is included.

Key words: Logic Programming, Termination Analysis, Abstract Interpretation

*Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium.
e-mail : {stefaan,dannyd}@cs.kuleuven.ac.be

[†]ICON, Lungadige Rubele 34, 37121 VERONA, Italy. email: fabris@icon.it

1 Introduction

In recent years, there has been a growing interest in the study of termination properties of logic programs (see [16] for a survey). Part of the research has been devoted to the development of techniques expressing sufficient and necessary conditions ([1], [2], [5], [6], [7], [8], [19], [43]) while other work on this topic has addressed the problem of deriving termination proofs automatically. Even within this latter more narrow context, a variety of different proposals have been made, e.g. [42], [38], [40], [34], [36], [47], [49], [48], [7], [37], [11], [39], [35], [46], [19], [9], [13]. Many of these works, e.g. [42], [34], [36], [47], [7], [46], [19], [9] either implicitly or explicitly rely on the notion of a *norm* to formulate and verify their termination conditions. Here, a norm is a function which maps each term (modulo variable renaming) to a corresponding natural number. The number is used as an estimate of the "size" of the term.

Many examples of norms can be found in the literature. When dealing with lists, it is often appropriate to use *list-length*, which computes the depth of the rightmost branch in the tree representation of a list and returns 0 for any other term. A norm which is used in a more general context is *term-size*. It counts the number of functors (with strictly positive arity) occurring in a term. Yet another norm is *term-depth*, which gives the maximum depth of the tree representation of a term.

Once a specific norm has been fixed, the techniques mentioned above proceed by selecting for each predicate in the program one or more argument positions. Typically, these positions correspond to the input positions for the predicate and are determined by some form of mode (and possibly type) analysis or by annotations. Using these argument positions, the norm can be extended as a function from atoms to natural numbers, by, for each atom, computing some combination (e.g. the sum or a positive linear combination) of the norms of the terms occurring at these argument positions fixed for the predicate. In what follows, we refer to such functions as *level mappings*.

Finally, the termination analysis techniques formulate and verify termination conditions expressed in terms of the level mapping. Roughly stated, given a top-level query Q of interest, the conditions will ensure that for each two consecutive calls to a same predicate in any derivation for Q , the level mapping for these calls decreases. As a result, by well-foundedness of the natural numbers, the derivations must be finite. Of course, no technique actually computes the derivations in order to establish such a decrease. Instead termination conditions are formulated in terms of a finite number of syntactic structures, such as for instance, the clauses of the given program.

To round up this short introduction on termination analysis, we must mention a more technical problem, which usually forms the heart of every automatic technique: the inference of interargument relations. Consider a clause of the type $p(X) : -q(X, Y), p(Y)$. Due to the presence of the existentially quantified variable Y , the only way to prove a decrease of the level mapping for two consecutive calls to $p/1$ is to prove that for any atom $q(s, t)$ in the success set of $q/2$, the norm of t is smaller than that of s . Then, assuming that the left-to-right computation rule of Prolog is used, the desired decrease for $p/1$ follows. In general, automatic termination analysis techniques spend considerable effort to derive such relations which hold between the norms of the arguments of atoms in the success set of the program. These relations are referred to as *interargument relations* or *size relations*. We refer to [16] for an overview on the topic.

From this brief discussion, it should be intuitively clear that the power of these au-

automatic termination analysis techniques strongly depends on the choice of the norm. As an example, consider the simple program:

Example 1.1

$$\begin{aligned} p(f(X, f(Y, Z)), R) &: - p(f(f(X, Y), Z), R). \\ p(f(X, Y), f(X, Y)) &: - \text{integer}(Y). \end{aligned}$$

which applies associativity of a functor $f/2$ to group f -occurrence to the left. For instance, a query $?-p(f(1, f(2, f(3, 4))), R)$ succeeds with $R = f(f(f(1, 2), 3), 4)$. If we use term-size and a level mapping which only considers the first argument position of $p/2$, termination cannot be proved for any query on the basis of the mentioned approaches. With term-size, the level mapping remains unchanged between the head and the body of the recursive rule. However, if we use a norm f -length, generalising list-length, which computes the depth of the right most branch in the tree representation of an f -term and gives 0 for any other term, then termination for queries with a ground first argument (and even more general ones) can be proved. ■

Similarly, on the level of inference of interargument relations, the selection of an appropriate norm can have a major impact. As an example, consider the program:

Example 1.2

$$\begin{aligned} q([], []). \\ q([0|T], [0|T1]) &: - q(T, T1). \\ q([N|T], [s(X)|T1]) &: - M \text{ is } N - 1, q([M|T], [X|T1]). \end{aligned}$$

which converts a list of integers in its first argument to an equivalent list of integers with successor representation in its second argument. Using term-size, no interargument relation can be derived. However, using list-length, $\{(X, X) | X \in \mathbb{N}\}$ is a correct interargument relation for $q/2$. ■

In spite of this crucial role that norms fulfil in the analysis, none of the existing techniques provide support for inferring the appropriate one. Some techniques, such as [42] are tailored to the use of one specific norm (for [42], list-length). Others, such as [44] provide the user with the possibility of specifying a desired norm, tuned to the program and query at hand.

In this paper we present two techniques for generating suitable norms automatically. Both rely on type information which is computed in a pre-processing phase through abstract interpretation. We make use of rigid types, as introduced in [30] and apply their abstract interpretation technique to infer them. The rigid call types for the arguments in the predicates of the program - inferred for a set of queries which is also specified through rigid types - provide us with very refined specification of the data that can be expected at any point in any derivation. Using the notion of rigidity, which has become a standard precondition for state-of-the-art analysis (see section 2, [7], [36] and [44]), we can induce in a very natural way the appropriate norms from the types.

We present two alternative techniques. The first and easiest one will derive one norm from the set of all call types. Moreover, that norm will be semi-linear and can thus

be used with no difficulty in other existing theoretical and automatical frameworks for termination analysis.

The other method will propose a set of interdependent norms, called *typed norms*. Such norms concentrate on measuring terms belonging to their type only. A level mapping is then defined that measures each argument using the specialised norm for that argument's call type. As can be expected when using different norms in one termination proof, this approach causes some consistency problems as soon as sideways information passing is involved.

In this paper we define a class of *well-typed* programs and we show that for such programs a set of conditions on the norms can be computed which prevent the above inconsistencies. In short, these conditions, if they can be satisfied, make sure that whenever a term could be measured under two different norms, then both sizes are the same. This property is called *matching*. We also discuss a method to construct such well-typed programs out of logic programs, and we show that this generation process is always possible.

As quoted above, in practical examples, interargument relations are crucial for being able to deliver a termination proof. We propose a system to derive such interargument relations. It is built around a generic abstract interpretation framework and it is basically an extension of the technique proposed in [18]. Up to minor changes, we used the same abstract domain of systems of linear equalities and also the operations on it are the same. However, where the derivation process in [18] is described as an instance of a top-down abstract interpretation framework, we decided to employ a bottom-up approach, which reduces the complexity of the description of the system. Our method extends [18] in the sense that we need to derive typed interargument relations, i.e. relations between the sizes of the arguments of a predicate, where each argument may be measured by a different norm. Again, the use of several different norms cause complications, specifically when one wants to apply the abstract domain operators of [18]. These complications can be prevented through our well-typedness property. Due to the introduction of types in the derivation process, we are able to generate very precise interargument relations. A drawback by basing the process on the framework of [18] is that all relations are in the form of linear equations. It is well-known that in some cases more expressive relations or inequalities are needed.

In the next section, we present some preliminaries on abstract interpretation, termination analysis, (semi-linear) norms and rigid types. In section 3, a method is proposed for deriving one semi-linear norm. This one norm is used to prove termination of the complete program. In the fourth section¹, *typed norms* are proposed. The problems which are resulting when using these norms are discussed and solved in section 5. In that section the well-typedness property for programs is introduced and we show how the set of conditions preventing inconsistencies can be computed from such a program. These conditions affect the definition of the typed norms introduced in section 4. In section 6, we extend an existing, automatic technique based on abstract interpretation for deriving typed interargument relations. In the next section, we illustrate the complete approach on three examples. We end with a discussion and conclusions.

¹A preliminary, short version of sections 3 and 4 appeared as [22].

2 Preliminaries

We first introduce some conventions and recall some basic terminology. The extended Herbrand Universe, U_P^E , and the extended Herbrand Base, B_P^E , associated to a program P , were introduced in [27]. They are defined as follows. Let $Term_P$ and $Atom_P$ denote the sets of respectively all terms and all atoms that can be constructed from the alphabet underlying to P . The variant relation, denoted \sim , defines an equivalence. U_P^E and B_P^E are respectively the quotient sets $Term_P/\sim$ and $Atom_P/\sim$. For any term t (or atom A), we denote its class in U_P^E (B_P^E) as $[t]$ ($[A]$). However, in order to reduce notational complexity, we drop the brackets when no real confusion is possible.

By \mathcal{L}_P , we denote the first order language associated to a program P . We use $Const_P$ and Fun_P to denote respectively the set of constants and the set of function symbols (without the constants). In this paper we extend the definition of Fun_P to also include all functor symbols which might occur in queries for P . The same holds for $Const_P$.

2.1 A bottom-up abstract interpretation framework

In this section, we take a look at a generic framework for abstract interpretation of logic programs. Formulating an actual abstract interpretation application as an instance of such a general framework is very interesting. The design of a new application becomes limited to the careful design of only a few components. Moreover, it provides one with a set of easy to manipulate conditions which guarantee several correctness results upon satisfaction. We only present a high level overview of the main aspects of such a framework. For more details, the interested reader is referred to [14]. In section 6 these ideas will be used to abstract predicates to systems of linear equations.

Abstract interpretation, originally developed by P. Cousot and R. Cousot ([15]), has become a very widespread technique in the area of static analysis of logic programs. Amongst the applications are e.g. the derivation of mode information at call and success time ([29]), deriving similar type information ([29]), freeness analysis ([25]), definiteness analysis([28]), combining freeness and definiteness analysis([26]) and derivation of inter-argument relations ([44]).

As the literature indicates, two frequently adopted approaches in the design of an application of abstract interpretation are a bottom-up and a top-down approach. In section 6, we have chosen a bottom-up approach because of its elegance in describing the components separate from the description of the procedure itself. In [19], a top-down approach has been described to solve a problem similar to the one addressed in this paper. In general, a top-down approach has a tendency of being fairly procedural and requires a profound knowledge of the underlying execution mechanism. We have chosen not to focus too deep into such technical details.

Abstract interpretation is a technique which has been introduced with the goal of extracting properties of programs without having to execute them. There is no strict restriction on the kind of properties which may be abstracted. These properties may describe aspects of the declarative semantics of the program or they can be related to the executional behaviour of the program.

The main idea behind abstract interpretation is that the behaviour of the program one is interested in can be defined as the least fixpoint of some function, which we will denote as F , on some domain of program properties. For example, if we aim at abstracting

the declarative semantics of the program, we could start from the concrete semantics and from the least fixpoint of the T_P operator, which is defined on the powerset of the Herbrand Base. Since an instantiation of the general framework will be developed later on for these semantics, their formal definition is recalled:

Definition 2.1 (T_P -semantics)

Let P be a definite logic program and $\text{ground}(P)$ the set of all ground instances of clauses in P . Then T_P is a function from the powerset of the Herbrand base to the powerset of the Herbrand base defined as follows:

$$T_P(S) = \{ A \mid A \leftarrow B_1, \dots, B_n \in \text{ground}(P) \text{ and } \forall i \in [1, n] : B_i \in S \}$$

where S stands for a subset of the Herbrand base. The T_P semantics gives the least Herbrand model of the program, which is the least fixpoint of the T_P function:

$$\text{lfp } T_P = \cup_{n \geq 0} T_P^n(\emptyset).$$

■

From a practical point of view, it is unfeasible to compute that fixpoint, as in general an infinite number of applications of the function are needed for obtaining it. The theory of abstract interpretation will instead define a second domain of program properties whose members form abstractions of subsets of members of the first domain. These domains are respectively referred to as the *concrete* and the *abstract domain*. The theory of abstract interpretation also requires that one defines a function abstracting elements of the concrete domain: the *abstraction function*. Its converse, associating concrete elements to abstract elements, is called a *concretisation function*. On the abstract domain a second function, F^α , the *abstract interpretation function*, is then defined. Its purpose is to act as an abstraction of the concrete interpretation function F on D . Before investigating how the least fixpoint of that new function can be assured to approximate in a reliable way the fixpoint of the concrete function F , we must define what constitutes a *safe* approximation. Intuitively, an abstract element is a safe approximation of a set of concrete elements if it abstracts *at least* all members of the set.

Definition 2.2 (safe approximation)

Let D be a domain which is partially ordered by a relation \sqsubseteq and let $d_1, d_2 \in D$. Then d_1 is a *safe approximation* of d_2 if $d_2 \sqsubseteq d_1$. ■

On the level of our concrete interpretation function F , for an abstract function F^α to safely approximate F , the following requirement must hold:

$$\forall d^\alpha : \gamma(F^\alpha(d^\alpha)) \supseteq F(\gamma(d^\alpha))$$

Then the ultimate goal of abstract interpretation is that the concretisation of the least fixpoint of the abstract function is a safe approximation of the least fixpoint of the concrete function. It is then hoped that it is easier to compute the abstract fixpoint. The price is off course precision.

From now on, we suppose that a concrete domain D , an abstract domain D^α , an abstraction function $\alpha : D \rightarrow D^\alpha$ and a concretisation function $\gamma : D^\alpha \rightarrow D$ have been

fixed. The framework described in [14] imposes a number of conditions on D, D^α, α (and γ). Their satisfaction guarantees that the least fixpoint of F^α is a safe approximation of the least fixpoint of F . We do not comment on these conditions but merely state them. More details are provided in [14]. The following properties are essential:

- There must exist an order relation on the elements of D . Let us denote this relation by \subseteq .
- Similarly, there must exist an order relation on D^α , which we will denote by \sqsubseteq .
- The abstraction function must be *total*: every element $d \in D$ must have an abstraction $d^\alpha \in D^\alpha$.
- Elements $d^\alpha \sqsupseteq \alpha(d)$ are safe approximations of $\alpha(d)$.
- Elements $\alpha(d_1)$ are safe approximations of $\alpha(d_2)$ with $d_2 \subseteq d_1$.

An algebraic structure which enforces all these conditions is the *Galois connection*. As our domains and abstraction function, which we will develop later on, will form a Galois connection, it is useful to include a definition here.

Definition 2.3 (Galois Connection)

Given (D, \subseteq) a domain D equipped with a partial order \subseteq and, (D^α, \sqsubseteq) a domain D^α equipped with a partial order \sqsubseteq , a Galois connection is a pair of functions α and γ satisfying:

- α is a total function from D to D^α
- γ is a total function from D^α to D
- $\forall d \in D : \forall d^\alpha \in D^\alpha : \alpha(d) \sqsubseteq d^\alpha$ iff $d \subseteq \gamma(d^\alpha)$

■

The existence of a Galois connection between an abstract and a concrete domain together with the fact that F^α safely approximates F imply a safe approximation of the **lfp** F by **lfp** F^α ([14]).

A final problem which is addressed here deals with practical concerns. For any automation of the procedure in mind, it would be very useful if it could be guaranteed that the computation of this fixpoint would involve only a finite number of steps. A simple solution, introduced in [14], is not to allow the abstract domain to contain infinite ascending chains of elements.

2.2 Termination analysis and interargument relations

In this section we briefly describe the main ideas of the termination analysis framework of [19]. Sticking to this particular approach is not a limitation in the sense that most of the results in the following sections can easily be integrated in other frameworks, such as e.g. [7]. To simplify the discussion, we recall the key notions of [19] in the context of directly recursive programs only and we specialise the results for one fixed selection rule: the left-to-right one. Full details on the framework in [19] can be found in [17].

Definition 2.4 (call set)

Let P be a definite program and S a set of atomic queries with one fixed predicate symbol. The *call set*, $Call(P, S)$, is the set of all atoms A such that a variant of A is a selected atom in some derivation for (P, Q) , for some $Q \in S$ and under the left-to-right selection rule. ■

The set of queries S can be an explicit enumeration of atoms but in practice it is often better specified as a call pattern using symbolic information. Mostly, mode or type abstraction mechanisms are used. We come back to this in section 2.4 when discussing one of these mechanisms: rigid types.

Definition 2.5 (level mapping)

A level mapping is a function $f : Call(P, S)/\sim \rightarrow \mathbb{N}$. ■

A levelmapping assigns to each relevant atom a natural number which can be interpreted as its *size*.

Definition 2.6 (acceptability wrt S)

Let S be a set of atomic queries with a fixed predicate and P a definite directly recursive program. P is *acceptable wrt S* if there exists a level mapping f and a model I for P , such that

- for any $A \in Call(P, S)$
- for any clause $A' \leftarrow B_1, \dots, B_n$ in P , such that $mgu(A, A') = \theta$ exists,
- for any atom B_i having the same predicate symbol as A and for any θ' such that $I \models B_j\theta\theta', 1 \leq j < i$:

$$f(A) > f(B_i\theta\theta')$$

■

The following proposition is one of the key results in [19].

Proposition 2.7

A directly recursive program P terminates under the left-to-right selection rule for any query in S if and only if P is acceptable wrt S . ■

The proposition hands us a practical method for checking the termination of a program with respect to a fixed set of queries. Off course, being able to prove acceptability depends largely on the creativity of coming up with a suitable levelmapping f , in addition to the model I and the call set $Call(P, S)$. Although not explicitly required by Definitions 2.5 and 2.6, automatic and semi-automatic approaches to termination analysis impose some further syntactic restrictions, as will be presented in the next subsection.

2.3 Norms

Note that neither Definition 2.5 nor Definition 2.6 impose any syntactical conditions on the levelmapping. In practice however, such a function is most often defined as a linear

combination of the sizes of the terms on fixed (per predicate) argument positions. That is, it is defined as

$$f(p(t_1, \dots, t_n)) = \sum_{i \in I_p} f'(t_i), \quad \text{for some } I_p \subseteq \{1, \dots, n\}$$

Obviously, the magic of this structure is incorporated in the set I_p and a new function f' . The set I_p is frequently referred to as the set of input positions. [44] describes how to automatically propose the set I_p and calls the resulting levelmappings *natural levelmappings*. It assumes however some kind of an oracle which proposes the function f' . Observe how the creative process of proving termination now reduces to finding a reliable oracle. These functions measuring the sizes of terms are commonly referred to as *norms*.

Definition 2.8 (norm)

A *norm* is a mapping $\|\cdot\| : U_P^E \rightarrow \mathbb{N}$. ■

Observe that norms are defined on the *extended Herbrand Universe*. With slight abuse of notation, we will often write $\|t\|$, with $t \in Term_P$.

Several examples of norms can be found in the literature. When dealing with lists, it is often appropriate to use *list-length*, which gives the depth of the rightmost branch in the tree representation of a list and 0 for any other term.

Definition 2.9 (list-length)

The *list-length* norm, denoted $\|\cdot\|_l$, is defined in the following way:

$$\begin{aligned} \|[x|y]\|_l &= 1 + \|y\|_l && \text{with } x \text{ and } y \text{ any term} \\ \|x\|_l &= 0 && \text{otherwise.} \end{aligned}$$
■

A more general norm is *term-size*, which counts the number of function symbols in a term.

Definition 2.10 (term-size)

The *term-size* norm, denoted $\|\cdot\|_t$, is defined in the following way:

$$\begin{aligned} \|f(t_1, \dots, t_n)\|_t &= 1 + \sum_{1 \leq i \leq n} \|t_i\|_t && \text{with } f \text{ any function symbol and } n > 0 \\ \|x\|_t &= 0 && \text{otherwise.} \end{aligned}$$
■

Another frequently used norm is *term-depth*, which gives the maximum depth of (the tree representation of) a term.

Definition 2.11 (term-depth)

The *term-depth* norm, denoted $\|\cdot\|_d$, is defined in the following way:

$$\begin{aligned} \|f(t_1, \dots, t_n)\|_d &= 1 + \max_{1 \leq i \leq n} \|t_i\|_d && \text{with } f \text{ any function symbol and } n > 0 \\ \|x\|_d &= 0 && \text{otherwise.} \end{aligned}$$
■

Studying the termination of queries containing non-ground input is known to cause problems in the analysis. The reason is the following: if a norm is applied to a ground term, then its value cannot be modified during the remainder of the computation. Therefore, one can measure an atom and compare it safely to the measure of another atom with the same predicate symbol occurring at a later step in the derivation. However, in the context of non-ground input terms, when measuring the second atom, the norm of the input terms in the previous atom may have changed, due to further instantiation of the term. This observation motivated the introduction of *rigid terms*.

Definition 2.12 (rigid term, see [7])

Let $\|\cdot\|$ be norm and t be a term. We say that t is *rigid* with respect to $\|\cdot\|$ if for any substitution σ , $\|t\sigma\| = \|t\|$. ■

Observe that rigidity of a term is not an absolute property, but instead, it is dependent on the chosen norm. If a term is known to be rigid with respect to a given norm, then its value under the norm will not change due to further instantiation of the term in subsequent derivation steps. In the termination analysis, the term can be treated as if it were ground. Of course, the notion of a rigid term is not very useful, unless we can provide support for detecting rigid terms syntactically from the program. This motivated the introduction of *semi-linear norms*.

Definition 2.13 (semi-linear norm, see [7])

A norm $\|\cdot\|$ is *semi-linear* if it is recursively defined by means of the following schema:

$$\begin{aligned} \|V\| &= 0 \text{ if } V \text{ is a variable, and} \\ \|f(t_1, \dots, t_n)\| &= c + \|t_{i_1}\| + \dots + \|t_{i_m}\| \\ &\text{with } c \in \mathbb{N}, \{i_1, \dots, i_m\} \subseteq \{1, \dots, n\} \\ &\text{and } c, i_1, \dots, i_m \text{ depend only on } f/n. \end{aligned}$$

■

In [7] it is shown how semi-linear norms allow to detect rigid terms syntactically. We briefly present a slight variant of their characterisation.

Definition 2.14 (functor arguments)

The set of functor arguments, $FunArgs_P$, associated to a program P is the set $\{f_i \mid f/n \in Fun_P \wedge 1 \leq i \leq n\}$. We release the subscript P when no confusion is possible. ■

In the next definitions, we assume the program P fixed.

Definition 2.15 (selected functor arguments)

The *selected functor arguments* of a semi-linear norm $\|\cdot\|$ is the set:

$$SFA(\|\cdot\|) = \{ f_i \in FunArgs \mid \|f(t_1, \dots, t_n)\| \text{ is defined as } \\ c + \|t_{i_1}\| + \dots + \|t_{i_m}\| \wedge \exists j \in \{1, \dots, m\} : i = i_j \}$$

■

From the above it must be clear that there is a direct correspondence between sets of functor arguments and semi-linear norms. Each set of arguments positions defines one semi-linear norm (up to the constant c) and vice versa. This characterisation of semi-linear norms by sets of argument positions now facilitates the detection of rigid terms.

Given the tree representation of a compound term $t = f(t_1, \dots, t_k)$, we say that an arc (n, m) in this tree representation *is covered by* an element f_i of $SFA(\|\cdot\|)$ if the label of n is f/k and the arc (n, m) corresponds to the i -th argument position of f .

Definition 2.16 (selected subterm)

A subterm u of a compound term $t = f(t_1, \dots, t_m)$ is a *selected subterm* of t under a semi-linear norm $\|\cdot\|$, if there exists a path in the tree representation of t , connecting its root to the root of the tree representation of u , and such that each arc in the path is covered by an element of $SFA(\|\cdot\|)$. ■

The characterisation of rigid terms can now be formulated for non-trivial semi-linear norms.

Definition 2.17 (trivial norm)

A semi-linear norm is *trivial* if it is a constant function. ■

Proposition 2.18

Given a non-trivial semi-linear norm $\|\cdot\|$, a term $t \in Term_P$ is rigid with respect to $\|\cdot\|$ if and only if t is a term which has no variable as a selected subterm. ■

Of course, for a trivial norm, every term is rigid.

Despite of their crucial importance for termination analysis ([36],[7],[44]), we are aware of no existing methods for automatically inferring these semi-linear norms. Section 3 presents one such method.

2.4 Rigid Types

The introduction of rigidity analysis and semi-linear norms as described in the previous section was a significant step forward in the direction of studying termination properties of programs for partially instantiated terms. It soon became clear however, that types could further refine the analysis. For this reason, types have fulfilled an increasingly important role in the state of the art approaches to Logic Program termination analysis ([44], [13], [8]).

As an example of the accurate information they provide, consider the following version of the one-level flattening program.

Example 2.19 (flatten)

```

flatten([], []).
flatten([L|Z], U)      ←  flatten(Z, U'), append(L, U', U).
append([], L, L).
append([H|L1], L2, [H|L3]) ←  append(L1, L2, L3).

```

Assume that we know that the queries of interest are $\leftarrow \text{flatten}(t_1, t_2)$, where the type of t_1 is τ_1 , the set of lists of lists of any terms and the type of t_2 is τ_2 , the set of lists of any terms. A type inference system (e.g. [30]) allows to derive that any descending call has the same types, while in any descending call of $\text{append}(t_3, t_4, t_5)$, all the terms t_3, t_4, t_5 have the same type τ_2 . Using an abstraction mechanism like modes, no matter which is the chosen semi-linear norm, it is not possible to prove termination of this program. The reason is that we are not able to prove that both the first term of flatten and the first term of append are rigid wrt the chosen semi-linear norm (this information provides the well-foundedness of the termination analysis in both cycles of the program). Hence, in this case, the problem is not due to semi-linear norms by themselves, but to the weak rigidity analysis modes provide. In fact, all terms in τ_1 and τ_2 are rigid wrt the semi-linear norm list-length, which is sufficient to prove termination of the program. Hence type analysis information provides a stronger rigidity analysis. ■

This is not the only issue for introducing type analysis: as we shall see, there are examples where the problem of failure of a termination proof is not due to the weakness of the rigidity analysis, but to the restriction imposed by the definition schema of semi-linear norms. This schema forces one to find for each function symbol a fixed set of selected argument positions. No information is used that takes into account *where* that particular symbol occurs. This may lead to problems in programs where a same functor is used in two different ways in two different types. By providing accurate information on the structure of terms occurring in derivations, types allow to select refined ways to measure the size of these terms.

The above mentioned issues motivated the use of type analysis in termination proofs in the work of [13], [44], [8]. These works also pointed out how the success or failure of a termination proof strongly depends on the choice (the definition) of the norm to measure terms. In section 3 we present a first simple but effective approach that

- addresses the fundamental problem of the automatic inference of the “right” norm;
- shows that type analysis can be used to solve it, in most of the cases.

In short, the proposal makes use of the characterisation of semi-linear norms by sets of argument positions. The main idea is to start from a maximal set of argument positions for each functor, thus considering all possible information. Then a (minimal) set of positions which account for possible non-rigid terms are removed from the set. Detecting which argument positions possibly take non-rigid arguments is detected through type analysis. By excluding sufficient positions, only the rigid part of the terms are measured.

Before illustrating this enhanced precision in the next section, we look a little deeper into one particular type formalism. As mentioned, the type formalism chosen in this paper are the rigid types of [30]. The reasons for this choice are

1. rigid types provide sufficient precision for the application,
2. automatic inference of rigid types through abstract interpretation has been fully developed and described in [30],
3. the tools for inferring them are available at our site, allowing for extensive experiments with the proposed techniques.

Due to space restrictions and since rigid types are not part of the contribution of this paper, we refer to [30] and [33] for the underlying intuitions. Here, we only recall the basic definitions and give an example.

There exist a number of *primitive types* (e.g. INT, REAL), which represent subsets of the set of constants in the language. We denote the set of all primitive types by \mathcal{P} , and we assume that there exists a function $Denote : \mathcal{P} \rightarrow 2^{Const_{\mathcal{P}}}$, mapping each primitive type to a corresponding set of constants. There also exists a special rigid type which is denoted by *Max*. Its denotation is the complete Herbrand universe.

