A Transformational Approach to Resource Analysis with Typed-Norms [⋆]

Elvira Albert[1], Samir Genaim[1], and Raúl Gutiérrez[2]

[1] Complutense University of Madrid, Spain
[2] DSIC, Universitat Politècnica de València,
Camino de Vera S/N, 46022 Valencia, Spain

Abstract. In order to automatically infer the resource consumption of programs, analyzers track how *data sizes* change along a program's execution. Typically, analyzers measure the sizes of data by applying *norms* which are mappings from data to natural numbers that represent the sizes of the corresponding data. When norms are defined by taking type information into account, they are named *typed-norms*. The main contribution of this paper is a transformational approach to resource analysis with typed-norms. The analysis is based on a transformation of the program into an *intermediate abstract program* in which each variable is abstracted with respect to all considered norms which are valid for its type. We also sketch a simple analysis that can be used to automatically infer the required, useful, typed-norms from programs.

1 Introduction

Automated resource analysis [17] needs to infer how the sizes of data are modified along a program's execution. Size is measured using so-called norms [5] which define how the size of a term is computed. Examples of norms are *list-length* which counts the number of elements of a list, *tree-depth* which counts the depth of a tree, *term-size* which counts the number of constructors, etc. Basically, in order to infer the resource consumption of executing a loop that traverses a data-structure, the analyzer tries to infer how the size of such data-structure decreases at each iteration w.r.t. the chosen norm. Given a tree t, using a term-size norm, we infer that a function like "def Int foo(Tree t)= case t {Leaf ↦0; Node(l,r) ↦ 1+foo(r);}" performs at most nodes(t) iterations, where function nodes returns the number of nodes in the tree. This is because size analysis infers that at each recursive call nodes(t) decreases. However, by using the tree-depth norm, we will infer that depth(t) is an upper bound on the number of iterations. The latter is obviously more precise than the former bound as depth(t)≤nodes(t).

[⋆] This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu) and by the Spanish projects TIN2008-05624 and TIN2012-38137. Raúl Gutiérrez is also partially supported by a Juan de la Cierva Fellowship from the Spanish MINECO, ref. JCI-2012-13528.

The last two decades have witnessed a wealth of research on using norms in termination analysis, especially in the context of logic programming [5, 6, 9]. Early work pointed out that the choice of norm affects the precision such that the analyzer may only succeed to prove termination if a certain norm is used, while it cannot prove it with others. Later on, there has been further investigation on applying multiple norms, i.e., using two or more norms by applying them simultaneously [5]. This means that the same data in the original program is replaced by two or more abstract data each one specifying its size information w.r.t. the corresponding norm. Even a further step has been taken on using *typed-norms* which allow defining norms based on type information (namely on recursive types) [6]. Inferring norms from type information makes sense as recursive types represent recursive data-structures and thus, in termination analysis, they identify some potential sources of infinite recursion and, in resource analysis, they might influence the number of iterations that the loops perform. Besides, typed-norms allow that the same term can be measured differently depending on its type. As pointed out in [9], this is particularly useful when the same function symbol may occur in different type contexts.

In the context of resource analysis, we found early work that already pointed out that the combination of norms affects the precision of lower-bound time analysis [11]. Sized-types provide a way to consider more than one norm for each type. They have been used in the context of functional [15, 16] and recently in logic programming [12]. In the former case, they are inferred by a type analysis and in the latter via abstract interpretation. In contrast, we propose a transformational approach which provides a simple and accurate way to use multiple typed-norms in resource analysis as follows: (1) we first transform the program into an *intermediate abstract program* in which each variable is abstracted with respect to all considered norms valid for its type, (2) such intermediate program is then analyzed to obtain upper and lower resource bounds automatically. Importantly, this second phase is done using existing techniques that do not need to be modified. Thus, formalizing our framework focuses only on the first step.

While allowing multiple norms might lead to more accurate bounds than adopting one norm, the efficiency of the analysis can be degraded considerably. This is because the process of finding resource bounds from abstractions that have more arguments (due to the use of multiple norms) is more costly. Thus, an essential aspect for the practical applicability of our method is to eliminate those abstractions that will not lead to further precision. As our second contribution, we outline an algorithm for the inference of typed-norms which, by inspecting the program, can detect which norms are useful to later infer the resource consumption, and discard norms that are useless for this purpose. This analysis is applied as a pre-process, such that once the relevant norms are inferred, the transformation into the abstract program is carried out w.r.t. the inferred norms.

Syntactic categories.	*Definitions.*
T in Ground Type	$T ::= B \mid D$
B in Basic Type	$B ::= \mathsf{Int} \mid \mathsf{String}$
D in Data type	$Dd ::= \mathtt{data}\ D = Cons[\overline{\mid Cons}];$
x in Variable	$Cons ::= Co[(\overline{T})]$
e in Expression	$F ::= \mathtt{def}\ T\ fn(\overline{T\ x}) = e;$
t in Ground Term	$e ::= x \mid t \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \mathtt{case}\ e\ \{\overline{br}\}$
br in Branch	$t ::= n \mid Co[(\bar{t})]$
p in Pattern	$br ::= p \Rightarrow e;$
n in Integer	$p ::= \_ \mid x \mid t \mid Co[(\overline{p})]$

Fig. 1. Syntax for the functional level. Terms $\overline{T}$ and $\overline{e}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements.

2 The Language

We present the simple functional language on which our framework is defined. It corresponds to the functional sublanguage of ABS [10], a modeling language for concurrent distributed systems which has been used to implement two industrial case studies (both of them of more than 1,000 lines of code). The functional sublanguage of ABS is used to define and manipulate the data structures used in the program, while the imperative sublanguage is used to handle its concurrency and distribution aspects. The reason why we chose ABS is double: first, because the funcional part of the language is appropriate to present our results in a clear and simple manner; and second, because our final goal is to integrate typed-norms in the complexity analysis of concurrent and distributed systems modeled in ABS. Sec. 2.1 defines the syntax of our functional language, and Sec. 2.2 introduces the intermediate form to which the programs are translated to define the analysis later.

