

# Towards a Framework for Resource Usage Verification and Debugging in the CiaoPP System

Pedro Lopez-Garcia      Luthfi Darmawan  
Francisco Bueno  
pedro.lopez@imdea.org    bueno@fi.upm.es

January 23, 2010

## Abstract

We present a framework for (static) verification of general resource usage program properties. The framework extends the criteria of correctness as the conformance of a program to a specification expressing non-functional global properties (e.g., upper and lower bounds on execution time, memory, power, or user defined resources). Such bounds are given as functions on input data sizes. A given specification can include both, lower and upper bound resource usage functions, i.e., it can express intervals where the resource usage is supposed to be included in. We have defined an abstract semantics for resource usage properties and operations to compare the (approximated) intended semantics of a program (i.e., the specification) with approximated semantics inferred by static analysis. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential or logarithmic functions). A novel aspect of our framework is that the static checking of assertions generate answers that include conditions under which a given specification can be proved or disproved. For example, these conditions can express intervals of input data sizes such that a given specification can be proved for some intervals but disproved for others. We have implemented our techniques within the Ciao/CiaoPP system in a natural way, so that the novel resource usage verification blends in with the CiaoPP framework that unifies static verification and static debugging (as well as run-time verification and unit testing).

## 1 Introduction and Motivation

The conventional understanding of software correctness is absence of errors or bugs, expressed in terms of conformance of all possible executions of the program with a functional specification (like type correctness) or behavioral specification (like termination or possible sequences of actions). However, in an increasing number of computing applications additional observables play an essential role.

For example, embedded systems must control and react to the environment, which also establishes constraints about the system’s behavior such as resource usage and reaction times. Therefore, it is necessary for these systems to extend the criteria for correctness with new aspects which include non-functional global properties such as maximum execution time, and usage of memory, power, or other types of resources.

In this paper we propose the extension of debugging and verification techniques based on static analysis [BDD<sup>+</sup>97, BCC<sup>+</sup>03, HPBG05] to deal with a quite general class of properties related to resource usage including upper and lower bounds on execution time, memory, power, and user-defined resources (the later in the sense of [NMLGH07]). Such bounds are given as functions on input data sizes (see [DL93] for the different metrics that can be used to measure data sizes, such as list-length, term-depth or term-size). The proposed extension blends in with the CiaoPP framework that unifies static verification and static debugging (as well as run-time verification and unit testing) [MLGH09]. For example, it can be used by CiaoPP to certify programs with resource consumption assurances and also to efficiently check such certificates [HALGP04]. We have defined an abstract semantics for resource usage properties and operations to compare the (approximated) intended semantics of a program (i.e., the specification, given as assertions in the program) with approximated semantics inferred by static analysis. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential or logarithmic functions). In traditional static checking, for each property or (part of) an assertion, the possible outcomes are *true* (property proved to hold), *false* (property proved not to hold), and *unknown* (the analysis cannot prove true or false). A novel aspect of the *resource verification and debugging* approach that we propose is that, to be useful, the answers of the checking process must go beyond these classical outcomes and will typically include conditions under which the truth or falsity of the property can be proved. Such conditions can be parameterized by attributes of inputs, such as input data size or value ranges. For example, it may be possible to say that the outcome is true if the input data size is in a given range.

Our verification framework automatically selects the appropriate analysis information necessary to check a given resource usage specification, depending on the kind of information expressed in such specification. This information could be for example lower and upper bounds, and even asymptotic values of the resource usage of the computation (given as functions on input data sizes). Moreover, a given specification can include both, lower and upper bound resource usage functions, i.e., it can express intervals where the resource usage is supposed to be included in.

Consider for example the naive reverse program in Figure 1, with the classical definition of predicate `append`. Assume that the programmer thinks that the cost of `nrev` is given by a linear function on the size (list-length) of its first argument, maybe because he has not taken into account the cost of the call to `append`. Since `append` is linear, it causes `nrev` to be quadratic. We will see how CiaoPP, the preprocessor of the Ciao system, is able to inform the

```

:- module(reverse, [nrev/2], [assertions]).
:- use_module(library('assertions/native_props')).
:- entry nrev(A,B) : (ground(A), list(A, term), var(B)).

nrev([], []).
nrev([H|L],R) :- nrev(L,R1), append(R1, [H], R).

```

Figure 1: A module for naive reverse.

programmer about this false idea of the cost of `nrev`. For example, assume that the programmer adds the following assertion to be checked:

```
:- check comp nrev(A,B) + steps_ub( length(A)+1 ).
```

CiaoPP issues the following error message:

```

ERROR: false comp assertion:
      :- comp nrev(A,B) : true => steps_ub(length(A)+1)
      because in the computation the following holds:
      steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1)

```

This message states that `nrev` will take *at least*  $0.5 (\text{length}(A))^2 + 1.5 \text{length}(A) + 1$  resolution steps (which is the resource usage analysis output), while the assertion requires that it take *at most*  $\text{length}(A) + 1$  resolution steps. The resource usage function in the user-provided assertion is compared with the lower-bound resource usage information inferred by analysis. This allows detecting the inconsistency and proving that the program does not satisfy the efficiency requirements imposed.

Consider now a pessimistic programmer who over-estimates the cost of predicate `nrev` and writes:

```
:- check comp nrev(A,B) + steps_ub( exp(length(A),2) ).
```

Function  $(\text{length}(A))^2$  (F2 in Fig. 2, where  $x = \text{length}(A)$ ) is hardly comparable with the upper bound resource usage function obtained by the analysis:  $0.5 (\text{length}(A))^2 + 1.5 \text{length}(A) + 1$  (F1 in Fig. 2, where  $x = \text{length}(A)$ ; the same as the lower bound obtained by it, incidentally). The approach up to now in these cases was to compare the terms of both functions which have the highest order to infer whether it can be determined that one is asymptotically greater than the other. In the example, the term  $x^2$  of F2 is greater than the term  $0.5 x^2$  of F1, so that the assertion holds (asymptotically).

However, a more precise (and useful) approach to the above example would be to consider the assertion proved only for the interval in which the resource usage function inferred by the analysis (F1) is lower than the one in the assertion (F2). Thus, a sensible output of the debugging tool would better be:

```

:- checked comp nrev(A,B) + steps_ub( exp(length(A),2) )
   in interval [-inf , -0.561552539563305] for length(A).