Rigid types are formally defined by means of *type graphs*, which are a particular instance of directed graphs. We assume that the reader is familiar with the basics of graph theory.

Definition 2.20 (rigid type graph; adapted from [30])

A *rigid type graph* T is a 5-tuple, $(Nodes, ForArcs, BackArcs, Label, ArgPos)$, where

1. $Nodes$ is a finite, non-empty set of nodes,
2. $ForArcs \subseteq Nodes \times Nodes$ such that $(Nodes, ForArcs)$ is a *tree*,
3. $BackArcs \subseteq Nodes \times Nodes$ such that for each arc $(m, n) \in BackArcs$, node n is an ancestor of node m in $ForArcs$,
4. $Label$ is a function $Nodes \rightarrow \mathcal{P} \cup Const_{\mathcal{P}} \cup Fun_{\mathcal{P}} \cup \{Max, OR\}$, and
5. $ArgPos$ is a function: $\bigcup_{k>0} (\{m \in Nodes \mid Label(m) = f/k \in Fun_{\mathcal{P}}\} \times \{1, \dots, k\}) \rightarrow Nodes \setminus \{root\}$, such that for each $m \in Nodes$, with $Label(m) = f/k$, $ArgPos(m, \cdot) : \{1, \dots, k\} \rightarrow Nodes$ is a bijection from $\{1, \dots, k\}$ onto $\{n \in Nodes \mid (m, n) \in ForArcs \cup BackArcs\}$.

Each node labelled with a function symbol with arity k has k immediate descendants, each node labelled OR has at least two immediate descendants, and the other nodes have no descendants and are called *terminal nodes*. For a functor node n , we use the shorthand n/i to denote the $ArgPos(n, i)$. Descendants of OR-nodes or functor-nodes are found using the *Desc* function. ■

Definition 2.21 (Desc function)

$Desc : \{n \in Nodes \mid Label(n) \in Fun_{\mathcal{P}} \cup \{OR\}\} \rightarrow 2^{Nodes} : Desc(n) = \{n' \mid (n, n') \in ForArcs \cup BackArcs\}$. ■

A rigid type graph T describes a (possibly infinite) set of finite terms. This set of finite terms is found by means of the denotation function, \mathbb{D} . The next couple of definitions were inspired by similar definitions in [33].

Definition 2.22 (adapted from [33]: definition 2.3.2)

Let \mathbb{T} be the function $\mathbb{T} : Nodes \times 2^{(Nodes \times Term_{\mathcal{P}})} \rightarrow 2^{Term_{\mathcal{P}}}$:

- if $Label(n) \in Const_{\mathcal{P}}$ then $\mathbb{T}(n, I) = \{Label(n)\}$
- if $Label(n) \in \mathcal{P}$ then $\mathbb{T}(n, I) = Denote(Label(n))$
- if $Label(n) = Max$ then $\mathbb{T}(n, I) = Term_{\mathcal{P}}$
- if $Label(n) = OR$ then $\mathbb{T}(n, I) = \bigcup_{n' \in Desc(n)} \{t \mid (n', t) \in I\}$
- if $Label(n) = f/k$ then $\mathbb{T}(n, I) = \{f(t_1, \dots, t_k) \mid ArgPos(n, i) = n_i, (n_i, t_i) \in I\}$. ■

The set $2^{(Nodes \times Term_P)}$ forms a complete lattice with respect to \subseteq , \cap and \cup . The bottom element is \emptyset , while $Nodes \times Term_P$ is the top element.

Definition 2.23 (adapted from [33])

Let \mathbb{T}_{Nodes} be the function $\mathbb{T}_{Nodes} : 2^{(Nodes \times Term_P)} \rightarrow 2^{(Nodes \times Term_P)}$:
 $\mathbb{T}_{Nodes}(I) = I \cup \{(n, t) \mid n \in Nodes, t \in \mathbb{T}(n, I)\}$. Observe that \mathbb{T}_{Nodes} is continuous. ■

Definition 2.24 (denotation of a node in a type graph)

Let $T = (Nodes, ForArcs, BackArcs, Label, ArgPos)$ be a rigid type graph. The denotation of $n \in Nodes$ is defined as $\mathbb{D}(n) = \{t \mid (n, t) \in \mathbb{T}_{Nodes} \uparrow \omega\}$. ■

Definition 2.25 (denotation of a rigid type)

Let T be a rigid type with root n_{root} . Then $\mathbb{D}(T) = \mathbb{D}(n_{root})$. ■

Sometimes we use $\mathbb{D}(p(T_1, \dots, T_n))$ as an abbreviation for the set $\{p(t_1, \dots, t_n) \mid \forall 1 \leq i \leq n : t_i \in \mathbb{D}(T_i)\}$.

A few rigid types are now presented by means of their graphical representation. In Figure 1 the set of all nil-terminated lists which take nil-terminated lists of any terms as elements is depicted as a type graph. It could be used to specify the call type of the first argument of the flatten predicate.

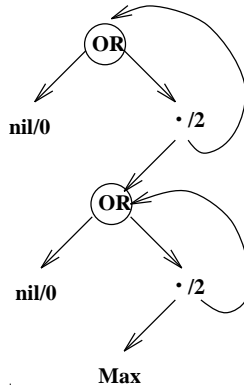


Figure 1: T_C , type graph representing lists of lists of terms.

Two further type graphs are shown in Figure 2. The first graph denotes the set of all f -terms accepting any term as its first argument and an integer or a similar f -term on its second position. The second one defines the left-associative version of the first one.

It is important to notice that variables, which are responsible for non-rigidity, are captured by *Max*-nodes.

In [29] and [30], rigid type graphs are used as a component of an abstract domain. Because several type graphs can have the same denotation, *compact* type graphs are introduced (see [30], where also an algorithm is presented for compactifying type graphs). This property does not limit the expressivity of type graphs. For reasons of efficiency, rigid type graphs must be further restricted to normal type graphs. This requires the following notion:

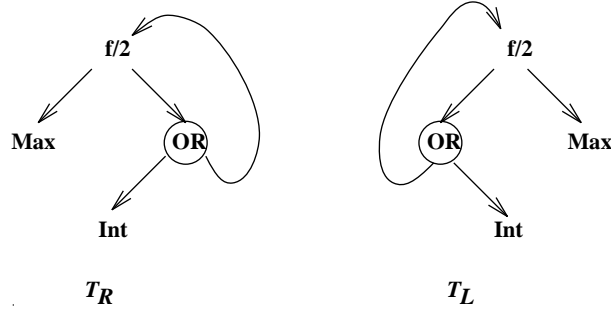


Figure 2: Left and right associative version of f -terms

Definition 2.26 (principal nodes; see [30]: definition 4.5)

Let $T=(Nodes,ForArcs,BackArcs,Label,ArgPos)$ be a compact rigid type graph. The *principal nodes* of a node $n \in Nodes$ are found by means of the function $Principal: Nodes \rightarrow 2^{Nodes}$:

$$\begin{aligned} Principal(n) &= \bigcup_{n' \in Desc(n)} Principal(n') && \text{if } Label(n) = OR \\ Principal(n) &= \{n\} && \text{otherwise.} \end{aligned}$$

■

Normal type graphs have the following properties :

1. no nodes with the empty set as denotation are allowed — except for a type graph containing one single node,
2. an OR-node cannot have an immediate descendant labelled Max,
3. the latter of two successive OR-nodes must have two or more ancestors,
4. no loops consisting of only OR-nodes are allowed, and
5. principal nodes in normal type graphs have distinct function symbols.

Only (5) reduces the expressivity with respect to rigid types. For more details we refer to [30]. Note that all types in Figures 1 and 2 satisfy this normality criterion.

Definition 2.27 (Recursive type)

A rigid type T is called *recursive* if its type graph contains at least one loop. ■

For the sake of termination, such types will be very important as, from a termination point of view, they represent recursive datastructures and they thus identify a first source of possible infinite recursion.

In [29] a system is proposed for statically analysing the flow of information in a program for some initial call description in terms of normal types. It computes for each predicate in the program a general call pattern and a corresponding success pattern.

Definition 2.28 (call and success pattern)

Let P be a definite Logic Program and let p/k be a predicate defined in P . Let S be a set of atomic queries. Then $p(T_1^c, \dots, T_k^c)$ is a *call pattern* for p/k wrt P and S if $\forall p(t_1, \dots, t_k) \in Call(P, S) : t_i \in \mathcal{ID}(T_i^c), \forall 1 \leq i \leq k$. $p(T_1^s, \dots, T_k^s)$ is a *success pattern* for p/k wrt P and S if $\forall p(t_1, \dots, t_k) \in Call(P, S), \forall \theta$, if $P \models p(t_1, \dots, t_k)\theta$ then $t_i\theta \in \mathcal{ID}(T_i^s), \forall 1 \leq i \leq k$. ■

Looking back at Example 2.19, it is able to compute the following information:

$$\text{flatten}(T_C, \text{Max}) \rightarrow \text{flatten}(T_C, T_{list}),$$

$$\text{append}(T_{list}, T_{list}, \text{Max}) \rightarrow \text{append}(T_{list}, T_{list}, T_{list})$$

where the symbol \rightarrow separates call and success descriptions wrt P and $\mathcal{D}(\text{flatten}(T_C, \text{Max}))$. T_C is the normal type illustrated in Figure 1 and T_{list} the subgraph of T_C rooted at the deepest OR-node.

3 Deriving one semi-linear norm

As mentioned in the previous sections, termination analysis techniques make use of a level mapping which only takes into account the norms of certain argument positions of the predicates, the so-called input positions. For the time being, we will assume that for each predicate a number of argument positions have been identified as 'input' positions, postponing the problem of inferring them. As a result, we now assume that a set $ST = \{T_1, \dots, T_n\}$ of normal types are given, which is the union of all call types that the type analysis inferred for the 'input' argument positions of predicates in P .

Definition 3.1 (rigidity of a type wrt a norm)

A normal type T is *rigid wrt a norm* $\|\cdot\|$, if every term $t \in \mathcal{D}(T)$ is rigid wrt $\|\cdot\|$. ■

The problem is now: given $ST = \{T_1, \dots, T_n\}$, derive a semi-linear norm $\|\cdot\|$ such that T_i is rigid wrt to $\|\cdot\|$, for all $i = 1, \dots, n$. First, we extend our representation of a type graph with a sixth component, the *ArcLabel*.

Definition 3.2 (*ArcLabel*)

ArcLabel is a function: $\{(m, n) \in \text{ForArcs} \cup \text{BackArcs} \mid \text{Label}(m) \in \text{Fun}_P\} \mapsto \text{FunArgs}$, defined as follows:

$$\text{ArcLabel}((m, n)) = f_i \leftrightarrow \text{Label}(m) = f/k \wedge \text{ArgPos}(m, i) = n.$$

In this way, all arcs except the ones that start from an *OR* node are labelled with a functor argument. ■

Proposition 2.18 was formulated as a criterion to syntactically characterise the rigid terms, given a semi-linear norm $\|\cdot\|$. But in fact, given a set of terms S , it can also be read as a criterion on semi-linear norms, characterising all such norms that make all terms in S rigid. The criterion is that $\|\cdot\|$ should be such that for every path in the tree representation of any term t in S which connects the root of the tree to a node labelled by a variable, at least one of the arcs in the path is not covered by the selected functor arguments, $SFA(\|\cdot\|)$.

As mentioned, in the context of rigid types, variables are denoted by nodes labelled by Max. Thus, given a set of normal types ST , the problem now is to find a semi-linear norm $\|\cdot\|$, such that for each path in any type graph $T \in ST$, which connects the root of T to a node labelled Max, there should at least be one *ArcLabel* (n) on this path, which is not in $SFA(\|\cdot\|)$. This is formalised as follows:

Definition 3.3 (critical path)

Let ST be a set of type graphs. A *critical path* in ST is a path in a type graph $T \in ST$, connecting the root of T with a terminal node of T , labelled by Max. ■

Proposition 3.4

Let ST be a set of normal type graphs and $\|\cdot\|$ a non-trivial semi-linear norm. All types in ST are rigid wrt $\|\cdot\|$, if and only if there is no critical path in any $T \in ST$, whose ArcLabels are all in $SFA(\|\cdot\|)$. ■

Proof

The result follows immediately from Proposition 2.18 and the discussion above. Due to Definition 3.1, however, which links the rigidity of a type T wrt a norm to the rigidity of all terms $t \in \mathcal{D}(T)$, the formal proof of the proposition is quite technical: it relies on the fixpoint definition of the denotation function \mathcal{D} . Since this produces no new conceptual insights, we omit it. ■

Next we discuss how to use Proposition 3.4 for the generation of the desired norm. First, observe that if we fix all constants c in the defining equations

$$\|f(t_1, \dots, t_n)\| = c + \|t_{i_1}\| + \dots + \|t_{i_m}\|$$

of a semi-linear norm to be 1, then the semi-linear norm is uniquely determined by its selected functor arguments, $SFA(\|\cdot\|)$.

Below, we will say that a path in a graph is cut, if an arc of the path is removed. We say that a path is disconnected from a graph, if all paths from the root of the graph to the root of the path are cut. Proposition 3.4 gives rise to a first approach to (non-deterministically) generate semi-linear norms: Let $FA(ST)$ be the set of functor arguments that are labels on an arc of a type graph in ST and let C be any subset of $FA(ST)$ such that if all the arcs labelled by an element in C are deleted, then all critical paths of ST are cut. Then, let $\|\cdot\|$ be defined by $SFA(\|\cdot\|) = (FA(ST) \setminus C)$.

At this point we need a heuristic to guide us in the definition of a useful semi-linear norm: the one we propose is to ensure that in the corresponding deletion of arcs from type graphs, the number of loops in the graphs that are cut or disconnected is minimised.

The reason for this heuristic is that loops in the type graphs represent recursive data structures. Usually, the recursive data structures in the input are the ones on which the termination of a program depends. Thus, we should select the norm in such a way that as many as possible of them remain in the focus of the norm.

Summarising, we can define a semi-linear norm derived from a set of types as follows:

Definition 3.5 (semi-linear norm derived from a set of types)

Let ST be a set of rigid type graphs. Let $FA(ST)$ be the set of functor arguments that are labels on an arc of a type graph in ST and let C be a subset of $FA(ST)$ such that if all the arcs labelled by an element in C are deleted then: (1) all critical paths of ST are cut; (2) the number of loops in the type graphs in ST that are cut or disconnected by C is minimal.

We say that the semi-linear norm $\|\cdot\|$ is *derived from ST (according to C)* if $SFA(\|\cdot\|) = (FA(ST) \setminus C)$. ■

Remark that in general there is more than one derived norm for a set of type graphs. However from our viewpoint they are equivalent. Our goal is to interrupt all critical paths saving as much loops as it is possible. A further refinement might be to remove a *minimal* number of elements from $FA(ST)$. This is motivated by the fact that all types in ST are assumed to be input arguments. Thus, by minimising, we aim to obtain a norm which takes as much input data into account as possible.

Note that, even with the additional secondary criterion, there can still be several semi-linear norms derivable from a set ST . Each of them satisfies our initial goal of obtaining rigidity for all call types:

Corollary 3.6

Let $\|\cdot\|$ be a semi-linear norm derived for ST and $T \in ST$, then T is rigid wrt $\|\cdot\|$. ■

The result follows immediately from Proposition 3.4 and Definition 3.5.

Example 3.7 (one-level flatten)

Reconsider the flatten program of Example 2.19 and suppose the program is queried with the call pattern $flatten(T_C, Max)$ where T_C is the graph of Figure 1. From type analysis we derive the following call patterns for the recursive predicates:

$$flatten(T_C, Max), append(T_{list}, T_{list}, Max)$$

where T_{list} is the type "list of objects", rooted at the deepest OR-node in the typegraph for T_C . We derive a norm which renders rigid all terms in the denotation of types T_C and T_{list} . Since $FA(\{T_C, T_{list}\}) = \{.1, .2\}$ and deleting all arcs labelled with $.1$ disconnects all critical paths and disconnects only one loop (deleting all arcs with $.2$ disconnects all loops), $SFA(\|\cdot\|) = \{.2\}$ and the derived norm is list-length:

$$\begin{aligned} \text{if } t = [t_1|t_2] \text{ then } \|[t_1|t_2]\| &= 1 + \|t_2\| \\ \text{else} \qquad \qquad \qquad \|t\| &= 0. \end{aligned}$$

To prove termination we choose a very natural definition of level mapping: the value associated to a call atom is the sum of the values of its (rigid) input arguments according to the derived norm. It is easy to see that the value of the first argument of flatten is always one more than the value of the first argument of the next recursive call, so that the proposed level mapping decreases. The same is true also for the sum of the first two arguments of append. Note that in this example we did not need to account for a model I as stated in Definition 2.6. This is due to the particular form of recursion of append and flatten, which takes place on the first atom in the body of recursive clauses. ■

We conclude the section with a comment on how to infer the 'input' argument positions from the set of call types computed by the type analysis. In [19], this set is computed through a preliminary mode analysis, computing one call mode pattern for each predicate. An argument position is considered to be an input position if it corresponds to a ground entry in the mode pattern for that predicate.

A very simple approach, closely related to mode analysis, is to regard all call types labelled with Max as the output positions and all others as input. As mentioned before, $\mathcal{D}(Max)$ contains the variables, so that every argument position of a predicate which - at least once - is called with this argument free, obtained the label Max from the type

analysis. An alternative approach is to consider an argument position of a predicate as 'input' if and only if its corresponding call type is recursive. Although this may correspond less to our intuitive notion of 'input', it has the advantage that in the definition of the semi-linear norm only recursive data structures are taken into account. As mentioned before, these are the relevant ones in terms of termination criteria.

4 Deriving norms from rigid type graphs

A lot of practical termination provers and practical applications use *one* semi-linear norm to measure *all* call types for the entire program. This can sometimes be too restrictive, the main reason being that it reduces the norm on each function symbol to a fixed set of selected argument positions. It can be such that in a first graph a set of argument positions P_1 must be excluded to obtain the rigidity property while in a second type a set P_2 must be excluded. One semi-linear norm needs to exclude $P_1 \cup P_2$. Similarly, if the functor occurs at two different locations of the same type graph, $P_1 \cup P_2$ need to be excluded as well. We present two examples to illustrate that this may result in imprecise norms.

Example 4.1

First, reconsider Example 1.1

$$\begin{aligned} p(f(X, f(Y, Z)), R) & : - p(f(f(X, Y), Z), R). \\ p(f(X, Y), f(X, Y)) & : - integer(Y). \end{aligned}$$

Assume that the $p/2$ predicate is used to verify that its second argument is the left associative version of its first argument, i.e. as if it were queried by $p(t_R, t_L)$, where t_R and t_L are in the denotation of T_R and T_L , the type graphs in Figure 2. When trying to derive a semi-linear norm from both type graphs, the rigidity property for T_R forces us to exclude the first argument for the f -functor, while the second argument must be excluded for the same reason in T_L . In this case all arguments must be disregarded and we obtain a trivial norm.

This would not have been necessary if it was decided to take more information into account. It should be desirable to measure a term according to the type to which it belongs. For example if the term belongs to type T_R , its size should be computed in terms of the size of its second argument, if it belongs to T_L in terms of its first argument. To obtain this, we need to introduce two different norms $\|\cdot\|_{T_L}$ and $\|\cdot\|_{T_R}$, defined on $\mathcal{D}(T_L)$ and $\mathcal{D}(T_R)$ respectively. ■

Example 4.2

As a second example, consider a slightly different definition for the flat/2 predicate which no longer depends on the append/3 predicate:

$$\begin{aligned} flat([], []). \\ flat([[]|T], R) & \leftarrow flat(T, R). \\ flat([[H|T]|T'], [H|R]) & \leftarrow flat([T|T'], R). \end{aligned}$$

Let $flat$ be queried by $flat(t_C, t_{Max})$ where t_C is any term in the denotation of T_C , the type represented in Figure 1, and t_{Max} is any term. If we automatically generate a

semi-linear norm wrt to these types, as in the previous section, the listlength norm will be proposed. However, list-length does not allow to prove termination for the third clause for flat. What is needed is a norm that computes the sum of the lengths of all members in lists of type T_C . With this norm, termination can be proved. Notice that again, the problem is due to the fact that a same functor (the list constructor) occurs twice in T_C . Removing the second argument to make the type rooted at the deepest occurrence of the list constructor rigid also forces us to remove this argument for the entire type. Again, allowing two different norms, one for the type T_C itself and one for the type rooted at the second occurrence of the list-constructor in T_C will solve the problem. ■

Below, we regard a rigid type graph as a specification of a set of rigid types: every node in the type graph can be considered as the root of a new type graph, defined by the subgraph rooted at that node. To be able to refer to such "subtypes" explicitly, we need the following definition, which extends the (original) rigid type graph by one extra component. The `TypeLabel` component is the one responsible for splitting up a type into its various subtypes.

Definition 4.3 (labelled rigid type graph; adapted from [30])

A labelled rigid type graph T is a 6-tuple,

$$(Nodes, ForArcs, BackArcs, Label, ArgPos, TypeLabel),$$

where $(Nodes, ForArcs, BackArcs, Label, ArgPos)$ is a rigid type graph and $TypeLabel$ is a function $Nodes \rightarrow TypeNames$ and $TypeNames$ is a set of unique identifiers. ■

Example 4.4

Figure 3 represents a labelled version of the graph T_C in Figure 1. Type Labels are denoted by the Greek character τ and are indexed.

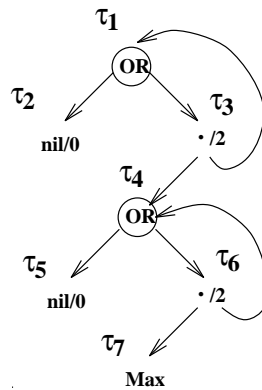


Figure 3: An example of a labelled graph

The next definition enables us to associate sets of terms to `TypeLabels`.

Definition 4.5 (denotation of a TypeLabel)

Let $\tau = TypeLabel(n)$ be the `TypeLabel` of a node n in a labelled type graph T . We extend the notion of denotation to `TypeLabels` as: $\mathcal{D}(\tau) = \mathcal{D}(n)$. ■

OR-nodes define a type whose denotation is the union of the denotation of some other types. Moreover, if we work with normal type graphs this is a union of disjoint sets of terms. Under the assumption that we deal with normal types and given any term in the denotation of an OR-node, the graph enables us to identify in an unambiguous way its corresponding TypeLabel as one of the TypeLabels of the principal nodes of the OR-node. The next definition collects for some OR-node all TypeLabels which are relevant (in the above sense) for it.

Definition 4.6 (successor typeLabel set of an OR-node)

Let T be a labelled type graph and n a node in T such that $label(n) = OR$. The *successor typeLabel set of n* , $succ_Label(n)$ is defined as

$$succ_Label(n) = \{ \tau \mid \exists (n, m) \in ForArcs \cup BackArcs \wedge m' \in Principal(m) \wedge TypeLabel(m') = \tau \}.$$

■

Given a normal type T , we face the problem of deriving a norm (not necessarily semi-linear any more) which makes all terms in the denotation of the type rigid. The general idea is that a type should not decide on how to measure a term belonging to some other type. For instance, considering the type with TypeLabel τ_1 in Example 4.4, the norm $\|\cdot\|_{\tau_1}$ should shift the responsibility of measuring a term t of type τ_1 to norms $\|\cdot\|_{\tau_2}$ and $\|\cdot\|_{\tau_3}$, depending on which type (τ_2 or τ_3), t actually has. If $t = [t_2|t_3]$ is of type τ_3 , then the definition of $\|\cdot\|_{\tau_3}$ will take responsibility on how to measure the top-level constructor, $\cdot/2$, only, but leave it up to $\|\cdot\|_{\tau_4}$ and $\|\cdot\|_{\tau_1}$ to measure t_2 and t_3 . This results in the following definitions.

Definition 4.7 (FuncNodes and NodeArgs)

Let T be a normal rigid type graph.

$$\begin{aligned} FuncNodes(T) &= \{ n \in Nodes \mid Label(n) = f/k, \text{ for some } f/k \in FuncP \} \\ NodeArgs(T) &= \{ (n, i) \mid n \in FuncNodes(T), \text{ with } Label(n) = f/k, f_i \in FuncArgs \} \end{aligned}$$

■

Definition 4.8 (coefficient assignment)

Let T be a normal rigid type graph. A coefficient assignment on T is a function $a : NodeArgs(T) \rightarrow \{0, 1\}$.

■

Definition 4.9 (offset assignment)

Let T be a normal rigid type graph. An offset assignment on T is a function $b : FuncNodes(T) \rightarrow \{0, 1\}$.

■

Definition 4.10 (typed norm induced by a TypeLabel in a type graph)

Let T be a labelled normal type and $\tau = TypeLabel(n)$ the TypeLabel of a node n in T . Let $a : NodeArgs(T) \rightarrow \{0, 1\}$ be a coefficient assignment on T and $b : FuncNodes(T) \rightarrow \{0, 1\}$ an offset assignment on T . The *typed norm induced by τ , a and b* is the function $\|\cdot\|_{\tau,a,b} : Term_P \rightarrow \mathbb{N}$ defined as:

$$\begin{aligned} \text{if } t \notin \mathcal{D}(\tau) & \text{ then } \|t\|_{\tau,a,b} = 0 \\ \text{else if } label(n) = OR & \text{ then } \|t\|_{\tau,a,b} = \sum_{\tau_i \in Succ_Label(n)} \|t\|_{\tau_i,a,b} \\ \text{else if } label(n) = f/k & \text{ then } \|f(t_1, \dots, t_k)\|_{\tau,a,b} = b(n) + \sum_{i=1}^k a(n, i) * \|t_i\|_{\tau_i,a,b} \\ & \text{with } \tau_i = TypeLabel(ArgPos(n, i)) \\ \text{else } & \|t\|_{\tau,a,b} = 0 \end{aligned}$$

■

Because of the restriction to normal types, the case in which $label(n) = \text{OR}$ reduces $||t||_{\tau,a,b}$ to exactly one of the $||t||_{\tau_i,a,b}$. As mentioned above, for normal types, the top-level functor of t corresponds to a unique $m' \in Principal(n)$ and its $TypeLabel(m')$. Since moreover $||t||_{\tau,a,b} = 0$ for terms t such that $t \notin \mathcal{D}(\tau)$, at most one of the summands in $\sum_{\tau_i \in Succ_Label(n)} ||t||_{\tau_i,a,b}$ is non-zero.

The last case in the definition captures primitive types and the Max type. These types get assigned the constant zero-norm. Doing so, Max nodes are cut off from the graph where they essentially should be cut off: just after their parent node.

Each normal type is now assigned a norm as follows:

Definition 4.11 (typed norm induced by a normal type graph)

Let T be a labelled normal type graph with root node n_{root} . The *norm induced by T , the coefficient assignment a and the offset assignment b* , is the norm induced by $\tau = TypeLabel(n_{root})$, a and b . ■

It is very important to remark the generic character of Definition 4.10. The import of the coefficient assignment and the offset assignment associates to each labelled rigid type a class of norms. Each member of this class will possess the heavily requested rigidity property, as will become clear further.

One frequently used member of this class is the one which maps all coefficients to 1 and thus considers all possible information. Such a norm is called natural.

