# Resource Bounds Analysis

Jorge Navas, Edison Mera, Pedro López-García, Manuel V. Hermenegildo

TR CLIP6/2006.0
November 03, 2006

**Abstract**

We present a generic analysis that infers both upper and lower bounds on the usage that a program makes of a set of user-definable resources. The inferred bounds will in general be functions of input data sizes. A resource in our approach is a quite general, user-defined notion which associates a basic cost function with elementary operations. The analysis then derives the related (upper- and lower-bound) cost functions for all procedures in the program. We also present an assertion language which is used to define both such resources and resource-related properties that the system can then check based on the results of the analysis. We have performed some experiments with some concrete resource-related properties such as execution steps, bits sent or received by an application, number of arithmetic operations performed, number of calls to a procedure, number of transactions, etc. presenting the resource usage functions inferred and the times taken to perform the analysis. Applications of our analysis include resource consumption verification and debugging (including for mobile code), resource control in parallel/distributed computing, and resource-oriented specialization.

## 1 Introduction

It is generally recognized that inferring information about the costs of computations can be useful for a variety of applications. These costs are usually related to execution steps and, sometimes, time or memory. We propose a generic analyzer which allows automatically inferring both upper and lower bounds on the usage that a program makes of a set of a more general concept of *user-definable resources*. Examples of such user-definable resources are bits sent or received by an application over a socket, number of calls to a procedure, number of files opened, number of licenses consumed, monetary

units spent, disk space used, etc., as well as the more traditional execution steps, execution time, or memory. We expect the inference of this kind of information to be instrumental in a variety of applications, such as resource usage verification and debugging, certification of resource consumption in mobile code, resource/granularity control in parallel/distributed computing, or resource-oriented specialization.

In our approach a resource is a user-defined notion which associates a basic cost function with elementary operations in the base language and/or to some methods in libraries. In this sense, each *resource* is essentially a user-defined *counter*. The user gives a name (such as, e.g., `bits_sent`) to the counter and then defines via assertions how each elementary operation in the program (e.g., assignments, field accesses, calls to builtins, external calls, etc.) increments or decrements that counter. The use of resources obviously depends in practice on the sizes or values of certain inputs to programs or procedures. Thus, in the assertions describing elementary operations the counters may be incremented or decremented not only by constants but also by amounts that are *functions* of input data sizes or values. Correspondingly, the objective of our method is to statically derive from these elementary assertions and the program text *functions* that yield upper- and lower-bounds on the amount of those resources that each of the procedures in the program (and the program as a whole) will consume or provide. The input to these functions will also be the sizes or value ranges of the topmost input data to the program or procedure being analyzed.

The method proposed builds on previous work on inferring functions providing upper- and lower-bounds on the number of *execution steps* performed by procedures (based again on the sizes or value ranges of input) [15, 6, 16, 5, 11, 7, 8], but generalizes that work to deal with a much more general class of user-defined resources. It also extends the range of languages to which the methods are applicable. In [13], and inspired by [2] and [12], Nielson presented a complexity analysis for Horn clauses. In [3] a method is presented for reserving resources before their actual use. However, the programmer (or program optimizer) needs to annotate the program with acquire and consume primitives, as well as loop invariants and function pre- and post-conditions. Interesting type-based related work has also been performed around the GRAIL system [1], also oriented towards resource analysis, but it has concentrated mainly on ensuring memory bounds.

| | | |
|---|---|---|
| $< program >$ | ::= | $< construct >^*$ |
| $< construct >$ | ::= | $< block > \mid < resource\_delta > \mid < approx\_def > \mid < primitive\_assrt >$ |
| $< block >$ | ::= | $< head > < assrt >^* < body >$ |
| $< head >$ | ::= | $block\_name(var^*)$ |
| $< body >$ | ::= | $\{ < cond\_stmt >^* \}$ |
| $< cond\_stmt >$ | ::= | $cond \rightarrow < stmt\_body >$ |
| $< stmt\_body >$ | ::= | $< stmt >^*$ |
| $< stmt >$ | ::= | $block\_name(var^*) \mid primitive\_name(var^*)$ |

Figure 1: Syntax of $\mathcal{L}$ (Blocks)

## 2 The Language $\mathcal{L}$

We start by describing a block-oriented kernel language called $\mathcal{L}$, which includes an assertion schema. The rules for the grammar of this language are listed in Fig. 1 and those for the assertion schema in Fig. 2. Therein we assume $var \in Vars$, where $Vars$ is the set of variable identifiers and *block_name* is a block identifier, and *primitive_name* is the name of a primitive operation of the language. Calls to external methods or procedures will also be represented with *primitive_name*s. In that way all statements are normalized to be either calls to blocks or calls to primitives. The grammar is purposely made independent of the basic statements of the language (e.g., assignment, field storage, expression evaluation, etc.) whose semantics (with respect to resource consumption and argument size relationships) will be described via assertions, as described later. Arithmetic or boolean expressions at the statement level are also assumed to be normalized, so that they are reduced to individual calls to atomic arithmetic or boolean primitives such as addition, subtraction, conjunction, etc. The resource-related semantics of such atomic primitives are also described via assertions.