```

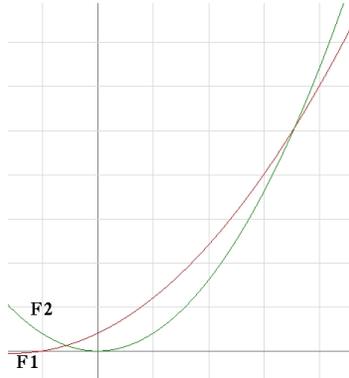


Figure 2: Functions  $F1=0.5x^2+1.5x + 1$  and  $F2=x^2$

```

:- check comp nrev(A,B) + steps_ub( exp(length(A),2) )
   in interval [-0.561552539563305 , 3.56155281280883] for length(A).
:- checked comp nrev(A,B) + steps_ub( exp(length(A),2) )
   in interval [3.56155281280883 , +inf] for length(A).

```

showing that the assertion remains to be proved or disproved in the intermediate interval between the roots of the function  $F1-F2$ .

Consider now an assertion with the resource usage function  $0.2x^2$  mentioned before as upper bound, plus another one stating a linear exact resource usage function, to motivate our proposal, as follows:

```

:- check comp nrev(A,B) + steps_ub( 0.2*exp(length(A),2) ).
:- check comp nrev(A,B) + steps_o( length(A) ).

```

For the first one, the tool will now issue an output that shows the relevant intervals computed for function comparison, in the following form:

```

:- check comp nrev(A,B) + steps_ub( 0.2*exp(length(A),2) )
   with [ steps_ub( 0.5*exp(length(A),2)+1.5*length(A)+1 ),
          steps_lb( 0.5*exp(length(A),2)+1.5*length(A)+1 ) ]
   in intervals
      [ -5.32469507659596, -4.207825127659933, -0.874999999999997,
        -0.7921748311282393, 1.324695143552614 ] .

```

This assertion shows the original one adorned with the results from analysis. In this case, that analysis has inferred a lower (and upper) bound resource usage function of  $0.5*length(A)^2+1.5*length(A) + 1$ . The functions to be compared are subtracted and the roots of the resulting function computed. If there exist roots, it means that the two functions have intersection, therefore there are closed intervals in which the result of their comparison (and, thus, that of the debugging/verification process) may be different. The intersection of the two functions in our example is illustrated in Fig. 3 (F1 and F2). In this case the roots are -4.21 and -0.79, therefore there are three intervals:  $[\infty,-4.21]$ ,  $[-4.21,-0.79]$ , and  $[-0.79,\infty]$ . These are shown (with approximate values) in the above

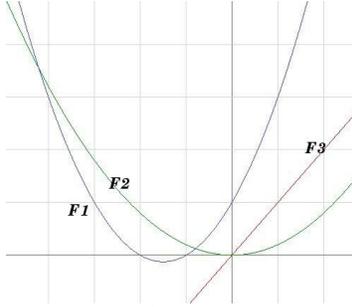


Figure 3: Functions  $F1 = 0.5*x^2 + 1.5*x + 1$ ,  $F2 = 0.2*x^2$  and  $F3 = x$

output assertion alternating with sample values of the difference function on each interval: first value is a sample for interval  $[\infty, -4.21]$ , the third value a sample for  $[-4.21, -0.79]$ , and the fifth and last value a sample for  $[-0.79, \infty]$ .

Interval information is only shown when the resource usage assertion is incompatible or it can not be decided. In case that the assertion is compatible no interval information is shown. The information might be shown but be actually empty, like in the following example for the second resource usage assertion above:

```
:- check comp nrev(A,_1) + steps_o( length(A) )
   with [ steps_ub( 0.5*exp(length(A),2)+1.5*length(A)+1 ),
          steps_lb( 0.5*exp(length(A),2)+1.5*length(A)+1 ) ]
   in intervals [ ] .
```

In this case, cost complexity function  $length(A)$  does not have intersection with cost lower bound function  $0.5*length(A)^2 + 1.5*length + 1$ , so that no interval information is shown (Fig. 3, F1 and F3).

Upper-bound resource usage assertions can also be proved to hold, i.e., can be *checked*, by using upper-bound resource usage analysis rather than lower-bound resource usage analysis. In such case, if the upper-bound computed by analysis is lower or equal than the upper-bound stated by the user in the assertion. The converse holds for lower-bound resource usage assertions.

In the following, in Sect. 2 we first present the CiaoPP verification framework that we take as starting point. Sect. 3 describes how it is used and extended for the verification of general resource usage program properties. Sect. 4 then explain the technique that we have developed for resource usage function comparison. Finally, Sect. 5 briefly reports on the implementation of our techniques within the Ciao/CiaoPP system, and Sect. 6 summarizes our conclusions and explains some improvements of our framework left for future work.

## 2 The Framework

We take as starting point an existing framework for static *verification* and *debugging* [PBH00b], which has been implemented and integrated into the

Property	Definition
$P$ is partially correct w.r.t. $I$	$\llbracket P \rrbracket \subseteq I$
$P$ is complete w.r.t. $I$	$I \subseteq \llbracket P \rrbracket$
$P$ is incorrect w.r.t. $I$	$\llbracket P \rrbracket \not\subseteq I$
$P$ is incomplete w.r.t. $I$	$I \not\subseteq \llbracket P \rrbracket$

Table 1: Set theoretic formulation of verification problems

CiaoPP system. In this work we extend the framework in order to deal with resource usage properties. The verification and debugging framework uses abstract interpretation based analysis, which are provably correct and also practical, in order to statically compute semantic approximations of programs. These semantic approximations are compared with (partial) specifications, in the form of assertions that are written by the programmer, in order to detect inconsistencies or to prove such assertions.

Both program verification and debugging compare the *actual semantics*  $\llbracket P \rrbracket$  of a program  $P$  with an *intended semantics* for the same program, which we will denote by  $I$ . This intended semantics embodies the user’s requirements, i.e., it is an expression of the user’s expectations. In Table 1 we show classical verification problems in a set-theoretic formulation as simple relations between  $\llbracket P \rrbracket$  and  $I$ . Using the exact actual or intended semantics for automatic verification and debugging is in general not realistic, since the exact semantics can be typically only partially known, infinite, too expensive to compute, etc. On the other hand the abstract interpretation technique allows computing *safe* approximations of the program semantics. The key idea in our approach is to use the abstract approximation  $\llbracket P \rrbracket_\alpha$  directly in program verification and debugging tasks.