Definition 4.12 (natural norm induced by a normal type graph)

Let T be a labelled normal type graph and let a and b be the constant 1-assignments on T . The *natural norm induced by T* , is the norm induced by $\tau = TypeLabel(n_{root})$, a and b . This norm is denoted by $||\cdot||_{\tau}^n$. ■

Example 4.13

Consider again the *flat* program of Example 4.2. Definition 4.10 can be applied to each node in the labelled graph presented in Figure 3. To focus on the important aspects of typed norms, we consider here the natural norms induced by the type graphs. This gives the following definitions:

$$\begin{aligned} ||\cdot||_{\tau_2}^n &= ||\cdot||_{\tau_5}^n = ||\cdot||_{\tau_7}^n \equiv 0 \\ ||t||_{\tau_1}^n &= ||t||_{\tau_2}^n + ||t||_{\tau_3}^n \\ ||t||_{\tau_4}^n &= ||t||_{\tau_5}^n + ||t||_{\tau_6}^n \\ |[t_1|t_2]|_{\tau_3}^n &= 1 + ||t_1||_{\tau_4}^n + ||t_2||_{\tau_1}^n \\ |[t_1|t_2]|_{\tau_6}^n &= 1 + ||t_1||_{\tau_7}^n + ||t_2||_{\tau_4}^n \\ ||t||_{\tau_3}^n &= ||t||_{\tau_6}^n = 0, \text{ for any other term } t. \end{aligned}$$

To clarify the definition, we can resolve this system in function of $||\cdot||_{\tau_1}^n$ and $||\cdot||_{\tau_4}^n$, and obtain:

$$\begin{aligned} |[t_1|t_2]|_{\tau_1}^n &= 1 + ||t_1||_{\tau_4}^n + ||t_2||_{\tau_1}^n \\ |[t_1|t_2]|_{\tau_4}^n &= 1 + ||t_2||_{\tau_4}^n \\ ||t||_{\tau_1}^n &= ||t||_{\tau_4}^n = 0, \text{ for any other term } t. \end{aligned}$$

■

The following proposition is the main result of our construction.

Proposition 4.14

Let T be a labelled normal type, n a node in T , $\tau = \text{TypeLabel}(n)$, a a coefficient assignment on T and b an offset assignment on T . Then $\forall t \in \mathcal{ID}(\tau) : t$ is rigid with respect to $\|\cdot\|_{\tau,a,b}$. ■

Proof

The proof is by induction on the \mathcal{IT}_{Nodes} -operator.

We know: $\mathcal{ID}(\tau) = \mathcal{ID}(n)$ if $\text{TypeLabel}(n) = \tau$ and $\mathcal{ID}(n) = \{t|(n, t) \in \mathcal{IT}_{Nodes} \uparrow \omega\}$.

The base case $\forall t \in \{t|(n, t) \in \mathcal{IT}_{Nodes} \uparrow 0\}$: $\|t\|_{\tau,a,b} = \|t\theta\|_{\tau,a,b}, \forall \theta$, trivially holds.

Suppose we have proved our proposition for $\mathcal{IT}_{Nodes} \uparrow i$. We must prove it for $\mathcal{IT}_{Nodes} \uparrow (i + 1)$: $\mathcal{IT}_{Nodes} \uparrow (i + 1)$ has been defined as $\{(n, t)|n \in Nodes \wedge t \in \mathcal{IT}(n, \mathcal{IT}_{Nodes} \uparrow i)\}$. A case analysis on node n is sufficient.

- $Label(n) = OR$
 $\|t\|_{\tau,a,b}$ has been defined as $\sum_{\tau_k \in Succ_Label(n)} \|t\|_{\tau_k,a,b}$. Because of the construction of the \mathcal{IT} operator, $\exists j$ with $(n_j, t) \in \mathcal{IT}_{Nodes} \uparrow i$, such that $(n, n_j) \in ForArcs \cup BackArcs$. Because of the normality property of the rigid type graph, there exists only one such a j . Therefore $\|t\|_{\tau,a,b}$ can be written as $\|t\|_{\tau_j,a,b}$. From the induction hypothesis we obtain that $\|t\|_{\tau_j,a,b} = \|t\theta\|_{\tau_j,a,b}$.
- $Label(n) = f/k$
The norm $\|f(t_1, \dots, t_k)\|_{\tau,a,b}$ has been defined as $b(n) + \sum_{i=1}^k a(n, i) * \|t_i\|_{\tau_i,a,b}$. From the construction of the \mathcal{IT} operator we know that $\exists(n_j, t_j), \forall 1 \leq j \leq k$ such that $(n_j, t_j) \in \mathcal{IT}_{Nodes} \uparrow i$. It suffices to apply the induction hypothesis to obtain that $\forall i : \|t_i\|_{\tau_i,a,b} = \|t_i\theta\|_{\tau_i,a,b} \forall \theta$.
- $Label(n) \in Const_P, Label(n) \in \mathcal{P}$ or $Label(n) = Max$
In each of these cases, the norm has been defined as $\|\cdot\|_{\tau,a,b} \equiv 0$. Consequently, the condition is satisfied. ■

The purpose of constructing typed norms is to use them in a termination proof. From the above it becomes clear that the extension from norms to typed norms as such has no impact on the termination conditions of subsection 2.2. The reason is that, although different terms may be measured using different norms depending on their types, the two atoms A and $B_i\theta\theta'$ in the acceptability condition are measured in the same way. This is because they have the same predicate, there is only one tuple of call types associated to each predicate and the level mapping is defined as a fixed combination of the typed norms associated to the call types. Thus, the call and descending recursive call are measured in the same way, and Definition 2.6 still captures well-founding and termination.

Example 4.15

Let T_C denote the type graph of Figure 3 (with $\text{TypeLabel} \tau_1$) and T the subgraph with $\text{TypeLabel} \tau_4$ as its root. Note that all terms denoted by T_C and T are indeed rigid with respect to respectively $\|\cdot\|_{\tau_1}^n$ and $\|\cdot\|_{\tau_4}^n$.

The norm can now be used to prove termination of the flat predicate of Example 4.2 as follows: type analysis gives us the following call types: $flat(T_C, Max)$. Consider the levelmapping

$$|flat(t_1, t_2)| = \|t_1\|_{\tau_1}^n + \|t_2\|_{Max}^n = \|t_1\|_{\tau_1}^n$$

and the set S consisting of all atoms in the given call pattern. As a corollary of Proposition 4.14, we have that for any $flat(s, t) \in S$ and for any substitution θ : $|flat(s, t)\theta| = |flat(s, t)|$ (in [19] this property is referred to as 'rigidity of the levelmapping with respect to the set S ').

The termination property will now be proved for the third flat clause. Take any atom $flat(s, t)$ of the set S such that $\theta = mgu(flat(s, t), flat([H|T]|T', [H|R]))$ exists. We must prove that the size of $flat(s, t)$ is strictly larger than that of $flat([T|T'], R)\theta$:

$$\begin{aligned} |flat(s, t)\theta| &= |flat([H|T]|T', [H|R])\theta| = \|[[H|T]|T']\theta\|_{\tau_1}^n \\ &= 1 + \|[H|T]\theta\|_{\tau_4}^n + \|T'\theta\|_{\tau_1}^n \\ &= 1 + 1 + \|T\theta\|_{\tau_4}^n + \|T'\theta\|_{\tau_1}^n \\ |flat([T|T'], R)\theta| &= \|[T|T']\theta\|_{\tau_1}^n \\ &= 1 + \|T\theta\|_{\tau_4}^n + \|T'\theta\|_{\tau_1}^n \end{aligned}$$

Because of rigidity, $|flat(s, t)\theta| = |flat(s, t)|$, so that the desired decrease is proved. The same can be done for the second clause, giving us the desired termination property for the flat program. \blacksquare

5 Sets of Norms, Adequate for Proving Termination

5.1 The problem

As illustrated in the previous section, typed norms can be used in a termination proof without much complications. However, this is only the case when calls can be compared directly. The termination issue becomes more complicated as soon as sideways information passing is involved, in other words, when the model I of Definition 2.6 is needed because of intermediate atoms. In [19], in the context of one semi-linear norm, the model I is provided in the form of *interargument relations*, also referred to as *size relations*.

Definition 5.1 (interargument relation)

Let P be a definite program, p/n a predicate in P and $\|\cdot\| : Term_P/\sim \rightarrow \mathbb{N}$ a norm. An *interargument relation for p wrt $\|\cdot\|$* is a relation $R_p \subseteq \mathbb{N}^n$, such that for every computed answer, $p(t_1, \dots, t_n)$, for any possible call to p , $(\|t_1\|, \dots, \|t_n\|) \in R_p$. \blacksquare

It is important to notice that the above definition of interargument relation is strongly dependent on one fixed norm. Reconsidering typed norms, it does not seem very interesting to use such detailed norms in measuring atoms whilst losing this level of precision through the mono-typed description of interargument relations. Instead, we are interested in deriving an interargument relation for atoms which holds wrt the norms induced by the types each of its arguments has at the moment of the recursive call. Said in another way, deriving termination proofs using typed norms calls for the use and derivation of typed size relations. The following definition is a straightforward extension of the original one.

Definition 5.2 (interargument relation wrt a tuple of typed norms)

Let p/n be a predicate of P and $(\|\cdot\|_{\tau_1, a_1, b_1}, \dots, \|\cdot\|_{\tau_n, a_n, b_n})$ an n -tuple of typed norms, An interargument relation for p/n wrt $(\|\cdot\|_{\tau_1, a_1, b_1}, \dots, \|\cdot\|_{\tau_n, a_n, b_n})$ is a relation $R_p \subseteq \mathbb{N}^n$ such that for any $p(t_1, \dots, t_n)$ satisfying $P \models p(t_1, \dots, t_n)$ and $t_i \in \mathcal{D}(\tau_i), \forall i = 1, \dots, n$, $(\|t_1\|_{\tau_1, a_1, b_1}, \dots, \|t_n\|_{\tau_n, a_n, b_n}) \in R_p$. ■

Notice that the notion is more specific than that of an interargument relation in the sense that the relation is not required to hold for the entire success set for p in P .

Typed sizerelements allow for a high level of precision as is illustrated in the following example.

Example 5.3 (varlist/2)

The following program accepts a natural number on its first position and it constructs a list of variables of that length.

```
varlist(0, []).
varlist(s(N), [H|T]) ← varlist(N, T).
```

Let τ_1 and τ_4 be the labelled types of Figure 4. Define a_1, b_1 and b_2 to be the constant 1-mappings and let a_2 be 0 on the tuple $(\tau_6, 1)$ and 1 everywhere else. Then the following relation is a correct interargument relation for *varlist* wrt $(\|\cdot\|_{\tau_1, a_1, b_1}, \|\cdot\|_{\tau_4, a_2, b_2})$.

$$\{(x, y) \in \mathbb{N}^2 \mid x = y\}$$

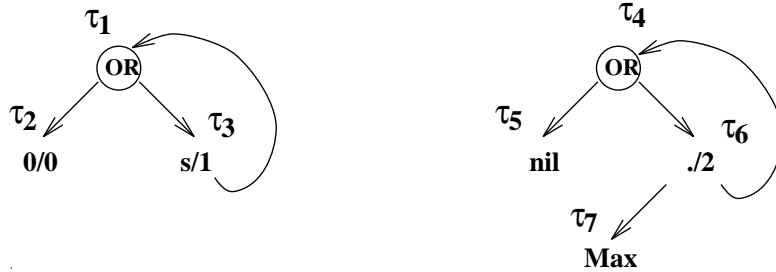


Figure 4: Success types for varlist/2: natural numbers and lists of variables.

One of the main problems we now need to solve is to produce a set of typed norms, as input for the definition of the interargument relation, that is sufficiently precise and consistent to provide a useful model for Definition 2.6. At first sight, an arguable choice seem to constitute the global success types for the predicate. However, using such size relations results in very hard consistency problems, as the following example illustrates.

Example 5.4 (permute)

```
permute([], []).
permute(L, [El|T]) ← delete(El, L, L1), permute(L1, T).
delete(X, [X|T], T).
delete(X, [H|T], [H|T']) ← delete(X, T, T').
```

Suppose we are interested in proving termination for $permute(t_1, t_2)$ atoms obeying the call pattern $permute(\tau_4, Max)$, where τ_4 is the type of Figure 4, representing lists of any terms. Along the lines of the previous section, the call pattern fixes our levelmapping as $|permute(t_1, t_2)| = ||t_1||_{\tau_4}^n$, where $||\cdot||_{\tau_4}^n$ is the natural norm induced by τ_4 .

A type analysis learns that the delete predicate is always called obeying the pattern $delete(\tau_7, \tau_4, Max)$, and that its success pattern is $delete(\tau_7, \tau_6, \tau_4)$. A correct interargument relation for delete wrt $(||\cdot||_{\tau_7}^n, ||\cdot||_{\tau_6}^n, ||\cdot||_{\tau_4}^n)$ is the following:

$$\{(x, y, z) \in \mathbb{N}^3 \mid y = z + 1\}$$

Unfortunately, this does not allow one to prove a decrease between $||L\theta||_{\tau_4}^n$ and $||L_1\theta||_{\tau_4}^n$ with $\theta = mgu(permute(t_1, t_2), permute(L, [El|T]))$, simply because there is no relation between $||L\theta||_{\tau_4}^n$ (the result of measuring $L\theta$ in the permute-call) and $||L\theta||_{\tau_6}^n$ (the result of measuring $L\theta$ in the delete atom).

In this case it is easy to observe that $||L\theta||_{\tau_4}^n$ will be the same as $||L\theta||_{\tau_6}^n$, as τ_6 is a subtype of τ_4 . However, more complex cases may occur for which such a relationship does not hold any longer. ■

One attractive solution could be to feed in local success types instead of the global ones, but very soon, this leads to other even more complicated problems. We refer to the discussion for more details.

The main idea we adopt in this paper to solve the problem, is that whenever a term *can* be measured under two different norms, its size under these norms should be the same. In the example, this would fall down to saying that $\forall t \in \mathcal{D}(\tau_4) \cap \mathcal{D}(\tau_6) : ||t||_{\tau_4} = ||t||_{\tau_6}$. Such a property is called *matching*.

Definition 5.5 (matching norms)

Let $||\cdot||_{\tau_1}$ and $||\cdot||_{\tau_2}$ be two typed norms. $||\cdot||_{\tau_1}$ and $||\cdot||_{\tau_2}$ are *matching* if and only if

$$\forall t \in \mathcal{D}(\tau_1) \cap \mathcal{D}(\tau_2) : ||t||_{\tau_1} = ||t||_{\tau_2}$$

■

To summarise the above discussion it should be clear that although types enhance the power of the analysis, they pose three relevant questions:

- which types are the relevant ones in a termination proof, i.e. which types identify a set of typed norms useful to the analysis;
- how to enforce typed norms to be matching, and
- how can we derive typed interargument relations.

In the following subsections we deal with these problems:

- In the first subsection, well-typed programs are introduced. This class of programs gives clues on how a variable could be typed at different occurrences. This way, such a program allows detecting where terms could be differently measured.

- In the next subsection, these well typed programs form the basis for identifying all couples of norms on which the matching property should hold. For each such a set of couples, a set of equations is generated and it is shown that if the system of equations is solvable, then all norms induced by these types can be matched.
- Next, a match algorithm is introduced. It's output is given as a set of constraints on the coefficient and offset assignments of any input types. Any assignments satisfying these constraints lead to matching norms.
- Finally, some alternative solutions are discussed. One of those derives one semi-linear norm from all typed norms. Although not always applicable, its main merit is obviously its compatibility with existing strategies for termination proofs.

5.2 Well-typed programs

In this subsection we will discuss how to derive from a normal definite program an analogous one where all available type information is explicitly manifested. These programs will prove extremely important for deriving interargument relations for predicates.

For this section, we will assume that all programs are in normal form. This means that all unifications are made explicit by means of the unification builtin $= /2$ as follows.

Definition 5.6 (normal program)

A definite program P is in *normalised form*, if

- every occurrence of an atom with a user defined predicate p/n in it has the form $p(X_1, \dots, X_n)$, with X_1, \dots, X_n distinct variables,
- every occurrence of an atom with predicate $= /2$ in it has the form $X = Y$ or $X = f(Y_1, \dots, Y_n)$, with X, Y, Y_1, \dots, Y_n distinct variables and f a functor of arity $n \geq 0$.

■

This assumption does not restrict expressiveness and it is straightforward how to transform a program to fit this requirement.

Example 5.7 (flat without append)

Consider the following version of the flat predicate:

$$\begin{aligned}
 & flat([], [], 0). \\
 & flat([[H|T]|T'], [H|R], N) \leftarrow flat([T|T'], R, N). \\
 & flat([[[]|T], R, s(N)) \leftarrow flat(T, R, N).
 \end{aligned}$$

After normalising, it looks as follows:

$$\begin{aligned}
flat(A, B, C) &\leftarrow \begin{aligned} A &= [], \\ B &= [], \\ C &= []. \end{aligned} \\
flat(D, E, F) &\leftarrow \begin{aligned} D &= [G|H], \\ G &= [I|L], \\ E &= [I|M], \\ N &= [L|H], \\ flat(N, M, F). \end{aligned} \\
flat(O, P, Q) &\leftarrow \begin{aligned} O &= [R|S], \\ R &= [], \\ Q &= s(N), \\ flat(S, P, T). \end{aligned}
\end{aligned}$$

■

The following notion will play an important role in establishing the well-behavedness of annotated programs. In the sequel, let $\bar{\tau}$ stand for a tuple of Typelabels (τ_1, \dots, τ_n) of rigid types (T_1, \dots, T_n) .

Definition 5.8 (well-annotated atom wrt a tuple of types)

An atom A is well-annotated wrt a tuple of types $\bar{\tau}$ if either

- A is a user defined atom $p(X_1, \dots, X_k)$, with X_1, \dots, X_k distinct variables and $\bar{\tau}$ has dimension k ($\bar{\tau} = (\tau_1, \dots, \tau_k)$), or
- A is of the form $X = Y$, with X and Y distinct variables and $\bar{\tau}$ has dimension 2, or
- A is of the form $X = f(Y_1, \dots, Y_k)$ with X, Y_1, \dots, Y_k distinct variables, $\bar{\tau}$ has dimension $k + 1$, say $\bar{\tau} = (\tau_0, \tau_1, \dots, \tau_k)$, and, given m , a node with $TypeLabel(m) = \tau_0$, then there exists a node $n \in Principal(m)$, such that $Label(n) = f/k$.

■

Throughout the remainder, we frequently use *well-annotated atom* $A^{\bar{\tau}}$ as a shorthand for ' A is a well-annotated atom wrt $\bar{\tau}$ '.

We now introduce the following definition:

Definition 5.9 (well-annotated clause)

Given a normal clause $C = A_0 \leftarrow A_1, \dots, A_n$. A clause

$$\bar{\tau}_0^i A_0^{\bar{\tau}_0^o} \leftarrow A_1^{\bar{\tau}_1^o}, \dots, A_n^{\bar{\tau}_n^o}$$

is *well-annotated wrt* C if the atoms A_0 and A_1, \dots, A_n are well-annotated wrt $\bar{\tau}_0^i$ and $\bar{\tau}_0^o, \dots, \bar{\tau}_n^o$. ■

This definition forms the basis to define well-annotated programs.

Definition 5.10 (well-annotated program)

Let P be a definite logic program. A well-annotated program P^* associated to P is a set of well-annotated clauses, one-to-one associated to the clauses of P , such that for every predicate symbol p/k (different from $=/2$) in P , letting $\bar{\tau}_{0_1}^i A_{0_1}^{\bar{\tau}_{0_1}^i}, \dots, \bar{\tau}_{0_n}^i A_{0_n}^{\bar{\tau}_{0_n}^i}, A_{j_1}^{\bar{\tau}_{j_1}^o}, \dots, A_{j_m}^{\bar{\tau}_{j_m}^o}$ be all occurrences of labelled atoms with predicate symbol p/k in P^* , then:

- $\bar{\tau}_{0_1}^o = \dots = \bar{\tau}_{0_n}^o = \bar{\tau}_{j_1}^o = \dots = \bar{\tau}_{j_m}^o$
- $\bar{\tau}_{0_1}^i = \dots = \bar{\tau}_{0_n}^i$

■

Example 5.11 (flat*)

Reconsider the *flat* program of Example 5.7: The following program *flat** is a well-annotated program associated to *flat*. We refer to Figure 3 and Figure 5 for a definition of the types.

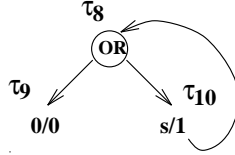


Figure 5: The natural numbers in successor notation

$$\begin{aligned}
 \tau_1, \text{Max,Max flat}(A, B, C)^{\tau_1, \tau_4, \tau_8} &\leftarrow \begin{aligned} A &= \text{nil}^{\tau_2}, \\ B &= \text{nil}^{\tau_5}, \\ C &= 0^{\tau_9}. \end{aligned} \\
 \tau_1, \text{Max,Max flat}(D, E, F)^{\tau_1, \tau_4, \tau_8} &\leftarrow \begin{aligned} D &= [G|H]^{\tau_3, \tau_6, \tau_1} \\ G &= [I|L]^{\tau_6, \tau_7, \tau_4}, \\ E &= [I|M]^{\tau_6, \tau_7, \tau_4}, \\ N &= [L|H]^{\tau_3, \tau_4, \tau_1}, \\ &\text{flat}(N, M, F)^{\tau_1, \tau_4, \tau_8} \end{aligned} \\
 \tau_1, \text{Max,Max flat}(O, P, Q)^{\tau_1, \tau_4, \tau_8} &\leftarrow \begin{aligned} O &= [R|S]^{\tau_3, \tau_4, \tau_1} \\ R &= \text{nil}^{\tau_2}, \\ Q &= s(T)^{\tau_{10}, \tau_8} \\ &\text{flat}(S, P, T)^{\tau_1, \tau_4, \tau_8} \end{aligned}
 \end{aligned}$$

■

Definition 5.12 (input (output) type of a predicate)

Let P^* be a well-annotated program associated to a logic program P and p/k a user defined predicate symbol occurring in P . If P^* contains a clause $\bar{\tau}_0^i A^{\bar{\tau}_0^i} \leftarrow A_1^{\bar{\tau}_1}, \dots, A_n^{\bar{\tau}_n}$, such that A has predicate symbol p/k , then $\bar{\tau}_0^i$ is called the *input type* of p/k , $in(p/k)$, $\bar{\tau}_0^o$ is its *output type*, $out(p/k)$. If P^* does not contain such a clause, then the *output type* of p/k is $\bar{\tau}_j^o$, where $\bar{\tau}_j^o$ is the annotation of some annotated body atom $A_j^{\bar{\tau}_j^o}$ occurring in P^* , with the predicate symbol of A_j being p/k . ■

Remark that for well-annotated programs, these types are unique for each predicate. In what follows, whenever we refer to the input or output type of a user defined atom A ($in(A)$ or $out(A)$), we mean the input or output type of its predicate symbol.

Example 5.13

Reconsider the well-annotated program $flat^*$ of Example 5.11. The input type of $flat$ is (τ_1, Max, Max) , while (τ_1, τ_4, τ_8) is its output type. ■

Next, we refine the notion of well-annotated program to respect the type flow in the program. We introduce a notion of well-typed programs, similar to that in [13], but specialised for our purposes.

Definition 5.14 (well-typed atom)

- A user defined atom $p(t_1, \dots, t_k)$ is called *well-typed wrt a well-annotated atom* $p(X_1, \dots, X_k)^{(\tau_1, \dots, \tau_k)}$ if $t_i \in \mathcal{ID}(\tau_i)$, for $i = 1, \dots, k$.
- An atom $t_1 = t_2$ is called *well-typed wrt a well-annotated atom* $X = Y^{(\tau_1, \tau_2)}$ if $t_i \in \mathcal{ID}(\tau_i)$, for $i = 1, 2$.
- An atom $s = f(t_1, \dots, t_k)$ is called *well-typed wrt a well-annotated atom* $X = f(Y_1, \dots, Y_k)^{(\tau_0, \tau_1, \dots, \tau_k)}$ if $s \in \mathcal{ID}(\tau_0)$ and $t_i \in \mathcal{ID}(\tau_i)$, for $i = 1, \dots, k$.

■

Definition 5.15 (well-typed program)

Let P^* be a well-annotated program associated to a program P . P^* is *well-typed* if

- for each clause

$$\bar{\tau}_0^i A_0^{\bar{\tau}_0^o} \leftarrow A_1^{\bar{\tau}_1^o}, \dots, A_n^{\bar{\tau}_n^o}$$

in P^* , associated to a clause $C = A_0 \leftarrow A_1, \dots, A_n$ in P ,

- for each instance $A_0\rho_0$ of A_0 , such that $A_0\rho_0$ is well-typed wrt the annotated atom $A_0^{\bar{\tau}_0^o}$,
- denoting $\rho_i = \rho_{i-1}\sigma_i$, $i = 1, \dots, n$, where σ_i is a computed answer substitution for the goal $\leftarrow A_i\rho_{i-1}$

we have

- $A_i\rho_r$ is well-typed wrt the well-annotated atom $A_i^{\bar{\tau}_i^o}$, $\forall 1 \leq i < r$ if the clause is recursive and A_r is the (only) recursive atom (note that we allow only direct recursion),
- $A_i\rho_n$ is well-typed wrt the well-annotated atom $A_i^{\bar{\tau}_i^o}$, $\forall 1 \leq i \leq n$,
- $A_0\rho_n$ is well-typed wrt the annotated atom $A_0^{\bar{\tau}_0^o}$,
- for each A_i , $i = 1, \dots, n$ with a user defined predicate and such that $in(A_i)$ exists, $A_i\rho_{i-1}$ is well-typed wrt the annotated atom $A_i^{in(A_i)}$.

■

The first condition is the crucial one. It states that the types adorning all atoms left of a recursive call should capture all successful instantiations of these atoms at the point immediately before the recursive call. The second condition expresses the fact that whenever all atoms in the body succeed, their arguments satisfy their output typing. At first sight, condition (a) and (b) might seem to impose incompatible constraints on the annotations of the atoms to the left of a recursive call. Note however that, because of the closedness of our rigid types, any atom which satisfies condition (a) also satisfies the second condition. The third condition says that if all atoms in the body of the clause succeed, then the success substitution satisfies the output typing of the head of the clause. The final condition guarantees that all calls to body atoms are well-typed, meaning that the arguments satisfy the input typing of the atom. In the following, we will denote such programs by a superscript t , i.e. P^t stands for a well-typed program associated to a program P .

The following proposition will be useful.

Proposition 5.16

Let $\bar{\tau}_0^i A_0^{\bar{\tau}_0^i} \leftarrow A_1^{\bar{\tau}_1^i}, \dots, A_n^{\bar{\tau}_n^i}$ be a well-typed clause associated with a clause C and $\rho_i, 0 \leq i \leq n$, the substitutions defined in Definition 5.15, then:

- a. $A_0\rho_n$ is in the success set of $\leftarrow A_0\rho_0$.
- b. $A_i\rho_i$ is in the success set of $\leftarrow A_i\rho_{i-1}, \forall i, 1 \leq i \leq n$.

■

Proof

The results directly follow from the definition of $\rho_i, 0 \leq i \leq n$.

■

Example 5.17 (flat)

Reconsider the *flat* program introduced in Example 5.7 and assume *flat/3* to be queried by *flat*(τ_1, Max, Max). The type inference mechanism of [29] returns the following call/success information when provided with the above type description:

$$\mathbf{flat}(\tau_1, Max, Max) \rightarrow \mathbf{flat}(\tau_1, \tau_4, \tau_8)$$

We refer to Figure 3 and Figure 5 for a definition of the types. The program *flat* ^{t} as defined in Example 5.11 is a well-typed version of *flat*.

■

Such programs will be useful to derive typed interargument relations. But first, due to the explicit typing of unification and atoms, their semantics must be redefined.

5.3 Semantics of well-typed programs

We now define the semantics of these well-typed programs and analyse their relationship with traditional semantics. In the following, when we refer to the (extended) Herbrand Base or an (extended) Herbrand Interpretation of a well-typed program P^t , we mean the corresponding concept for P . We will assume that these interpretations all include the atoms $\{ t = t \mid t \in U_P^E \}$ (as required for Herbrand interpretations).