2.1 A Simple Functional Language

The language defines data types and functions, as shown in Fig. 1. Ground types T consist of basic types B as well as names D for data types. In data type declarations Dd, a data type D has at least one constructor $Cons$, which has a name Co and a list of ground types T for its arguments. Function declarations F consist of a return type T, a function name fn, a list of variable declarations $\overline{x}$ of types $\overline{T}$, and an expression e. *Expressions* e include variables x, (ground) terms t, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$ and case expressions $\mathtt{case}\ e\ \{\overline{br}\}$. Ground terms t are integer numbers and constructors applied to ground terms $Co(\bar{t})$. Case expressions have a list of branches $p \Rightarrow e$, where p is a pattern. The branches are evaluated in the listed order. Patterns include wild cards $\_$, variables x, terms t, and constructor patterns $Co(\overline{p})$. Abusing notation, fn in e can be a function name or a built-in function $(+, -, >, =, \geq)$. We assume that the considered programs are well-typed and unambiguous.

```
module Library;
type Author = String ;
type Title = String ;
data Authors = Nil
    | Cons(Author,Authors);
data Titles = Nil
    | Cons(Title,Titles);
data Book=Pair(Title,Authors);
data Books = EmptyMap
    | InsertAssoc(Book,Books);
data Ref = Pair(Author,Titles);
data Refs = EmptyMap
    | InsertAssoc(Ref,Refs);
```

```
def Int is_coauthor(Author a,Authors as)
 = case as {
    Nil => 0;
    Cons(a,as‘) => 1;
    Cons(a‘,as‘)=> is_coauthor(a, as‘); };

def Titles written_by(String a,Books bs)
 = case bs {
    EmptyMap => Nil;
    InsertAssoc(b,bs‘)
     => case b {
          Pair(t,as)
           => case is_coauthor(a, as) {
                1 => Cons(t,written_by(a,bs‘));
                0 => written_by(a,bs‘); }; }; };
```

```
def Refs sort_books_by_author(Authors as,Books bs)
= case as {
    Nil => EmptyMap;
    Cons(a,as‘)
     => InsertAssoc(Pair(a,written_by(a,bs)),sort_books_by_author(as‘, bs)); };
```

Fig. 2. Motivating example (data type declarations and three functions)

Example 1. Our running example is showed in Fig. 2. It defines a function sort_books_by_author (and several auxiliary functions) for sorting books by author given a list of authors and a list of books.

2.2 Intermediate Form

From now on, we develop our analysis on a typed intermediate representation (IR) similar to those defined in [2, 8, 13, 14]. The translation from our simple functional language to the IR is straightforward and follows exactly the same steps as the one formalized in [2]. Essentially, the IR of each function is obtained by translating each basic block in its control flow graph (CFG) into a procedure, defined by means of *rules* that adhere to the following grammar:

$r ::= m(\bar{x}, y) \mapsto g, b_1, \ldots, b_n.$
$b ::= x{:=}t \mid\ m(\bar{x}, y)$
$g ::= \mathit{true} \mid g \wedge g \mid e\ \mathit{op}\ e \mid \mathit{match}(x, t) \mid \mathit{nonmatch}(x, t)$
$t ::= e \mid \mathit{Co}(\bar{t})$
$e ::= x \mid n \mid e{+}e \mid e{-}e$

where $\mathit{op} \in \{>, =, \geq\}$, $m(\bar{x}, \bar{y})$ is the *head* of the rule, g specifies the conditions for the rule to be applicable and $b_1, \ldots, b_n$ is the rule's *body*. Calls are of the form $m(\bar{x}, y)$ where the variables $\bar{x}$ are the properly typed formal parameters and the variable y is the properly typed return value. Guards $\mathit{match}(x, t)$ and $\mathit{nonmatch}(x, t)$ simulate `case`-expressions and x and t are of the same type. We assume $x \notin vars(t)$. Terms are constructed using $Co(\bar{t})$, where Co is a data symbol and $\bar{t}$ are the arguments (e.g., Cons(x, y)), variables x, integer numbers n and arithmetic expressions ($e + e$ and $e - e$). A function is thus defined by a (global) set of rules. The dynamics of the data-structures are preserved by using the guard *match*, which fixes the shape of the input variables in the rules.

Example 2. Fig. 3 shows the IR of function is_coauthor. For each function definition, we have a rule with the same number of arguments plus a new argument at the end that represents the output of the function call. The case expression is split into three new rules, one rule for each possible matching alternative.

def Int is_coauthor(Author a,
Authors as)
  = **case** as {
    Nil => 0;
    Cons(a,as') => 1;
    Cons(a',as')
    => is_coauthor(a, as');
  };
	is_coauthor(a,as,y) $\mapsto$ case$_0$(a,as,y).
	case$_0$(a,as,y) $\mapsto$ match(as,Nil),
	y := 0.
	case$_0$(a,as,y) $\mapsto$ nonmatch(as,Nil),
	match(as,Cons(a,as')),
	y := 1.
	case$_0$(a,as,y) $\mapsto$ nonmatch(as,Nil),
	nonmatch(as,Cons(a,as')),
	match(as,Cons(a',as')),
	is_coauthor(a,as',y).

Fig. 3. IR of function is_coauthor from the example in Fig. 2

3 Size Abstraction Using Typed-Norms

The cost analysis framework that we rely on [2] is performed in two steps: (1) the program is first transformed into an abstract version that is used to track how the sizes of the different data-structures change, when moving from one control point to another; and (2) the abstract program is then analyzed to infer lower and upper bounds on the resource consumption. As the second step remains unchanged, we focus only on the first step.