Regarding control primitives, the language supports a slightly modified version of the traditional `if-elseif-else` statement through the rule $< cond\_stmt >$`::=` $cond \rightarrow < stmt\_body >$ where $cond$ is as $< stmt >$ but restricted to return only boolean values. For a given branch $cond_i \rightarrow stmt\_body_i$, $cond_i$ is executed to determine the corresponding boolean boolean output value $b_i$. If $b_i$ is true the $stmt\_body_i$ is executed; otherwise control moves on to the next conditional statement $cond_{i+1} \rightarrow stmt\_body_{i+1}$. We assume that if any $cond_i$ is `'_'` it represents a condition that always returns true and if any $stmt\_body_i$ is `'_'` it means that represents that the body is empty. The language also allows arbitrary recursive calls between blocks. Loops are not supported directly. Instead, they are encoded as tail-recursive

blocks.

Assertions (Fig. 2) are associated with primitives or with blocks ($<$ *primitive_assrt* $>$ or $<$ *assrt* $>$).[1] Such assertions can refer to properties of the execution states when the primitive or block is called (the **requires** part), properties of the execution states when the block terminates execution (the **ensures** part), and properties which refer to the whole computation of the block, rather than the input-output behavior (which herein will be used only for resource-related properties and is thus labeled as the **costs** part).

In our particular application the assertion language is used essentially to provide input to the resource analysis. The first and most fundamental such use of assertions is to provide the cost functions for each language primitive used in the program (as well as for the external calls). To this end one or more **costs** assertions are used for each primitive, specifying properties of the form **res_usage**($<$ *approx* $>$, *res_name*, $<$ *arith_expr* $>$). The *res_name* field in these properties states which resource the assertion refers to. These *res_name*s are user-provided identifiers which are used to name a particular resource that needs to be tracked. This resource does not need to be defined in any other way –the set of resources that the system is aware of is simply the set of such names that appear in assertions. Most importantly, $<$ *arith_expr* $>$ is an arithmetic expression which when evaluated returns how much of a given resource that primitive consumes (or provides). It is a function of the sizes of that primitive's input data. These expressions are built using numeric constants (*num*), basic arithmetic operators ($\ominus$), and functions (*sz_metric*) which given input data return the size of such data under a that metric. Typical size metrics are the actual value of a number, the length of a list or array, the size (number of nodes and fields) of a data structure, etc. Finally, the $<$ *approx* $>$ field states whether $<$ *arith_expr* $>$ is providing an upper-bound or a lower-bound (with **o** meaning it is a "big O" expression, i.e., with only the order information).

The analysis is aimed at deriving the resource usage cost (as well as argument size relations) for the non-primitive blocks. However, it is also possible to state **costs** assertions for such blocks. These can be used to guide the analysis. In particular, the analysis will compute the greatest lower bound between the cost function provided by the assertion and the cost function inferred by the analysis.

Assertions are also used, via the **requires** and **ensures** fields, to declare

[1] This language is also the language in which the compiler outputs the results of the resource analysis. Furthermore, this assertion language is also the one used to state resource-related specifications. These specifications are then proved or disproved based on the results of the analysis, verifying the program or finding bugs [10].

| | | |
|---|---|---|
| $< primitive\_assrt >$ | ::= | $primitive\_name(var^*) < assrt >^*$ |
| $< assrt >$ | ::= | **@ requires (** $< prop >^*$ **)** |
| | \| | **@ ensures (** $< prop >^*$ **)** |
| | \| | **@ costs (** $< resource\_usage >^*$ **)** |
| $< resource\_usage >$ | ::= | **res_usage**$(< approx >, res\_name, < arith\_expr >)$ |
| $< resource\_delta >$ | ::= | **res_delta**$(< approx >, res\_name, \Delta)$ |
| $< approx\_def >$ | ::= | **approx_def**$(< approx >, arith\_function)$ |
| $< prop >$ | ::= | **in**$(var)$ \| **out**$(var)$ \| $type$ |
| | \| | **size**$(var, < approx >, < sz\_metric >, < arith\_expr >)$ |
| | \| | **size_metric**$(var, < sz\_metric >)$ |
| $< approx >$ | ::= | **ub** \| **lb** \| **o** |
| $< sz\_metric >$ | ::= | **value** \| **length** \| **size** |
| $< arith\_expr >$ | ::= | $< sz\_val >$ \| $\ominus(< sz\_val >^*)$ |
| $< sz\_val >$ | ::= | $num$ \| $< sz\_metric > (var)$ |

Figure 2: Syntax of $\mathcal{L}$ (Assertions)

relationships between the data sizes of the inputs and outputs of (primitive) blocks, which are needed by our analysis, as will be described later. These assertions are also used to label block arguments as input or output, as well as to provide types or size metric information if needed (but note that many size metrics can in practice be derived from types and many types inferred). In the same way as with the **costs** assertions, for user-defined blocks these other assertions can be provided by the user or inferred by the analysis. Again, the analysis will compute the greatest lower bound of the two.

An additional kind of property, used in **costs** assertions, is **res_delta**$(< approx >, res\_name, \Delta)$. These properties are used in to describe how a block *updates* the value for certain resources (such as counting the number of arguments passed or total execution steps –see Section 5). By $\Delta : block\_text \rightarrow arith\_expr$ we refer to a function which takes as input the text of a block and returns an arithmetic expression as defined by the $< arith\_expr >$ rule.

The following example shows how a simple Java program can be represented in our language $\mathcal{L}$ in order to perform resource analysis. However, giving a formal description of how such a translation can be made is outside the scope of the paper.