A number of approaches have already been proposed which make use to some extent of abstract interpretation in verification and/or debugging tasks. Abstractions were used in the context of algorithmic debugging in [LS88]. Abstract interpretation for debugging of imperative programs has been studied by Bourdoncle [Bou93], by Comini et al. for the particular case of algorithmic debugging of logic programs [CLV95] (making use of partial specifications) [CLMV99], and by P. Cousot [Cou03]. Additional discussion and more details about the foundations and implementation issues of our approach can be found in [BDD<sup>+</sup>97, HPB99, HPBG05].

## 2.1 Abstract Verification and Debugging

In our framework the abstract approximation  $\llbracket P \rrbracket_\alpha$  of the concrete semantics  $\llbracket P \rrbracket$  of the program is actually computed and compared directly to the (also approximate) intention (which is given in terms of *assertions* [PBH00a]), following almost directly the scheme of Table 1. We safely assume that the program specification is given as an abstract value  $I_\alpha \in D_\alpha$  (where  $D_\alpha$  is the abstract domain of computation). Program verification is then performed by comparing  $I_\alpha$  and  $\llbracket P \rrbracket_\alpha$ . Table 2 shows sufficient conditions for correctness and completeness

Property	Definition	Sufficient condition
P is partially correct w.r.t. $I_\alpha$	$\alpha(\llbracket P \rrbracket) \subseteq I_\alpha$	$\llbracket P \rrbracket_{\alpha+} \subseteq I_\alpha$
P is complete w.r.t. $I_\alpha$	$I_\alpha \subseteq \alpha(\llbracket P \rrbracket)$	$I_\alpha \subseteq \llbracket P \rrbracket_{\alpha-}$
P is incorrect w.r.t. $I_\alpha$	$\alpha(\llbracket P \rrbracket) \not\subseteq I_\alpha$	$\llbracket P \rrbracket_{\alpha-} \not\subseteq I_\alpha$ , or $\llbracket P \rrbracket_{\alpha+} \cap I_\alpha = \emptyset \wedge \llbracket P \rrbracket_\alpha \neq \emptyset$
P is incomplete w.r.t. $I_\alpha$	$I_\alpha \not\subseteq \alpha(\llbracket P \rrbracket)$	$I_\alpha \not\subseteq \llbracket P \rrbracket_{\alpha+}$

Table 2: Verification problems using approximations

w.r.t.  $I_\alpha$ , which can be used when  $\llbracket P \rrbracket$  is approximated. Several instrumental conclusions can be drawn from these relations.

Analyses which over-approximate the actual semantics (i.e., those denoted as  $\llbracket P \rrbracket_{\alpha+}$ ), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification  $I_\alpha$ . It will also be sometimes possible to prove incorrectness in the case in which the semantics inferred for the program is incompatible with the abstract specification, i.e., when  $\llbracket P \rrbracket_{\alpha+} \cap I_\alpha = \emptyset$ . On the other hand, we use  $\llbracket P \rrbracket_{\alpha-}$  to denote the (less frequent) case in which analysis under-approximates the actual semantics. In such case, it will be possible to prove completeness and incorrectness.

We are interested in supporting properties that may be defined by means of user programs and extend beyond the predefined set which may be natively understandable by the available static analyzers. Also, only a small number of (even zero) assertions may be present in the program, i.e., the assertions are *optional*. In general, since most of the properties being inferred are in general undecidable at compile-time, the inference technique used, abstract interpretation, is necessarily approximate, i.e., possibly imprecise. Nevertheless, such approximations are also always guaranteed to be safe, in the sense that they are never incorrect.

## 2.2 Combined Static and Dynamic Verification/Debugging

CiaoPP is also capable of combined static and dynamic verification, and debugging, using the ideas outlined so far (see [HPB99] for details).

Program verification and detection of errors is first performed at compile-time by using the sufficient conditions shown in Table 2, i.e., by inferring properties of the program via abstract interpretation-based static analysis and comparing this information against (partial) specifications  $I_\alpha$  written in terms of assertions. For those assertions neither proved nor disproved the system may (optionally) instrument the program for checking them at run-time. Both the static and the dynamic checking are provably *safe* in the sense that all errors flagged are definite violations of the specifications.

### 3 Extending the Framework to Resource Usage Properties

We now define all the elements of the framework for its application to resource usage properties.

#### 3.1 Resource usage semantics

Given a program  $p$ , let  $\mathcal{C}_p$  be the set of all calls to  $p$ . The concrete resource usage semantics of a program  $p$ , for a particular resource of interest,  $\llbracket P \rrbracket$ , is a set of pairs  $(p(\bar{t}), r)$  such that  $\bar{t}$  is a tuple of terms,  $p(\bar{t}) \in \mathcal{C}_p$  is a call to predicate  $p$  with actual parameters  $\bar{t}$ , and  $r$  is a number expressing the amount of resource usage of the computation of the call  $p(\bar{t})$ . Such semantic object can be computed by a suitable operational semantics, such as SLD-resolution, adorned with the computation of the resource usage. We abstract away such computation, since it will be in general dependent on the particular resource  $r$  refers to. The concrete resource usage semantics can be defined as a function  $\llbracket P \rrbracket : \mathcal{C}_p \rightarrow \mathcal{R}$  where  $\mathcal{R}$  is the set of real numbers (note that depending on the type of resource we can take other set of numbers, e.g., the set of natural numbers).