Abstract programs are obtained from the source program (in the intermediate form) as follows: (1) the program variables are replaced by numerical variables that represent their corresponding sizes; and (2) the instructions are replaced by linear constraints, over the new variables, to simulate the effect of their execution on the sizes of the corresponding data-structures. When data refer to numerical values, their sizes are defined as their values, and when they refer to data-structures then size functions, commonly known as *norms*, are used to measure their sizes. Note that our goal is not to obtain the real size of data-structures, but to use the data-size information to obtain a more accurate complexity of the recursions in the program.

3.1 Preliminaries on Typed-Norms

Among all norms in the literature, the *term-size* norm is probably the most well-known one. It has been introduced, and intensively used, in the context of termination analysis of logic programs. Intuitively, it counts the number of data constructors in a given data-structure, and can be defined as follows:

$$\|t\|_{ts} = \begin{cases} 1 + \sum_{i=1}^{n} \|t_i\|_{ts} & \text{if } t = Co(t_1, \ldots, t_n) \\ 1 & \text{otherwise} \end{cases} . \qquad (1)$$

The main shortcoming of the term-size norm is that it considers all data types equal, which leads to imprecision when used in the context of cost analysis.

Example 3. The recursive function `written_by` in the example traverses Authors and Books recursive data-structures. Using term-size norm, a static analysis obtains that the complexity is $O(n^2)$, because each recursion in the data-structure is abstracted to n. However, it is more accurate if we can say that the complexity is $O(bs \times as)$ where bs refers to the number of books and as the maximum length of the lists Authors for each of the books in bs, because recursions are applied to different data-types.

To overcome the imprecision issues discussed above we use typed-norms, which are designed to distinguish data constructors according to their types. For example, they can measure the length of a list, and the size of its elements separately. Such norms have been used before in the context of termination analysis (see [6] and its references), and can be defined as follows:

$$\|t\|_{\sigma} = \begin{cases} t & \sigma = \texttt{Int} \text{ and } t \text{ is an integer} \\ \mathit{length}(t) & \sigma = \texttt{String} \text{ and } t \text{ is a string} \\ 1 + \sum_{i=1}^{n} \|t_i\|_{\sigma} & \text{if } t = Co(t_1, \ldots, t_n) \text{ and } \mathit{type}(t) = \sigma \\ \sum_{i=1}^{n} \|t_i\|_{\sigma} & \text{if } t = Co(t_1, \ldots, t_n) \text{ and } \mathit{type}(t) \neq \sigma \end{cases} . \qquad (2)$$

Intuitively, $\|t\|_{\sigma}$ counts the number of data-constructs of type σ in t. Basic types are treated in a special way: integers keep their values, and strings are abstracted to their lengths. This means that $\|t\|_{\texttt{Int}}$ equals to the sum of all integer values in the data-structure t. We modify the above typed-norm scheme to the following one

$$\|t\|_{\sigma} = \begin{cases} t & \sigma = \texttt{Int} \text{ and } t \text{ is an integer} \\ \mathit{length}(t) & \sigma = \texttt{String} \text{ and } t \text{ is an string} \\ 1 + \sum_{i=1}^{n} \|t_i\|_{\sigma} & \text{if } t = Co(t_1, \ldots, t_n) \text{ and } \mathit{type}(t) = \sigma \\ \max_{i=1}^{n} \|t_i\|_{\sigma} & \text{if } t = Co(t_1, \ldots, t_n) \text{ and } \mathit{type}(t) \neq \sigma \end{cases} . \qquad (3)$$

The difference from (2) is that, instead of summing the sizes of the inner elements, it just keeps the maximal one. For instance, consider the recursive function written_by. By using (3), we will be able to infer that the cost is bounded by $O(bs \times as)$ where bs denotes the length of the recursive data-structure Books and as is the maximal length of the recursive data-structure Authors for each book.

This is because when abstracting the list using the Authors norm, the fourth case applies and the maximum value of all elements of the list is taken as worst case cost. Using (2), we add the length of Authors as many times as Books we have (at most bs books). Thus, obtaining the less accurate bound $O(bs^2 \times as)$. We argue that scheme (3) is more suitable than (2) for the cost analysis framework we rely on. This is because this framework is based on compositional reasoning that assumes worst-case for each iteration (i.e., when processing the inner elements of a data-structure), and then multiplies it by the number of iterations (which usually depends on the size of the skeleton). Note that one could also define in an analogous way a norm that estimates the minimum value, by replacing max with min in (3). This is in particular useful for inferring lower-bounds [12]. A variation of (3) is implicitly used in works on sized types [15, 12] (see Sec. 6 for more details).

3.2 Our Transformational Approach

Next we describe our abstraction procedure based on typed-norms. Our approach allows maintaining several abstractions even for the same variable at the same time as in [6]. Thus, it allows estimating the size of a variable using different measures. This is important since two different parts of the program might traverse two different parts of the same data-structure. Having both measures allows us to provide tighter bounds. Note that although we are interested in using typed-norms following scheme (3), our techniques are also valid for scheme (2).

b	b^α
$g_1 \wedge g_2$	$g_1^\alpha \wedge g_2^\alpha$
$match(x,t)$	$\wedge\{X_\sigma = \lVert t \rVert_\sigma \mid \sigma \in \texttt{typed\_norms}(x)\}$
$nonmatch(x,t)$	$true$
$e_1\ op\ e_2$	$(e_1\ op\ e_2)[y/Y_{\texttt{Int}}]$ if $\texttt{Int} \in \texttt{typed\_norms}(x)$; otherwise $true$
$p(\bar{x},\bar{y})$	$p(\bar{X},\bar{Y})$
$x := t$	$\wedge\{X_\sigma = \lVert t \rVert_\sigma \mid \sigma \in \texttt{typed\_norms}(x)\}$
$true$	$true$