**Example 2.1** Consider the Java method `exchangeBuffer` defined in class `Buffer` in Fig 3, which sends a buffer of bytes through a socket and receives another (possibly transformed) data buffer. Assume that we would

```
public class Buffer{
    private ByteList buffer;
    public void exchangeBuffer (int s,ByteList ext_buf){
       Byte c;
       while ( ! buffer.empty()){
           c = buffer.first();
           c.exchangeByte(s,data);
           buffer.next();
           ext_buf.insert(data);
       }}}

/*@ Byte.exchangeByte(s) costs res_usage(ub,bits_received,size(this.val)) */
```

Figure 3: Java source for the `exchangeBuffer` method and assertion for `exchangeByte` method.

like to obtain an upper bound on the number of bits received by the method `exchangeBuffer` from the application –a *resource* that we will call `bits_received`. Assume also that the `ByteList` class is defined in a library and implements standard list operations such as `empty`, `first`, `next`, and `insert`. In Fig. 4 we show the same Java program transformed into our language $\mathcal{L}$. The loops are converted into recursive blocks (as in GRAIL [1]) and a standard static single assignment (SSA) transformation is carried out so that each variable is assigned exactly once (which allows us to have a clear notion of input and output).[2] Another aim of the transformation is to incorporate into the $\mathcal{L}$ program the (resource-related part of) the semantics of the base language (in this case, a part of Java) in such a way that the analysis has information required in order to infer the usage functions for the resource desired. As we will see, the analysis needs to know the for each argument in the program the metric and whether it is input or output in order to perform properly the size and resource usage analyses described in the Sect. 4 and 5. Input/output and metric information can be induced by the language (typed language), given by the user (via assertions), or inferred automatically via analysis. In the example, we assume that part of information is inferred from the language and another part is asserted in the language library.

[2]Note that this transformation, used in most modern compiler intermediate forms generally does not need to affect most resource-related analyses, even if sensitive resources such as execution time are measured or even memory consumption: the number of new variables created artificially by the transformation is statically determined and can be compensated for.

```
exchangeBuffer(buffer_i,stream,buffer_o)
    @ requires (in(buffer_i),in(stream),out(buffer_o),size_metric(buffer_i,length))
{
    empty(buffer_i) -> _,
    _               -> first(buffer_i,c)
                       exchangeByte(c,s,data)
                       next(buffer_i,buffer_0)
                       exchangeBuffer(buffer_0,stream,buffer_1)
                       insert(buffer_1,data,buffer_o)
}

exchangeBytes(c,s,data)
    @ costs     (resource_usage(ub,bits_received,size(data))
    @ requires  (size(c,ub,size,8))
    @ ensures   (size(data,ub,size,size(c)))


first(buffer,elem)
    @ requires (in(buffer), out(elem), size_metric(this.buffer,length))
    @ ensures  (size(elem,ub,size,size(elem)))

empty(buffer)
    @ requires (in(buffer))

next(buffer_i,buffer_o)
    @ requires (in(buffer_in), out(buffer_o), size_metric(buffer_i,length))
    @ ensures  (size(buffer_o,ub,length,length(buffer_i)-1))

insert(buffer_i,elem,buffer_o)
    @ requires (in(buffer_i), in(elem), out(buffer_o), size_metric(this.buffer,length))
    @ ensures  (size(buffer_o,ub,length,length(buffer_i) + 1))
```

Figure 4: $\mathcal{L}$ code for the `exchangeBuffer` method, assertion for the `exchangeByte` method, and assertions for library list operations.

# 3 Overview of the Approach

Our basic approach is as follows: given a procedure (block) call $p$, an expression $\Phi_p(r, n)$ is determined (at compile-time) that approximates $\texttt{Cost}_p(r, n)$: the units of resource $r$ consumed or produced during the computation of $p$ for an input of size $n$. We will refer to such $\Phi_p(r, n)$ expressions as *resource usage bound functions*. Certain program information (such as, for example, size metrics for arguments) is first automatically inferred by other (abstract interpretation-based) analyzers and then provided as input to the size and cost analysis (the techniques involved in inferring this information are beyond the scope of this paper —see, e.g., [9] and its references for some

examples). Based on this information, our analysis first finds bounds on the size of input arguments to the calls in the body of the procedure being analyzed, relative to the sizes of the input arguments to this procedure, using the inferred metrics. The size of an output argument in a block call depends in general on the size of the input arguments in that call. For this reason, for each output argument we infer an expression which yields its size as a function of the input data sizes. To this end, and using the input-output argument information, data dependency graphs are used to set up recurrence equations whose solution yields size relationships between input and output arguments of block calls. This information regarding argument sizes is then used to set up another set of recurrence equations whose solution provides bound functions on resource usage.

# 4 Size Analysis

In this section, we present a data dependency-based method for inferring the sizes of output arguments in the head of a block as a function of the sizes of input arguments to the block. We use a propagation approach (inspired by that of [5]) starting from the size relations of the statements of the block.

**Size metrics:** Various metrics are used for the "size" of an argument. To simplify the discussion, we assume (following $\mathcal{L}$) that the measures used are *value*, *length*, and *size* (but note that other metrics could also be used). Let $t$ be an argument position:

- The *value* metric defines the argument value as follows:

$$\texttt{val}(t) = \begin{cases} n & \text{if } t \text{ is an integer } n \\ \ominus(val(t_1), \ldots, val(t_n)) & \text{if } t = \ominus(t_1, \ldots, t_n). \end{cases}$$

  where $\ominus$ is the concatenation operator.

- Let a *proper list* be either the atom *nil* or of the form $cons(car(L), cdr(L))$, where $L$ is a proper list. The *length* metric of a proper list $L$ is defined as follows:

$$\texttt{length}(L) = \begin{cases} 0 & \text{if } L = nil \\ length(cdr(\mathrm{L})) + 1 & \text{if } L = cons(car(L), cdr(L)). \end{cases}$$

- The *size* metric of an argument $t$ is defined by $\texttt{size}(t)$ and represents the amount of memory (e.g., in bytes) taken by the contents of $t$ (e.g., an integer argument is 1, a long argument is 2, etc.).

