Static Performance Guarantees for Programs with Run-time Checks

Maximiliano Klemen
IMDEA Software Institute and
ETSIINF, U. Politécnica Madrid
maximiliano.klemen@imdea.org

Nataliia Stulova
IMDEA Software Institute and
ETSIINF, U. Politécnica Madrid
nataliia.stulova@imdea.org

Pedro López-García
IMDEA Software Institute and
Consejo Sup. Inv. Cient. (CSIC)
pedro.lopez@imdea.org

José Morales
IMDEA Software Institute
josef.morales@imdea.org

Manuel V. Hermenegildo
IMDEA Software Institute and
ETSIINF, U. Politécnica Madrid
manuel.hermenegildo@imdea.org

ABSTRACT

Instrumenting programs for performing run-time checking of properties, such as regular shapes, is a common and useful technique that helps programmers detect incorrect program behaviors. This is specially true in dynamic languages such as Prolog. However, such run-time checks inevitably introduce run-time overhead (in execution time, memory, energy, etc.). Several approaches have been proposed for reducing this overhead, such as eliminating the checks that can statically be proved to always succeed, and/or optimizing the way in which the (remaining) checks are performed. However, there are cases in which it is not possible to remove all checks statically (e.g., open libraries which must check their interfaces, complex properties, unknown code, etc.) and in which, even after optimizations, these remaining checks may still introduce an unacceptable level of overhead. It is thus important for programmers to be able to determine the additional cost due to the run-time checks and compare it to some notion of admissible cost. The common practice used for estimating run-time checking overhead is profiling, which is not exhaustive by nature. Instead, we propose a method that uses static analysis to estimate such overhead, with the advantage that the estimations are functions parameterized by input data sizes. Unlike profiling, this approach can provide guarantees for all possible execution traces, and allows assessing how the overhead grows as the size of the input grows. Our method also extends an existing assertion verification framework to express "admissible" overheads, and statically and automatically checks whether the instrumented program conforms with such specifications. Finally, we present an experimental evaluation of our approach that suggests that our method is feasible and promising.

PPDP '18, September 3–5, 2018, Frankfurt am Main, Germany

ACM ISBN 978-1-4503-6441-6/18/09…$15.00
https://doi.org/10.1145/3236950.3236970

CCS CONCEPTS

• **Theory of computation** → *Program analysis*; *Assertions*; *Pre- and post-conditions*; *Invariants*; *Program semantics*;

KEYWORDS

Run-time Checks, Assertions, Abstract Interpretation, Resource Usage Analysis, Program Analysis, (Constraint) Logic Programming

ACM Reference Format:
, Maximiliano Klemen, , Nataliia Stulova, , Pedro López-García, , José Morales, and , Manuel V. Hermenegildo. 2018. Static Performance Guarantees, for Programs with Run-time Checks. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3236950.3236970

1 INTRODUCTION AND MOTIVATION

Dynamic programming languages are a popular programming tool for many applications, due to their flexibility. They are often the first choice for web programming, prototyping, and scripting. The lack of inherent mechanisms for ensuring program data manipulation correctness (e.g., via full static typing or other forms of full static built-in verification) has sparked the evolution of flexible solutions, including assertion-based approaches [5, 6, 14, 15, 23, 29, 50, 53] in (constraint) logic languages, soft- and gradual-typing [7, 13, 17, 47, 48, 54, 58, 62–65, 67] in functional languages (also applied to, e.g., Prolog [56] or Ruby [27]), and contract-based approaches [16, 30, 31, 34, 42, 47, 49] in imperative languages.

A trait that many of these approaches share is that some parts of the specifications may be the subject of *run-time checking* (e.g., those that cannot be discharged statically in the case of systems that support this functionality). However, such run-time checking comes at the price of overhead during program execution, that can affect a number of resources, such as execution time, memory use, energy consumption, etc., often in a significant way [54, 63]. If these overheads become too high, the whole program execution becomes impractical and programmers may opt for sacrificing the checks to keep the required level of performance.

Dealing with excessive run-time overhead is a challenging problem. Proposed approaches in order to address this problem include discharging as many checks as possible via static analysis [6, 16, 21, 23, 53, 61], optimizing the dynamic checks themselves [28, 49, 55, 60], or limiting run-time checking points [40]. Nevertheless, there are cases in which a number of checks cannot be optimized away and must remain in place, because of software architecture choices (e.g., the case of the external interfaces of reusable libraries or servers), the need to ensure a high level of safety (e.g., in safety-critical systems), etc.

At the same time, low program performance may not always be due to the run-time checks. Consider for example two basic database access operations: insertion and query. Consider also a program that follows the pattern of rare inserts and frequent querying. In this case it can perhaps be fine to perform complex run-time checks in the first operation, provided that the checks in the second are inexpensive enough.

A technique that can help in this context is *profiling*, often used to detect performance "hot spots" and guide program optimization. Prior work on using profiling in the context of optimizing the performance of programs with run-time checks [18, 41, 59] clearly demonstrates the benefits of this approach. Still, profiling infers information that is valid only for some particular input data values (and their execution traces). I.e, the profiling results thus obtained may not be valid for other input data values. Since the technique is by nature not exhaustive, detecting the worst cases can take a long time, and is impossible in general.