Fig. 4. Size abstraction for the instructions

We first introduce some concepts. Given two types σ_1 and σ_2, we write $\sigma_1 \preceq \sigma_2$ if the definition of type σ_2 uses (either directly or transitively) type σ_1. If $\sigma \preceq \sigma$ we say that the type is recursive. For simplicity, we assume that recursive types are in direct recursive form (thus, its form can be checked by just inspecting its definition). We use $type(x)$ to refer to the type of x, and $\texttt{typed\_norms}(x)$ to refer to the set of types w.r.t. which we want to measure the size of x. In Sec. 4 we explain how to automatically infer $\texttt{typed\_norms}(x)$. For $\texttt{typed\_norms}$ to be *valid*, we require that $\sigma' \preceq \sigma$ if $type(x) = \sigma$ and $\sigma' \in \texttt{typed\_norms}(x)$. For instance, $\texttt{typed\_norms}(x) = \{\mathsf{Authors}, \mathsf{String}\}$ is a valid typed-norm for x with $\mathit{type}(x) = \mathsf{Authors}$. Given a type $\sigma \in \texttt{typed\_norms}(x)$, we let X_σ be an integer valued variable representing the size of (the value of) x w.r.t the typed-norm $\|.\|_\sigma$. If $\sigma \neq \texttt{Int}$, then we implicitly assume $X_\sigma \geq 0$. For a sequence of variables $\bar{x}$, we let $\bar{X}$ be a sequence that results from replacing each x_i by $X_{\sigma_1}, \ldots, X_{\sigma_n}$, where $\texttt{typed\_norms}(x_i) = \{\sigma_1, \ldots, \sigma_n\}$. Given an arithmetic expression e, we abstract e as $e[y/Y_{\texttt{Int}}]$, where we use $e[y/Y_{\texttt{Int}}]$ to denote the expression that results from replacing each variable y in e by $Y_{\texttt{Int}}$.

Given a typed-norm as in scheme (2) or (3), its *symbolic* version is an extension to handle terms that include variables, e.g., $\mathit{Cons}(x, \mathit{xs})$ where x and xs are variables. It is obtained from the corresponding typed-norms definition by adding the following extra cases: when t is a variable of type σ_1, then $\|t\|_\sigma = T_\sigma$ if $\sigma \preceq \sigma_1$ and $\|t\|_\sigma = 0$ otherwise. In what follows, we abuse notation and use $\|t\|_\sigma$ to refer to this symbolic version of typed-norm.

For the sake of simplifying the presentation, we assume that the input program is in *single static assignment* form. A *size abstraction* is a conjunction of linear constraints that describe the effect of the corresponding instruction. Given an instruction b, its abstract version b^α is defined as in Fig. 4. Let us explain the abstraction for the different instructions: conjunctions are abstracted by recursively abstracting each of their conjuncts; a match guard on x adds as many constraints as typed-norms apply to the variable x, each constraint assigns to the abstract variable the abstraction of the matched term w.r.t. the considered norm; as we do not keep inequality constraints, *nonmatch* guards are abstracted to *true*; in the expressions involving arithmetic operations, each variable y is replaced by an abstract variable $Y_{\texttt{Int}}$; the arguments in the calls are replaced by their corresponding abstract names; assignments are abstracted analogously to match guards.

Definition 1. *Given a program P, its size abstraction P^α is a program obtained by replacing each rule $p(\bar{x}, \bar{y}) \mapsto g, b_1, \ldots, b_n \in P$ by $p(\bar{X}, \bar{Y}) \mapsto g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$.*

When using the typed-norm scheme (3), then P^α might include constraints of the form $X_\sigma = E$ where E is an arithmetic expression that involves max. Such non-linear constraints can be approximated by linear ones as follows: replace the sub-expression $\max(B_1, \ldots, B_n)$ by a new auxiliary variable A, and add the constraints $A \geq B_1 \wedge \cdots \wedge A \geq B_n$; this might be applied repeatedly in case of nested or multiple occurrences of max. When the max has zero operands, it can be safely replaced by 0. Note also that if non-linear arithmetic is allowed in our language, then P^α might include non-linear constraints. These can also be approximated by linear ones as in [4].

Example 4. Fig. 5 shows, in the right column, the abstraction of the instructions which appear in the corresponding left column for function $\mathsf{is\_coauthor}$. We use underlining to denote abstractions that are useless, as it will be explained in the next section. The $\texttt{typed\_norms}$ that we use in $\mathsf{is\_coauthor}$ is: $\texttt{typed\_norms}(x) = \{\mathsf{String}\}$ if $\mathit{type}(x) = \mathsf{String}$; $\texttt{typed\_norms}(x) = \{\mathsf{String}, \mathsf{Authors}\}$ if $\mathit{type}(x) = \mathsf{Authors}$; and $\texttt{typed\_norms}(x) = \{\mathsf{Int}\}$ if $\mathit{type}(x) = \mathsf{Int}$. Observe that the first

is_coauthor(a,as,y)$\mapsto$	is_coauthor($\underline{a_1}$,$\underline{as_1}$,as_2,$\underline{y_1}$) $\mapsto$
case$_0$(a,as,y).	case$_0$($\underline{a_1}$,$\underline{as_1}$,as_2,$\underline{y_1}$).
case$_0$(a,as,y)$\mapsto$	case$_0$($\underline{a_1}$,$\underline{as_1}$,as_2,$\underline{y_1}$)$\mapsto$
match(as,Nil), $y := 0$.	$\{\underline{as_1} = 0, as_2 = 1\}$, $\{\underline{y_1} = 0\}$.
case$_0$(a,as,y)$\mapsto$	case$_0$($\underline{a_1}$,$\underline{as_1}$,as_2,$\underline{y_1}$)$\mapsto$
nonmatch(as,Nil),	$\{\}$,
match(as,Cons(a,as')),	$\{\underline{as_1} \geq \underline{a_1}, \underline{a_1} \geq 0, \underline{as_1} \geq \underline{as'_1},$
	$\underline{as'_1} \geq 0, as_2 = as'_2 + 1, as'_2 \geq 1\}$,
$y := 1$.	$\{\underline{y_1} = 1\}$.
case$_0$(a,as,y)$\mapsto$	case$_0$($\underline{a_1}$,$\underline{as_1}$,as_2,$\underline{y_1}$) $\mapsto$
nonmatch(as,Nil),nonmatch(as,Cons(a,as')),	$\{\}$, $\{\}$,
match(as,Cons(a',as')),	$\{\underline{as_1} \geq \underline{a'_1}, \underline{a'_1} \geq 0, \underline{as_1} \geq \underline{as'_1},$
	$\underline{as'_1} \geq 0, as_2 = as'_2 + 1, as'_2 \geq 1\}$,
is_coauthor(a,as',y).	is_coauthor($\underline{a_1}$,$\underline{as'_1}$,as'_2,$\underline{y_1}$).