We also define the $\texttt{size\_metric}(t)$ function as follows:

$$\texttt{size\_metric}(t) = \begin{cases} \texttt{val}(t) & \text{if size metric is } \textit{val} \\ \texttt{length}(t) & \text{if size metric is } \textit{length} \\ \texttt{size}(t) & \text{if size metric is } \textit{size} \\ \bot & \text{otherwise,} \end{cases}$$

The method for defining the argument sizes in a statement in terms of the input argument sizes of the block head is based on setting up the size differences between the statements arguments and the arguments of the block head. To this end, we define the $\texttt{diff}(x_1, x_2)$ operation which returns the size difference between $x_1$ and $x_2$. Again, we define it for each metric:

- If the metric is *val* then

$$\texttt{diff}(n_1, n_2) = \begin{cases} n_1 - n_2 & \text{if } n_1 \text{ and } n_2 \textit{ are integers} \\ \bot & \text{otherwise,} \end{cases}$$

- If the metric is *length* then the size difference between two lists $L_1$ and $L_2$ is defined as:

$$\texttt{diff}(L_1, L_2) = \begin{cases} \textit{length}(L) & \text{if there is a proper list } L \text{ such that } L_2 = L \cdot L_1 \\ \bot & \text{otherwise,} \end{cases}$$

  where $\cdot$ is the concatenation operator.

- If the metric is *size* then

$$\texttt{diff}(t_1, t_2) = \begin{cases} \textit{size}(t_2) - \textit{size}(t_1) & \\ \bot & \text{otherwise,} \end{cases}$$

**Data Dependency Graph:** A directed *data dependency graph* $G = (N, E)$ ($N$ a set of nodes and $E$ a set of edges) is used to represent the data dependency between statements in a conditional statement, and between statements and the head of the block. A node in the graph denotes a statement and is represented by the set of argument positions in the statement. An edge is created from a node $S_1$ to a node $S_2$ if the statement denoted by $S_2$ is dependent on the statement denoted by $S_1$. The node $S_1$ is said to be a *predecessor* of the node $S_2$ and the node $S_2$ a *successor* of the node $S_1$. The arguments in the head of a block are specially treated in the graph. They are divided into two nodes: the *start* node consisting of the set of input argument positions has no predecessor and the *end* node consisting of the set of output argument positions has no successor.

In the following $\texttt{I}n(G, n)$ returns the set of input argument positions in node $n$ and $\texttt{O}ut(G, n)$ returns the set of output argument positions in node

$n$. Let $S$ be a statement corresponding to a node $n$ in $G$, with $\mathtt{In}(G,n) = \{t_1,\ldots,t_m\}$. Let $\Upsilon_t$ denote the size function of an input argument position $t$. The size function of an output argument position in a statement depends, in general, on the size of the input argument positions in that statement: let the $i^{th}$ argument position of $S$ be an output argument, then its size function is denoted by $\Psi_S^{(i)}(\Upsilon_{t_1},\ldots,\Upsilon_{t_m})$.

Let $s$ and $e$ denote the start node and the end node of $G$, and $B = N-\{s,e\}$ the set of nodes for the statements. We distinguish between intra-statement argument size relations, which refer to size relations between the argument positions of a single statement, and inter-statement argument size relations, which refer to relations between argument positions of different statements. Then $D = \mathtt{Out}(G,e) \cup \bigcup_{n\in B} \mathtt{In}(G,n)$ denotes the set of argument positions for which the inter-statement argument size relations need to be computed, and $I = \bigcup_{n\in B} \mathtt{Out}(G,n)$ denotes the set of argument positions for which the intra-statement argument size relations need to be computed.

**Size Relation Equations:** The aim of this phase is to set up the size relation equations necessary to yield a closed form solution for the sizes corresponding to the output arguments in the head of a block and defined in terms of its input argument sizes. To do this we need to establish size relation equations for arguments in statements and for the head arguments. Regarding the arguments in a statement $S$ we have:

- *Output arguments.* Let $\mathtt{sz}_t$ denote the size of an output argument position $t$, and $s_1,\ldots,s_m$ the input arguments of the statement $S$. We construct a general equation as follows:
$$\mathtt{sz}_t \leq \Psi_S^t(\Upsilon(s_1),\ldots,\Upsilon(s_m))$$
Note that this equation can be used both with non-recursive statements and recursive statements. However, in the case of non-recursive statements the sizes for each of the $s_i$ have already been computed previously.

- *Input arguments.* Given an input argument position $t$ in $S$, let $\mathtt{predecessors}(t)$ be the set of predecessors of $t$ in the data dependency graph, we have the following possibilities:

  1. Compute $\mathtt{size\_metric}(t)$. If $\mathtt{size\_metric}(t) \neq \bot$ then the result is $\mathtt{sz}_t \leq \mathtt{size\_metric}(t)$.

2. Otherwise, if $\exists r \in \texttt{predecessors}(t)$ such that the size metrics corresponding to $r$ and $t$ are compatible then $\texttt{sz}_t \leq \texttt{sz}_r + \texttt{diff}(t, r)$.
3. Otherwise, $\texttt{sz}_t = \bot$.