The abstract resource usage semantics is a set of 4-tuples:

$$(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$$

The first element of the tuple,  $p(\bar{v}) : c(\bar{v})$ , is an abstraction of a set of calls.  $\bar{v}$  is a tuple of variables and  $c(\bar{v})$  is an abstraction representing a set of tuples of terms which are instances of  $\bar{v}$ .  $c(\bar{v})$  is an element of some abstract domain expressing instantiation states. The second argument of the tuple,  $\Phi$ , is an abstraction of the resource usage of the calls represented by  $p(\bar{v}) : c(\bar{v})$ . We refer to it as a *resource usage interval function* for  $p$ , defined as follows:

**Definition 1** A resource usage interval function for  $p$  is an arithmetic function,  $\Phi : \mathcal{R}^k \mapsto \mathcal{RI}$ , where  $\mathcal{R}$  is the set of real numbers,  $k$  is the number of input arguments to predicate  $p$  and  $\mathcal{RI}$  is the set of intervals of real numbers, such that  $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$  for all  $\bar{n} \in \mathcal{R}^k$ , where  $\Phi^l(\bar{n})$  and  $\Phi^u(\bar{n})$  are resource usage bound functions (see Definition 2) that denote the lower and upper endpoints of the interval  $\Phi(\bar{n})$  respectively for the tuple of input data sizes  $\bar{n}$ . Although  $\bar{n}$  is typically a tuple of natural numbers, we do not want to restrict our framework. We require that  $\Phi$  be well defined so that  $\forall \bar{n} (\Phi^l(\bar{n}) \leq \Phi^u(\bar{n}))$ .

**Definition 2** A resource usage bound function for  $p$  is a monotonic arithmetic function,  $\Psi : \mathcal{R}^k \mapsto \mathcal{R}_\infty$ , where  $\mathcal{R}$  is the set of real numbers,  $k$  is the number of input arguments to predicate  $p$  and  $\mathcal{R}_\infty$  is the set of real numbers augmented with the special symbols  $\infty$  and  $-\infty$ . We use such functions to express lower and upper bounds on the resource usage of predicate  $p$  depending on input data sizes.

Function  $input_p$  takes a tuple of terms  $\bar{t}$  and returns a tuple with the input arguments to  $p$ . This function can be inferred by using existing mode analysis or can be given by the user by means of assertions. Function  $size_p(\bar{t})$  takes a tuple of terms  $\bar{t}$  and returns a tuple with the sizes of those terms under a given metric. The metric used for measuring the size of each argument of  $p$  can be automatically inferred (based on type analysis information) or can be given by the user by means of assertions [NMLGH07].

In order to make the presentation simpler, we will omit the  $input_p$  and  $size_p$  functions in abstract tuples, with the understanding that they are present in all of such tuples.

**Example 1** *Assuming that the first argument of  $nrev$  and the two first arguments of  $append$  are input and that the size measure for all of them is list-length, we have that:*

$$input_{nrev}((x, y)) = (x), \quad input_{app}((x, y, z)) = (x, y), \\ size_{nrev}(x) = (length(x)) \quad \text{and} \quad size_{app}((x, y)) = (length(x), length(y)).$$

**Intended meaning** The intended approximated meaning  $I_\alpha$  of a program is an abstract semantic object with the same kind of tuples:  $(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$ , which are given in the form of assertions. The basic form of resource usage assertion<sup>1</sup> is:

$$\boxed{:- \text{comp } Pred \text{ [ : } Precond \text{ ] } + ResUsage.}$$

which expresses that for any call to  $Pred$ , if  $Precond$  is satisfied in the calling state, then  $ResUsage$  should also be satisfied for the computation of  $Pred$ .  $ResUsage$  defines in general an interval of numbers for the particular resource usage of the computation of the call to  $Pred$  (i.e.,  $ResUsage$  is satisfied by the computation of the call to  $Pred$  if the resource usage of such computation is in the defined interval). For example:

$$:- \text{comp } nrev(A,B) : (\text{ground}(A), \text{list}(A, \text{term}), \text{var}(B)) \\ + \text{resource}(\text{ub}, \text{steps}, 1 + \exp(\text{length}(A), 2)).$$

expresses that for any call to  $nrev(A,B)$  with the first argument bound to a ground list and the second one a free variable, an upper bound ( $\text{ub}$ ) on the number of resolution  $\text{steps}$  performed by the computation is  $1 + n^2$ , where  $n = \text{length}(A)$ . In this case, the interval approximating the number of resolution steps is  $[0, 1 + n^2]$ . Since the number of resolution steps cannot be negative, the minimum of the interval is zero. If we assume that the resource usage can be negative, the interval would be  $(-\infty, 1 + n^2]$ . In we had a lower bound ( $\text{lb}$ ) in the assertion, the interval would be  $[1 + n^2, \infty)$ .

That assertion describes a tuple in  $I_\alpha$  which is given by  $(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$ , where  $p(\bar{v}) : c(\bar{v})$  is defined by  $Pred$  and  $Precond$ , and  $\Phi$  is defined by  $ResUsage$ . For simplicity, we assume that  $Pred$  is actually

<sup>1</sup>Assertions might be prefixed with an indicator that it is to be checked, it has been already checked, detected false of true. We omit such a prefix in the format given [PBH00a].

$p(\bar{v})$  and that there is a syntactic correspondence from  $Precond$  to  $c(\bar{v})$ , and from  $ResUsage$  to  $\Phi$ . The information about  $input_p$  and  $size_p$  is implicit in  $ResUsage$ . The concretization of  $I_\alpha$ ,  $\gamma(I_\alpha)$ , is the set of all pairs  $(p(\bar{t}), r)$  such that  $\bar{t}$  is a tuple of terms and  $p(\bar{t})$  is an instance of  $Pred$  that meets precondition  $Precond$ , and  $r$  is a number that meets the condition expressed by  $ResUsage$  (i.e.,  $r$  lies in the interval defined by  $ResUsage$ ) for some assertion. For example, the previous assertion captures the following concrete semantic tuples:

(  $nrev([a,b,c,d,e,f,g],X)$ , 35 )                      (  $nrev([],Y)$ , 1 )

but it does not capture the following ones:

(  $nrev([A,B,C,D,E,F,G],X)$ , 35 )                      (  $nrev(W,Y)$ , 1 )  
 (  $nrev([a,b,c,d,e,f,g],X)$ , 53 )                      (  $nrev([],Y)$ , 11 )

those in the first line above because they correspond to calls which are outside of the scope of the assertion (i.e., they do not meet the precondition  $Precond$ ); those on the second line (which will never occur on execution) because they violate the assertion (i.e., they meet the precondition  $Precond$ , but do not meet the condition expressed by  $ResUsage$ ).

### Partial correctness

**Definition 3** Given a program  $p$  and an intended resource usage semantics  $I$ , where  $I : \mathcal{C}_p \mapsto \mathcal{R}$ , we say that  $p$  is partially correct w.r.t.  $I$  if for all  $p(\bar{t}) \in \mathcal{C}_p$  such that  $r$  is the amount of resource usage of the computation of the call  $p(\bar{t})$ , we have that  $I(p(\bar{t})) = r$ , i.e.,  $(p(\bar{t}), r) \in I$ . This is equivalent to the condition  $\llbracket P \rrbracket \subseteq I$  given in Table 1.