Fig. 5. Abstraction of function is_coauthor

argument a of is_coauthor is abstracted by the variable a_1 using the type String and the second argument as is abstracted in variables as_1 and as_2, one for each element of typed_norms(as). It is interesting to see that the abstraction of the guard *match(as,Cons(a',as'))* on the third $case_0$ rule uses as_1 to denote the maximum length of a String in the recursive data-structure as, so we have to add the constraints $as_1 \geq a'_1$ (a'_1 represents the abstraction of the first argument of $Cons$) and $as_1 \geq as'_1$. Note that if we use (2) in Sec. 3.1 then as_1 corresponds to the length of the concatenation of every String in as, i.e., $as_1 = a'_1 + as'_1$. Since a'_1 and as'_1 represent String lengths, their value cannot be lower than 0 and we add constraints for that. Also, as'_2 represents the length of Authors (and Nil corresponds to size 1), then as'_2 must be at least of size 1. In order to assess the impact of our approach, we show in Fig. 6 the exact upper bounds obtained from an abstraction using only the term-size norm (left) and the abstraction using typed-norms (right) for our three functions. The upper bounds are given as functions of the sizes of the input parameters w.r.t. the different abstractions (hence the output parameter is not included). As explained in Ex. 3, the upper bounds obtained for written_by are more accurate using typed-norms. The largest gain is obtained for sort_books_by_authors as it uses the upper bounds of the two other functions, namely we achieve $O(n \times m \times l)$, where n represents the number of authors in as, m represents the number of books in bs and l represents the maximum length of Authors for each book in bs.

Intuitively, the analyzer obtains this upper bound following this reasoning. As function sort_books_by_authors has a recursive call that decreases the number of authors of as, we have that the maximum number of recursive calls is bound by n (number of authors in as), thus its cost is $O(n * cost\_body_1)$ where $cost\_body_1$ is the cost of each application of the body of the function. Now, in order to compute $cost\_body_1$, we have to analyze the cost of function written_by as it is called in the body. In this case, we also have recursive calls that decrease the size of the second argument bs (i.e., the number of books denoted as m). By applying

term-size norm	typed-norms
is_coauthor(a, as) $= 4 + 5 \times (\frac{as}{2} - \frac{1}{2})$	is_coauthor(a_1, as_1, as_2) $= 4 + 5 \times (as_2 - 1)$
written_by(a, bs) $= 3 + (\frac{bs}{4} - \frac{1}{4}) \times (14 + 5 \times (\frac{bs}{2} - \frac{5}{2}))$	written_by(a_1, bs_1, bs_2, bs_3) $= 3 + (bs_2 - 1) \times (14 + 5 \times (bs_3 - 1))$
sort_books_by_author(as, bs) $= 3 + (\frac{as}{2} - \frac{1}{2}) \times (10 + (\frac{bs}{4} - \frac{1}{4})$ $\times (14 + 5 \times (\frac{bs}{2} - \frac{5}{2})))$	sort_books_by_author$(as_1, as_2, bs_1, bs_2, bs_3)$ $= 3 + (as_2 - 1) \times (10 + (bs_3 - 1)$ $\times (14 + 5 \times (bs_2 - 1)))$

Fig. 6. Upper bounds comparison term-size vs. typed-norms (a_1, as_1 and bs_1 represent String-norms, as_2 and bs_2 represent Authors-norms and bs_3 represents Books-norm)

a similar reasoning, the cost of written_by is bound by $O(m * cost\_body_2)$. Again, we need to compute the cost of the call to is_coauthor, as it determines the cost of the body of written_by. Finally, we have a recursive call in is_coauthor that decreases the size of l (maximum size of Authors). By replacing each *cost_body* by the computed cost, we get the cubic cost above as upper bound. By using term-size, we obtain $O(n * m^2)$ where n is the size of as and m the size of bs. The difference is that the whole data structure is abstracted by m, thus the cost of method is_coauthor is bound by the whole m, instead of by the length of the author's lists (denoted l above) which are a subterm of m. This might lead to an important loss of precision when the data structure m is large.

4 Inference of Typed-Norms

In Sec. 3, we have assumed that each variable x is assigned a set of types, given by `typed_norms`(x), whose size we want to track. In principle, one could abstract each variable w.r.t. all norms valid for its type. However, this would threaten the efficiency of the analysis, as the complexity of the solving procedure for finding resource bounds from abstractions exponentially grows with the number of variables. In this section we develop an analysis that eliminates useless abstractions in two dimensions: (1) As it was observed in [3], one can remove variables that do not affect the cost. In particular, the cost of a given program (mainly) depends on the number of recursions performed, which in turn is controlled by the corresponding guards (conditions to stop the recursion). This means that any variable that does not affect, directly or indirectly, the value of a guard, can be completely ignored. (2) We push this observation further, and besides eliminating *useless* variables (and their abstractions), we also eliminate useless (typed) size information for those variables that are useful and thus have not been eliminated in (1). In some sense we eliminate *useless* types, and thus typed-norms, from each variable.