Given the size relations for the body statements in a conditional statement computed above, the key step in the size analysis is to define the size relations for the output argument positions in the head of the block into functions in terms of sizes of the input argument positions in the head. An algorithm called *normalization* is defined in [5] which basically repeatedly transforms size relations for body statementsinto size relations for head arguments. When the algorithm terminates all size relations for body statements are defined in terms of head arguments. Note that in case of a recursive block the size of the output arguments of the recursive statements in the body are expressed symbolically in terms of its input sizes. Let $t$ be an head output argument, and let $\Phi_t$ denote the size function for the $t_{th}$ output argument in the head. We can define the size equations for the head output arguments as follows:

1. If $\exists r \in \texttt{predecessor}(t)$ then $\Phi_t \leq \texttt{sz}_r + \texttt{diff}(t, r)$
2. Otherwise, $\Phi_t = \bot$

**Example 4.1** Consider again the $\mathcal{L}$ program obtained in Fig. 4. Let *block_name* be the name of a block and $block\_name_i$ be the size of its $i^{th}$ argument. Let $head_i$ be the size of the conditional statement head $i^{th}$ argument. First, we set up the size relation equations for the output/input arguments corresponding statements and the output arguments of the head:

$$
\begin{array}{lcl}
\texttt{first}_1 & = & size\_metric(\texttt{buffer\_i}) = \texttt{head}_1 + diff(\texttt{buffer\_i}, \texttt{buffer\_i}) = \texttt{head}_1 \\
\texttt{first}_2 & = & size\_metric(\texttt{c}) = 8 \\
\texttt{exchangeByte}_1 & = & \texttt{first}_2 \\
\texttt{exchangeByte}_2 & = & size\_metric(\texttt{stream}) = \texttt{head}_2 + diff(\texttt{stream}, \texttt{stream}) = \texttt{head}_2 \\
\texttt{exchangeByte}_3 & = & size\_metric(\texttt{data}) = size\_metric(\texttt{c}) = 8 \\
\texttt{next}_1 & = & size\_metric(\texttt{buffer\_i}) = \texttt{head}_1 + diff(\texttt{buffer\_i}, \texttt{buffer\_i}) = \texttt{head}_1 \\
\texttt{next}_2 & = & size\_metric(\texttt{buffer\_0}) = \texttt{head}_1 + diff(\texttt{buffer\_i}, \texttt{buffer\_0}) = \texttt{head}_1 - 1 \\
\texttt{exchangeBuffer}_1 & = & \texttt{next}_2 + diff(\texttt{buffer\_0}, \texttt{buffer\_0}) = \texttt{next}_2 \\
\texttt{exchangeBuffer}_2 & = & size\_metric(\texttt{stream}) = \texttt{head}_2 + diff(\texttt{stream}, \texttt{stream}) = \texttt{head}_2 \\
\texttt{exchangeBuffer}_3 & = & \Psi^3_{\texttt{exchangeBuffer}}(\texttt{sendBuffer}_1, \texttt{sendBuffer}_2) \\
\texttt{insert}_1 & = & size\_metric(\texttt{buffer\_1}) = \texttt{exchangeBuffer}_3 + diff(\texttt{buffer\_1}, \texttt{buffer\_1}) \\
 & = & \texttt{exchangeBuffer}_3 \\
\texttt{insert}_2 & = & size\_metric(\texttt{data}) = \texttt{exchangeByte}_2 + diff(\texttt{data}, \texttt{data}) = \texttt{exchangeByte}_2 \\
\texttt{insert}_3 & = & \Psi^3_{\texttt{insert}}(\texttt{insert}_1, \texttt{insert}_2) \\
\texttt{head}_3 & = & size\_metric(\texttt{buffer\_o}) + diff(\texttt{buffer\_o}, \texttt{buffer\_o}) = size\_metric(\texttt{buffer\_o})
\end{array}
$$

Normalizing the above equations, we obtain:

$$
\begin{array}{lcl}
\texttt{first}_1 & = & \texttt{head}_1 \\
\texttt{first}_2 & = & 8 \\
\texttt{exchangeByte}_1 & = & 8 \\
\texttt{exchangeByte}_2 & = & \texttt{head}_2 \\
\texttt{exchangeByte}_3 & = & 8 \\
\texttt{next}_1 & = & \texttt{head}_1 \\
\texttt{next}_2 & = & \texttt{head}_1 - 1 \\
\texttt{exchangeBuffer}_1 & = & \texttt{head}_1 - 1 \\
\texttt{exchangeBuffer}_2 & = & \texttt{head}_2 \\
\texttt{exchangeBuffer}_3 & = & \Psi^3_{\texttt{exchangeBuffer}}(\texttt{head}_1 - 1, \texttt{head}_2) \\
\texttt{insert}_1 & = & \Psi^3_{\texttt{exchangeBuffer}}(\texttt{head}_1 - 1, \texttt{head}_2) \\
\texttt{insert}_2 & = & \texttt{head}_2 \\
\texttt{insert}_3 & = & 1 + \Psi^3_{\texttt{exchangeBuffer}}(\texttt{head}_1 - 1, \texttt{head}_2) \\
\texttt{head}_3 & = & 1 + \Psi^3_{\texttt{exchangeBuffer}}(\texttt{head}_1 - 1, \texttt{head}_2)
\end{array}
$$