**Definition 4** Given an intended abstract resource usage semantics  $I_\alpha$  expressed as a set of tuples of the form  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$  (each tuple is expressed by an assertion in the program), we say that  $p$  is partially correct with respect to  $I_\alpha$  if for all  $p(\bar{t}) \in \mathcal{C}_p$  such that  $r$  is the amount of resource usage of the computation of the call  $p(\bar{t})$ , there is a tuple  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$  in  $I_\alpha$  such that  $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$  and  $r \in \Phi_I(\bar{s})$ , where  $\bar{s} = size_p(input_p(\bar{t}))$ .

**Definition 5** We say that  $p$  is partially correct with respect to a tuple of the form  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$  if for all  $p(\bar{t}) \in \mathcal{C}_p$  such that  $r$  is the amount of resource usage of the computation of the call  $p(\bar{t})$ , it holds that: if  $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$  then  $r \in \Phi_I(\bar{s})$ , where  $\bar{s} = size_p(input_p(\bar{t}))$ .

**Lemma 1**  $p$  is partially correct with respect to  $I_\alpha$  if:

- The set of all elements  $p(\bar{v}) : c_I(\bar{v})$  of all tuples in  $I_\alpha$  cover all the calls in  $\mathcal{C}_p$ .
- $p$  is partially correct with respect to every tuple in  $I_\alpha$ .

**Definition 6** Given two resource usage interval functions  $\Phi_1$  and  $\Phi_2$ , such that  $\Phi_1, \Phi_2 : \mathcal{R}^k \mapsto \mathcal{RI}$ , we define the inclusion relation  $\sqsubseteq_f$  and the intersection operation  $\sqcap_f$  as follows:

- $\Phi_1 \sqsubseteq_f \Phi_2$  iff for all  $\bar{n} \in \mathcal{R}^k$ ,  $\Phi_1(\bar{n}) \subseteq \Phi_2(\bar{n})$ .
- $\Phi_1 \sqcap_f \Phi_2 = \Phi_3$  iff for all  $\bar{n} \in \mathcal{R}^k$ ,  $\Phi_1(\bar{n}) \cap \Phi_2(\bar{n}) = \Phi_3(\bar{n})$ .

Consider a tuple  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$  in the intended meaning  $I_\alpha$ , and a tuple  $(p(\bar{v}) : c(\bar{v}), \Phi)$  in the computed abstract semantics  $\llbracket P \rrbracket_{\alpha+}$  (for simplicity, we assume the same tuple of variables  $\bar{v}$  in all abstract objects).

**Definition 7** We say that  $(p(\bar{v}) : c(\bar{v}), \Phi) \sqsubseteq (p(\bar{v}) : c_I(\bar{v}), \Phi_I)$  if  $c_I(\bar{v}) \sqsubseteq c(\bar{v})$  and  $\Phi \sqsubseteq_f \Phi_I$ .

Note that the condition  $c_I(\bar{v}) \sqsubseteq c(\bar{v})$  can be checked using the CiaoPP capabilities for comparing program state properties such as types.

**Definition 8** We say that  $(p(\bar{v}) : c(\bar{v}), \Phi) \sqcap (p(\bar{v}) : c_I(\bar{v}), \Phi_I) = \emptyset$  if  $c_I(\bar{v}) \sqsubseteq c(\bar{v})$  and  $\Phi \sqcap_f \Phi_I = \Phi_\emptyset$ , where  $\Phi_\emptyset$  is the empty function defined as follows:  $\Phi_\emptyset(\bar{n}) = \emptyset$  for all  $\bar{n} \in \mathcal{R}^k$ .

**Lemma 2** If  $(p(\bar{v}) : c(\bar{v}), \Phi) \sqsubseteq (p(\bar{v}) : c_I(\bar{v}), \Phi_I)$  then  $p$  is partially correct with respect to  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ .

Proof If  $(p(\bar{v}) : c(\bar{v}), \Phi) \sqsubseteq (p(\bar{v}) : c_I(\bar{v}), \Phi_I)$  then  $c_I(\bar{v}) \sqsubseteq c(\bar{v})$  and  $\Phi \sqsubseteq_f \Phi_I$ . For all  $p(\bar{t}) \in \mathcal{C}_p$  such that  $r$  is the amount of resource usage of the computation of the call  $p(\bar{t})$ , it holds that: if  $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$  then  $p(\bar{t}) \in \gamma(p(\bar{v}) : c(\bar{v}))$  (because  $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ ), and thus  $r \in \Phi(\bar{s})$ , where  $\bar{s} = \text{size}_p(\text{input}_p(\bar{t}))$  (because of the safety of the analysis). Since  $\Phi \sqsubseteq_f \Phi_I$ , we have that  $r \in \Phi_I(\bar{s})$ .

**Lemma 3** If  $(p(\bar{v}) : c(\bar{v}), \Phi) \sqcap (p(\bar{v}) : c_I(\bar{v}), \Phi_I) = \emptyset$  and  $(p(\bar{v}) : c(\bar{v}), \Phi) \neq \emptyset$  then  $p$  is incorrect w.r.t.  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ .

### 3.2 Comparing Resource Usage Interval Functions

During verification/debugging within the framework described in the previous section, we will need to compare abstract tuples following Table 2. Thus, whenever  $c_I(\bar{v}) \sqsubseteq c(\bar{v})$  we will have to determine whether  $\Phi \sqsubseteq_f \Phi_I$  or  $\Phi \sqcap_f \Phi_I = \Phi_\emptyset$ .

**Definition 9** Given two resource usage bound functions  $\Psi_1$  and  $\Psi_2$  (as in Definition 2,  $\Psi_1, \Psi_2 : \mathcal{R}^k \mapsto \mathcal{RI}$ ), we define the  $\leq_f$  relation as follows:

$$\Psi_1 \leq_f \Psi_2 \text{ iff for all } \bar{n} \in \mathcal{R}^k, \Psi_1(\bar{n}) \leq \Psi_2(\bar{n})$$

where  $\leq$  represents the standard relation between real numbers augmented with the special symbols  $\infty$  and  $-\infty$ . Similarly, we define  $<_f, >_f$  and  $\geq_f$ .