We introduce the following notation:

Definition 5.18

Let I be an extended Herbrand interpretation of a well-typed program P^t , $A^{\bar{\tau}}$ a well-annotated atom and σ a substitution:

We denote:

$$\begin{aligned} I \models_{\sigma} A^{\bar{\tau}} \\ \text{iff} \\ A\sigma \in I \text{ and } A\sigma \text{ is well-typed wrt } A^{\bar{\tau}}. \end{aligned}$$

■

Other model-theoretic concepts, such as the semantics and truth of well-annotated clauses etc., can be taken over with no further adaptations. We don't recall their definitions but we refer instead to [32] for a more fundamental treatment on these semantics.

The next step is to define an operator mimicking the immediate consequence operator:

Definition 5.19 ($T_P^*(I)$)

Let P be a definite logic program and let P^t be a well-typed program associated to P . Let I be an extended Herbrand interpretation of P^t

$$\begin{aligned} T_P^*(I) = \{ & A_0\sigma \mid \bar{\tau}_i A_0^{\bar{\tau}_0} \leftarrow A_1^{\bar{\tau}_1}, \dots, A_n^{\bar{\tau}_n} \text{ is a clause in } P^t, \\ & \sigma \text{ is a substitution,} \\ & \forall i, 1 \leq i \leq n, I \models_{\sigma} A_i^{\bar{\tau}_i} \text{ and} \\ & A_0\sigma \text{ is well-typed wrt } A_0^{\bar{\tau}_0} \} \\ \cup \\ \{ & t = t \mid t \in U_P^E \} \end{aligned}$$

■

Proposition 5.20

$T_P^*(I)$ is monotonic and finitary.

■

Proof

1. $T_P^*(I)$ is monotonic.

Take any two Herbrand interpretations $I \subseteq J$ and suppose that $T_P^*(I) \not\subseteq T_P^*(J)$. Then there exists an atom $A_0\sigma \in T_P^*(I)$ and $A_0\sigma \notin T_P^*(J)$. Thus, there exists a clause $\bar{\tau}_i A_0^{\bar{\tau}_0} \leftarrow A_1^{\bar{\tau}_1}, \dots, A_n^{\bar{\tau}_n}$ in P^t such that $\forall i, I \models_{\sigma} A_i^{\bar{\tau}_i}$ and $\exists j \in [1, \dots, n], J \not\models_{\sigma} A_j^{\bar{\tau}_j}$. Now A_j can not be user-defined as $I \subseteq J$. Thus A_j is an explicit unification. In this case however, the semantics do not depend on I and J but on the substitution σ , which in both cases is the same.

2. $T_P^*(I)$ is finitary.

Take any infinite sequence of Herbrand interpretations $I_0 \subseteq I_1 \subseteq I_2 \subseteq \dots$ and suppose that $A_0\sigma \in T_P^*(\bigcup_{n=0}^{\infty} I_n)$. Then there must exist a clause $\bar{\tau}_i A_0^{\bar{\tau}_0} \leftarrow A_1^{\bar{\tau}_1}, \dots, A_n^{\bar{\tau}_n}$ in P^t such that $\forall i, \bigcup_{n=0}^{\infty} I_n \models_{\sigma} A_i^{\bar{\tau}_i}$, thus $\forall i, A_i^{\bar{\tau}_i} \in \bigcup_{n=0}^{\infty} I_n$. This implies however that for some $i_n, I_{i_n} \models_{\sigma} A_i^{\bar{\tau}_i}, \forall i$ or that $A_0\sigma \in T_P^*(I_{i_n}) \subseteq \bigcup_{n=0}^{\infty} T_P^*(I_n)$.

■

Corollary 5.21

$T_P^*(I)$ is continuous. ■

Definition 5.22 (T_P^* semantics)

We define the model $\mathcal{M}(P^t)$ of a well-typed program P^t associated to some program P as follows:

$$\mathcal{M}(P^t) = T_P^* \uparrow \omega.$$
■

Proposition 5.23

Let P be a program and P^t a well-typed version associated to P . Let A be an atom such that $in(A)$ and $out(A)$ exist. If A is well-typed wrt $A^{in(A)}$ and σ is a computed answer substitution for $\leftarrow A$ in P , then $\mathcal{M}(P^t) \models_\sigma A^{out(A)}$. ■

Proof

By induction on the number N of selected atoms (different from explicit unification) in the derivation of A .

- **Base case:** $N = 0$

There is a clause $A_0 \leftarrow$ in P such that $\sigma = mgu(A, A_0)$. In P^t there is an associated clause $\tau^i A_0^{\tau^o} \leftarrow$. Because there are no atoms in the body for that clause and because $\tau^i A_0^{\tau^o} \sigma$ is well-typed wrt $A_0^{\tau^o}$ (because of Definition 5.15, condition c), $T_P^*(\emptyset) \models_\sigma A_0^{\tau^o}$.

- **Induction step:** $N \geq 1$

Let $A_0 \leftarrow A_1, \dots, A_n$ be the clause in P which has been used to resolve $\leftarrow A$. Then, $\sigma = \theta_c \theta_s$, with $\theta_c = mgu(A, A_0)$ and $P \models (A_1 \wedge \dots \wedge A_n) \theta_c \theta_s$. Due to the construction, P^t contains an associated clause $\bar{\tau}^i A_0^{\bar{\tau}^o} \leftarrow A_1^{\bar{\tau}^i}, \dots, A_n^{\bar{\tau}^i}$. Let σ be the same substitution as above. Then $T_P^*(T_P^* \uparrow (N - 1)) \models_\sigma A_0^{\bar{\tau}^o}$ because

1. $A_0 \sigma$ is well-typed wrt $A_0^{\bar{\tau}^o}$.

This is because of Definition 5.15,c.

2. $T_P^* \uparrow (N - 1) \models_\sigma A_i^{\bar{\tau}^i}, \forall i, 1 \leq i \leq n$.

If $A_i \sigma$ is an explicit unification, this holds because P is normalised, and because of the Definition of \models_σ and Definition 5.15,b.

If $A_i \sigma$ is a user defined atom, the induction hypothesis applies because all $A_i \sigma$ are well-typed wrt $A_i^{in(A_i)}$ (due to Definition 5.15,d).

Because $A_0 \theta_c = A \theta_c$ and $\bar{\tau}^o = out(A) : T_P^*(T_P^* \uparrow (N - 1)) \models_\sigma A_0^{\bar{\tau}^o}$. ■

5.4 Automatic generation of well-typed programs

Well-typed programs can be automatically constructed from a normalised program. Typically, three different kinds of information must be available: global call information, global success information and, so to say, *local success information* (which, in our case,

captures the types just before a recursive call is activated), all expressed in terms of rigid type patterns. As an example, we refer to [29], which proposes a framework for deriving call/success information for all predicates wrt an initial call description. The outcome is given in the form of one tuple of rigid call/success types per predicate. This means that for each predicate p/k which is defined in P , a call pattern $p(T_1^c, \dots, T_k^c)$ and a success pattern $p(T_1^s, \dots, T_k^s)$ is derived. We denote the tuple (T_1^c, \dots, T_k^c) by $in(p/k)$ and (T_1^s, \dots, T_k^s) by $out(p/k)$. We now define what we understand by *local success information*. The concept is defined in terms of a set of atoms S , which can be seen as a set of possible calls.

Definition 5.24 (local success pattern of an atom wrt a set of atoms)

Let $C \equiv p(U_1, \dots, U_k) \leftarrow A_1, \dots, A_i, \dots, A_n$ be a clause in a normalised program P and let $Vars(C) = (X_1, \dots, X_m)$. Let S be a set of atoms.

$Loc_S(C, A_i) = \{X_1 \leftarrow T_1, \dots, X_m \leftarrow T_m\}$ is a *local success pattern wrt A_i and S in C* if $\forall p(U_1, \dots, U_k)\theta_c \in S, \forall \theta_s : (P \models (A_1 \wedge \dots \wedge A_i)\theta_c \theta_s \Rightarrow X_j \theta_c \theta_s \in \mathcal{D}(T_j), \forall 1 \leq j \leq m)$. ■

The framework described in [29] is able to derive such local success patterns for any possible atom. Here, local reflects the fact that also for all local variables which do not occur in the atom, instantiation information is available.

Algorithm 5.25 (construction of a well-typed program)

Let P be a normalised logic program and let S be the call set $Call(P, S_q)$ for some set of queries P of interest. For each predicate p/k in P , let there be available a call pattern $p(T_1^c, \dots, T_k^c)$ wrt S and a success pattern $p(T_1^s, \dots, T_k^s)$ wrt S . Assume also that for each clause $C = A_0 \leftarrow A_1, \dots, A_n : Loc_S(C, A_n)$ is available and, if the clause is recursive, with recursive atom $A_i, Loc_S(C, A_n)$ is computed.

Let $P^t = \emptyset$. For each clause $C \equiv p(U_1, \dots, U_k) \leftarrow A_1, \dots, A_i, \dots, A_n$ in P , we add a clause $in(p/k)p(U_1, \dots, U_k)out(p/k) \leftarrow A_1^{\bar{\tau}_1}, \dots, A_i^{\bar{\tau}_i}, \dots, A_n^{\bar{\tau}_n}$ to P^t as follows:

```

if       $\exists 1 \leq i \leq n, Pred(A_i) = p/k$ 
then     $r \leftarrow i - 1$ 
else     $r \leftarrow 0$ 

```

```

 $j \leftarrow 1$ 
while  $j \leq n$ 
do

```

case:

- $A_j \equiv q(U_1, \dots, U_m), q/m \neq /2$
 $\bar{\tau}_j = out(p/k)$
- $A_j \equiv X = Y$

```

if       $j < r$ 
then     $\bar{\tau}_j = (\tau_x, \tau_y)$  where  $X \leftarrow \tau_x \in Loc_S(C, A_{i-1})$  and  $Y \leftarrow \tau_y \in Loc_S(C, A_{i-1})$ 
else     $\bar{\tau}_j = (\tau_x, \tau_y)$  where  $X \leftarrow \tau_x \in Loc_S(C, A_n)$  and  $Y \leftarrow \tau_y \in Loc_S(C, A_n)$ 

```

- $A_j \equiv X_0 = f(X_1, \dots, X_m)$

if $j < r$
then $\bar{\tau}_j = (\tau_0, \dots, \tau_m)$ where $X_k \leftarrow \tau_k \in Loc_S(C, A_{i-1}), 1 \leq k \leq m$
else $\bar{\tau}_j = (\tau_0, \dots, \tau_m)$ where $X_k \leftarrow \tau_k \in Loc_S(C, A_n), 1 \leq k \leq m$

$j \leftarrow j + 1$ ■

Given an initial call pattern $p(\tau_1, \dots, \tau_k)$, the framework of [29] is able to derive one call pattern and one success pattern for each predicate, wrt $S = \mathcal{ID}(p(\tau_1, \dots, \tau_k))$. Also, for each clause $C = A_0 \leftarrow A_1, \dots, A_n$, the framework can derive a local success pattern $Loc_S(C, A_n)$ and, if A_i is a recursive atom, $Loc_S(C, A_{i-1})$ can be computed. Using this information, the following proposition holds.

Proposition 5.26

If P^t is a program obtained from a normalised program P by using the above algorithm, then P^t is well-typed wrt P . ■

Proof

(a) is immediate because of construction. (b) is immediate for all atoms $A_l, l > r$. For $l < r$, this holds because rigid types are closed under substitution. (c) holds because the head is labelled with a global success pattern of the predicate and (d) holds because each atom with a user-defined predicate p/k is labelled with a global call pattern $p(\tau_1, \dots, \tau_n)$, i.e. $\forall p(t_1, \dots, t_n) \in Call(P, S), p(t_1, \dots, t_n) \in \mathcal{ID}(p(\tau_1, \dots, \tau_n))$. ■

5.5 Matching relevant types

Let P denote the definite logic program for which termination must be examined wrt some call pattern $p(\tau_1, \dots, \tau_n)$. Additionally suppose that a type analysis along the lines of [29] has provided one global call- and success description for each predicate in P . Let P^t be a well-typed program associated to P constructed along the lines of the previous sections wrt this call- and success-information.

We will postpone the problem of deriving typed interargument relations until section 6. Instead, assume that there exists some way of deriving a correct typed sizerelation for each predicate in the program. As can be expected when dealing with success information, the relation must hold between the sizes of the arguments where each argument is measured under its global success type. What concerns explicit unification of the form $X_0 = f(Y_1, \dots, Y_n)$ or $X = Y$, we know that after the unification both terms become identical. Thus a trivially correct relation is $\|X\sigma\theta\|_\tau = \|f(Y_1, \dots, Y_n)\sigma\theta\|_\tau, \forall \theta, \sigma = mgu(X, f(Y_1, \dots, Y_n))$ and where the local success type of X in the unification is τ (the one used to label the lefthandside of the unification in P^t). Note that this relation is valid for any other typed norm $\|\cdot\|_{\tau'}$. However, $\|\cdot\|_\tau$ is the obvious choice because it is a success type.

In this section, attention will be directed towards the identification of all norms, and thus of all types, that will have to become matched. As explained earlier, such a matching is necessary to obtain a consistent termination proof, more in particular, to retain the well-foundedness of the ordering requested in Definition 2.6.

First we define the type of a variable in a well-annotated atom.

Definition 5.27 (type of a variable)

Let $A^{(\tau_1, \dots, \tau_k)}$ be a well-annotated atom and let (X_1, \dots, X_k) be the bag of variables occurring in A . Then we say that X_i is typed τ_i in $A^{(\tau_1, \dots, \tau_k)}$ ■

To be able to make explicit the flow of information among types, we assemble all variables of the program into one global set and we couple them with their respective types. Note that one variable may occur multiple times with each of the times being differently typed. Off course, we assume the program being standardised apart.

Definition 5.28 (set of typed variables $TV(P^t)$)

Let us denote by $TV(P^t)$ the set of typed occurrences of variables in P^t , i.e. $TV(P^t)$ is the set of couples (X, τ) such that there exists a clause $\bar{\tau}_0^i A_0^{\bar{\tau}_0^o} \leftarrow A_1^{\bar{\tau}_1^o}, \dots, A_n^{\bar{\tau}_n^o}$, with a recursive atom $A_j^{\bar{\tau}_j^o}, 1 \leq j \leq n$, such that

1. X is typed τ in some atom $A_i^{\bar{\tau}_i^o}, 1 \leq i < j$ wrt $\bar{\tau}_i^o$, or
2. X is typed τ in A_j wrt $\bar{\tau}_0^i$, or
3. X is typed τ in A_0 wrt $\bar{\tau}_0^i$.

■

The set $TV(P^t)$ forms the basis for identifying relevant norms: norms on which a matching condition must be imposed. It provides an answer to the question of which are the different types (norms) under which one and the same term may be measured. Essentially, there are three possible scenarios of how one term could become measured differently. A first one is the double occurrence of a variable being differently typed on both places. Two other sources of such a relationship between types is provided by the explicit unifications, more specifically by the way it has been decided to abstract such unification. The types composed by the left- and righthandside of the unification must be measured consistently. Due to the choice of norm under which to measure these terms, inconsistencies may occur. The matching set, which we will introduce immediately, gathers all information of this kind. This concept forms the key relation behind the approach of this section, as it identifies all couples of types whose derived norms will have to become matching.

Definition 5.29 (matching set $Match(P^t)$)

$(\tau_1, \tau_2) \in Match(P^t)$ iff

1. $(X, \tau_1) \in TV(P^t), (X, \tau_2) \in TV(P^t)$ and $\tau_1 \neq \tau_2$, or
2. $X = Y$ is in a clause of P , $(X, \tau_1) \in TV(P^t), (Y, \tau_2) \in TV(P^t)$ and $\tau_1 \neq \tau_2$, or
3. $X = f(Y_1, \dots, Y_n)$ is in a clause of P , X is typed τ_X , $\tau_X = TypeLabel(m)$, Y_i is typed τ_2 , $n \in Principal(m)$, $Label(n) = f/k$, $ArgPos(n, i) = n'$, $TypeLabel(n') = \tau_1$ and $\tau_1 \neq \tau_2$.

■

Example 5.30 (flat)

The matching set for the flat example (Example 5.11) is the following:

$$\text{Match}(\text{flat}^T) = \{(\tau_1, \tau_3), (\tau_3, \tau_1), (\tau_4, \tau_6), (\tau_3, \tau_2), (\tau_8, \tau_{10})\}$$

As the first clause is not recursive, it does not contribute to the match set. The first three entries arise from the second clause, while the third clause provides the last two couples. The couple (τ_1, τ_3) e.g. originates from the double occurrence (and its double typing) of the variable D in the second clause. An interesting case is the third couple (τ_4, τ_6) . The unification $D = [G|H]^{\tau_3, \tau_6, \tau_1}$ is responsible for this entry. Here, the variable D is typed with τ_3 whilst H is typed with τ_1 in this atom and further in the clause. If the norm induced by τ_3 should be used for unfolding the term $[G|H]$, the variable H would be assigned to the τ_4 norm. The couple reflects this behaviour, that one term could be assigned two types. ■

The coefficient and offset assignment which have been imported in the definition of typed norm now enable us to fine-tune our norms wrt the matching conditions implicitly stated by the match set. The following definition presents a set of conditions on the assignments whose satisfaction implies the matching property.

Definition 5.31 (system of matching conditions for two typed norms)

Let τ_1 and τ_2 be the TypeLabels of two labelled type graphs. A set of matching conditions for τ_1 and τ_2 is a system of boolean equations $\text{match}(\tau_1, \tau_2)$ over the values of the coefficient and offset assignments a_1, b_1, a_2, b_2 of τ_1 and τ_2 such that for every a_1, b_1, a_2, b_2 that are solutions of $\text{match}(\tau_1, \tau_2)$, the norms $\|\cdot\|_{\tau_1, a_1, b_1}$ and $\|\cdot\|_{\tau_2, a_2, b_2}$ are matching. ■

Example 5.32

Consider the types pictured in Figure 6. Let n_1 be the node corresponding to Typelabel τ_3 and n_2 be the one corresponding to τ_7 . A possible set of matching conditions $\text{match}(\tau_1, \tau_5)$ is the set of equations $\{b_1(n_1) = b_2(n_2), a_2(n_2, 1) = 0, a_2(n_2, 2) = a_1(n_1, 2)\}$. The set $\{b_1(n_1) = b_2(n_2), a_1(n_1, 2) = a_2(n_2, 2), b_2(n_3) = 0, a_2(n_3, 1) = 0, a_2(n_3, 2) = 0\}$ is another set of matching conditions for these types, where n_3 is the node with the TypeLabel τ_{10} . ■

Such a system of equations is needed for each couple (τ_i, τ_j) in $\text{Match}(P^t)$. The union of the systems provides a system that imposes all matches simultaneously.

Definition 5.33 (matching conditions)

Let P^t be a well-typed program associated to a program P , $\text{Match}(P^t)$ be its Match set, and let $\text{match}/2$ be a function accepting two labelled types and returning a system of matching equations for these. The *set of matching conditions* associated to $\text{Match}(P^t)$ is the system

$$\mathcal{MC}(P^t) = \cup_{(\tau_1, \tau_2) \in \text{Match}(P^t)} \text{match}(\tau_1, \tau_2)$$

■

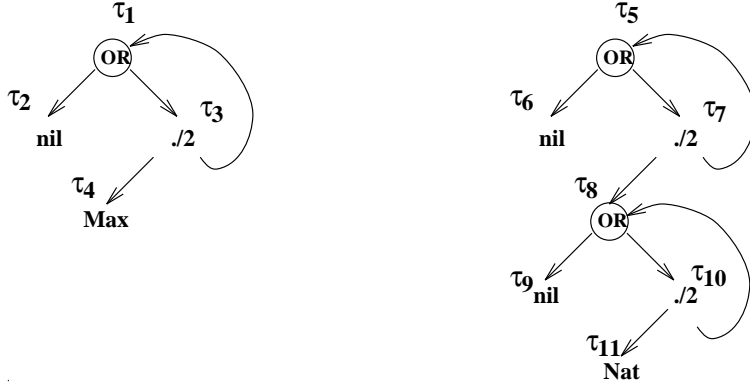


Figure 6: There are several alternatives to match these types.

5.6 A matching algorithm

In the following we report an algorithm to calculate a set of matching conditions for the typed norms induced by two TypeLabels τ_1 and τ_2 . The function compares the nodes of the typographs associated to the typed norms, starting from the root. In analogy to the definition of Typed Norm, appropriate actions are taken depending on the kind of nodes and new conditions are formulated in terms of the descendants of these nodes.

The key step in the algorithm is the case in which one node is a Max node. To enforce the matching and to retain rigidity, the norm induced by the non-Max type must be made the constant zero function. Note that this is the step which reduces the precision of Typed Norms. Another important case is when both nodes are functor nodes. However, this step has no direct impact on the precision.

The algorithm itself makes use of three different sets. S^{sc} is the set of *selected couples*. It contains couples of nodes which have been compared already. S^{uc} is the set of *unselected couples*. This set is some kind of a queue, where couples are waiting to be matched. E_q is the set of equations which are generated so far.

Algorithm 5.34 ($Eq = Match(\tau_1, \tau_2)$)

Let T_1 and T_2 be two type graphs with roots n_1 and n_2 and let $TypeLabel(n_1) = \tau_1$ and $TypeLabel(n_2) = \tau_2$.

Initialisation

$$S^{uc} = \{(n_1, n_2)\}$$

$$S^{sc} = \emptyset$$

$$E_q = \emptyset$$

repeat

Select $(m_1, m_2) \in S^{uc}$.

$$\text{if } m_1 = m_2 \vee (m_1, m_2) \in S^{sc} \vee \mathcal{ID}(m_1) \cap \mathcal{ID}(m_2) = \emptyset \quad (1)$$

$$\text{then } S^{uc} = S^{uc} \setminus \{(m_1, m_2)\}$$

$$S^{sc} = S^{sc} \cup \{(m_1, m_2)\}$$

$$\text{else if } Label(m_1) = OR \vee Label(m_2) = OR \quad (2)$$

$$\begin{aligned}
& \text{then } S^{uc} = S^{uc} \setminus \{(m_1, m_2)\} \cup \{(q_1, q_2) \mid q_1 \in \text{Principal}(m_1), q_2 \in \text{Principal}(m_2)\} \\
& \quad S^{sc} = S^{sc} \cup \{(m_1, m_2)\} \\
& \text{else if } \text{Label}(m_1) = \text{Max} \wedge \text{Label}(m_2) = f/k \tag{3} \\
& \text{then } S^{uc} = S^{uc} \setminus \{(m_1, m_2)\} \\
& \quad S^{sc} = S^{sc} \cup \{(m_1, m_2)\} \\
& \quad \text{Eq} = \text{Eq} \cup \{b_2(m_2) = 0\} \cup \{a_2(m_2, i) = 0 \mid 1 \leq i \leq k\} \\
& \text{else if } \text{Label}(m_2) = \text{Max} \wedge \text{Label}(m_1) = f/k \tag{4} \\
& \text{then } S^{uc} = S^{uc} \setminus \{(m_1, m_2)\} \\
& \quad S^{sc} = S^{sc} \cup \{(m_1, m_2)\} \\
& \quad \text{Eq} = \text{Eq} \cup \{b_1(m_1) = 0\} \cup \{a_1(m_1, i) = 0 \mid 1 \leq i \leq k\} \\
& \text{else if } \text{Label}(m_1) = \text{Label}(m_2) = f/k \tag{5} \\
& \text{then } S^{uc} = S^{uc} \setminus \{(m_1, m_2)\} \cup \{(m_1/i, m_2/i) \mid 1 \leq i \leq k\} \\
& \quad S^{sc} = S^{sc} \cup \{(m_1, m_2)\} \\
& \quad \text{Eq} = \text{Eq} \cup \{b_1(m_1) = b_2(m_2)\} \cup \{a_1(m_1, i) = a_2(m_2, i) \mid 1 \leq i \leq k\} \\
& \text{else } S^{sc} = S^{sc} \cup \{(m_1, m_2)\} \tag{6} \\
& \quad S^{uc} = S^{uc} \setminus \{(m_1, m_2)\}
\end{aligned}$$

until $S^{uc} = \emptyset$ ■

The last case in the algorithm is necessary to handle in a correct way those types which are constants or primitive.

Example 5.35

If we apply this algorithm to the type graphs represented in Figure 6, the set of equations $\{b_1(n_1) = b_2(n_2), a_1(n_1, 1) = a_2(n_2, 1), a_1(n_1, 2) = a_2(n_2, 2), b_2(n_3) = 0, a_2(n_3, 1) = 0, a_2(n_3, 2) = 0\}$ is obtained, where n_1 is the node with *TypeLabel* τ_1 , n_2 has *TypeLabel* τ_5 and n_3 is the functor node with *TypeLabel* τ_{10} . ■

Proposition 5.36

Let n_1 and n_2 be the roots of two typegraphs and let τ_1 and τ_2 be their *TypeLabel*. Then $\text{Match}(n_1, n_2)$ is a set of matching conditions for the typed norms associated to τ_1 and τ_2 . ■

Proof

termination of the algorithm

It is easy to see that the algorithm terminates as there are only a finite number of distinct couples of nodes $(n_1, n_2) \in N_1 \times N_2$. Each time an element of S^{uc} is selected, it becomes added to S^{sc} . Because no element is ever removed from S^{sc} , step 1 assures that either $S^{uc} = \emptyset$ at some time or $S^{sc} = N_1 \times N_2$.

the matching

We will prove a stronger claim. Let a_1, b_1 and a_2, b_2 be coefficient and offset assignments on n_1 and n_2 which satisfy $\text{Match}(n_1, n_2)$. Then: $\forall (m_1, m_2) \in S^{sc}, \forall t \in \mathcal{ID}(m_1) \cap \mathcal{ID}(m_2) : \|t\|_{\tau_1, a_1, b_1} = \|t\|_{\tau_2, a_2, b_2}$, where S^{sc} denotes the *final* S^{sc} set, upon termination of the algorithm.

Because initially S^{uc} only contains the tuple (n_1, n_2) , this element is selected immediately. It is easy to see that every tuple which becomes selected in S^{uc} , is added to S^{sc} , which implies our claim.

The following observation will simplify the proof.

If $t = f(t_1, \dots, t_n)$ and $t \in \mathcal{ID}(n)$ with $Label(n) = OR$, then there exists one and only one $m \in Principal(n)$ such that $t \in \mathcal{ID}(m)$. Moreover, $Label(m) = f/k$.

Because of the definition of typed norm, for such a type the following claim is immediate:

$$||t||_{\tau_n, a, b} = ||t||_{\tau_m, a, b}$$

where $\tau_n = TypeLabel(n)$ and $\tau_m = TypeLabel(m)$.

Let $t \in \mathcal{ID}(m_1) \cap \mathcal{ID}(m_2)$. The proof proceeds by structural induction on t .

Base cases

We distinguish between the following cases:

- $t = X$.
Both m_1 and m_2 must be Max nodes, because variables can only be denoted by Max types. The normality condition imposes that no Max nodes can be principal nodes of an OR-node, such that we can exclude these cases. The condition is trivially satisfied as the norm induced by a Max node is trivially zero and thus does not depend on the assignments.
- $t \in ConstP$
Here we must consider the cases in which the nodes are constants or primitive. Also one or both nodes can be an OR-node.
In the first case, the associated norm is defined to be the constant zero function and thus does not depend on the assignment.
In the case where one of the nodes is an OR-node, say m_1 , we know by normality that there is only one node $m \in Principal(m_1)$ with $TypeLabel(m) = \tau$ such that $t \in \mathcal{ID}(m)$. Because of the definition of Typed Norm and because $t \in \mathcal{ID}(\tau_1) \cap \mathcal{ID}(\tau_2)$, the norm $||t||_{\tau_1, a_1, a_2}$ reduces to $||t||_{\tau, a_1, a_2}$ which is zero. The same holds for the second node: either it is a constant node, in which case its associated norm is zero, or it is an OR-node, in which case its norm, using the same reasoning as above, reduces to the norm induced by a constant or a primitive type, which is again zero.