We say that a guard instruction g is *cost-significant* if it appears in a guard. In practice, we identify such instructions by examining the (recursive) strongly connected components of the corresponding control flow graph. The variables that are involved in the guards are the source for the size information that we want to track. For example, if a cost-significant guard is of the form $\mathit{match}(x,t)$, and $\mathit{type}(x)=\sigma$ where σ is a *recursive* type, then $\|.\|_\sigma$ is a norm that we should use for x (because the corresponding recursion might be traversing this part of x). Our analysis is done in two steps: (1) first the cost-significant guards are used to initialize $\texttt{typed\_norms}(x)$ for the variables involved in these guards, and (2) this information is propagated to other variables in the program by means of backwards data-flow analysis. Below we sketch these two steps.

Initialization. This step starts by setting $\texttt{typed\_norms}(x)$ to $\emptyset$ for each variable x in the program. Then, it identifies the set of cost-significant guards, and uses each such guard to modify related $\texttt{typed\_norms}(x)$ as follows:

- If the guard is $\mathit{match}(x,t)$, variable x has a type σ, and σ is a recursive type, then σ is added to $\texttt{typed\_norms}(x)$.
- If the guard is of the form e_1 *op* e_2, and variable x appears in e_1 or e_2, then `Int` is added to $\texttt{typed\_norms}(x)$.

Note that in the case of $\mathit{match}(x,t)$, if σ is not recursive then it is simply ignored. This is because non-recursive types cannot directly affect the number of recursions. However, they might have some inner recursive types that do, those will be propagated to x (from other guards) in the second step.

Propagation. The initial information computed in the first step must be propagated backwards to other variables in the program. Intuitively, the propagation step works as follows: suppose we have an instruction $x{:=}\mathit{Cons}(y,\mathit{ys})$, and we know that $\sigma\in\texttt{typed\_norms}(x)$ (after the instruction). This means that we want to track the size of x w.r.t. the type σ, but to do so precisely we must track this information in all parts of x, i.e., in y and ys, thus we add σ to $\texttt{typed\_norms}(y)$ and $\texttt{typed\_norms}(\mathit{ys})$, if they are valid norms for the corresponding types. The propagation rules for the different instructions are defined as follows:

- For $\mathit{match}(x,t)$ and $\mathit{nonmatch}(x,t)$, if $y\in\mathit{vars}(t)$, and $\sigma\in\texttt{typed\_norms}(y)$, then we add σ to $\texttt{typed\_norms}(x)$.
- For $x{:=}t$, if $\sigma\in\texttt{typed\_norms}(x)$ we add σ to $\texttt{typed\_norms}(y)$ for each variable $y\in\mathit{vars}(t)$ as far as $\mathit{type}(y)\preceq\sigma$.
- For $m(x_1,\dots,x_n,y)$, if there is a rule $m(w_1,\dots,w_n,z)\mapsto g,b_1,\dots,b_m$ and $\sigma\in\texttt{typed\_norms}(w_i)$ we add σ to $\texttt{typed\_norms}(x_i)$, for each $1\le i\le n$.
- For $m(x_1,\dots,x_n,y)$, if there is a rule $m(w_1,\dots,w_n,z)\mapsto g,b_1,\dots,b_m$ and $\sigma\in\texttt{typed\_norms}(y)$ we add σ to $\texttt{typed\_norms}(z)$.
- For any pair of rules $m(x_1,\dots,x_n,y)\mapsto g,b_1,\dots,b_m$ and $m(w_1,\dots,w_n,z)\mapsto g',b'_1,\dots,b'_k$, if $\sigma\in\texttt{typed\_norms}(x_i)$ then $\sigma\in\texttt{typed\_norms}(w_i)$, and if $\sigma\in\texttt{typed\_norms}(y)$ then $\sigma\in\texttt{typed\_norms}(z)$ (this forces rules with the same name and number of arguments to be abstracted to rules with the same name and same number of abstracted arguments).
- There are some built-in functions that are treated as built-in instructions, e.g., $\mathit{length}(s,x)$ which binds x to the length of the string s. In such case, if $\texttt{Int}\in\texttt{typed\_norms}(x)$ then we add `String` to $\texttt{typed\_norms}(s)$.

Function	Initialization	Propagation
is_coauthor	$\{\}_a$,$\{\}_{as}$,$\{\}_y$	$\{\}_a$,$\{\text{Authors}\}_{as}$,$\{\}_y$
case$_0$ (1st rule)	$\{\}_a$,$\{\text{Authors}\}_{as}$,$\{\}_y$	$\{\}_a$,$\{\text{Authors}\}_{as}$,$\{\}_y$
case$_0$ (2nd rule)	$\{\}_a$,$\{\text{Authors}\}_{as}$,$\{\}_y$,$\{\}_{as'}$	$\{\}_a$,$\{\text{Authors}\}_{as}$,$\{\}_y$,$\{\}_{as'}$
case$_0$ (3rd rule)	$\{\}_a$,$\{\text{Authors}\}_{as}$,$\{\}_y$,$\{\}_{a'}$,$\{\}_{as'}$	$\{\}_a$,$\{\text{Authors}\}_{as}$,$\{\}_y$,$\{\}_{a'}$,$\{\text{Authors}\}_{as'}$

Fig. 7. Inference on is_coauthor

– All other instructions do not modify any information.

The propagation step is applied iteratively, using standard backwards data-flow analysis, until a fix-point is reached, i.e., the values of all `typed_norms`(x) become stable. Note that this data-flow analysis also propagates information between the rules (no special treatment is required). Termination is guaranteed because the number of typed-norms is finite.