Finally, we establish the recurrence equation for the output argument since it belongs to a recursive block, and we obtain its closed form. Note that we use the first statement as boundary condition.[3]

$$
\begin{array}{lcl}
\Psi^3_{\texttt{exchangeBuffer}}(0, \texttt{head}_2) & = & 0 \\
\Psi^3_{\texttt{exchangeBuffer}}(\texttt{head}_1, \texttt{head}_2) & = & 1 + \Psi^3_{\texttt{exchangeBuffer}}(\texttt{head}_1 - 1, \texttt{head}_2)
\end{array}
$$

$$
\Psi^3_{\texttt{exchangeBuffer}}(\texttt{head}_1, \texttt{head}_2) = \texttt{head}_1
$$

# 5 Resource Usage Analysis

As mentioned before, our static resource usage cost analysis approach is based on that of [6, 5] (for estimation of upper bounds on execution steps), further extended in [7] for lower bounds. In these approaches the cost of a block definition (generally taken as a number of execution steps) can be bounded by the cost of the basic operations in the body of the block (including the parameter passing cost), combined with bounds on the cost of each of the block calls in the body. However, in our approach, the basic metric is open and can be tailored to the use of other metrics as the unit of cost in the analysis.

Assume that the program is analyzed in a single traversal of the call graph in reverse topological order. Consider a block definition `p` given as `H` $\{\texttt{C}_1, ..., \texttt{C}_\texttt{m}\}$' where each $\texttt{C}_\texttt{i}$ is a conditional statement. Assume that each $\texttt{C}_\texttt{i}$, $1 \leq i \leq m$ is of the form $stcond_i \rightarrow \texttt{S}^\texttt{i}_1, ..., \texttt{S}^\texttt{i}_\texttt{k}$ where $\texttt{S}^\texttt{i}_\texttt{j}$, $1 \leq j \leq k$, is

[3] Note that $head_3$ is defined by the $\Psi^3_{exchangeBuffer}$ function.

a statement (either a block call or a primitive operation of the language). Assume that $\overline{n}$ is a vector such that each element corresponds to the size of an input argument position to block p. Assume that $\psi_j(\overline{n})$ is a vector with the sizes of all the input arguments to the statement $\mathtt{S}^{\mathtt{i}}_{\mathtt{j}}$, given as functions of the sizes of the input arguments to the head of block p (the one being analyzed). Note that these $\psi_j(\overline{n})$ size relations have previously been computed during size analysis for all input arguments to statements in the body of all conditional statements.

Then, the cost (expressed in units of resource $r$ with approximation $ap$) of a call to p, $\mathtt{Cost}_{\mathtt{block}}(p, ap, r, \overline{n})$ for an input of size $\overline{n}$ (assuming that only a conditional statement is executed), can be expressed as:

$$\mathtt{Cost}_{\mathtt{block}}(p, ap, r, \overline{n}) = \bigodot(ap)_{1 \leq i \leq m}\{\mathtt{Cost}_{\mathtt{cond}}(C_i, p, ap, r, \overline{n})\} \qquad (1)$$

where $\bigodot(ap)$ is a function that takes an approximation identifier $ap$ and returns a function which applies over all $\mathtt{Cost}_{\mathtt{cond}}(C_i, p, ap, r, \overline{n})$, for $1 \leq i \leq m$. For example, if $ap$ is the identifier for approximation "upper bound" ($ub$), then $\bigodot(ap)$ is the $max$ function, which returns the maximum of all $\mathtt{Cost}_{\mathtt{cond}}(C_i, p, ap, r, \overline{n})$, for $1 \leq i \leq m$. If $ap$ is the identifier for approximation "lower bounds" ($lb$), then $\bigodot(ap)$ is the $min$ function. The function $\bigodot(ap)$ is provided by means of assertions in the program.

In turn, the cost (expressed in units of resource $r$ with approximation $ap$) of conditional statement $\mathtt{C_i}$, can be expressed as:

$$\begin{aligned} \mathtt{Cost}_{\mathtt{cond}}(\mathtt{C_i}, p, ap, r, \overline{n}) = \quad & \delta(ap, r)(p) + \\ & \mathtt{Cost}_{\mathtt{sc}}(stcond_i, ap, r, \overline{n}) + \\ & \sum_{j=1}^{k} \mathtt{Cost}_{\mathtt{stm}}(S^i_j, ap, r, \psi_j(\overline{n})). \end{aligned} \qquad (2)$$

where $\delta(ap, r)$ is a function that takes an approximation identifier $ap$ and a resource identifier $r$ and returns a function $\Delta : block\_name \rightarrow arith\_expr$ which takes a block name and returns an arithmetic cost expression $< arith\_expr >$ as defined in Section 2. Thus, $\delta(ap, r)(p)$ represents $\Delta(p)$. For example, if the resource we want to measure is the number of block calls (steps), we can define $\delta(ub, steps) = return\_1$, and define the function $return\_1$ as $return\_1(x) = 1$. If the resource we want to measure is $nc\_exchangeBytes$, the number of calls to block $exchangeBytes$, then we can define the function $\delta(ub, nc\_exchangeBytes) = exchb$, and define the function $exchb$ as:

$$exchb(p) = \begin{cases} 1 & \text{if } p = nc\_exchangeBytes \\ 0 & otherwise \end{cases}$$

$\texttt{Cost}_{\texttt{stm}}(\texttt{S}^{\texttt{i}}_{\texttt{j}}, ap, r, \psi_j(\overline{n}))$ is:

- If $\texttt{S}^{\texttt{i}}_{\texttt{j}}$ is recursive (i.e., performs a call to a block which is in the strongly-connected component of the call graph being analyzed), then $\texttt{Cost}_{\texttt{stm}}(\texttt{S}^{\texttt{i}}_{\texttt{j}}, ap, r, \psi_j(\overline{n}))$ is expressed as a symbolic expression: $\texttt{Cost}_{\texttt{block}}(\texttt{S}^{\texttt{i}}_{\texttt{j}}, ap, r, \psi_j(\overline{n}))$
- If $\texttt{S}^{\texttt{i}}_{\texttt{j}}$ is not recursive, assume that is a call to q (where q can be a primitive name or a block name):
  - If there is a resource usage assertion for q, $res\_usage(ap, r, \Phi)$, then $\texttt{Cost}_{\texttt{stm}}(\texttt{S}^{\texttt{i}}_{\texttt{j}}, ap, r, \psi_j(\overline{n}))$ is replaced by the arithmetic cost expression (in closed form) $\Phi(\psi_j(\overline{n}))$.
  - Otherwise, q has been already analyzed, i.e., the cost function for q, has been recursively computed as $\Phi'$ (a closed form cost function) and $\texttt{Cost}_{\texttt{stm}}(\texttt{S}^{\texttt{i}}_{\texttt{j}}, ap, r, \overline{n})$ can be expressed explicitly in terms of the function $\Phi'$, and it is thus replaced by $\Phi(\psi_j(\overline{n}))$.

$\texttt{Cost}_{\texttt{sc}}(stcond_i, ap, r, \overline{n})$ is the cost of evaluating the condition $stcond_i$ plus the cost of evaluating all the conditions of the preceding conditional statements.

It can be proved by induction on the number of statements in the body of conditional statement $\texttt{C}_{\texttt{i}}$ that:

1. If conditional statement $\texttt{C}_{\texttt{i}}$ is nonrecursive, then, expression 2 results in a closed form function of the sizes of the input argument positions in the head of block p;
2. If conditional statement $\texttt{C}_{\texttt{i}}$ is simply recursive, then, expression 2 results in a recurrence equation in terms of the sizes of the input argument positions in the head of block p;
3. If conditional statement $\texttt{C}_{\texttt{i}}$ is mutually recursive, then expression 2 results in a recurrence equation which is part of a system of equations for mutually recursive conditional statements in terms of the sizes of the input argument positions in the head of block p.

If these recurrence equations can be solved then $\texttt{Cost}_{\texttt{cond}}(\texttt{C}_{\texttt{i}}, p, ap, r, \overline{n})$ in expression 2 can be expressed in a closed form, which is a function of the sizes of the input argument positions in the head of block p. Thus, after the strongly-connected component to which p belongs in the call graph has been analyzed, we have that expression 1, results in a closed form function of the sizes of the input argument positions in the head of block p.

Note that our analysis is parameterized by the functions $\delta(ap, r)$ and $\bigodot(ap)$, whose definition can be given by means of assertions of type $<$ *resource_delta* $>$ and $<$ *approx_def* $>$ respectively, as given in Figure 2. These functions make our analysis parametric w.r.t. resources (as execution steps or calls to a procedure) and types of approximations (as lower and upper bounds).

**Example 5.1** Consider the same program defined in Fig. 4 and the size relations computed in Ex. 4.1. Assume that $\psi_S(\overline{n})$ is the vector with the sizes of all the input arguments to the statement `S`, and it is defined as follows:

$$\begin{aligned}\psi_{\texttt{exhange\_buffer}}(\overline{n}) = \; & (\langle size(\texttt{buffer\_i}, ub, length, length(\texttt{buffer\_i}))\rangle,\\ & \langle size(\texttt{c}, ub, size, size(\texttt{c}))\rangle, \langle size(\texttt{s}, ub, size, 1)\rangle,\\ & \langle size(\texttt{s}, ub, size, 1)\rangle, \langle size(\texttt{data}, ub, size, size(\texttt{c}))\rangle,\\ & \langle size(\texttt{buffer\_0}, ub, length, length(\texttt{buffer\_i}) - 1)\rangle,\\ & \langle size(\texttt{buffer\_1}, ub, length, length(\texttt{buffer\_0}))\rangle,\\ & \langle size(\texttt{buffer\_o}, ub, length, length(\texttt{buffer\_1}) + 1)\rangle)\end{aligned}$$

We define the following cost equations for each conditional statement:

$$\begin{aligned}Cost_{\texttt{exchangeBuffer}_1}(0, \_) &= 0\\ Cost_{\texttt{exchangeBuffer}_2}(\texttt{b\_i}, \texttt{s}) &= \delta(ub, bits\_received)(\texttt{exchangeBuffer}) +\\ &\quad Cost(\texttt{empty}, ub, bits\_received, \texttt{b\_i}) +\\ &\quad Cost(\texttt{first}, ub, bits\_received, \psi_{\texttt{exhange\_buffer}}(\overline{n})) +\\ &\quad Cost(\texttt{exchangeByte}, ub, bits\_received, \psi_{\texttt{exhange\_buffer}}(\overline{n})) +\\ &\quad Cost(\texttt{next}, ub, bits\_received, \psi_{\texttt{exhange\_buffer}}(\overline{n})) +\\ &\quad Cost(\texttt{exchangeBuffer}, ub, bits\_received, \psi_{\texttt{exhange\_buffer}}(\overline{n})) +\\ &\quad Cost(\texttt{insert}, ub, bits\_received, \psi_{\texttt{exhange\_buffer}}(\overline{n}))\\ &= 8 + Cost(\texttt{exchangeBuffer}, ub, bits\_received, \psi_{\texttt{exhange\_buffer}}(\overline{n}))\\ &= 8 + length(\texttt{b\_i})\end{aligned}$$