Inductive case: $t = f(t_1, \dots, t_n)$

Nine different situations are possible: $Label(m_1)$ can be any of Max , f/k or OR , as can be $Label(m_2)$. We drop all symmetric cases which results in still six cases to consider: $(Label(n_1), Label(n_2)) \in \{(OR, OR), (OR, Max), (OR, f/k), (Max, f/k), (Max, Max), (f/k, f/k)\}$.

Because of the second case of the algorithm and because of the observation we made at the beginning of the proof, we can discard the first three other cases, as they reduce to the following three remaining ones (case 2 of the algorithm assures us that these couples are all added to S^{uc} and thus eventually to S^{sc}),.

- $Label(m_1) = Max, Label(m_2) = f/k$
This tuple can only be in S^{sc} because it has been selected from S^{uc} . Because of the third case, the constraints $\{b_2(m_2) = 0\} \cup \{a_2(m_2, i) = 0 | 1 \leq i \leq k\}$ are part of E_q . Because of the definition of Typed Norm, $\|\cdot\|_{m_2, a_2, b_2}$, with $TypeLabel(n_2) = \tau$, reduces to the constant zero function, which is also the norm induced by the Max node.
- $Label(m_1) = Max, Label(m_2) = Max$
This is trivial as the typed norm induced by a Max node is independent of the assignments and is defined as the constant zero function.
- $Label(m_1) = f/k, Label(m_2) = f/k$
Because of the selection of this tuple, corresponding to case 5 of the algorithm, the set E_q contains at least the equations $\{b_1(m_1) = b_2(m_2), a_1(m_1, 1) = a_2(m_2, 1), \dots, a_1(m_1, k) = a_2(m_2, k)\}$. Because of the definition of typed norm, it remains to prove that $\forall i, \|t_i\|_{m_1/i, a_1, b_1} = \|t_i\|_{m_2/i, a_2, b_2}$, where $\tau_1/i = TypeLabel(n_1/i)$ and $\tau_2/i = TypeLabel(n_2/i)$. Because of the same case 5 of the algorithm, all tuples $(m_1/i, m_2/i)$ have been added to S^{uc} and thus are eventually selected and added to S^{sc} . We can thus apply our induction hypothesis on these tuples: $\|t_i\|_{m_1/i, a_1, b_1} = \|t_i\|_{m_2/i, a_2, b_2}$.

■

Although the above algorithm generates correct matching sets, an obvious bottleneck (from an efficiency point of view) is the inclusion of the empty denotation test in the first step. This is a very inefficient test, as at each step two typegraphs are topologically compared against each other. The observation that always the same parts of the same graphs are compared, leads to the following efficient, though restricted algorithm. When provided with two type graphs, it computes a set of couples of nodes whose denotation is not empty. It will turn out that these couples are exactly the couples which we need.

Apart from the sets S^{uc} and S^{sc} , which are as in the matching algorithm, two other sets are used. *Ned* is the set of couples of nodes whose intersection is non-empty. *Dep* expresses how the intersection of the denotation of one couple of nodes depends on the denotation of other tuples of nodes. It contains elements of the form $dep((m_1, m_2), S)$, where S is a set of couples of nodes. Such an element captures the fact that m_1 and m_2 have a non-empty denotation if all couples in S have a non-empty denotation.

Algorithm 5.37 (efficient intersection of two types)

Initialisation

Dep $\leftarrow \emptyset$
Ned $\leftarrow \emptyset$
 $S^{uc} \leftarrow \{(n_1, n_2)\}$
 $S^{sc} \leftarrow \emptyset$

• Phase 1

repeat

Select $(m_1, m_2) \in S^{uc}$

if $(m_1, m_2) \in S^{sc}$ (1)
then $S^{uc} \leftarrow S^{uc} \setminus \{(m_1, m_2)\}$

else if $\text{Label}(m_1) = \text{Max} \vee \text{Label}(m_2) = \text{Max}$ (2)
then $\text{Ned} \leftarrow \text{Ned} \cup \{(m_1, m_2)\}$
 $S^{uc} \leftarrow S^{uc} \setminus \{(m_1, m_2)\}$
 $S^{sc} \leftarrow S^{sc} \cup \{(m_1, m_2)\}$

else if $\text{Label}(m_1) = \text{OR} \vee \text{Label}(m_2) = \text{OR}$ (3)
then $\text{Dep} \leftarrow \text{Dep} \cup \{\text{dep}((m_1, m_2), \{(r_1, r_2)\}) \mid r_1 \in \text{Principal}(m_1), r_2 \in \text{Principal}(m_2)\}$
 $S^{uc} \leftarrow S^{uc} \setminus \{(m_1, m_2)\} \cup \{(r_1, r_2) \mid r_1 \in \text{Principal}(m_1), r_2 \in \text{Principal}(m_2)\}$
 $S^{sc} \leftarrow S^{sc} \cup \{(m_1, m_2)\}$

else if $\text{Label}(m_1) = \text{Label}(m_2) = f/k$ (4)
then $\text{Dep} \leftarrow \text{Dep} \cup \{\text{dep}((m_1, m_2), \{(m_1/i, m_2/i) \mid 1 \leq i \leq k\})\}$
 $S^{uc} \leftarrow S^{uc} \setminus \{(m_1, m_2)\} \cup \{(m_1/i, m_2/i) \mid 1 \leq i \leq k\}$
 $S^{sc} \leftarrow S^{sc} \cup \{(m_1, m_2)\}$

else if $\text{Label}(m_1) = \text{Label}(m_2)$ (5)
then $\text{Ned} \leftarrow \text{Ned} \cup \{(m_1, m_2)\}$
 $S^{uc} \leftarrow S^{uc} \setminus \{(m_1, m_2)\}$
 $S^{sc} \leftarrow S^{sc} \cup \{(m_1, m_2)\}$

else $S^{uc} \leftarrow S^{uc} \setminus \{(m_1, m_2)\}$ (6)
 $S^{sc} \leftarrow S^{sc} \cup \{(m_1, m_2)\}$

Until $S^{uc} = \emptyset$

• **Phase 2**

repeat

Select $\text{dep}((m_1, m_2), S) \in \text{Dep}$ such that $\forall (r_1, r_2) \in S : (r_1, r_2) \in \text{Ned}$.

$\text{Ned} \leftarrow \text{Ned} \cup \{(m_1, m_2)\}$
 $\text{Dep} \leftarrow \text{Dep} \setminus \{\text{dep}((m_1, m_2), S) \mid \text{dep}((m_1, m_2), S) \in \text{Dep}\}$

until no further selection possible ■

The following lemma highlights the importance of the algorithm.

Lemma 5.38

Let Ned and S^{sc} denote the final values of Ned and S^{sc} obtained from Algorithm 5.37. Then:

$$\forall (m_1, m_2) \in S^{sc} : (m_1, m_2) \in \text{Ned} \text{ iff } \mathcal{ID}(m_1) \cap \mathcal{ID}(m_2) \neq \emptyset$$

■

Proof

First note that the algorithm always terminates. For the first step, the proof is completely similar to that for Algorithm 5.34. Step two always terminates as Dep is a finite set and at each step elements are removed from Dep . Moreover, no new elements are added.

1. $\forall (m_1, m_2) \in S^{sc} : (m_1, m_2) \in Ned \Rightarrow \mathcal{ID}(m_1) \cap \mathcal{ID}(m_2) \neq \emptyset$

We will prove a stronger claim: $\forall (m_1, m_2) \in Ned : \mathcal{ID}(m_1) \cap \mathcal{ID}(m_2) \neq \emptyset$.

The proof proceeds by induction on the number of repeat loops that have been applied in phase two.

base case: 0 applications.

Consider the set Ned after the first phase in the algorithm: we know that if a couple of nodes (m_1, m_2) belongs to Ned , then their intersection is not empty: either one node is a Max node or both nodes are the same constant or denote the same primitive type.

induction step

Suppose that the couple (m_1, m_2) is added to Ned . This can only be the case if there was an element $dep((m_1, m_2), S) \in Dep$ such that $\forall (r_1, r_2) \in S : (r_1, r_2) \in Ned$. Because of the induction hypothesis, $\mathcal{ID}(r_1) \cap \mathcal{ID}(r_2) \neq \emptyset$. There are two cases in which $dep((m_1, m_2), S)$ could have been added to Dep .

- $Label(m_1) = OR \vee Label(m_2) = OR$

Then because of the definition of denotation of an OR-node and because r_1 and r_2 are principal nodes of m_1 and m_2 : $\mathcal{ID}(m_1) \cap \mathcal{ID}(m_2) \neq \emptyset$

- $Label(m_1) = Label(m_2) = f/k$

In this case, $\mathcal{ID}(m_1) \cap \mathcal{ID}(m_2) \neq \emptyset$ if $\forall i, \mathcal{ID}(m_1/i) \cap \mathcal{ID}(m_2/i) \neq \emptyset$. Now, S is exactly the set of couples $(m_1/i, m_2/i)$.

2. $\forall (m_1, m_2) \in S^{sc} : \mathcal{ID}(m_1) \cap \mathcal{ID}(m_2) \neq \emptyset \Rightarrow (m_1, m_2) \in Ned$

We will prove the following equivalent statement: $\forall (m_1, m_2) \in S^{sc} : \text{if } \exists t \in \mathcal{ID}(m_1) \cap \mathcal{ID}(m_2) \text{ then } (m_1, m_2) \in Ned$. The proof proceeds via induction on the structure of t .

base case.

We distinguish two cases.

1. $t = X$. In this case, either m_1 or m_2 must be a Max node. Because of step 1 in the algorithm, (m_1, m_2) is added immediately to Ned , and nowhere any elements are removed from Ned .

2. $t \in Const_P$. In such a case, t can be a constant $c/0$ or it may belong to a primitive type as $Int, Real, \dots$. We discuss the constant case. The primitive case is analogous. If both $Label(m_1) = Label(m_2) = c/0$, then in the first phase, (m_1, m_2) is added immediately to Ned . One or both nodes however can be an OR-node. In such case, phase 1 of the algorithm adds elements $dep((m_1, m_2), \{(r_1, r_2)\})$ to Dep and (r_1, r_2) to S^{uc} . Because of normality, $Label(r_1) = Label(r_2) = c/0$, for some r_1 and r_2 , and analogously to the case above, we can conclude that (r_1, r_2) is added immediately to Ned . But then in phase 2, the element $dep((m_1, m_2), \{(r_1, r_2)\})$ can be selected. Thus (m_1, m_2) is added to Ned .

inductive case: $t = f(t_1, \dots, t_n)$.

There are three possible scenarios.

1. m_1 or m_2 can be a Max node. In such a case the couple (m_1, m_2) is added to Ned already in the first phase.

2. $Label(m_1) = Label(m_2) = f/k$. Because of the first phase in the algorithm, Dep contains a tuple $dep((m_1, m_2), \{(m_1/i, m_2/i) | 1 \leq i \leq k\})$ and the tuples $(m_1/i, m_2/i), \forall i, 1 \leq i \leq k$ have been added to S^{uc} . On these couples, we can apply the induction hypothesis, because we know that $t_i \in \mathcal{ID}(m_1/i) \cap \mathcal{ID}(m_2/i), \forall i, 1 \leq i \leq k$, thus $(m_1/i, m_2/i) \in Ned$

$\forall i, 1 \leq i \leq k$. Then, the above *dep* element is an element which satisfies the condition of phase 2 and (m_1, m_2) is added to *Ned*.

3. One of the nodes can be an OR-node. Because of normality and because of case 3 in phase one of the algorithm, *Dep* contains an element $dep((m_1, m_2), \{(r_1, r_2)\})$, with $Label(r_1) = Label(r_2) = f/k$. We can apply the same reasoning as in case 2 above to conclude that $(r_1, r_2) \in Ned$. Because of this, in phase 2 of the algorithm, the above *Dep* element is selected and (m_1, m_2) is added to *Ned*. ■

The following proposition shows how the set *Ned* can be used to by-pass the time-consuming intersection test in Algorithm 5.34.

Proposition 5.39

$\forall (m_1, m_2) \in S^{sc}$ of Algorithm 5.34: $(m_1, m_2) \in S^{sc}$ of algorithm 5.37. ■

Proof

This is trivial because (i) both algorithms start from the same couple of nodes, (ii) both have the same two reduction steps ((2) vs (3) and (4) vs (4)), and (iii) the same reduction steps are made: the same couples are added to S^{uc} . ■

5.7 Reducing the number of constraints

To reduce the number of equalities that will be generated, we put the following restriction on the definition of coefficient assignment.

Definition 5.40 (rigid coefficient assignment)

Let T be a normal type with $TypeLabel(T) = \tau$ and $Nodes$ its set of nodes. A coefficient assignment a on τ is *rigid* if $\forall n \in Nodes$ with $Label(n) = f/k$, and for all i such that $Label(ArgPos(n, i)) = Max : a(n, i) = 0$. ■

Without this rigidisation, the match algorithm would nullify the arguments of all principal nodes of a node n when compared to a *Max* node. Now, only the arc from a functor node m leading to this node n will be constrained to zero. This considerably reduces the number of implicit constraints (e.g. due to transitivity). Only the case (5) in the match algorithm must be adapted to model this:

$$\begin{aligned} &\text{else if } Label(m_1) = Label(m_2) = f/k && (5) \\ \text{then } &S^{uc} \leftarrow S^{uc} \setminus \{(m_1, m_2)\} \cup \{(m_1/i, m_2/i) \mid Label(m_1/i) \neq Max, Label(m_2/i) \neq Max\} \\ &S^{sc} \leftarrow S^{sc} \cup \{(m_1, m_2)\} \\ &E_q \leftarrow E_q \cup \{a_1(m_1, i) = a_2(m_2, i) \mid 1 \leq i \leq k\} \\ &\quad \cup \{b_1(m_1) = b_2(m_2)\} \\ &\quad \cup \{a_1(m_1, i) = 0 \mid 1 \leq i \leq k, Label(m_2/i) = Max\} \\ &\quad \cup \{a_2(m_2, i) = 0 \mid 1 \leq i \leq k, Label(m_1/i) = Max\} \end{aligned}$$

Example 5.41

Reconsider the types of Example 5.32. The above adaptation results in a match set $\{b_1(n_1) = b_2(n_2), a_2(n_2, 1) = 0, a_2(n_2, 2) = a_1(n_1, 2)\}$, where n_1 and n_2 are as defined in Example 5.35. The advantage of the modification is obvious when comparing this set with the one originally obtained. ■

From now on, we implicitly assume that every coefficient assignment is rigid.

6 Deriving typed interargument relations

For this section, we assume that P^t is a well-typed program associated to a definite program P and that n predicates (different from the annotated unification) are defined in it. Subsection 6.4 describes how to compute bottom-up for each of these predicates a linear interargument relation with respect to its output types. Recall that for well-typed programs, to each predicate there corresponds a unique output type. First, we provide a solid base for the procedure by describing the concrete and abstract domain, abstraction (and concretisation) function and some useful operations.

6.1 Concrete domain

Obviously the concrete domain consists of sets of annotated atoms $A\theta^{\bar{\tau}}$. All atoms in such a set obey their annotations, i.e. the atoms are well-typed wrt $A^{\bar{\tau}}$. We can equip D with a partial order relation \subseteq defined as the subset relation on sets.

6.2 Abstract domain

Analogously to [18], we abstract size relations as systems of linear equations. A linear equation is an equation of the form $a_1X_1 + \dots + a_mX_m = b$, where a_i, b are rational numbers, $\forall i \in \{1, \dots, m\}$. Geometrically interpreted, (the set of solutions of) such an equation corresponds to a hyperplane of dimension $m - 1$ in m -dimensional space. Then, a system of such equations corresponds to the intersection of the hyperplanes associated to each equation. Such systems can be represented by a matrix of the form $\bar{A} \cdot \bar{X} = \bar{c}$, where \bar{A} is an $m \times n$ matrix of rational numbers, \bar{X} is an $n \times 1$ matrix of variables and \bar{c} is a $1 \times m$ matrix of rational numbers. The vector X is called the *domain* of the system. Because different systems can have the same set of solutions, thus can denote the same hyperplane, a canonical form is introduced: the so-called *row echelon form* of a system. A system of equations $\bar{A} \cdot \bar{X} = \bar{c}$ is in row echelon form if its augmented matrix (i.e. the matrix $[\bar{A}|\bar{c}]$) is in reduced row echelon form. Three conditions must be satisfied for such a matrix $[\bar{A}|\bar{c}] = [a_{ij}]$: 1) the first non-zero entry in a row is 1, 2) for any row i_0 let j_0 be the first column with a non-zero entry, then for all $i > i_0, j \leq j_0, a_{ij} = 0$, 3) for any row i_0 , let j_0 be the first non-zero entry, then for all $i < i_0, a_{ij_0} = 0$.

This form can always be obtained by repeated application of one of three basic operations: multiplication of a row by a scalar, addition of one row to another and permutation of two rows.

Because there exist multiple ways to represent unsolvable systems, we adopt the convention to denote such a system by the symbol \perp . We refer to [18] for more details.

In the abstract interpretation procedure, the following operations on systems of equations will occur frequently.

- **Intersection.** Obviously, a tuple is a solution of the intersection of two systems if it is a solution of both systems. Thus, the intersection of two systems of equations corresponds to the set of solutions of the combined systems. Computing the intersection of two systems $\bar{A}_1 \cdot \bar{X} = \bar{c}_1$ and $\bar{A}_2 \cdot \bar{X} = \bar{c}_2$ corresponds to reducing to row

echelon form the augmented matrix.

$$\left[\begin{array}{c|c} \bar{A}_1 & \bar{c}_1 \\ \bar{A}_2 & \bar{c}_2 \end{array} \right]$$

- **Disjunction.** Given two systems of equations S_1 and S_2 , their disjunction is a new system which has as solutions the solutions of both systems. Of course, the union of two hyperplanes is very seldom a new hyperplane. We refer to [31] for the presentation of a technique which computes the most precise system of equations whose solution set comprises the two given hyperplanes.
- **Restriction.** Let \bar{A} be an $m \times n$ -matrix and let \bar{X} be the transposition of $[X_1, \dots, X_n]$. The restriction of a system $\bar{A} \cdot \bar{X} = \bar{c}$ to variables X_{k+1}, \dots, X_n is a system $\bar{A}_r \cdot \bar{X}_r = \bar{c}_r$ (\bar{A}_r has dimension $m' \times (n - k)$) such that

$$\begin{aligned} \exists x_1, \dots, x_n \in \mathbb{R} : (x_1, \dots, x_k, x_{k+1}, \dots, x_n) \text{ is a solution of } \bar{A} \cdot \bar{X} = \bar{c} \\ \Downarrow \\ (x_{k+1}, \dots, x_n) \text{ is a solution of } \bar{A}_r \cdot \bar{X}_r = \bar{c}_r \end{aligned}$$

[18] describes how to compute this operation for a system in reduced row-echelon form.

- **Extension.** This is the converse of the restriction operation. The operation maps a system $\bar{A} \cdot \bar{X} = \bar{c}$ to a system $\bar{A}_e \cdot \bar{X}_e = \bar{c}_e$ such that

$$\begin{aligned} (x_1, \dots, x_k) \text{ is a solution of } \bar{A} \cdot \bar{X} = \bar{c} \\ \Downarrow \\ \forall x_{k+1}, \dots, x_n \in \mathbb{R} : (x_1, \dots, x_k, x_{k+1}, \dots, x_n) \text{ is a solution of } \bar{A}_e \cdot \bar{X}_e = \bar{c}_e \end{aligned}$$

In practice, this is performed by adding $(n - k)$ columns of 0's to \bar{A} and adding k rows of a single 0 to \bar{c} .

A partial order can be established on these systems of linear equations as follows: let $S_1 = \bar{A}_1 \cdot \bar{X} = \bar{c}_1$ and $S_2 = \bar{A}_2 \cdot \bar{X} = \bar{c}_2$. Then $S_1 \leq S_2$ iff $S_1 = \perp$, or $S_2 = \top$, or the intersection of S_1 and S_2 is S_1 . We have $S_1 \equiv S_2$ iff $S_1 \leq S_2$ and $S_2 \leq S_1$. Obviously, \perp and \top are minimal and maximal for this order.

Two extremely useful operations on systems with the same domain, are the *least upper bound* and the *greatest lower bound* operation. They are defined as follows. Let S, S_1 and S_2 denote systems of linear equations with $S_1, S_2 \notin \{\perp, \top\}$:

- $\text{lub}(S, \top) = \text{lub}(\top, S) = \top$.
- $\text{lub}(S, \perp) = \text{lub}(\perp, S) = S$.
- $\text{lub}(S_1, S_2) = \text{disjunct}(S_1, S_2)$.

The greatest lower bound is defined as:

- $\text{glb}(S, \top) = \text{glb}(\top, S) = S$.

- $\text{glb}(S, \perp) = \text{glb}(\perp, S) = \perp$.
- $\text{glb}(S_1, S_2) = \text{intersect}(S_1, S_2)$.

As [18] indicates, there do not exist infinitely ascending chains of systems $S_0 < S_1 < \dots$. The maximum length of a strictly ascending chain is equal to the dimension of the domain.

We are now in a position to specify our abstract domain. If we denote the set of all possible systems of linear equations over k variables by E^k , then we fix our abstract domain to be $D^\alpha = (E^{k_1} \times E^{k_2} \times \dots \times E^{k_n})$, where k_1, \dots, k_n , are the arities of the predicate symbols in $\text{Pred}(P^t) = \{p^{\bar{\tau}_i}/k_i \mid k_i \in \mathbb{N}, 1 \leq i \leq n\}$. Elements of our abstract domain are n -tuples (S_1, \dots, S_n) of systems of linear equations of fixed dimensions. Note that there is a one-to-one correspondence between the i -th element of the n -tuple and the i -th predicate symbol in $\text{Pred}(P^t)$. The order on systems of equations naturally induces an order relation \sqsubseteq on our abstract domain: $\forall (S_1, \dots, S_n), (S'_1, \dots, S'_n) \in D^\alpha : (S_1, \dots, S_n) \sqsubseteq (S'_1, \dots, S'_n)$ iff $S_i \leq S'_i, \forall i \in \{1, \dots, n\}$. Similarly, we can lift the equivalence \equiv on systems of equations to this domain. This finishes the description of the abstract domain. Analogous as in [18], it has the structure of a complete lattice, with as bottom element (\perp, \dots, \perp) and as top element (\top, \dots, \top) .

Proposition 6.1

The abstract domain does not contain infinitely ascending chains. ■

Proof

Trivial, because there does not exist an infinite sequence of systems of linear equations $E_0 \sqsubset E_1 \sqsubset \dots$. The maximal length of strictly ascending tuples of systems is bounded by the sum of the number of variables in each system. ■

6.3 Concretisation and abstraction function

The concretisation function maps a tuple $\bar{S} = (S_1, \dots, S_n)$ of systems of equations to a set of well-typed atoms. The norms of the arguments of these well-typed atoms satisfy the system S_i of equations in the tuple \bar{S} which corresponds to the predicate symbol of the atom. For this reason we first define the component of the concretisation function which is associated to a particular set of equations (and the corresponding predicate symbol). Before giving the definition, we make the assumption that for each type τ_i occurring in P^t , a coefficient assignment a_i and an offset assignment b_i have been fixed. To enforce readability, we drop these assignments from all norm definitions, as no confusion is possible. Thus, when writing the norm $\|\cdot\|_{\tau_i}$, we actually mean the norm $\|\cdot\|_{\tau_i, a_i, b_i}$.

Definition 6.2 (punctual concretisation of a set of linear equations)

The *punctual concretisation* $\dot{\gamma}$ of a set of linear equations S over k variables and a predicate $p^{\tau_1, \dots, \tau_k}$, is defined as follows:

$$\dot{\gamma}(S, p^{(\tau_1, \dots, \tau_k)}) = \{ p^{(\tau_1, \dots, \tau_k)}(t_1, \dots, t_k) \mid (\|t_1\|_{\tau_1}, \dots, \|t_k\|_{\tau_k}) \text{ is a solution of } S \text{ and } \forall i, 1 \leq i \leq k, t_i \in \mathcal{ID}(\tau_i) \}.$$

■

It can easily be proved that $S_1 \leq S_2$ iff $\dot{\gamma}(S_1, p^{\bar{r}}) \subseteq \dot{\gamma}(S_2, p^{\bar{r}})$, for any $p^{\bar{r}}$ with appropriate arity.

We can now define the concretisation function of the tuple of systems: it simply takes the union of the sets which are the punctual concretisation of each system in the tuple.

Definition 6.3 (concretisation function)

Let $\bar{S} = (S_1, \dots, S_n)$ be a n -tuple of systems of equations, one system for each of the n predicates $p^{\bar{r}_i}/k_i$ defined in P^t . Let S_i correspond to the system for predicate $p^{\bar{r}_i}/k_i$. The *concretization function* γ on n -tuples of $\bar{S} = (S_1, \dots, S_n)$ is defined as follows

$$\gamma(\bar{S}) = \cup_{i=1}^n \dot{\gamma}(S_i, p^{\bar{r}_i}/k_i)$$

■

This definition of the concretisation function also fixes for a major part the abstraction function. We first define it on one system of equations.

Definition 6.4 (punctual abstraction of a set of annotated atoms)

The *punctual abstraction* $\dot{\alpha}$ of a set of atoms S with the same predicate symbol $p^{(\tau_1, \dots, \tau_k)}/k$ is the \sqsubseteq -least system having all tuples $(\|t_1\|_{\tau_1}, \dots, \|t_k\|_{\tau_k}), p^{(\tau_1, \dots, \tau_k)}(t_1, \dots, t_k) \in S$ as solutions.

■

In general, we can define the abstraction function as follows.

Definition 6.5 (abstraction function)

Let S be a set of well-typed atoms and let $p^{\bar{r}_i}/k_i, 1 \leq i \leq n$ be the predicates defined in P^t . Let $S = S_1 \cup \dots \cup S_n$, where each S_i contains all atoms in S with predicate symbol $p^{\bar{r}_i}/k_i$. Then $\alpha(S) = (\dot{\alpha}(S_1), \dots, \dot{\alpha}(S_n))$

■

Proposition 6.6

The partial orders \subseteq on D and \sqsubseteq on D^α and the concretisation function γ (abstraction function α) establish a Galois connection between both domains.

■

Proof

(1) α is total, i.e. that $\forall d \in D, \exists \alpha(d) \in D^\alpha$. Note that d can be written as a union of disjoint sets $S_1 \cup \dots \cup S_n$, where each S_i contains only atoms with the same predicate symbol $p^{(\tau_1, \dots, \tau_{k_i})}/k_i$. It is thus sufficient to prove that $\dot{\alpha}(S_i) \in E^{k_i}$ exists for $i = 1 \dots n$.

Consider S_i . Because E^{k_i} does not contain infinitely ascending chains of elements and because the top element \top has all possible n -tuples as a solution, there does exist an element having all tuples $(\|t_1\|_{\tau_1}, \dots, \|t_{k_i}\|_{\tau_{k_i}}), p^{(\tau_1, \dots, \tau_{k_i})}(t_1, \dots, t_{k_i}) \in S_i$ as a solution.

(2) γ is total. We must prove that $\forall d^\alpha = (E_1, \dots, E_n) \in D^\alpha, \exists \gamma(d^\alpha) \in D$. Obviously, $\gamma(d^\alpha)$ exists if $\dot{\gamma}(E_i, p^{(\tau_1, \dots, \tau_{k_i})}/k_i)$ exists, $i = 1 \dots n$, $p^{(\tau_1, \dots, \tau_{k_i})}/k_i$ being the predicate symbol associated to E_i , which is straightforward.