**Lemma 4** Given two resource usage interval functions  $\Phi_1$  and  $\Phi_2$ , we have that:

- $\Phi_1 \sqsubseteq_f \Phi_2$  if  $\Phi_2^l \leq_f \Phi_1^l$  and  $\Phi_1^u \leq_f \Phi_2^u$ .
- $\Phi_1 \sqcap_f \Phi_2 = \Phi_\emptyset$  if  $\Phi_1^u <_f \Phi_2^l$  or  $\Phi_2^u <_f \Phi_1^l$ .

**Corollary 1** *Let  $(p(\bar{v}) : c(\bar{v}), \Phi)$  and  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$  be tuples expressing an abstract semantics  $\llbracket P \rrbracket_{\alpha+}$  inferred by analysis and an intended abstract semantics  $I_\alpha$  (given in a specification) respectively, such that  $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ , and for all  $\bar{n} \in \mathcal{R}^k$ ,  $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$  and  $\Phi_I(\bar{n}) = [\Phi_I^l(\bar{n}), \Phi_I^u(\bar{n})]$ . We have that:*

- *If for all  $\bar{n} \in \mathcal{R}^k$ ,  $\Phi_I^l(\bar{n}) \leq \Phi^l(\bar{n})$  and  $\Phi^u(\bar{n}) \leq \Phi_I^u(\bar{n})$ , then  $p$  is partially correct with respect to  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ .*
- *If for all  $\bar{n} \in \mathcal{R}^k$   $\Phi^u(\bar{n}) < \Phi_I^l(\bar{n})$  or  $\Phi_I^u(\bar{n}) < \Phi^l(\bar{n})$ , then  $p$  is incorrect with respect to  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ .*

However, for simplicity, in this paper we assume that one of the endpoints of the interval is always the maximum (resp., minimum) of the possible values, i.e.,  $\forall \bar{n} (\Phi_I^u(\bar{n}) = \infty)$  (resp.,  $\Phi_I^l(\bar{n}) = -\infty$  or  $\Phi_I^l(\bar{n}) = 0$ , depending on the resource). Thus, one of the resource usage bound function comparisons in each of the two cases above is always trivial.

Therefore, we will be faced with only one such comparisons, between two functions, each denoting either a lower bound ( $l$ ) or an upper bound ( $u$ ).

When comparing resource usage bound functions, our approach computes subsets of  $\mathcal{R}^k$  for which a function is less, equal, or greater than another. This allows to give specifications of subsets of input data (sizes) for which a program  $p$  is partially correct (or incorrect) with respect to a given tuple  $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$  (i.e., representations of subsets of  $p(\bar{v}) : c_I(\bar{v})$ ). For the particular case where resource usage bound functions depend on one argument, such subsets can be represented as intervals of numbers in which the input data sizes lie in. This is explained in Section 4. It is straightforward to formalize this concept and to redefine partial correctness (or incorrectness) restricted to a set of input data sizes, by restricting all the instrumental definitions accordingly. We do not pursue it in this paper for space reasons.

## 4 Resource Usage Bound Function Comparison

Fundamental to our approach to verification is the operation that compares two resource usage bound functions, one of them inferred by the static analysis and the other one given in an assertion present in the program (i.e. given as a specification). Given two of such functions,  $\Psi_1(n)$  and  $\Psi_2(n)$ ,  $n \in \mathcal{R}$ , the objective of this operation is to determine intervals for  $n$  in which  $\Psi_1(n) > \Psi_2(n)$ ,  $\Psi_1(n) = \Psi_2(n)$ , or  $\Psi_1(n) < \Psi_2(n)$ .

Our approach consists on defining  $f(n) = \Psi_1(n) - \Psi_2(n)$  and finding the roots of the equation  $f(n) = 0$ . Assume that the equation has  $m$  roots,  $n_1, \dots, n_m$ . These roots are intersection points of  $\Psi_1(n)$  and  $\Psi_2(n)$ . We consider the intervals  $S_1 = [0, n_1)$ ,  $S_2 = (n_1, n_2)$ ,  $S_m = \dots (n_{m-1}, n_m)$ ,  $S_{m+1} = (n_m, \infty)$ .

For each interval  $S_i$ ,  $1 \leq i \leq m$ , we select a value  $v_i$  in the interval. If  $f(v_i) > 0$  (respectively  $f(v_i) < 0$ ), then  $\Psi_1(n) > \Psi_2(n)$  (respectively  $\Psi_1(n) < \Psi_2(n)$ ) for all  $n \in S_i$ .

Since our resource analysis is able to infer different types of functions (e.g., polynomial, exponential and logarithmic), it is also desirable to be able to compare all of these functions. For polynomial functions there exist powerful algorithms for obtaining roots. For the other functions, we have to approximate them using polynomials. In this case, we should guarantee that the error falls in the safe side when comparing the corresponding resource usage bound functions.

## 4.1 Comparing Polynomial Functions

There are general methods for finding roots of polynomial equations. Root equation finding of polynomial functions can be done analytically until polynomial order four. For higher order polynomial functions, numerical methods must be used. According to a fundamental theorem of algebra, a polynomial equation of order  $m$  has  $m$  roots, whether real or complex numbers. There are numerical methods that allow to compute all these roots (although the complex numbers are not needed in our approach).

## 4.2 Approximation of Non-Polynomial Functions

There are two non-polynomial resource usage function that CiaoPP analysis can infer: exponential and logarithmic. For approximating these functions we use Taylor series.

**Exponential function approximation using polynomial** This approximation is carried out using these formulae:

$$e^x \approx \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \quad \text{for all } x$$

$$a^x = e^{x \ln a} \approx 1 + x \ln a + \frac{(x \ln a)^2}{2!} + \frac{(x \ln a)^3}{3!} + \dots$$

In the implementation these series are limited up to order 8. This decision has been taken based on experiments that show that higher orders do not give a significant difference. Also, in the implementation, the computation of the factorials is done separately and the results are kept in a table in order to reuse them.

**Logarithmic function approximation using polynomial** Unfortunately this approximation can not be done in straightforward way as the approximation to exponential. Taylor series for this function for whole interval does not exist, the series only holds for interval  $-1 < x < 1$ . One possibility to work within this restriction is using range reduction [Ylo06].