Example 5. Fig. 7 shows the obtained typed norms on each variable after initialization and propagation on is_coauthor and `case`$_0$ rules. We use $\{\}_x$ notation to represent `typed_norms`(x) in a compact way. The algorithm works in the following way:

- Initialization sets `typed_norms`$(as) = \{\text{Author}\}$ and `typed_norms`$(x) = \emptyset$ for any other variable x in the program because all the guards in the program are of the form $match(as, t)$.
- Then, $\{\text{Author}\}$ is propagated in the following way:
  1. The second argument of `case`$_0$ propagates $\{\text{Authors}\}$ to `is_coauthor` rule, making `typed_norms`$(as) = \{\text{Author}\}$ on `is_coauthor`.
  2. The second argument of `is_coauthor` propagates $\{\text{Authors}\}$ to the third `case`$_0$ rule, making `typed_norms`$(as') = \{\text{Author}\}$ on the third `case`$_0$.
  3. Guard `match`$(as,$ `Cons`$(a', as'))$ on the third `case`$_0$ rule adds $\{\text{Authors}\}$ to `typed_norms`(as), but `typed_norms`(as) already contains $\{\text{Authors}\}$, and the process stops.

When a variable has an empty set of candidate norms, it means that it is not relevant to obtain the cost expression. In our example, String-norm and Int-norm are useless to obtain an upper bound. The result of applying our inference of typed-norms on the running program is the abstraction in Fig. 5 removing all underlined variables and associated constraints.

5 Experimental Evaluation

We have implemented the resource analysis detailed in this paper in the static analyzer for ABS programs SACO (http://costa.ls.fi.upm.es/saco). Our analysis is currently being integrated in the web interface of SACO and will be available by selecting the `typed-norms` option within the settings section soon. Our experiments aim at evaluating both the accuracy and efficiency of our analysis. Experimental evaluation has been carried out on the functional modules of the *Replication System* case study (an industrial case study whose source code is available from the SACO website). A total of 88 functions are used in the replication system. We have used three different configurations for the analysis with norms: (1) term-size, (2) typed-norms considering all possible norms, and (3) significant typed-norms obtained by the inference algorithm as described in Sec. 4. An upper bound was obtained on 61 out of the 88 functions in configuration (1) and in 62 out of the 88 functions on configurations (2) and (3). A notable result of our experiment is that for one function (*'itemMapToSchedule'*) an upper bound has been obtained using configurations (2) and (3) but cannot be obtained in (1) since it requires a more refined abstraction than term-size.

As regards accuracy, in Fig. 8 we compare the quality of the upper bounds obtained using term-size and typed-norms (note that in (2) and (3) we infer the same upper bounds). Since the term-size norm measures the size of the input in a different way from the typed-norms, a fair comparison of the results can be done by actually evaluating the corresponding upper bounds on some (random) concrete input. We used quickCheck [7] to generate 10 random concrete inputs for each upper bound, so for each case we obtain 10 different quotients.

For each random input, the diagram in Fig. 8 shows the quotient between the value of the upper bound obtained using term size, and the value of the upper bound using typed-norms. The x-axis corresponds to the benchmark number, and to improve readability we have sorted the benchmarks according to the corresponding values in the y-axis. We have ignored constant upper bounds since they correspond to functions without any recursion (i.e., the term-size norm and typed-norm should give the same answer), and thus remained with 32 non-constant upper bounds (the horizontal axe of the diagram corresponds to these 32 upper bounds). Values below 1 mean the analysis based on the typed-norms is more precise than the term-size one (the smaller the value, the bigger is the improvement), which is the case in all 32 cases.

We have also compared the performance of the different configurations. The run-time of each configuration (for all benchmarks together, using the average of 5 runs) is depicted in Table 1. We divide the total time into 3 parts: $\mathbf{T_{sa}}$ is the time for processing the input program in order to define the typed-norms, for configuration (3) this also includes the typed-norms inference, and for configuration (1) this step does not exist and thus it costs 0; $\mathbf{T_{ac}}$ is the time for generating the abstract program; and $\mathbf{T_{ub}}$ is the time for solving the abstract program into an upper bound. As expected, using all typed-norms introduces a significant overhead in configuration (2) when compared to (1). Importantly, by using the typed-norms inference we reduce the number of typed-norms significantly and thus the overhead becomes reasonable in configuration (3) when compared to (1). The experiments have been performed on an Intel Core 2 Duo at 2.4GHz with 8GB of RAM, running OS X 10.9.

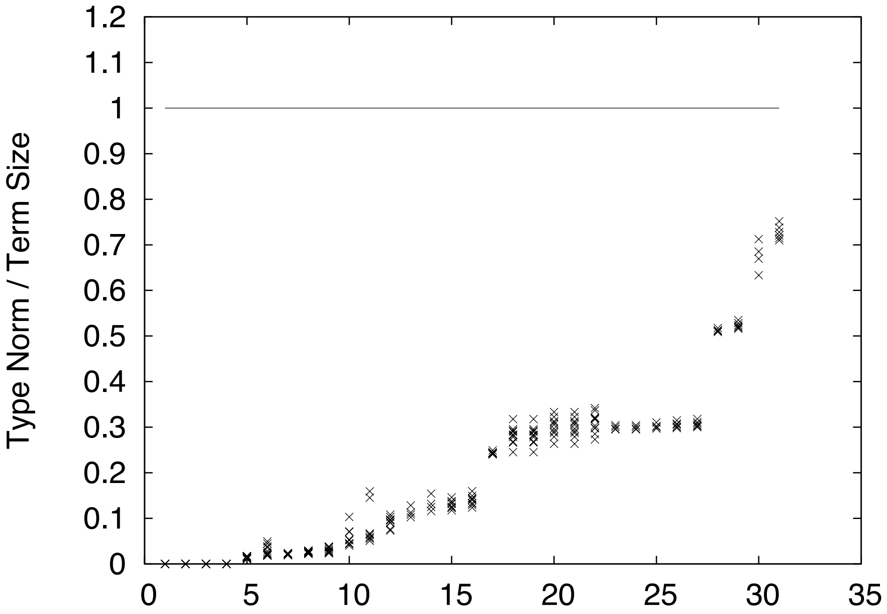

Fig. 8. Upper Bound Comparison

Table 1. Run-Time statistics (in ms.) on 61 functions: (1) using term-size; (2) using all type-norms; and (3) using only significant typed-norms obtained as in Sec. 4