(3) $\alpha(d) \sqsubseteq d^\alpha$ iff $d \subseteq \gamma(d^\alpha)$. Recall that $S_1 \leq S_2$ iff $\dot{\gamma}(S_1, p_1^{\tau_1}/k_1) \subseteq \dot{\gamma}(S_2, p_2^{\tau_2}/k_2)$.

(\Rightarrow) Let $d = S_1 \cup \dots \cup S_n$ and $\alpha(d) = (E_1, \dots, E_n) \sqsubseteq d^\alpha = (E'_1, \dots, E'_n)$. This means $E_i \leq E'_i$ for $i = 1 \dots n$ and thus also $\dot{\gamma}(E_i, p_i^{\bar{r}_i}/k_i) \subseteq \dot{\gamma}(E'_i, p_i^{\bar{r}_i}/k_i)$.

(\Leftarrow) The same argument holds for this case.

■

6.4 Bottom-up abstract interpretation procedure

The goal of abstract interpretation is to approximate the semantics of programs. In this paper we consider the $\mathcal{M}(P^t)$ semantics defined in section 5.3 and we adopt the bottom-up abstract interpretation of [4]. The basic operation is the abstract interpretation of unification. In [18], using a top-down framework, given a call to unification $X = t$, and an abstract call substitution S (a system of linear equations), the abstract success substitution is $S \cup \{X = \text{abs}_{\|\cdot\|}(t)\}$. Here, $\text{abs}_{\|\cdot\|}$ is called the abstract norm and it is a function which maps each term t to a linear arithmetic expression, which can best be described as a partial evaluation of the concrete norm on the term. In our setting, the notion is generalised to type-annotated unification and typed norms as in the following definition.

Definition 6.7 (size expression induced by a typed norm $\|\cdot\|_\tau$)

Let T be a labelled normal type, let $\tau = \text{TypeLabel}(n)$ be the TypeLabel of a node n in T and $\|\cdot\|_\tau$ the associated norm. A term t is mapped to the *size expression* $\text{abs}_\tau(t)$ induced by the norm $\|\cdot\|_\tau$ by means of the following function $\text{abs}_\tau : \text{Term}_P/\sim \rightarrow \mathcal{L}_{<0,1;+;\leq}$ defined as

$$\begin{array}{ll}
 \text{if } & \text{label}(n) = OR \text{ then} \\
 & \text{if } t = X, \text{ a variable} \quad \text{then } \text{abs}_\tau(t) = X \\
 & \text{otherwise} \quad \text{abs}_\tau(t) = \sum_{\tau_i \in \text{Succ-Label}(n)} \text{abs}_{\tau_i}(t) \\
 \text{else if } & \text{label}(n) = f/k \text{ then} \\
 & \text{if } t = X, \text{ a variable} \quad \text{then } \text{abs}_\tau(t) = X \\
 & \text{if } t = f(t_1, \dots, t_n) \quad \text{then } \text{abs}_\tau(t) = b(n) + \sum_{i=1}^n a(n, i) * \text{abs}_{\tau_i}(t_i) \\
 & \quad \quad \quad \text{with } \tau_i = \text{TypeLabel}(\text{ArgPos}(n, i)) \\
 \text{else} & \text{abs}_\tau(t) = 0
 \end{array}$$

■

Example 6.8

Consider the type of lists of any elements displayed in Figure 5.5. Let a and b be the natural, rigid assignments on the type. Then:

$$\begin{array}{l}
 \text{abs}_{\tau_1}([H|T]) = 1 + T. \\
 \text{abs}_{\tau_1}([[U|V]|T]) = 1 + T. \\
 \text{abs}_{\tau_1}(X) = X. \\
 \text{abs}_{\tau_4}(X) = 0.
 \end{array}$$

Observe how this mapping abstracts a term to the already available size information. Variables in such a size expression denote how the size of the term may be affected through instantiation. ■

The abstract \mathcal{T}_P^α operator is defined in three stages. First, we introduce an auxiliary function \mathcal{A} which, given an atom (either user defined or unification) and an element $\bar{S} = (S_1, \dots, S_n)$ of the abstract domain, selects from \bar{S} the information relevant to this atom.

Definition 6.9 \mathcal{A}

Let $p^{\bar{\tau}}/k$ be a predicate defined in P^t and let $\bar{S} \in D^\alpha$ be a tuple of systems of equations, one system for each of the n predicates occurring in P^t . By $\bar{S}[p^{\bar{\tau}}/k]$ we denote the system of equations in S corresponding to $p^{\bar{\tau}}/k$. The operator $\mathcal{A} : \bar{B}_P \times D^\alpha \rightarrow \cup_{i=1}^n E^{k_i}$ abstracts atoms as follows:

$$\mathcal{A}(X = Y^{(\tau_0, \tau_0)}, \bar{S}) = \{X = Y\}$$

$$\mathcal{A}(X = f(Y_1, \dots, Y_n)^{(\tau_0, \tau_1, \dots, \tau_n)}, \bar{S}) = \{X = \text{abs}_{\tau_0}(f(Y_1, \dots, Y_n))\}$$

$$\mathcal{A}(p^{\bar{\tau}}(X_1, \dots, X_k), \bar{S}) = \bar{S}[p^{\bar{\tau}}/k]\theta$$

where θ is a renaming of the variables in $\bar{S}[p^{\bar{\tau}}/k]$ according to X_1, \dots, X_k . ■

\mathcal{A} abstracts unification in the same way as presented in the previous section, more specifically by taking the abstraction of both the lefthandside and the righthandside under the norm identified by the lefthandside.

To define the abstract operator $T_P^\alpha : D^\alpha \rightarrow D^\alpha$, we need to define a T_P^α operator which, for each $\bar{S} \in D^\alpha$, computes the i -th element of the n -tuple $T_P^\alpha(\bar{S}) \in D^\alpha$. More in particular, $T_P^\alpha : \text{Pred}(P^t) \times D^\alpha \rightarrow \cup_{i=1}^n E^{k_i}$, with $T_P^\alpha(p^{\bar{\tau}_i}/k_i, \bar{S}) \in E^{k_i}$.

Definition 6.10 T_P^α

Let $p^{(\tau_1, \dots, \tau_k)} \in \text{Pred}(P^t)$ be defined by m clauses C_1, \dots, C_m whose heads we assume all identical to $p^{(\tau_1, \dots, \tau_k)}(X_1, \dots, X_k)$. Let $\text{Var}(C_i)$ be the set of variables occurring in C_i .

$$T_P^\alpha(p^{\bar{\tau}}/k, \bar{S}) = \text{lub}(\mathcal{A}(p^{\bar{\tau}}(X_1, \dots, X_k), \bar{S}), \text{lub}_{i=1, \dots, m}(S_{\text{Head}(C_i)}))$$

where

$$S_{\text{Head}(C_i)} = \text{restriction}(\{X_1, \dots, X_k\}, S_{\text{Body}(C_i)})$$

$$S_{\text{Body}(C_i)} = \text{glb}_{B \in \text{Body}(C_i)}(S_B)$$

$$S_B = \text{extension}(\text{Var}(C_i), \mathcal{A}(B, \bar{S})).$$

■

Lemma 6.11

$$\mathcal{A}(p^{\bar{\tau}}(X_1, \dots, X_k), \bar{S}) \leq T_P^\alpha(p^{\bar{\tau}}/k, \bar{S}).$$

■

Proof

Trivial. ■

From T_P^α , we can now easily define $T_P^\alpha : D^\alpha \rightarrow D^\alpha$.

Definition 6.12 T_P^α

$$T_P^\alpha : D^\alpha \rightarrow D^\alpha : \bar{S} \rightarrow (T_P^\alpha(p^{\bar{\tau}_1}/k_1, \bar{S}), \dots, T_P^\alpha(p^{\bar{\tau}_n}/k_n, \bar{S})).$$

■

Lemma 6.13

T_P^α is monotonic. ■

Proof

Let $E = (E_q^{k_1}, \dots, E_q^{k_n}), E' = (E_q'^{k_1}, \dots, E_q'^{k_n}) \in D^\alpha$ and $E \sqsubseteq E'$. We need to prove $\mathcal{T}_P^\alpha(E) \sqsubseteq \mathcal{T}_P^\alpha(E')$. By definition of \sqsubseteq and \mathcal{T}_P^α it is sufficient to prove that for all $i = 1, \dots, n, E \sqsubseteq E'$ implies $\mathcal{T}_P^\alpha(p_i^{\bar{\tau}^i}, E) \leq \mathcal{T}_P^\alpha(p_i^{\bar{\tau}^i}, E')$. Obviously, $\mathcal{A}(p_i^{(\tau_1, \dots, \tau_n)}(X_1, \dots, X_n), E) \leq \mathcal{A}(p_i^{(\tau_1, \dots, \tau_n)}(X_1, \dots, X_n), E')$. It remains to be shown that $e \rightarrow \text{lub}_{i=1, \dots, m}(\text{restrict}((X_1, \dots, X_k), \text{glb}_{B \in \text{Body}(C_i)}(\text{extend}((Z_1^i, \dots, Z_{l_i}^i), \mathcal{A}(B, e))))))$ is monotonic.

First note that $e \rightarrow \mathcal{A}(B, e)$ is monotonic. If B is a unification, then $\mathcal{A}(B, e)$ is independent of e . Else, let $B = p_j^{\bar{\tau}^j}(\bar{U})$ and assume that $E \sqsubseteq E'$. We have: $\mathcal{A}(B, E) = E[p_j]$ renamed to \bar{Y} which is equal to $E_q^{k_j}$ renamed to $\bar{Y} \leq E_q'^{k_j}$. Finally, all of the operations extend, restrict, lub and glb are monotonic, which concludes the proof. ■

Definition 6.14 ω -first powers of \mathcal{T}_P^α

The ω -first powers of the operator are defined as usual:

$$\mathcal{T}_P^\alpha \uparrow 0 = (\perp, \dots, \perp).$$

$$\mathcal{T}_P^\alpha \uparrow (i + 1) = \mathcal{T}_P^\alpha(\mathcal{T}_P^\alpha \uparrow i).$$

$$\mathcal{T}_P^\alpha \uparrow \omega = \text{lub}_{i \in \mathbb{N}} \mathcal{T}_P^\alpha \uparrow i. \quad \blacksquare$$

Proposition 6.15

There exists an $n \in \mathbb{N}$ such that $\mathcal{T}_P^\alpha \uparrow n$ is a least fixpoint. ■

Proof

This is an immediate consequence from:

- 1) $\bar{S} \sqsubseteq \mathcal{T}_P^\alpha(\bar{S})$ (because of Lemma 6.11)
- 2) \mathcal{T}_P^α is monotonic
- 3) D^α has no infinite ascending chains. ■

Before we can state some correctness results relating \mathcal{T}_P^α to the model $\mathcal{M}(P^t)$, we must extend the definition of the matching set introduced in the previous section. In that section, on clause level, abstraction was only made of possible calls, recursive calls and all intermediate atoms. Now, all atoms in a clause are involved. This calls for additional matching operations to preserve consistency. The following adapted definition of the *set of typed variables* mirrors this argument:

Definition 6.16 (Set of typed variables $TV(P^t)$)

$TV(P^t)$ is the set of couples (X, τ) such that there exists a clause $\bar{\tau}_0^i A_0^{\bar{\tau}_0^i} \leftarrow A_1^{\bar{\tau}_1^i}, \dots, A_n^{\bar{\tau}_n^i}$, such that X is typed τ in some atom $A_i, 0 \leq i \leq n$ wrt $\bar{\tau}_i^i$. ■

An immediate remark here is which set must be used to produce norms for the complete termination proof, including the derivation of interargument relations. A naive answer would simply be to merge both sets of typed variables to produce one matching set. In a lot of cases this merging will have a disastrous effect of decreasing the precision of the norms. We refer to the Deeppermute example in section 7.2 for an illustration. The solution is to derive two matching sets, one for each set of typed variables, and join these matching sets. Again, we refer to the Deeppermute example.

Proposition 6.17

Let P be a definite logic program and P^t a well-typed program associated to P . If

$\forall(\tau_1, \tau_2) \in Match(P^t), \|\cdot\|_{\tau_1}$ and $\|\cdot\|_{\tau_2}$ are matching, then \mathcal{T}_P^α safely approximates the \mathcal{T}_P^* -operator: for every $n \geq 0$

$$\mathcal{T}_P^* \uparrow n \models_\sigma p(X_1, \dots, X_k)^{(\tau_1, \dots, \tau_k)} \Rightarrow p(X_1, \dots, X_k)\sigma^{(\tau_1, \dots, \tau_k)} \in \gamma(\mathcal{T}_P^\alpha \uparrow n)$$

■

Proof

The proof proceeds via induction on n .

base case: $n = 0$.

This is trivial as $\mathcal{T}_P^* \uparrow 0 = \emptyset$.

induction step:

We prove this step only for atoms which are added in iteration n . This is sufficient as \mathcal{T}_P^* is monotonic and starts from \emptyset .

Because $\mathcal{T}_P^* \uparrow n \models_\sigma p(X_1, \dots, X_k)^{(\tau_1, \dots, \tau_k)}$ the following must hold:

(1) there exists a clause, lets say $C \equiv \bar{\tau}^i p(X_1, \dots, X_k)^{\bar{\tau}^o} \leftarrow A_1^{\bar{\tau}_1}, \dots, A_n^{\bar{\tau}_n}$ in P^t , with $\bar{\tau}^o = (\tau_1, \dots, \tau_k)$

(2) $\mathcal{T}_P^* \uparrow (n-1) \models_\sigma A_i^{\bar{\tau}_i}$,

(3) $p(X_1, \dots, X_k)\sigma$ is well-typed wrt $p(X_1, \dots, X_k)^{\bar{\tau}^o}$.

We must prove that $p(X_1, \dots, X_k)\sigma \in \gamma(\mathcal{T}_P^\alpha \uparrow n)$, or simpler, that $p(X_1, \dots, X_k)\sigma \in \dot{\gamma}(\mathcal{T}_P^\alpha \uparrow n[p^{(\tau_1, \dots, \tau_k)}/k], p^{(\tau_1, \dots, \tau_k)}/k)$. Because of the definition of $\dot{\gamma}$, $(\|X_1\sigma\|_{\tau_1}, \dots, \|X_k\sigma\|_{\tau_k})$ must be a solution of $\mathcal{T}_P^\alpha \uparrow n[p^{(\tau_1, \dots, \tau_k)}/k]$ and $\forall i, 1 \leq i \leq k, X_i\sigma \in \mathcal{ID}(\tau_i)$. The second part always holds because of (3).

So it remains to prove that $(\|X_1\sigma\|_{\tau_1}, \dots, \|X_k\sigma\|_{\tau_k})$ is a solution of $\mathcal{T}_P^\alpha \uparrow n[p^{(\tau_1, \dots, \tau_k)}/k] = \mathcal{T}_P^{\alpha P}(p^{(\tau_1, \dots, \tau_k)}/k, \mathcal{T}_P^\alpha \uparrow (n-1))$. Because of the definition of the *lub*, it is sufficient to prove that $(\|X_1\sigma\|_{\tau_1}, \dots, \|X_k\sigma\|_{\tau_k})$ is a solution of $\text{lub}_{i=1 \dots m} S_{Head(C_i)}$. Note that because all heads are identical, all systems $Head(C_i)$ have the same domain: (X_1, \dots, X_k) . Because of the way the *lub* is defined, it is sufficient to show that for one i , $(\|X_1\sigma\|_{\tau_1}, \dots, \|X_k\sigma\|_{\tau_k})$ is a solution of $S_{Head(C_i)}$.

Now take $C_i = C$. Let X_{k+1}, \dots, X_m be all local variables of clause C . Then we must prove that for some $\tau_{k+1}, \dots, \tau_m$, $(\|X_1\sigma\|_{\tau_1}, \dots, \|X_k\sigma\|_{\tau_k}, \|X_{k+1}\sigma\|_{\tau_{k+1}}, \dots, \|X_m\sigma\|_{\tau_m})$ is a solution of $S_{Body(C)}$, as the restriction operation does not affect solutions nor the way in which terms are measured.

It remains to prove that $(\|X_1\sigma\|_{\tau_1}, \dots, \|X_k\sigma\|_{\tau_k}, \|X_{k+1}\sigma\|_{\tau_{k+1}}, \dots, \|X_m\sigma\|_{\tau_m})$ is a solution of $\text{glb}_{B \in Body(C)} S_B$, thus the tuple must be a solution of each separate S_B and each system S_B must have the same domain and the solution must be preserved under the *glb*-operation. The second point holds because of the extension operation.

To prove that the tuple is a solution of each S_B , take some atom $B \in Body(C)$. Then there are three possibilities:

- $B \equiv q(U_1, \dots, U_r)^{(\tau_1, \dots, \tau_r)}$

Because of (2), and the induction hypothesis, we know that $q(U_1, \dots, U_r)\sigma^{(\tau_1, \dots, \tau_r)} \in \gamma(\mathcal{T}_P^\alpha \uparrow (n-1))$, and thus that it is a solution of $\mathcal{A}(B, \mathcal{T}_P^\alpha \uparrow (n-1))$, and thus also of the extension to $Var(C)$.

- $B \equiv X = Y^{(\tau_X, \tau_Y)}$

For this case, we must prove that $(\|X\theta\|_{\tau_X}, \|Y\theta\|_{\tau_Y})$ is a solution of $\mathcal{A}(X =$

$Y^{(\tau_X, \tau_Y)}, \mathcal{T}_P^\alpha \uparrow (n-1)$). By definition, $\mathcal{A}(X = Y^{(\tau_X, \tau_Y)}, \mathcal{T}_P^\alpha \uparrow (n-1)) = \{X = Y\}$. Because of (2), we know that $P \models_\sigma X = Y$ and because the program is well-typed, and because rigid types are closed under substitution, $X\sigma \in \mathcal{ID}(\tau_X)$ and $Y\sigma \in \mathcal{ID}(\tau_Y)$. Because $X\sigma = Y\sigma$, also $\|X\sigma\|_{\tau_X} = \|Y\sigma\|_{\tau_X}$.

- $B \equiv X = f(Y_1, \dots, Y_r)^{(\tau_0, \tau_1, \dots, \tau_r)}$

The same reasoning as for the second case applies here.

Now, we still have to prove that this tuple is a solution of the glb of all these systems S_{B_i} . More in particular, we must prove that all variables in each separate system stand for the same variables. This is guaranteed because of the matchings which have been imposed. Consider any possible double occurrence of a variable, let's say X . The types, say τ_1 and τ_2 , under which the variable is abstracted are exactly those types on which a match has been imposed. To show this, *first* notice that whenever a variable which is typed τ_1 in an atom, is abstracted under a norm $\|\cdot\|_{\tau_2}$, then a match between τ_1 and τ_2 has been performed. This can easily be seen by considering the following four possible cases: (1) $p(Y_1, \dots, Y_n)^{(\tau_1, \dots, \tau_n)}$ and $X = Y_i$ for some $i, 1 \leq i \leq n$. Here the typing of X and the type under which it is abstracted are the same. (2) $X = Y^{(\tau_x, \tau_y)}$ or $X = f(Y_1, \dots, Y_n)^{(\tau_0, \tau_1, \dots, \tau_n)}$. Again both types are the same. (3) $Y = X^{(\tau_y, \tau_x)}$. Here the typing of X is τ_x while it is abstracted under τ_y . However, (τ_y, τ_x) is added to $Match(P^t)$ because of case 2 of the matching set definition. (4) $U = f(Y_1, \dots, Y_n)^{(\tau_0, \tau_1, \dots, \tau_n)}$ and $X = Y_i$ for some $i, 1 \leq i \leq n$. Here the typing of X is τ_i while it becomes abstracted under the type τ_0/i . The third case of the matching set definition is responsible for adding the couple $(\tau_0/i, \tau_i)$ to $Match(P^t)$.

Second, wherever there is a double occurrence of X , typed both τ_1 and τ_2 , we know that $(X, \tau_1) \in TV(P^t)$ and $(X, \tau_2) \in TV(P^t)$. The first case in the matching set construction then adds a couple (τ_1, τ_2) to $Match(P^t)$. Moreover, because of the first part, these types have also been matched with the types under which X is abstracted.

Finally, if X is typed τ_1 and τ_2 , then because of the well-typedness of P^t and the substitution closedness of rigid types, $X\sigma \in \mathcal{ID}(\tau_1)$ and $X\sigma \in \mathcal{ID}(\tau_2)$, and because of the matching $\|X\sigma\|_{\tau_1} = \|X\sigma\|_{\tau_2}$. ■

Corollary 6.18

Let P be a definite program and let P^t be a well-typed program associated to P . If $\mathcal{M}(P^t) \models_\sigma p(X_1, \dots, X_k)^{(\tau_1, \dots, \tau_k)}$ then $(\|X_1\sigma\|_{\tau_1}, \dots, \|X_k\sigma\|_{\tau_k})$ is a solution of $\mathcal{T}_P^\alpha \uparrow \omega[p^{(\tau_1, \dots, \tau_k)}/k]$. ■

The previous corollary and Proposition 5.23 give the main result of this section.

Proposition 6.19

Let P be a definite program and let P^t be a well-typed program associated to P . Let $p(X_1, \dots, X_k)\theta_c\theta_s$ be a computed instance in P of a call $p(X_1, \dots, X_k)\theta_c$ which is well-typed wrt $in(p/k) = (\tau_1, \dots, \tau_k)$. Then:

$(\|X_1\theta_c\theta_s\|_{\tau_1}, \dots, \|X_k\theta_c\theta_s\|_{\tau_k})$ is a solution of $\mathcal{T}_P^\alpha[p^{(\tau_1, \dots, \tau_k)}/k]$. ■

7 Examples

7.1 Expand

This example illustrates where typed norms can be useful. It also highlights the practicality of the approach in the sense that one does not have to worry about rigidity any longer, as it is automatically ensured.

The program *expand* accepts two arguments. The first argument is a list of an even number of elements, in which the odd entries are natural numbers, whereas on the even positions any terms can occur. When called with the second argument uninstantiated, the predicate constructs a list where each term on an even position $2n$ in the original list is reproduced as many times as the number on position $2n - 1$ indicates.

```

expand(nil, nil)           ←
expand([0, V|T], ET)      ← expand(T, ET).
expand([s(N), V|T], [V|ET]) ← expand([N, V|T], ET).

```

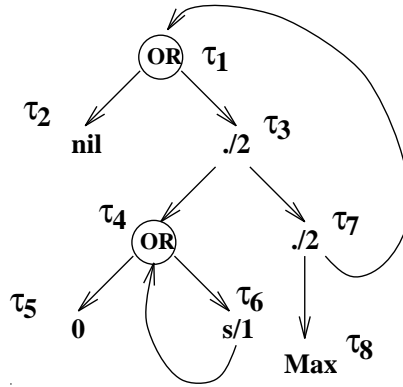


Figure 7: Lists where the type of an element depends on its position.

Let us now prove termination for this program for the set of queries S formalised by the call pattern $expand(\tau_1, Max)$, where τ_1 is shown in Figure 7. Let T_{oe} denote this typegraph. This pattern also comprises the call set $Call_{expand}(S)$. Using only one semi-linear norm is not sufficient to prove termination. Instead, Typed Norms provide a solution.

With this type there corresponds a class of typed norms, each of which is rigid wrt the terms in its denotation. As no interargument relations are necessary, no matches have to be imposed and any selection of a coefficient assignment a and offset assignment b on T_{oe} satisfies our purpose. We take the natural coefficient assignment for a and take b to be the constant one-mapping. Thus:

$$\begin{aligned}
 ||t||_{\tau_1} &= ||t||_{\tau_3} \\
 |[t_1|t_2]|_{\tau_3} &= 1 + ||t_1||_{\tau_4} + ||t_2||_{\tau_1} \\
 ||t||_{\tau_4} &= ||t||_{\tau_6} \\
 ||s(t)||_{\tau_6} &= 1 + ||t||_{\tau_4} \\
 |[t_1|t_2]|_{\tau_7} &= 1 + ||t_2||_{\tau_1}
 \end{aligned}$$

With the above call information and this norm, we can derive the following levelmapping:

$$|\mathit{expand}(t_1, t_2)| = \|t_1\|_{\tau_1} + \|t_2\|_{Max} = \|t_1\|_{\tau_1}$$

of which we know that it is rigid wrt S .

To see that the program is terminating wrt the set S , take any query $\mathit{expand}(t_1, t_2)$ of S . Two possibilities can be distinguished. Either it unifies with the head of the first recursive clause, $\mathit{expand}(t_1, t_2)\theta = \mathit{expand}([0, V|T], ET)\theta$, with $\theta = \mathit{mgu}(\mathit{expand}(t_1, t_2), \mathit{expand}([0, V|T], ET))$. Then $|\mathit{expand}([0, V|T], ET)\theta| = \|[0, V|T]\theta\|_{\tau_1} = 1 + 1 + \|T\theta\|_{\tau_1} > \|T\theta\|_{\tau_1} = |\mathit{expand}(T, ET)\theta|$. Because $|\mathit{expand}(t_1, t_2)\theta| = |\mathit{expand}(t_1, t_2)|$, our original call, we always obtain a decrease in level between an original and a recursive call using the first clause.

The second possibility is where the second recursive clause is used to resolve the call. Take any call $\mathit{expand}(t_1, t_2)$ of S such that $\theta = \mathit{mgu}(\mathit{expand}([s(N), V|T], [V|ET]), \mathit{expand}(t_1, t_2))$ exists. Then $|\mathit{expand}([s(N), V|T], [V|ET])\theta| = \|[s(N), V|T]\theta\|_{\tau_1} = 1 + 1 + \|N\theta\|_{\tau_4} + 1 + \|T\theta\|_{\tau_1} > 1 + \|N\theta\|_{\tau_4} + 1 + \|T\theta\|_{\tau_1} = |\mathit{expand}([N, V|T], ET)|$. Again, $|\mathit{expand}(t_1, t_2)\theta| = |\mathit{expand}(t_1, t_2)|$ and we can always prove a reduction between a call in S and a recursive call.

7.2 Deep permute

This example shows that in some cases matches are necessary. The predicate *Deeppermute* succeeds whenever the first argument is a list of lists of any elements which is a deep permutation of a similar list in the second argument. A deep permutation means that not only both top level lists are a permutation of each other, but that also all their elements (again lists) are permuted.

$$\begin{aligned} \mathit{Deeppermute}(\mathit{nil}, \mathit{nil}) &\leftarrow \\ \mathit{Deeppermute}([E_1|T_1], L_2) &\leftarrow \mathit{permute}(E_1, E_2), \\ &\quad \mathit{permute}(L_2, [E_2|T_2]), \\ &\quad \mathit{Deeppermute}(T_1, T_2). \\ \\ \mathit{permute}(\mathit{nil}, \mathit{nil}) &\leftarrow \\ \mathit{permute}(L_1, [H|L_2]) &\leftarrow \mathit{delete}(H, L_1, DL_1), \\ &\quad \mathit{permute}(DL_1, L_2). \\ \\ \mathit{delete}(E, [E|T], T) &\leftarrow \\ \mathit{delete}(El, [E|T_1], [E|T_2]) &\leftarrow \mathit{delete}(El, T_1, T_2). \end{aligned}$$

The goal is to prove termination for all *Deeppermute* queries which satisfy the following call pattern: $\mathit{Deeppermute}(\tau_1, \tau_1)$. We refer to Figure 3 for a definition of τ_1 . Let S denote $\mathcal{D}(\mathit{Deeppermute}(\tau_1, \tau_1))$, the queries for which we want to prove termination. If we perform a type analysis from this information, the following patterns result:

$$\begin{aligned} \mathit{Deeppermute}(\tau_1, \tau_1) &\rightarrow \mathit{Deeppermute}(\tau_1, \tau_1) \\ \mathit{permute}(\tau_4, Max) &\rightarrow \mathit{permute}(\tau_4, \tau_4) \\ \mathit{delete}(Max, \tau_4, Max) &\rightarrow \mathit{delete}(Max, \tau_4, \tau_4) \end{aligned}$$

Here, τ_4 again refers to the corresponding Typelabel in Figure 3. Notice that *permute* is called in two different ways in the recursive clause for *Deeppermute*. Because of the restriction that only one global pattern is retained, our matching operation will be needed.