### 4.3 Safety of the Approximation

Since the roots obtained for the function comparison are in some cases approximations of the real roots, we must guarantee that their values are safe, i.e., that they can be used for verification purposes as in table 2, and in particular, for safely checking the conditions of Corollary 1.

We have equation  $f(x) = 0$  and want to find its roots. In general, we approximate  $f(x)$  using a polynomial  $P(x)$ , so that  $f(x) = P(x) - e$ , with  $e$  an approximation error. Let the roots of equation  $f(x) = 0$  be  $x_0, \dots, x_n$ , and the roots of equation  $P(x) = 0$  be  $xp_0, \dots, xp_n$ . Using a root finding algorithm on equation  $P(x) = 0$ , we obtain  $xp_0, \dots, xp_n$ , so that we have  $P(xp_i) = 0$ , and therefore  $f(xp_i) \in [-e, +e]$ . The approximation is safe if there is a  $\epsilon$  such that  $f(xp_i - \epsilon) < 0$  whenever  $P(xp_i - \epsilon) < 0$ , and  $f(xp_i + \epsilon) > 0$  whenever  $P(xp_i + \epsilon) > 0$ . This means that  $x_i \in [xp_i - \epsilon, xp_i + \epsilon]$ . However, in our case we will reduce to only one of those values:  $xp_i + \epsilon$ , and we will approximate it by iteratively incrementing  $xp_i$  by some  $\delta$ .

Consider again Corollary 1. Assume for example that we are going to safely check whether  $\Phi^u(x) \leq \Phi_I^u(x)$  (where  $\Phi^u$  and  $\Phi_I^u$  are resource usage bound functions the first one is a result of program analysis and the second one an assertion declared in the program). In this case, we define  $f(x) = \Phi_I^u(x) - \Phi^u(x)$ , so that we can safely say that if  $f(x) > 0$ , then  $\Phi^u(x) \leq \Phi_I^u(x)$ . Assume also that  $\Phi_I^l$  is not given in the assertion, meaning that specification do not state any lower bound for the resource usage (i.e., the lower endpoint of any resource usage interval is  $-\infty$ , which means that  $\Phi_I^l(x) \leq \Phi^l(x)$  is always true). Thus, if  $f(x) > 0$  we can safely state that the assertion is definitely true. In the same way, if we define  $f(x) = \Phi^l(x) - \Phi_I^u(x)$  we can safely assert that  $\Phi_I^u(\bar{n}) < \Phi^l(\bar{n})$  if  $f(x) > 0$ , proving that the assertion is definitely false. We can reason similarly in the comparisons involving a lower bound in the assertion ( $\Phi_I^l$ ). Thus, we focus exclusively on safely checking that  $f(x) > 0$ , where  $f(x)$  is conveniently defined in each case.

Then, we have to determine, for each approximated root  $xp_i$ ,  $1 \leq i \leq n$ , a value  $\epsilon$  such that  $f(xp_i + \epsilon) > 0$  and  $x_i \in [xp_i - \epsilon, xp_i + \epsilon]$ . We do this by first determining the relative position of  $xp_i$  and  $x_i$  (i.e., whether  $xp_i$  is “to the right” or “to the left” of  $x_i$ ) and then starting an iterative process that increments (or decrements)  $xp_i$  by some  $\delta$  until we have that, after  $m$  iterations,  $f(xp_i + m \delta) > 0$ .

**Determining the relative position of the exact root** To determine the relative position of the exact root and its approximated value we use the gradient of  $f(x)$  around  $x = xp_i$ . For determining the gradient we use the values of  $e = f(xp_i)$  and  $e' = f(xp_i + \delta')$ , with  $\delta' > 0$  a relatively small number. Whether the approximated root is greater or less than the exact root depends on the following conditions:

1. if  $e < 0$  and  $e' > e$  then  $x_i > xp_i$
2. if  $e > 0$  and  $e' > e$  then  $x_i < xp_i$

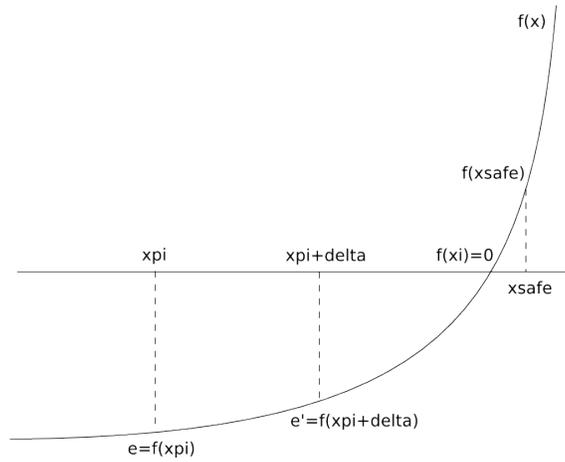


Figure 4: Case 1.  $x_i > xp_i$  (since  $e' > e$ ). A safe approximate root found is  $x_{safe}$ .

3. if  $e > 0$  and  $e' < e$  then  $x_i > xp_i$
4. if  $e < 0$  and  $e' < e$  then  $x_i < xp_i$

From Fig. 4 we can see the rationale behind the first case (the other cases follow an analogous reasoning). If  $e' > e$  then  $f(x)$  is increasing, but, since  $e < 0$ , then  $f(x) > 0$  can only occur for values of  $x$  greater than  $xp_i$ . Therefore,  $x_i > xp_i$ . In such cases we use a positive value of  $\delta$  for the iterative process. When  $x_i < xp_i$  we use a negative value of  $\delta$ .

Note that this method of guessing approximates to the exact root only works if the approximated value does not *go wrong*. We say that an approximated value for a root  $x_1$  goes wrong when there is a local maximum or minimum of the function in between  $x_1$  and its approximation  $xp_1$  but there is no such local maximum or minimum in between the approximation and another root different from  $x_1$  (see Fig. 5). In these cases, the approximated value will be “moved” to the other root ( $x_2$  in Fig. 5). Thus, we will be unable to determine a verification result for the interval between the root  $x_1$  and the next root “to the left” (in fact, for the interval in between such root and the approximation to  $x_2$ ).