# 6 Experimental results

In order to study the kinds of resource usage bound functions inferred by our analysis as well as the time required to infer them we have completed a prototype implementation of our approach. This implementation is written in Ciao and uses a number of modules and facilities from the `CiaoPP` multiparadigm preprocessor. The results are shown in Table 1. The column labeled **Approx** shows the type of approximation performed by the analysis (for brevity we provide results only on upper bounds). The column labeled **Resource** shows the actual resource for which bounds are being inferred by the analysis for a given benchmark. While any of the resources could be used for any of the benchmarks we show only the results for the most natural or

| Program | Approx | Resource | Cost function | Metric for x | T |
|---|---|---|---|---|---|
| `trans` | ub | "ext. accesses" | $\lambda x.x^2 + 1$ | length | 66 |
| `nfib` | ub | "arith ops" | $\lambda x.2.17 \times 1.618^x + 0.82 \times (-0.618)^x - 3$ | value | 64 |
| `disk` | ub | "head moves" | $\lambda x.2^x - 1$ | value | 56 |
| `file` | ub | "open files"<br>"closed files"<br>"unclosed files" | $\lambda x.3x$<br>$\lambda x.2x + 1$<br>$\lambda x.x - 1$ | length | 52 |
| `nrev` | ub | "steps" | $\lambda x.0.5x^2 + 1.5x + 1$ | length | 32 |
| `send` | ub | "bits sent" | $\lambda x.8x$ | length | 24 |
| `exch` | ub | "bits exchanged" | $\lambda x.16x$ | length | 28 |
| `qsort` | ub | "lists paral." | $\lambda x.4.2^x - 2x - 4$ | length | 140 |

Table 1: Accuracy and efficiency of the resource analysis.

interesting resource for each one of them. We have tried to use a relatively wide range of resources: number of bytes sent by an application, number of calls to a particular procedure, robot arm movements, money spent in a commercial transaction, number of accesses to a database, etc. The column **Cost function** shows the actual cost function (which depends on the size of the input arguments) inferred by the analysis, given as a lambda term. Finally, the column labeled **T** shows analysis times in milliseconds, on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 4Gb of RAM memory, running Fedora Core 5.0. These are quite reasonable for this relatively small benchmarks (but scalability should obviously be studied in future work).

In `trans` (a database transaction which carries out accesses to different tables), we decided to measure the number of accesses to an external table as a function of the length of the input data. In `nfib` (the naive, doubly recursive implementation of finding the nth. Fibonacci number) we decided to count the number of arithmetic operations performed. The result is given as a function of the (integer) value of the input argument to fib representing the ordinal of the Fibonacci number to be computed (the metric, as in the rest of the cases, was inferred automatically from type analysis). In `disk` (a standard implementation of the towers of Hanoi problem but where we introduced calls to a virtual robotic arm that is assumed to be moving the disks) we decided to count the number of movements that the robotic arm has to make. The result is given as a function of the value of the input argu-

ment to disk: the number of disks to move. In `file` (a typical piece of code in an operative system kernel), we kept track of the number of unclosed file descriptors as a function of the length of the input file descriptors. In `nrev` (the standard naive recursive implementation of naive reversal of a list) we simply decided to measure the number of steps which is obtained as a function of the length of the input list. This experiment essentially reproduces the results of previous analyses aimed specifically at this steps measure. In `send` (an extended version of the program of Figure 3) we decided to measure the number of bits sent, which is obtained as a function of the length of the input buffer. In `exch` (which is a more complex communications program where information is both sent and received) we also measure number of bits sent (in both directions). Finally in `qsort` (standard quick-sort algorithm) we decided to count the number of list splits, which determines the number recursions or iterations that can be executed in parallel. The (upper bound) results are obtained as a function of the length of the input lists.

# 7 Conclusions

We have presented a generic analysis that infers upper or lower bounds on the usage that a program makes of a quite general notion of user-definable resources. The inferred bounds are in general functions of input data sizes. We have also presented the assertion language which is used to define such resources for the basic components of the language. The analysis then derives the related (upper- and lower-bound) cost functions for all procedures in the program. Our experimental results are encouraging because they show that interesting resource bound functions can be obtained automatically and in reasonable time, at least for our (small) benchmarks. Also, we expect the applications of our analysis to be rather interesting, including resource consumption verification and debugging (including for mobile code), resource control in parallel/distributed computing, and resource-oriented specialization [4, 14].

# References


[1] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resource verification. In *TPHOLs2004*, volume 3223 of *LNCS*, pages 34–49, Heidelberg, September 2004. Springer Verlag.

[2] D. Basin and H. Ganzinger. Complexity Analysis based on Ordered Resolution. In *11th. IEEE Symposium on Logic in Computer Science*, 1996.

[3] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *ESOP*, number 3444 in LNCS, pages 311–325. Springer-Verlag, 2005.

[4] S.J. Craig and M. Leuschel. Self-tuning resource aware specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.

[5] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[6] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[7] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[8] M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[9] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[10] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[11] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[12] David A. McAllester. On the complexity analysis of static analyses. In *Static Analysis Symposium*, pages 312–329, 1999.

[13] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Automatic complexity analysis. In *European Symposium on Programming*, pages 243–261, 2002.

[14] G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 261–271. ACM Press, July 2006.

[15] M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

[16] M. Rosendahl. The essence of computation. complexity, analysis, transformation. In Springer-Verlag LNCS 2566, editor, *Essays Dedicated to Neil D. Jones*, pages 404 – 419, 2002.