Using type inference, we can construct a well-typed (and standardised apart) program Deeppermute^t associated to Deeppermute (again referring to the types in Figure 3):

$$\begin{array}{ll}
\tau_1, \tau_1 \text{ Deeppermute}^{\tau_1, \tau_1}(A, B) & \leftarrow A =^{\tau_2} \text{nil}, B =^{\tau_2} \text{nil}. \\
\tau_1, \tau_1 \text{ Deeppermute}^{\tau_1, \tau_1}(C, D) & \leftarrow C =^{\tau_3, \tau_4, \tau_1} [E_1 | T_1], \\
& \text{permute}^{\tau_4, \tau_4}(E_1, E_2), \\
& Y_p =^{\tau_3, \tau_4, \tau_1} [E_2 | T_2], \\
& \text{permute}^{\tau_4, \tau_4}(D, Y_p), \\
& \text{Deeppermute}^{\tau_1, \tau_1}(T_1, T_2). \\
\tau_4, \text{Max permute}^{\tau_4, \tau_4}(E, F) & \leftarrow E =^{\tau_5} \text{nil}, F =^{\tau_5} \text{nil}. \\
\tau_4, \text{Max permute}^{\tau_4, \tau_4}(I, J) & \leftarrow \text{delete}^{\text{Max}, \tau_4, \tau_4}(H, I, DL_1), \\
& \text{permute}^{\tau_4, \tau_4}(DL_1, T), \\
& J =^{\tau_6, \tau_7, \tau_4} [H | T]. \\
\text{Max}, \tau_4, \text{Max delete}^{\text{Max}, \tau_4, \tau_4}(K, L, M) & \leftarrow L =^{\tau_6, \tau_7, \tau_4} [K | M] \\
\text{Max}, \tau_4, \text{Max delete}^{\text{Max}, \tau_4, \tau_4}(O, P, Q) & \leftarrow P =^{\tau_6, \tau_7, \tau_4} [R | S], \\
& \text{delete}^{\text{Max}, \tau_4, \tau_4}(O, S, T_s), \\
& Q =^{\tau_6, \tau_7, \tau_4} [R | T_s].
\end{array}$$

To prove left-termination for Deeppermute wrt to the set of queries S , we first have to derive suitable Typed Norms. Obviously, we will need semantic information about the success of the permute atoms. This forces us to impose some matches on our norms. Once the norms are fixed, we will use them to derive typed interargument relations for the permute and the delete predicates. These are then used to show left-termination. These steps are illustrated in the next three sections.

7.2.1 Matching

The goal of the matching phase is to avoid measuring the same terms under different norms. Possible type clashes are assembled in the matching set. As explained in the previous sections, we compute two different sets of Typed Variables, each one producing a different matching set. From the viewpoint of termination, the set of Typed variables $TV(\text{Deeppermute}^t)$ results in a matching set $\{(\tau_1, \tau_3), (\tau_1, \tau_4), (\tau_3, \tau_4), (\tau_4, \tau_6)\}$. The set of Typed variables for the interargument relation derivation produces the following matching set: $\{(\tau_1, \tau_2), (\tau_1, \tau_3), (\tau_1, \tau_4), (\tau_3, \tau_4), (\tau_4, \tau_5), (\tau_4, \tau_6)\}$. Most of these types feature a type/subtype relationship. Two cases however generate matching conditions: (τ_1, τ_4) and (τ_3, τ_4) . If we use the matching algorithm of section 5.6 on the first tuple, we obtain the following system of matching conditions: $\text{match}(\tau_1, \tau_4) = \{a(n_3, 1) = 0, b(n_3) = b(n_6), a(n_3, 2) = a(n_6, 2)\}$. Here, n_i stands for the node with TypeLabel τ_i . The same system of matching conditions is generated by the second tuple. So $\mathcal{MC}(\text{Deeppermute}^t) = \{a(n_3, 1) = 0, b(n_3) = b(n_6), a(n_3, 2) = a(n_6, 2)\}$.

A norm which satisfies $\mathcal{MC}(\text{Deeppermute}^t)$ is the following one, obtained by taking a and b one on the tuples which are not in $\mathcal{MC}(\text{Deeppermute}^t)$, and by making a a rigid

coefficient assignment:

$$\begin{aligned}
\|t\|_{\tau_1} &= \|t\|_{\tau_3} \\
\|[t_1|t_2]\|_{\tau_3} &= 1 + \|t_2\|_{\tau_3} \\
\|t\|_{\tau_4} &= \|t\|_{\tau_6} \\
\|[t_1|t_2]\|_{\tau_6} &= 1 + \|t_2\|_{\tau_6} \\
\|t\|_{\tau_2} &= \|t\|_{\tau_5} = \|t\|_{\tau_7} = 0
\end{aligned}$$

In other words, both $\|\cdot\|_{\tau_1}$ and $\|\cdot\|_{\tau_4}$ reduce to the listlength norm on their respective denotation.

This example also illustrates why it is beneficial to produce two matching sets instead of just merging the sets of Typed Variables. In the first set of Typed Variables, there is an occurrence (J, Max) coming from the second clause. The second set of Typed Variables has the entries (J, τ_4) and (J, τ_6) . Aggregating both sets would impose matchings between τ_4 and Max and between τ_6 and Max . This would have a disastrous effect in the sense that only the constant zero-norm is a norm satisfying the resulting conditions.

7.2.2 Deriving interargument relations

Now that we have derived convenient norms, we can start the process of deriving typed interargument relations. Obviously, we need two of them: one for permute wrt (τ_4, τ_4) and one for delete wrt (Max, τ_4, τ_4) . This means that $T_{D\text{eeppermute}}^\alpha$ (abbreviated to T_P^α in the remainder of this subsection) is defined on $\langle E_q^2, E_q^3 \rangle$. We adopt the convention that in a couple (S_p, S_d) , S_p corresponds to a system of equations for permute $^{\tau_4, \tau_4}$ and S_d to one for delete $^{Max, \tau_4, \tau_4}$. In order to enhance the readability of this example, we do not work with the matrix representation of the systems of equations. Neither do we show where the extension operation comes in.

Let us rename the permute and delete clause such that all heads become identical.

$$\begin{aligned}
\tau_4, Max \text{ permute}^{\tau_4, \tau_4}(X, Y) &\leftarrow X =^{\tau_5} nil, Y =^{\tau_5} nil. \\
\tau_4, Max \text{ permute}^{\tau_4, \tau_4}(X, Y) &\leftarrow \text{delete}^{Max, \tau_4, \tau_4}(H, X, DL_1), \\
&\quad \text{permute}^{\tau_4, \tau_4}(DL_1, T), \\
&\quad Y =^{\tau_6, \tau_7, \tau_4} [H|T]. \\
Max, \tau_4, Max \text{ delete}^{Max, \tau_4, \tau_4}(X, Y, Z) &\leftarrow Y =^{\tau_6, \tau_7, \tau_4} [X|Z] \\
Max, \tau_4, Max \text{ delete}^{Max, \tau_4, \tau_4}(X, Y, Z) &\leftarrow Y =^{\tau_6, \tau_7, \tau_4} [R|S], \\
&\quad \text{delete}^{Max, \tau_4, \tau_4}(X, S, T_s), \\
&\quad Z =^{\tau_6, \tau_7, \tau_4} [R|T_s].
\end{aligned}$$

For convenience, let us number these clauses Cl_1 till Cl_4 .

Then:

$$T_P^\alpha \uparrow 0 = \langle \perp, \perp \rangle.$$

$T_P^\alpha \uparrow 1$: Here we have to compute both $T_P^\alpha(\text{permute}^{\tau_4, \tau_4}/2, (\perp, \perp))$ and $T_P^\alpha(\text{delete}^{Max, \tau_4, \tau_4}/2, \langle \perp, \perp \rangle)$. Starting with permute $^{\tau_4, \tau_4}/2$, we compute for both of its clauses Cl_i ($glb_{B \in \text{Body}(Cl_i)} \mathcal{A}(B, (\perp, \perp))|_{\{X, Y\}}$). Cl_1 gives $glb(\{X = 0, Y = 0\})|_{\{X, Y\}} = \{X = 0, Y = 0\}$.

For Cl_2 , we obtain $glb(\perp, \perp, \{Y = 1 + T\})|_{\{X, Y\}} = \perp$. Hence, $lub(\perp, \{X = 0, Y =$

$0\}, \perp) = \{X = 0, Y = 0\}$.

For Cl_3 , for $\text{delete}^{Max, \tau_4, \tau_4}$, $\text{glb}(\{Y = 1 + Z\})|_{\{X, Y, Z\}} = \{Y = 1 + Z\}$. Cl_4 gives $\text{glb}(\{Y = 1 + S\}, \perp, \{Z = 1 + T_s\})|_{\{X, Y, Z\}} = \perp$.

$\text{lub}(\perp, \{Y = 1 + Z\}, \perp) = \{Y = 1 + Z\}$.

$\mathcal{T}_P^g \uparrow 2$: We again consider all clauses, now making use of the information computed in the previous iteration. In short, for Cl_1 we obtain $\{X = 0, Y = 0\}$, and for Cl_2 , $\{X = 1, Y = 1\}$. $\text{lub}(\{X = 0, Y = 0\}, \{X = 0, Y = 0\}, \{X = 1, Y = 1\}) = \{X = Y\}$.

For Cl_3 the set $\{Y = 1 + Z\}$ is obtained, Cl_4 generates $\{Y = 1 + Z\}$. $\text{lub}(\{Y = 1 + Z\}, \{Y = 1 + Z\}, \{Y = 1 + Z\}) = \{Y = 1 + Z\}$.

$\mathcal{T}_P^g \uparrow 3$: This iteration does not result in any new information, so that the least fixpoint is reached in $\mathcal{T}_P^g \uparrow 2$.

The resulting size relation says that whenever an atom $\text{permute}(t_1, t_2)$ is a logical consequence of the *Deeppermute* program, and its arguments satisfy $t_1 \in \mathcal{ID}(\tau_4)$ and $t_2 \in \mathcal{ID}(\tau_4)$, then $\|t_1\|_{\tau_4} = \|t_2\|_{\tau_4}$.

7.2.3 Proving termination

The final step is proving termination. We illustrate the termination proof for the *Deeppermute* predicate only. Termination of *permute* and *delete* illustrate no additional problems. Using the derived call information, we can propose the following levelmapping to measure *Deeppermute* atoms:

$$|\text{Deeppermute}(t_1, t_2)| = \|t_1\|_{\tau_1} + \|t_2\|_{\tau_1}$$

Take any atom $\text{Deeppermute}(t_1, t_2) \in \text{Call}(\text{Deeppermute}, S)$, such that

$\theta = \text{mgu}(\text{Deeppermute}(t_1, t_2), \text{Deeppermute}([E_1|T_1], L_2))$ exists.

The goal is to prove that $|\text{Deeppermute}(t_1, t_2)| > |\text{Deeppermute}(T_1, T_2)\theta\sigma|$, for any σ such that $P \models (\text{permute}(E_1, E_2), \text{permute}(L_2, [E_2|T_2]))\theta\sigma$.

Then: $|\text{Deeppermute}(t_1, t_2)| = |\text{Deeppermute}(t_1, t_2)\theta| = \|t_1\theta\|_{\tau_1} + \|t_2\theta\|_{\tau_1} = 1 + \|T_1\theta\|_{\tau_1} + \|L_2\theta\|_{\tau_1} f$. The interargument relation derived for *permute*, gives us the information that $\|[E_2|T_2]\theta\sigma'\|_{\tau_4} = \|[E_2|T_2]\theta\sigma'\|_{\tau_4} = 1 + \|T_2\theta\sigma'\|_{\tau_4}$, for any σ' such that $[E_2|T_2]\theta\sigma' \in \mathcal{ID}(\tau_4)$ and $L_2\theta\sigma' \in \mathcal{ID}(\tau_4)$. This is the case for $\sigma' = \sigma$, the above substitution. Because our norms are rigid on their denotation, $\|L_2\theta\|_{\tau_4} = \|L_2\theta\sigma\|_{\tau_4}$. Thus $\|t_1\theta\|_{\tau_1} + \|t_2\theta\|_{\tau_1} > \|T_1\theta\sigma\|_{\tau_1} + \|T_2\theta\sigma\|_{\tau_1} = |\text{Deeppermute}(T_1\theta\sigma, T_2\theta\sigma)|$, which concludes our proof.

7.3 Occurs

The following predicate checks whether all elements in a list of lists occur in a second list. The example again illustrates the usefulness of Typed Norms. Using one semi-linear norm is not sufficient to prove left-termination.

$$\begin{array}{ll} \text{occurs}([], L) & \leftarrow \\ \text{occurs}([nil|T], L) & \leftarrow \text{occurs}(T, L). \\ \text{occurs}([H|T], L) & \leftarrow \begin{array}{l} \text{delete}(El, H, Hd), \\ \text{delete}(El, L, Ld), \\ \text{occurs}([Hd|T], L). \end{array} \\ \\ \text{delete}(El, [El|T], T) & \leftarrow \\ \text{delete}(El, [E|T_1], [E|T_2]) & \leftarrow \text{delete}(El, T_1, T_2). \end{array}$$

Let us try to prove left-termination for the set of queries $S = \mathcal{D}(\text{occurs}(\tau_1, \tau_4))$, where τ_1 and τ_4 are the types of Figure 3.

If one semi-linear norm were used, listlength is required to derive an interargument relation for delete, but the listlength norm is not able to show a reduction between two consecutive recursive occurs calls, using the third clause.

Type analysis gives the following call/success descriptions:

$$\begin{aligned} \text{occurs}(\tau_1, \tau_4) &\rightarrow \text{occurs}(\tau_1, \tau_4) \\ \text{delete}(Max, \tau_4, Max) &\rightarrow \text{delete}(Max, \tau_6, \tau_4) \end{aligned}$$

Feeding this information in Algorithm 5.25 gives the following well-typed program occurs^t :

$$\begin{aligned} \tau_1, \tau_4 \text{occurs}^{\tau_1, \tau_4}(L_1, L_2) &\leftarrow L_1 =^{\tau_2} \text{nil}. \\ \tau_1, \tau_4 \text{occurs}^{\tau_1, \tau_4}(L_1, L_2) &\leftarrow L_1 =^{\tau_3, \tau_2, \tau_1} [U|T], \\ &U =^{\tau_2} \text{nil}, \\ &\text{occurs}^{\tau_1, \tau_4}(T, L_2). \\ \tau_1, \tau_4 \text{occurs}^{\tau_1, \tau_4}(L_1, L_2) &\leftarrow L_1 =^{\tau_3, \tau_4, \tau_1} [H|T], \\ &\text{delete}^{Max, \tau_6, \tau_4}(El, H, Hd), \\ &\text{delete}^{Max, \tau_6, \tau_4}(El, L_2, L_d), \\ &L_s =^{\tau_3, \tau_4, \tau_1} [Hd|T], \\ &\text{occurs}^{\tau_1, \tau_4}(L_s, L_2). \\ \\ Max, \tau_4, Max \text{delete}^{Max, \tau_6, \tau_4}(X, Y, Z) &\leftarrow Y =^{\tau_6, \tau_7, \tau_4} [X|Z]. \\ Max, \tau_4, Max \text{delete}^{Max, \tau_6, \tau_4}(X, Y, Z) &\leftarrow Y =^{\tau_6, \tau_7, \tau_4} [R|S], \\ &\text{delete}^{Max, \tau_6, \tau_4}(X, S, T_s), \\ &Z =^{\tau_6, \tau_7, \tau_4} [R|T_s]. \end{aligned}$$

7.3.1 Matching

From occurs^t we can derive two different matching sets: $\{(\tau_1, \tau_2), (\tau_1, \tau_3), (\tau_4, \tau_6)\}$ and $\{(\tau_1, \tau_3), (\tau_4, \tau_6)\}$. The types in all couples are type/subtype related so that no matching conditions result.

The most immediate typed norm is the one obtained by considering as much information as possible, thus taking a and b constant 1, within the limitations of a being rigid. The following Typed Norm results:

$$\begin{aligned} \|t\|_{\tau_1} &= \|t\|_{\tau_3} \\ \|[t_1|t_2]\|_{\tau_3} &= 1 + \|t_1\|_{\tau_4} + \|t_2\|_{\tau_3} \\ \|t\|_{\tau_4} &= \|t\|_{\tau_6} \\ \|[t_1|t_2]\|_{\tau_6} &= 1 + \|t_2\|_{\tau_6} \\ \|t\|_{\tau_2} &= \|t\|_{\tau_5} = \|t\|_{\tau_7} = 0 \end{aligned}$$

7.3.2 Deriving interargument relations

With the norms derived above, we would like to derive a typed interargument relation for delete wrt (Max, τ_6, τ_4) . Because $\mathcal{I}_{\text{occurs}}^\alpha$ is defined on $\langle E_q^3 \rangle$, which consists of tuples

of one element, T_{occurs}^α reaches a fixpoint when T_{occurs}^α reaches one. In the following, we illustrate the behaviour of T_{occurs}^α . Let us number the clauses Cl_1 till Cl_5 .

$T_{occurs}^\alpha \uparrow 0 = \perp$.

$T_{occurs}^\alpha \uparrow 1$: Here we have to compute $T_{occurs}^\alpha(delete^{(Max, \tau_6, \tau_4)}/3, \perp)$. For Cl_4 , we obtain that $glb(\{Y = 1 + Z\})_{\{X, Y, Z\}} = \{Y = 1 + Z\}$. Cl_5 gives $glb(\{Y = 1 + S\}, \perp, \{Z = 1 + T_s\})_{\{X, Y, Z\}} = \perp$. $lub(\perp, \{Y = 1 + Z\}, \perp) = \{Y = 1 + Z\}$.

$T_{occurs}^\alpha \uparrow 2$: No new information is added through this iteration, so that $T_{occurs}^\alpha \uparrow 1$ is a fixpoint.

We have thus derived an interargument relation for delete wrt (Max, τ_6, τ_4) which expresses that, for any atom $delete(t_1, t_2, t_3)$ which is logically implied by $occurs$ and which is such that $t_1 \in \mathcal{D}(Max)$, $t_2 \in \mathcal{D}(\tau_6)$ and $t_3 \in \mathcal{D}(\tau_4)$, then $\|t_2\|_{\tau_6} = 1 + \|t_3\|_{\tau_4}$.

7.3.3 Proving termination

To prove termination wrt the above set of queries $S = \mathcal{D}(occurs(\tau_1, \tau_4))$, we can use the derived call type information to propose the following levelmapping:

$$\begin{aligned} |occurs(t_1, t_2)| &= \|t_1\|_{\tau_1} + \|t_2\|_{\tau_4} \\ |delete(t_1, t_2, t_3)| &= \|t_1\|_{Max} + \|t_2\|_{\tau_4} + \|t_3\|_{Max} = \|t_2\|_{\tau_4} \end{aligned}$$

We prove termination for the third clause Cl_3 .

We can take any atom $occurs(t_1, t_2) \in Call(occurs, S)$, where $\theta = mgu(occurs(t_1, t_2), occurs([H|T], L))$ exists and we must prove that for such an atom, $|occurs(t_1, t_2)| > |occurs([H_d|T], L)\theta\sigma|$, for any σ which is such that $occurs \models (delete(El, H, H_d), delete(El, L, L_d))\theta\sigma$. Making use of the interargument relation we derived for delete wrt (Max, τ_6, τ_4) , we may assume that $\|H\theta\sigma\|_{\tau_6} = 1 + \|H_d\theta\sigma\|_{\tau_4}$. Note that the correctness of the type analysis guarantees that the constraints on the types of the arguments for delete are indeed satisfied.

Then: $|occurs(t_1, t_2)\theta| = \|t_1\theta\|_{\tau_1} + \|t_2\theta\|_{\tau_4} = \|[H|T]\theta\|_{\tau_1} + \|L\theta\|_{\tau_4} = 1 + \|H\theta\|_{\tau_4} + \|T\theta\|_{\tau_1} + \|L\theta\|_{\tau_4} = 1 + \|H\theta\sigma\|_{\tau_4} + \|T\theta\sigma\|_{\tau_1} + \|L\theta\sigma\|_{\tau_4} = 1 + 1 + \|H_d\theta\sigma\|_{\tau_4} + \|T\theta\sigma\|_{\tau_1} + \|L\theta\sigma\|_{\tau_4} > 1 + \|H_d\theta\sigma\|_{\tau_4} + \|T\theta\sigma\|_{\tau_1} + \|L\theta\sigma\|_{\tau_4} = \|[H_d|T]\theta\sigma\|_{\tau_1} + \|L\theta\sigma\|_{\tau_4} = |occurs([H_d|T], L)\theta\sigma|$, which proves our claim.

A similar proof can be provided for the second occurs clause and for the permute predicate.

The overall conclusion is that occurs terminates wrt the set of queries $S = \mathcal{D}(occurs(\tau_1, \tau_4))$.

8 Extensions

8.1 Deriving one semi-linear norm from a set of typed norms

In a lot of cases, it will not be necessary to define multiple typed norms to obtain a valid termination proof. In this section we briefly present a method for these situations. It will be described when one semi-linear norm will be sufficient and it will be shown how to derive it, thus removing the need for the more complex schema of deriving typed size relations.

We first introduce the definition of the set of functor patterns $FP(\|\cdot\|_{\tau,a,b})$ which depends on the typed norm $\|\cdot\|_{\tau,a,b}$ induced by the type τ , the coefficient assignment a and the offset assignment b on τ . This set describes for every functor f/k all possible different ways in which a subterm $f(t_1, \dots, t_k)$ of $t \in \mathcal{D}(\tau)$ may be measured according to its position in the tree representation of t .

Definition 8.1 (Set of functor patterns)

Let T be a normal type with N its set of Nodes and τ the TypeLabel of its root. Let $\|\cdot\|_{\tau,a,b}$ be the norm induced by τ , the coefficient assignment a and the offset assignment b . The set of functor patterns $FP(\|\cdot\|_{\tau,a,b})$ associated to $\|\cdot\|_{\tau,a,b}$ is defined as:

$$FP(\|\cdot\|_{\tau,a,b}) = \{f < b(n), a(n, 1), \dots, a(n, k) > \mid \exists n \in N \text{ with } label(n) = f/k\}.$$

■

Proposition 8.2

Let $S = \{\|\cdot\|_{\tau_1, a_1, b_1}, \dots, \|\cdot\|_{\tau_n, a_n, b_n}\}$ be a set of typed norms and let all $a_i, 1 \leq i \leq n$, be rigid coefficient assignments. If for every functor symbol f/k there is at most one functor pattern $f < x_0, \dots, x_k > \in \cup_{i=1}^n FP(\|\cdot\|_{\tau_i, a_i, b_i})$, then the following semi-linear norm

$$\begin{aligned} \|f(t_1, \dots, t_k)\| &= x_0 + x_1 \|t_1\| + \dots + x_k \|t_k\| & \text{iff } f < x_0, \dots, x_k > \in \cup_{i=1}^n FP(\|\cdot\|_{\tau_i, a_i, b_i}) \\ \|t\| = 0 & & \text{otherwise} \end{aligned}$$

has the following property:

$$\forall i, 1 \leq i \leq n, \forall t \in \mathcal{D}(\tau_i) : \|t\|_{\tau_i, a_i, b_i} = \|t\|.$$

■

Proof

Let $Nodes = \cup_{1 \leq i \leq n} Nodes_i$ with $Nodes_i$ the set of nodes associated to the normal type with TypeLabel τ_i . We will prove a stronger claim: $\forall n \in Nodes, \forall t \in \mathcal{D}(n) : \|t\|_{\tau, a, b} = \|t\|$ where $TypeLabel(n) = \tau$.

We use structural induction on t to prove the claim.

base case.

t can only be a variable or a constant or primitive. Both norms measure such a term as zero or reduce the case to a sum of zero norms (OR case).

induction step.

Here, $t = f(t_1, \dots, t_n)$. Let n be a functor node. Then, by definition, $\|t\|_{\tau, a, b} = b(n) + \sum_{1 \leq i \leq k} a(n, i) \cdot \|t_i\|_{\tau, a, b}$. By construction, $\|t\| = x_0 + \sum_{1 \leq i \leq k} x_i \cdot \|t_i\|$. Because of our induction hypothesis, we know that $\forall i, 1 \leq i \leq k, \|t_i\|_{\tau, a, b} = \|t_i\|$. Because of the definition of the typed norm $\|\cdot\|_{\tau, a, b}$, $FP(\|\cdot\|_{\tau, a, b})$ contains the element $f < b(n), a(n, 1), \dots, a(n, k) >$. Moreover, this is the only element, which proves our claim. The case in which n is an OR-node reduces to the above functor case (this is because of normality and the definition of Typed Norm). ■

Moreover, Proposition 4.14 guarantees that all terms $t \in \mathcal{D}(\tau)$ are rigid wrt $\|\cdot\|$. In practice, it happens a lot that the set of natural typed norms induced by the relevant recursive types meet the condition of the previous proposition. A list of examples is the following: append, quick-sort, permute, treetolist, leaves, ...

Example 8.3 (flat)

Reconsider the flat program. Let a be the *rigid* coefficient assignment which is derived from the natural coefficient assignment (from now on, when talking about the natural coefficient assignment, we mean the rigid assignment). It is easy to check that each pair of types in the *Match* set for $flat^t$ satisfies the condition of Proposition 8.2. So all natural norms associated to these types are matching norms. ■

It is easy to see that one might use the semi-linear norm defined in Proposition 8.2 to prove termination of all these example programs.

An immediate question is then what happens when the condition of Proposition 8.2 does not apply. This happens when we consider all functor patterns of all relevant types and we discover that a functor symbol f/k has more than one occurrence. One could force this condition as proposed in section 3, but this does not always work. This is the point of the next subsection.

8.2 A sufficient condition for matching

In a lot of cases, it will not be necessary to apply the match algorithm, as the norms on which the algorithm must be applied are already matching. In this section we describe a sufficient (and easy to check) condition for two typed norms to be matching.

The following property is inspired by Proposition 8.2.

Property 8.4

Let a_1, a_2 be *rigid* coefficient assignments and b_1, b_2 offset assignments on the types τ_1 and τ_2 respectively and let $\|\cdot\|_{\tau_1, a_1, b_1}$ and $\|\cdot\|_{\tau_2, a_2, b_2}$ be two typed norms induced by τ_1, a_1, b_1 and τ_2, a_2, b_2 respectively. Then: $\|\cdot\|_{\tau_1, a_1, b_1}$ and $\|\cdot\|_{\tau_2, a_2, b_2}$ are matching iff for every functor symbol f/k there is at most one functor pattern $f < x_0, \dots, x_k > \in FP(\|\cdot\|_{\tau_1, a_1, b_1}) \cup FP(\|\cdot\|_{\tau_2, a_2, b_2})$. ■

Note how a second, apparently more simple Match algorithm is hidden in the property.

Definition 8.5 (FunctorNodes $FN(\|\cdot\|_{\tau, a, b})$)

Let T be a normal type with *TypeLabel* τ and let *Nodes* be its set of nodes. The set of FunctorNodes $FN(\|\cdot\|_{\tau, a, b})$ is the set $\{f/k < n > \mid n \in Nodes, Label(n) = f/k\}$. ■

Algorithm 8.6 ($E_q = Match'(\tau_1, \tau_2)$)

Let τ_1 and τ_2 be the *TypeLabels* of two normal types and let a_1, a_2 be (not necessarily rigid) coefficient assignments and b_1, b_2 be offset assignments on τ_1 and τ_2 respectively.