Configuration	$\mathbf{T}_{sa}$	**Avg.** $\mathbf{T}_{sa}$	$\mathbf{T}_{ac}$	**Avg.** $\mathbf{T}_{ac}$	$\mathbf{T}_{ub}$	**Avg.** $\mathbf{T}_{ub}$	$\mathbf{T}_{sa}+\mathbf{T}_{ac}+\mathbf{T}_{ub}$
(1)	0	0	120	2	3911	65	4031
(2)	1633	27	631	11	14230	234	16494
(3)	2161	36	255	5	4488	74	6904

6 Conclusions, Related and Future Work

We have presented a novel transformational approach to resource analysis with typed-norms which has the advantage that its formalization can be done by only adapting the first phase of cost analysis in which the program is transformed into an intermediate abstract program. Besides its simple formal development, the implementation has been easily integrated into the previous system as a pre-phase to the existing analysis.

Our work is inspired by [9] where the authors introduce the notion of typed-based norm in the context of termination analysis, and show how types can be very useful for finding suitable norms even for untyped languages like Prolog. They also illustrate that typed-based norms sometimes must be combined to get a termination proof. In [15], Vasconcelos introduces an enriched typing to get upper bounds and uses it on resource analysis. Unlike our approach, in this approach one can handle multiple typed-norms on variables only by having parametric data-structures. The techniques of Vasconcelos have been extended to the context of logic programs [12]. When compared to an approach based on abstract interpretation like [12], our transformational approach is simpler to define and to implement because we do not need to re-do all the abstract interpretation theory (defining specific concretization, abstraction functions, etc.). Instead, we simply have to add explicit arguments for the sizes of data structures and define a size abstraction which is rather straightforward. The implementation simply requires a pre-process to add the arguments and properly abstract them. Then, standard size analysis works on the transformed program. As regards accuracy, when compared to [12], we define an additional step to infer the required typed-norms. This allows us to handle accurately examples like our running example in which the same term requires the use of more than one typed-norm in order to be as accurate as possible.

With respect to the inference of typed-norms, we extend the results in [1] to deal with typed-norms in addition to useless arguments. As our experiments have showed, this analysis is essential to be scalable in practice (the analysis time is reduced 58.14%) and, to the best of our knowledge, it is the first time that it is applied on norms.

In future work, we plan to include parametric data types, which pose some challenges in the definition of the framework. Also, we want to enrich types with positions so that we can measure differently the same type when it appears in different type contexts. E.g., the type $\mathbf{data}\ t = \mathit{Pair}(\mathtt{Int}, \mathtt{Int})$ is enriched to $\mathbf{data}\ t = \mathit{Pair}(\mathtt{Int}^1, \mathtt{Int}^2)$, and thus we will have the two different norms $\|.\|_{\mathtt{Int}^1}$ and $\|.\|_{\mathtt{Int}^2}$.

References

1. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO programs. In: Proc. of APLAS'11. LNCS, vol. 7078, pp. 238–254. Springer (Dec 2011)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: Proc. of ESOP'07. LNCS, vol. 4421, pp. 157–172. Springer (2007)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Removing Useless Variables in Cost Analysis of Java Bytecode. In: Proc. of SAC'08. pp. 368–375. ACM (2008)
4. Alonso, D., Arenas, P., Genaim, S.: Handling Non-linear Operations in the Value Analysis of COSTA. In: Proc. of BYTECODE'11. ENTCS, vol. 279, pp. 3–17. Elsevier (2011)
5. Bossi, A., Cocco, N., Fabris, M.: Proving Termination of Logic Programs by Exploiting Term Properties. In: Proc. of TAPSOFT'91. LNCS, vol. 494, pp. 153–180. Springer (1991)
6. Bruynooghe, M., Codish, M., Gallagher, J., Genaim, S., Vanhoof, W.: Termination Analysis of Logic Programs through Combination of Type-Based norms. TOPLAS 29(2), Art. 10 (2007)
7. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proc. of ICFP'00. pp. 268–279. ACM (2000)

8. Fähndrich, M.: Static Verification for Code Contracts. In: Proc. of SAS'10. LNCS, vol. 6337, pp. 2–5. Springer (2010)
9. Genaim, S., Codish, M., Gallagher, J., Lagoon, V.: Combining Norms to Prove Termination. In: Proc. of VMCAI'02. LNCS, vol. 2294, pp. 123–138. Springer (2002)
10. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Proc. of FMCO'10 (Revised Papers). LNCS, vol. 6957, pp. 142–164. Springer (2012)
11. King, A., Shen, K., Benoy, F.: Lower-bound Time-complexity Analysis of Logic Programs. In: Proc. of ILPS'97. pp. 261–275. MIT Press (1997)
12. Serrano, A., Lopez-Garcia, P., Bueno, F., Hermenegildo, M.: Sized Type Analysis for Logic Programs. In: Tech. Comms. of ICLP'13. Cambridge U. Press (2013), to Appear
13. Spoto, F., Mesnard, F., Payet, É.: A Termination Analyser for Java Bytecode based on Path-Length. TOPLAS 32(3), Art. 8 (2010)
14. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: Proc. of CASCON'99. pp. 125–135. IBM (1999)
15. Vasconcelos, P.: Space Cost Analysis using Sized Types. Ph.D. thesis, School of CS, University of St. Andrews (2008)
16. Vasconcelos, P., Hammond, K.: Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In: Proc. of IFL'03. LNCS, vol. 3145. Springer (2003)
17. Wegbreit, B.: Mechanical Program Analysis. Commun. ACM 18(9), 528–539 (1975)