**Iterative process for computing the safe root** Once we have determined the relative position of the exact root and its approximated value, we first set up the appropriate sign for  $\delta$ , where  $|\delta|$  is a relatively small number:  $\delta < 0$  if the iteration should go to the left ( $x_i < xp_i$ ), or  $\delta > 0$  if it should go to the right ( $x_i > xp_i$ ). Then we iterate on the addition  $xp'_i = xp_i + \delta$  until  $f(xp'_i) > 0$ . Such an iteration is apparent in the following pseudo-code:

- 1:  $x_{safe} \leftarrow xp_i$
- 2: **while**  $f(x_{safe}) < 0$  **do**  $x_{safe} \leftarrow x_{safe} + \delta$
- 3: **end while**

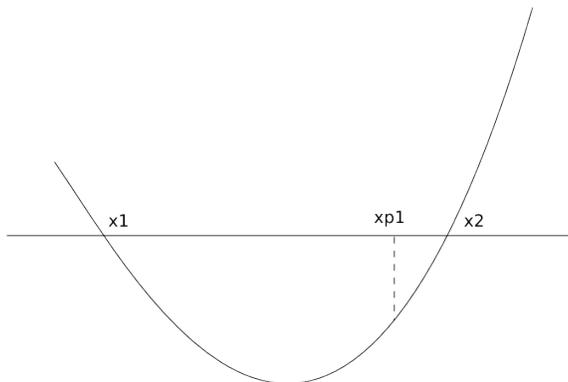


Figure 5: A root approximation that goes wrong:  $x_{p1}$  (approximation of  $x_1$ ) goes to  $x_2$ .

4: **return** *xsafe*

**Example 2** Consider an assertion which declares an upper bound on the resource usage of the classical `fibonacci` program given by function  $\Phi_I^u(x) = 2^x - 1000$ . Let the analysis infer a lower bound given by function  $\Phi^l(x) = 1.45 \times 1.62^x - 1$ . Their intersection occurs at  $x \approx 10.22$ . However, the root obtained by our root finding algorithm is  $x \approx 10.89$ . By doing an iterative approximation from 10.89 to the left, we finally obtain a safe approximate root of  $x \approx 10.18$ .

Note that usually (as in this example), resource usage functions are on variables which range on natural numbers. Because of this, the iterative approximation process for safe roots can be substituted by simply taking the closest natural number to the left or right of the approximated root (depending on the gradient) to obtain a safe value. In the previous example, we will simply take 10, without any iteration. Thus, the output of our assertion checking implementation within the *CiaoPP* system is:

```
:- false comp fib(N,F) + steps_ub( exp(2,int(N))-1000 )
   in interval [0, 10] for int(N).
:- check comp fib(N,F) + steps_ub( exp(2,int(N))-1000 )
   in interval [11, +inf] for int(N).
```

Meaning that the system has proved that the assertion is false for values of the input argument  $N$  in the interval  $[0, 10]$ , but nothing can be (statically) ensured for the values outside this interval.

## 5 Implementation

We have implemented a prototype that performs the verification of resource usage functions and integrated it into the *CiaoPP* system. For this purpose, we have used the GNU Scientific Library [GDT<sup>+</sup>09], which offers a specific

polynomial function library that uses analytical methods for finding roots of polynomials up to order four, and uses numerical methods for higher order polynomials. Since the functions in the GSL library can not be called directly from Ciao, some glue code has been written in C so that they can be invoked through this glue code.

## 6 Conclusions and Future Work

We have extended an existing framework for verification/debugging (implemented in the CiaoPP system), to deal with specifications about the resource usage of programs. We have provided a formalization which blends in with the previous framework for verification of functional or program state properties. A key aspect of the framework is to be able to compare mathematical functions. We have instrumented a method for function comparison which is safe, in the sense that the results of verification/debugging cannot go wrong. In the case where the resource usage functions being compared depend on one variable (which represents some input argument size) our method reveals particular numerical intervals for such variable, if they exist, which might result in different answers to the verification problem: a given specification might be proved for some intervals but disproved for others. Our current method accurately computes such intervals for polynomial and exponential resource usage functions, and in general for functions that can be approximated by polynomials. Moreover, we have proposed an iterative post-process to safely tune up the interval bounds by taking as starting values the previously computed roots of the polynomials. The current method allows for several improvements. For example, a better handling of logarithmic functions. In this sense, we will study a combined approach: using range reduction [Ylo06] and Taylor approximation to polynomials. Also, for the case in which the resource usage functions depend on more than one variable, we plan to extend the method in order to compute subsets of input data size tuples for which a given specification can be proved or disproved.

Finally, we also plan to develop a diagnosis scheme for resource usage under-performance. The diagnoser will heavily exploit analysis information in order to trace an incorrectness symptom backwards up to a fragment of the program responsible for it.

## References

- [BCC<sup>+</sup>03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proc. of PLDI'03*. ACM Press, 2003.
- [BDD<sup>+</sup>97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of*

- the 3rd. Int'l WS on Automated Debugging-AADEBUG*, pages 155–170. U. Linköping Press, May 1997.
- [Bou93] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation '93*, pages 46–55, 1993.
- [CLMV99] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.
- [CLV95] M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.
- [Cou03] P. Cousot. Automatic Verification by Abstract Interpretation, Invited Tutorial. In *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, number 2575 in LNCS, pages 20–24. Springer, January 2003.
- [DL93] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5), 1993.
- [GDT<sup>+</sup>09] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*. Network Theory Ltd, 2009. Library and Manual also available at <http://www.gnu.org/software/gsl/>.
- [HALGP04] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Some Techniques for Automated, Resource-Aware Distributed and Mobile Computing in a Multi-Paradigm Programming System. In *Proc. of EURO-PAR 2004*, number 3149 in LNCS, pages 21–37. Springer-Verlag, August 2004.
- [HPB99] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 161–192. Springer-Verlag, 1999.
- [HPBG05] 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 Comp. Progr.*, 58(1–2), 2005.
- [LS88] Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington, August 1988. MIT.
- [MLGH09] E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *International Conference on Logic Programming (ICLP)*, volume 5649 of LNCS, pages 281–295. Springer-Verlag, July 2009.
- [NMLGH07] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *ICLP*, LNCS, 2007.

- [PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.
- [PBH00b] G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.
- [Ylo06] Jyri Ylostalo. Function approximation using polynomials. *Signal Processing Magazine*, 23:99–102, September 2006.