$Match'(\tau_1, \tau_2) =$

$$\begin{aligned} & \{b_1(n_1) = b_2(n_2) \mid f/k < n_1 > \in FN(\|\cdot\|_{\tau_1, a_1, b_1}), f/k < n_2 > \in FN(\|\cdot\|_{\tau_2, a_2, b_2})\} \\ & \cup \\ & \{a_1(n_1, i) = a_2(n_2, i) \mid f/k < n_1 > \in FN(\|\cdot\|_{\tau_1, a_1, b_1}), f/k < n_2 > \in FN(\|\cdot\|_{\tau_2, a_2, b_2}), \\ & \quad 1 \leq i \leq k\} \end{aligned}$$

■

It is however important to understand that the constraints resulting from this new condition are different from the result of the original match algorithm, as types are now

compared globally, while the *Match* algorithm takes into account the structure of the types.

As an example, consider the types of Figure 8 which must be matched. If we apply the original *Match* algorithm on these types, the following set of equalities will be proposed: $Match(\tau_1, \tau_2) = \{a_2(n_2, 1) = 0, a_1(n_1, 2) = 0\}$, where τ_1 and τ_2 are the left and right typegraph and n_1 and n_2 their respective root. The *Match'* algorithm will also impose these same constraints but another two of them in addition: $Match'(\tau_1, \tau_2) = \{a_2(n_2, 1) = 0, a_1(n_1, 2) = 0, a_1(n_3, 1) = 0, a_1(n_3, 2) = 0\}$, where n_3 is the node with the label $g/2$ of the left typegraph. Of course, these extra constraints will have their impact on the resulting class of norms and they may in the worst case lead to an unsolvable system of equations.

Although this second *Match'* algorithm seems much simpler, the resulting system of equations is *not*. As explained, this is because in general too much constraints are generated, quickly decreasing the precision degree of the resulting class of norms.

The Property 8.4 is very useful however. It can be applied to check whether for all couples of types in the *Match* set of a program, both types are matching. If they are, there is no need for calling the *Match* algorithm. Only when there is one couple in the *Match* set on which the sufficient condition is not satisfied, the matching algorithm must be called. Of course, it must be run on the complete *Match* set.

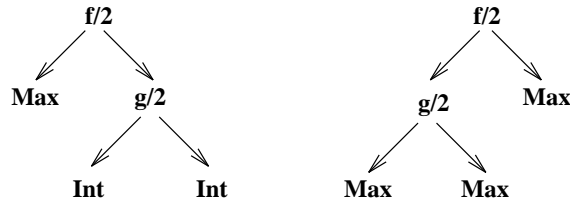


Figure 8: Matching should not bother about g -terms, as performed by *Match'*

8.3 Exploiting subtype relationship in unification

In this section, we observe a little closer the construction of the matching set. Recall Definition 5.29 where it was defined as a three component structure. If in some way or another it could be assumed that its second and third components were always satisfied, a lot of interdependency would have been eliminated. This can be achieved by imposing a type/subtype relationship on the annotation of a unification.

Intuitively, one would expect such a relationship to hold between the type of the variable on the lefthandside and the types corresponding to the variables of the righthandside in any case. In practice, this assumption can not always be made, as it depends among others on the way unification is labelled and on the precision of the type analysis and on the type analysis framework itself. Consider the unification $X = f(Y)$ and assume that X and Y are typed as on the lefthandside of Figure 9, i.e. before the unification X is typed Max and Y is instantiated to a term of the type $f(Int)$. The precise type for X after the unification would be $T_x = f(f(Int))$. However, to ensure working with a domain of finite abstract substitutions during type analysis, a depth restriction is imposed on all types. It states an upperbound on the number of functor symbols that may occur consecutively in a type. Figure 9 shows the effect of the restriction when this upperbound is taken to

be 1. This is also the upperbound in the framework of [29].

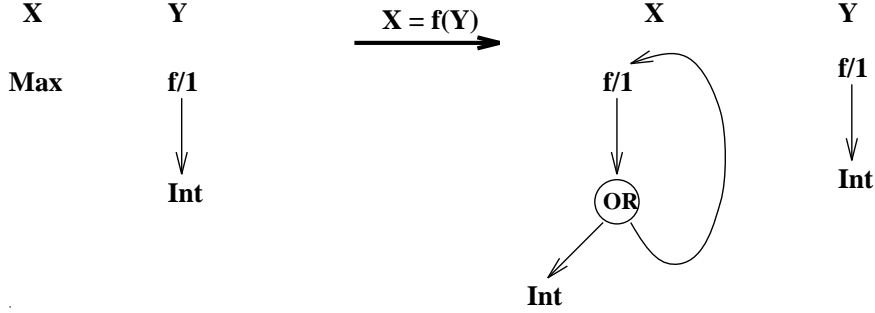


Figure 9: effect of depth restriction on type/subtype relationship

We first discuss what kind of type/subtype relationship could be beneficial for our purpose. We call such unification type-compact.

Definition 8.7 (type-compact unification)

Let $\tau_x, \tau_{y_1}, \dots, \tau_{y_n}$ be the TypeLabels of $n + 1$ normal types $T_x, T_{y_1}, \dots, T_{y_n}$ and let $Nodes$ be the set of nodes associated with T_x . A unification $X = f(Y_1, \dots, Y_n)^{\tau_x, \tau_{y_1}, \dots, \tau_{y_n}}$ is *type-compact* iff $\forall i, 1 \leq i \leq n, \exists n_i \in Nodes, TypeLabel(n_i) = \tau_{y_i}$. A unification $X = Y^{\tau_x, \tau_y}$ is *type-compact* iff $\tau_x = \tau_y$. ■

Example 8.8

Reconsider the *flat^t* example. All unifications are type-compact. ■

This notion is now introduced into well-typed programs.

Definition 8.9 (type-compact well-typed programs)

A well-typed program P^t is *type-compact* if every unification occurring in it is type-compact. ■

The gain from working with type-compact unification is obvious as unification is labelled with parts of the *same* typegraph. Previously, the set of coefficients occurring in the types of the left- and the righthandside of unification, could be totally different. With this type/subtype relationship, all constraints generated by the Match algorithm will reduce to trivial constraints of the form $a(n, i) = a(n, i)$. Thus working with type-compact unification has the advantage that *no* match relations must be imposed on the types. As a result, the set of matching constraints will now be smaller.

The new definition of the matching set, adapted to programs satisfying the type-compactness property becomes very simple.

Definition 8.10 (matching set $Match(P^t)$)

$(\tau_1, \tau_2) \in Match(P^t)$ iff $(X, \tau_1) \in TV(P^t), (X, \tau_2) \in TV(P^t)$ and $\tau_1 \neq \tau_2$. ■

Type-compact programs can be constructed without much of a problem. Suppose we have a clause C where a unification $X = f(Y_1, \dots, Y_n)$ must be labelled. We assume that the results of a type analysis as described in [29] are available. Let τ_x be the local success type of X for the clause C . Where previously the variables Y_i were labelled by their own local successtype, we now simply discard this information and we derive their local successtype from the type τ_x .

Algorithm 8.11 (labelling unification)

There are two cases, corresponding to the two types of unification. Let C be the clause in which the unification occurs.

- $X = f(Y_1, \dots, Y_n)$.
Let τ_x be the local success type of X in C and let n_x be the node whose `TypeLabel` is τ_x . If $\nexists n \in \text{Principal}(n_x), \text{Label}(n) = f/k$, the algorithm fails. Else, the atom is labelled by $\tau_x, \tau_{y_1}, \dots, \tau_{y_n}$, where $\tau_{y_i} = \text{TypeLabel}(n_x/i)$.
- $X = Y$.
Let τ_x be the local success type of X in C . Then the unification is labelled by τ_x, τ_x . ■

Proposition 8.12

All unifications which are labelled by Algorithm 8.11 are type-compact. ■

Proof

This is true by construction. ■

Proposition 8.13

Let P be a normal program and let P^t be a program obtained from P by performing the above labelling. Then P^t is well-typed. ■

The following proposition shows that Algorithm 8.11 never fails if the type-inference mechanism of [29] is used. Thus, it is always possible to construct a type-compact well-typed program from a normal program.

Proposition 8.14

Let C be a clause in a program P and let $X = f(Y_1, \dots, Y_n)$ be an atom of it. Let τ_x be the local success type of X as provided by the framework of [29] Then: τ_x is of the form $f(\tau_1, \dots, \tau_k)$. ■

Proof

Due to the abstract interpretation of the unification, the success-type of X will be of the form $f(\tau_1, \dots, \tau_n)$, for some τ_1, \dots, τ_n . This is the case after the interpretation of the unification. Let us now look what happens when other atoms are abstractly executed. Let us look at the abstract interpretation of a user-defined atom wrt some input substitution: After the abstract interpretation of the atom, resulting in an output type for all variables of the domain of the input substitution, in [29], an intersection operation between all input and output types is effected. The output type for T_x could be one of six cases. (1) this output type is *Max* or (2) it is a node with Label f/k or (3) it is a node with Label g/l , $f/k \neq g/l$ or (4) it is an OR-type having an f/k node as one of its principal nodes or (5) it is an OR-type having no f/k node as one of its principal nodes or (6) it is a primitive type. Obviously, the intersection is \perp in the cases (3), (5) and (6). In the other cases, the type of the intersection is $f(\tau'_1, \dots, \tau'_n)$. Thus when the last atom in the clause is interpreted in this fashion, the type of X is still an f/k -type (a type having as root a node with label f/k). The local success type is found by taking the upperbound of all such end-of-this-clause types for X . However, the upperbound of f/k -types is still an f/k -type. ■

The main result of this proposition is that we can always apply our algorithm to label unification.

9 Discussion

The contribution in this article is the automatic inference of suitable norms for termination analysis. The paper also fills in the technical gap needed to extend automatic termination analysis techniques such as the one in [44] to be able to take advantage of typed norms: the inference of typed interargument relations. In previous contributions to termination analysis, norms were assumed to be fixed or specified by a user. Using automatic type inference, we automatically supply suitable norms that measure as much as possible the information available in "input arguments". We applied both the semi-linear norm of Section 3 and the typed norms of Section 4 to all examples presented in the termination analysis literature. Results are quite convincing, since with both approaches we succeed in producing the "right" norm(s) in all examples discussed in automatic termination analysis and most examples proposed in non-automatic termination analysis. Among the few failing examples we registered are the Game example of [3], which relies on non-syntactical information, the Bubblesort of [10] and the Check program proposed in [8]. In [8] a class of norms larger than the semi-linear one is proposed (also called 'typed norms', which we denote between quotes to avoid confusion) to deal with some limitations of semi-linear norms. We can not treat this last example due to limitations on the expressiveness of our type graphs, imposed for efficiency reason (normalisation). We stress however that the 'typed norms' of Bossi et al. are *not* inferred automatically, but they must be supplied. In spite of this, our definitions cover most of their examples automatically.

With respect to related work on inference of interargument relations, we refer to [16] for an overview. That paper also discusses the differences in expressivity. With respect to [18], the main differences are the improved expressivity by integrating types and the use of a conceptually simpler abstract interpretation scheme. For obvious reasons (being an extension of the technique in [18]), this technique shares the same disadvantages of [18]. Being restricted to linear relations is one of them. As is discussed there, in some cases a relation of a more expressive nature might be required. The framework of [18] was based itself on ideas in [20] and [21] in the context of estimating the complexity of a program. There, the linearity constraint is omitted and general inequalities are allowed. In [12], Brodsky and Sagiv present a bottom-up technique for deriving disjunctions of conjunctions of basic inequalities between two arguments.

A first attempt to derive typed interargument relations in the context of typed norms, was done in [23]. There, the problem is solved through the introduction of so-called *consistently type-annotated programs*. In such programs, every atom is annotated by a tuple of safe and correct success types. The power of the approach is enforced through a 'consistency property', which says that whenever a variable occurs in two different atoms of a clause, it must be labelled with the same type. It is not difficult to see that for such programs no matches are necessary. To enforce such a property however, atoms with the same predicate symbol, may be labelled differently. For each differently typed predicate a separate interargument relation is derived, with respect to its annotation.

The main advantage of this approach is obviously one of precision as interargument relations are derived which mirror local needs. However this solution also suffers from severe shortcomings, which make it unacceptable in practice. First of all, the transformation and derivation process call for an overwhelming amount of complex and technical gluing. Moreover, a lot of pre-processing is needed for deriving the annotated programs

in a consistent way. Another problematic issue is the apparent need for several abstract interpretation phases in combination with the need for re-execution to obtain acceptable results. Finally, it remains an open question how to *automatically* derive consistently type-annotated programs. We refer to [23] for a deeper discussion on these topics.

This is not the first paper to introduce types in termination analysis. [45] first used type inference as a replacement for mode analysis in the context of termination of non-ground queries. [44] and [46] have further elaborated on this extension.

Another approach in which types play a central role is the one proposed by [13]. This paper introduces types to allow the incorporation of syntactic term-orderings (as an alternative to norms) in LP-termination analysis. It is not clear to what extent this framework has been or can be automated. The introduction of syntactic orderings is new and interesting, but they still need to compute interargument relationships using (in general) norms to measure terms and the usual order on natural numbers. Hence, the mixing of orders might seem a complication. The type system proposed in [13] is more similar to the integrated types of [30] than to rigid types. This increases expressibility to some extent, but on the other hand it implies the loss of invariance under substitution, which is a useful property in the context of the current application.

A prototype implementation has been developed at our site. It is written in Prolog with minor parts written in Fortran (the primitive operations on the elements of the abstract domain for the derivation of interargument relations). We have tested the tool on a lot of examples and we give a survey of its performance on a few selected ones. All examples are either drawn from [44] or from this paper. We do not recall the call types here, but wherever possible, we have chosen to add as much genericity as possible.

The following examples appear in the tests: vectorsum, append, insertion sort, delete, duplicate, palindrome, permute using delete, naive reverse, expand, occurs, heapsort and mergesort.

The first table shows the time spent on proving termination for each example. As should be clear from this paper, the complete termination analysis consists of four distinct phases: a type analysis phase, a matching phase, a third part in which interargument relations are derived and the actual termination proof itself. We have collected timings for each of these four phases. In the second table, we compare the heap consumption of each phase.

Obviously, the type analysis phase is the most time consuming part. This is not surprising as not only a complete AND/OR tree structure must be constructed from the top-level call but also several geometrical operations must be applied on typegraphs. Note however that time increases with respect to the number of variables occurring. Notable exceptions are insertsort (simple recursive structure in combination with the call type 'list of integers' which has not much genericity) and expand (we had to increase the depth restriction to be able to express the call type). Remark also that for any two programs, their difference in CPU-usage is directly related to their difference in heap usage. Similar as in [18], in this implementation, the time divergence is caused indirectly by the memory usage. The abstract substitutions have a matrix representation where each variable occupies one column. Moreover, one of the prerequisites for the analysis is that the program must be in normal form, and such a transformation process spawns a lot of extra variables. Being the major time consuming phase however opens additional perspectives for using the analysis in recent, typed languages like Gödel and Mercury.

The same observation holds for the derivation of interargument relations. Again

the program must be in normal form and the abstract substitutions are matrices whose columns are related to the number of variables. This phase puts a smaller hold on the heap however because it is done bottom-up.

At first sight, the matching phase might be considered to put a severe constraint on CPU-usage. The current implementation is however responsible for this behaviour. In the current type analysis tool, in order to restrict the number of geometrical operations, it happens a lot that same types have a different representation and thus are not recognised as being the same. This has a drastic effect on the matching phase, as whenever two types are different (in representation), a match on them is performed. Currently, we use a very simple check to detect some of these situations. We believe however, as this is an implementation issue, that this problem should be solved in the type analysis tool. Comparing the matching phase to the other phases, it should be noticed that the time consumption scales much better with respect to program size. This is already the effect of adding the simple check. This effect should be more visible when a complete check is performed. The reason is that only a finite number of different types occur in a program. Extending a program is not likely to add lots of new types.

<i>program</i>	<i>clauses/ vars</i>	<i>type analysis</i>	<i>matching</i>	<i>size relations</i>	<i>termination</i>
vectorsum(A,B,C)	2/12	8.116	6.049	8.316	6.099
append(A,B,C)	2/10	9.233	6.100	7.383	5.866
insertsort1(A,B)	4/19	9.700	6.133	7.166	5.349
delete(A,B,C)	2/12	10.050	7.116	6.900	5.666
duplicate(A,B)	2/8	10.366	6.083	7.549	5.966
palindrome(A,B)	3/18	11.916	7.316	11.900	5.816
permute(A,B)	4/22	13.400	6.333	9.150	5.783
nreverse(A,B)	4/19	13.866	6.416	9.550	5.766
occurs(A,B)	5/23	27.149	7.866	7.150	5.283
expand(A,B)	3/17	33.583	9.400	9.733	5.866
heapsort(A,B)	13/78	38.016	28.633	24.550	4.666
mergesort(A,B)	14/85	169.083	16.033	52.449	4.866

Table 1: CPU-usage

<i>program</i>	<i>clauses/ vars</i>	<i>type analysis</i>	<i>matching</i>	<i>size relations</i>	<i>termination</i>
vectorsum(A,B,C)	2/12	134617	10282	93527	23072
append(A,B,C)	2/10	120510	9389	56181	6148
insertsort1(A,B)	4/19	222485	16502	59887	1616
delete(A,B,C)	2/12	213855	22893	52037	12472
duplicate(A,B)	2/8	150351	11726	60032	10755
palindrome(A,B)	3/18	275941	26374	216447	20746
permute(A,B)	4/22	342607	20909	131955	26501
nreverse(A,B)	4/19	327620	19395	134351	18609
occurs(A,B)	5/23	753291	35888	62312	7511
expand(A,B)	3/17	951122	52131	156094	40597
heapsort(A,B)	13/78	1710593	219400	657765	42119
mergesort(A,B)	14/85	4192236	144386	170141	29384

Table 2: Heap-usage

As a final comment, we point out some of the remaining problems. As has been described in the previous sections, our technique does not derive one single (set of) typed norm(s) but instead it produces a whole class of such norms, which for the purpose of termination are all equivalent: all of them are rigid wrt their type and all of them preserve some kind of consistency required for the well-foundedness of the termination proof. Each member of the class is identified through the selection of an argument and offset mapping satisfying the matching conditions. However, as discussed in the introduction, such a choice might be vital for obtaining a termination proof. Consider a reverse program (with accumulating parameter) such that the accumulating parameter is merged together with the input-list into one argument position. Say :

$$\begin{aligned} rev(f([H|T], Acc), R) &: -rev(f(T, [H|Acc]), R). \\ rev(f(Nil, Acc), Acc) &. \end{aligned}$$

The call type for the first, "input" argument is $T = f(\tau_1, \tau_1)$ where τ_1 is the type of Figure 3. Using our technique to derive norms, two of the proposed norms are the following:

$$\begin{aligned} \|f(t_1, t_2)\|_{\tau} &= \|t_1\|_{\tau_1} + \|t_2\|_{\tau_1} \\ \|f(t_1, t_2)\|_{\tau} &= \|t_1\|_{\tau_1} \end{aligned}$$

with $\|\cdot\|_{\tau_1}$ being the list-length norm. However, with the first norm, termination cannot be proved, because the list-length of the second argument of the f -functor will be taken into account. Since this is the - growing - accumulating parameter, we find no decrease for this norm. Termination can be proved by using the second norm, which excludes this parameter.

Using the full power of these norms involves a search problem : find the (boolean) values for the argument and offset mappings for which a termination proof can be constructed. Linear programming techniques – as used for symbolic level mappings in [41] – cannot be employed here, due to the fact that with a symbolic norm, computing the interargument relations with any reasonable degree of precision becomes unfeasible. One

alternative is exhaustive search with backtracking over all possible assignments for the booleans. This may become very expensive, but 1) it will produce the appropriate norm(s) if one (some) exist, 2) from the large set of examples we considered so far, the search normally stops at the first assignment (all $a(n, i) = 1$ and $b(n) = 1$).

Acknowledgements

Stefaan Decorte is supported by GOA, "Non-standard applications of abstract interpretation", DPWB, Belgium. Danny De Schreye is a senior research associate of the Belgian National Fund for Scientific Research. Massimo Fabris is partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant 89.00026.69.

The authors wish to thank Yehoshua Sagiv and Naomi Lindenstrauss for interesting discussions on the topic. We are also grateful towards several anonymous referees of [22].

References

- [1] K.R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9:335–363, 1991.
- [2] K.R. Apt and D. Pedreschi. Studies in pure Prolog: termination. In *Proceedings Esprit symposium on computational logic*, pages 150–176, Brussels, November 1990. Springer-Verlag.
- [3] K.R. Apt and D. Pedreschi. Proving termination of general Prolog programs. In *Proc. International Conference on Theoretical Aspects of Computer Science*, Sendai, Japan, 1991.
- [4] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1991.
- [5] M. Baudinet. Proving termination properties of Prolog programs: a semantic approach. *Journal Logic Programming*, 14:1–29, 1992.
- [6] M. Bezem. Characterizing termination of logic programs with level mappings. *Journal of Logic Programming*, 15(1 & 2):79–98, 1992.
- [7] A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In *Proc. CCPSD-TAPSOFT'91*, pages 153–180. Springer-Verlag, LNCS 494, 1991.
- [8] A. Bossi, N. Cocco, and M. Fabris. Typed norms. In B. Krieg-Brueckner, editor, *Proc. ESOP'92*, pages 73–92. Springer-Verlag, LNCS 582, 1992.
- [9] A. Bossi, N. Cocco, and M. Fabris. Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science*, 124(2):297–328, 1994.
- [10] I. Bratko. *Prolog programming for artificial intelligence*. Addison-Wesley, 1986.

- [11] A. Brodsky and Y. Sagiv. On termination of Datalog programs. In *First International Conference on Deductive and Object Oriented Databases*, pages 95–112, Kyoto, Japan, 1989.
- [12] A. Brodsky and Y. Sagiv. Inference of inequality constraints in logic programs. In *Proceedings of the 10th ACM SIGACT-SIGMOD-SIGART symposium on principles of Database Systems*, pages 227–240, Denver, Colorado, 1991.
- [13] F. Bronsard, T.K. Lakshman, and U.S. Reddy. A framework of directionality for proving termination of logic programs. In Krzysztof Apt, editor, *Proc. JICSLP '92*, pages 321–335. MIT Press, 1992.
- [14] M. Bruynooghe and D. Boulanger. Abstract interpretation for (constraint) logic programming. In J. Penjam B. Mayoh, E. Tyugu, editor, *Constraint Programming*, volume F/131 of *NATO ASI Series*, pages 228–258. Springer-Verlag, 1994.
- [15] P. Cousot and R. Cousot. Abstract interpretation and application to logic programming. *Journal of Logic Programming*, 13(2 & 3):103–180, 1992.
- [16] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19(20):199–260, 1994.
- [17] D. De Schreye and K. Verschaetse. Termination analysis of definite logic programs with respect to call patterns. Technical Report CW 138, Department Computer Science, K.U.Leuven, January 1992.
- [18] D. De Schreye and K. Verschaetse. Deriving linear size relations for logic programs by abstract interpretation. *New Generation Computing*, 13(2):117–154, 1995.
- [19] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proc. FGCS'92*, pages 481–488, ICOT Tokyo, 1992. ICOT.
- [20] S.K. Debray and N.-W. Lin. Automatic complexity analysis of logic programs. In *Proceedings ICLP'91*, pages 599–613, June 1991.
- [21] S.K. Debray, N.-W. Lin, and M. Hermenegildo. Task granularity analysis in logic programs. In *Proceedings ACM SIGPLAN'90 conference on programming language design and implementation*, pages 174–188, June 1990.
- [22] S. Decorte, D. De Schreye, and M. Fabris. Automatic inference of norms : a missing link in automatic termination analysis. In D. Miller, editor, *Proceedings ILPS'93*, pages 420–436, Vancouver, Canada, 1993.
- [23] S. Decorte, D. De Schreye, and M. Fabris. Exploiting the power of typed norms in automatic inference of interargument relations. Technical report, Department of Computer Science, K.U.Leuven, Belgium, 1994.
- [24] S. Decorte, D. De Schreye, and M. Fabris. Integrating types in termination analysis. Technical Report 222, Department Computer Science, K.U.Leuven, Belgium, Januari 1996.

- [25] Veroniek Dumortier. *Freeness and Related Analyses of Constraint Logic Programs using Abstract Interpretation*. PhD thesis, K.U.Leuven, Dept. of Computer Science, October 1994.
- [26] Veroniek Dumortier and Gerda Janssens. Towards a practical full mode inference system for CLP(H,N). In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 569–583, Italy, June 1994. MIT Press.
- [27] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
- [28] Maria García de la Banda and Manuel Hermenegildo. A Practical Approach to the Global Analysis of CLP Programs. In Dale Miller, editor, *Proceedings of the 10th International Logic Programming Symposium*, pages 437–455, Vancouver, Canada, October 1993. MIT Press.
- [29] G. Janssens. *Deriving run time properties of logic programs by means of abstract interpretation*. PhD thesis, Department of Computer Science, K.U.Leuven, 1990.
- [30] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992.
- [31] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
- [32] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 1987.
- [33] A. Mulkers. *Deriving Live Data Structures in Logic Programs by means of Abstract Interpretation*. Number 675 in LNCS. Springer-Verlag, 1993.
- [34] L. Plümer. *Termination proofs for logic programs*. Number 446 in LNAI. Springer-Verlag, 1990.
- [35] L. Plümer. Termination proofs for logic programs based on predicate inequalities. In *Proceedings ICLP'90*, pages 634–648, Jerusalem, June 1990. MIT Press.
- [36] L. Plümer. Automatic termination proofs for Prolog programs operating on non-ground terms. In *Proc. ILPS'91*, pages 503–517, San Diego, October 1991. MIT Press.
- [37] M.R.K. Krishna Rao, D. Kapur, and R. K. Shyamasundar. A transformational methodology for proving termination of logic programs. In *Proc. CSL'91, LNCS 626*. Springer, 1991.
- [38] M.R.K. Krishna Rao, P.K. Pandya, and R. K. Shyamasundar. Verification tools in the development of provably correct compilers. In *Proc. 5th symp. on formal methods europe, FME'93*, 1993. To appear.

- [39] Y. Sagiv. A termination test for logic programs. In *Proceedings ILPS'91*, pages 518–532, San Diego, 1991. MIT Press.
- [40] R. K. Shyamasundar, M.R.K. Krishna Rao, and D. Kapur. Rewriting concepts in the study of termination of logic programs. In *proc. ALPUK '92*, London, April 1992.
- [41] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Proceedings 10th symposium on principles of database systems*, pages 216–226. ACM Press, May 1991.
- [42] J.D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal ACM*, 35(2):345–373, April 1988.
- [43] T. Vasak and J. Potter. Characterisation of terminating logic programs. In *Proceedings 1986 symposium on logic programming*, pages 140–147, Salt Lake City, 1986.
- [44] K. Verschaeetse. *Static Termination Analysis for Definite Horn Clause Programs*. PhD thesis, Dept. Computer Science, K.U.Leuven, 1992.
- [45] K. Verschaeetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. In *Proc. ICLP'91*, pages 301–315, Paris, June 1991. MIT Press.
- [46] K. Verschaeetse, S. Decorte, and D. De Schreye. Automatic termination analysis. In *Proc. LOPSTR'92, LNCS*. Springer-Verlag, 1993.
- [47] B. Wang and R. K. Shyamasundar. Towards a characterization of termination of logic programs. In *Proc. PLILP'90*, number 456 in LNCS, pages 204–221, Linköping, August 1990. Springer-Verlag.
- [48] B. Wang and R. K. Shyamasundar. Methodology for proving the termination of logic programs. In *Proceedings STACS'91*, number 480 in LNCS, pages 214–227, Hamburg, Germany, 1991.
- [49] B. Wang and R.K. Shyamasundar. Proving termination of logic programs. In R. Narasimhan, editor, *Perspective in theoretical computer science, Commemorative Volume*, pages 380–397. World Scientific Publishers, 1989.