We develop and evaluate a static analysis-based approach aimed at delivering guarantees on the costs introduced by the run-time checks in a program (i.e., on the run-time checking overhead). The resulting method provides the programmer with feedback at compile-time regarding the impact that run-time checking will have on the program costs. Furthermore, we propose an assertion-based mechanism that allows programmers to specify bounds on the admissible run-time checking overhead introduced in programs. The approach then compares the inferred run-time checking overhead against the admissible one and provides guarantees on whether such specifications are met or not. Such guarantees can be given as constraints (e.g., intervals) on the size of the input data. We provide the formalization of the method and present also results from its implementation and experimental evaluation. As already said, our proposal builds on *static cost analysis* [1, 10–12, 37, 51, 57] instead of (or as a complement to) dynamic profiling. This type of analysis is aimed at inferring statically (i.e., without actually running the program with concrete data) *safe upper and lower bounds on execution costs*, i.e., bounds that are guaranteed and will never be violated in actual executions. Since such costs are data-dependent, these bounds take the form of functions that depend on certain characteristics (generally, data sizes) of the inputs to the program. These functions encode (bound) how the program costs change as the size of the input grows.

To the best of our knowledge, this is the first paper that proposes, implements, and benchmarks a method for expressing the admissible costs introduced by the run-time checks in a program (the run-time checking overhead) and producing statically (i.e., at compile time) guarantees of such overheads meeting these specifications or identifying errors with respect to them. In the following, we will present our proposal for concreteness in the context of the Ciao system and apply it to logic programs. However, the approach is general and can be applied directly to other languages and systems.

The rest of the paper proceeds as follows: as preliminaries, Section 2 presents the assertion language used and the types of run-time checks generated, and Section 3 briefly introduces static cost analysis in the context of those assertions. Then, Section 4 presents the proposed method for analyzing, specifying limits on, and verifying the run-time checking overhead. These issues are covered in subsections 4.1, 4.2, and 4.3. Also, subsection 4.4 proposes a method for applying accumulated cost analysis for detecting hot spots. Section 5 describes our implementation and presents results from the experimental evaluation. Finally, Section 6 presents our conclusions.

2 ASSERTIONS AND RUN-TIME CHECKING

Assertion Language. Assertions are linguistic constructions that allow expressing properties of programs. For concreteness we will use the `pred` assertions of the Ciao assertion language [22, 23, 52], following the presentation of [61]. Such `pred` assertions allow defining the set of all admissible preconditions for a given predicate, and for each such precondition a corresponding postcondition. These pre- and postconditions are formulas containing literals defined by predicates specially labeled as *properties*, to which we refer to as *prop* literals. A set of assertions for a predicate, identified by a normalized[1] atom *Head*, is as follows:

```
:- Status pred Head : Pre_1 => Post_1.
...
:- Status pred Head : Pre_n => Post_n.
```

where the Pre_i and $Post_i$ fields are logic formulas (e.g., conjunctions) of *prop* literals that refer to the variables of *Head*. Informally, such a set of assertions states that in any execution state immediately before the call to *Head* at least one of the Pre_i conditions should hold, and that, given the $(Pre_i, Post_i)$ pair(s) where Pre_i holds, then, if the predicate succeeds, the corresponding $Post_i$ should hold upon its success. The precondition and postcondition fields are both optional, and they are assumed to be true if not present. Similarly, the assertions themselves are also optional (partial), in the sense that a given predicate may or may not have assertions associated with it.

Example 1 (Program with Assertions). *Consider the following implementation of a predicate for reversing a list and its assertions (note that in this running example we are using* `app1` *which appends* one *new element at the end of a list):*

```
1 :- check pred rev(X,Y)       % \
2      : (list(X), var(Y))     %  A1
3     => (list(X), list(Y)).   % /
4
5 rev([], []).
6 rev([X|Xs], Y) :-
7      rev(Xs, Ys),
8      app1(Ys,X,Y).
```

[1] By normalized we mean the standard notion that all arguments are distinct variables.

```
:- check pred app1(Y,X,Z)            % \
     : (list(Y), term(X), var(Z))    %  A2
    => (list(Y), term(X), list(Z)). % /

app1([],X,[X]).
app1([E|Y],X,[E|T]) :-
     app1(Y,X,T).
```

Assertion A1 *states that if* rev/2 *is called with a list* X *and a free variable* Y*, on its success the second argument* Y *will also be a list. Assertion* A2 *says if* app1/3 *is called with a list* Y*, a term* X*, and a free variable* Z*, on success the third argument* Z *will be a list. The algorithmic complexity of* rev/2 *is* $O(N^2)$ *in the size (list length in this case)* N *of its input argument* X*. While this implementation is obviously not optimal, we use it as a representative of the frequent case of nested loops with linear costs.*

Every assertion also has a *Status* field which indicates whether the assertion refers to intended or actual properties. Programmer-provided assertions by default have status check, and only assertions with this status generate run-time checks. Static analysis can prove or disprove properties in assertions for a given class of input queries, statically verifying assertions (if all the prop literals are proved to be true, in which case their status is changed to checked) or flagging errors (if any prop literal is proved to be false, and then the status is changed to false). Assertions can also be simplified by eliminating the prop literals proved to be true, so that only the remaining ones need to be checked. Other information inferred by static analysis is communicated by means of true assertions (e.g., see Example 5).

Example 2 (Assertions After Static Checking). *The following listing shows a possible result after performing static assertion checking for the code fragment of Example 1. We assume that the code is in a module, exporting only* rev/2*, and that it is analyzed in isolation, i.e., we have no information on the callers to* rev/2*.*

```
:- check calls rev(X,Y)
     : (list(X), var(Y)).
:- checked pred rev(X,Y)
     : (list(X), var(Y))
    => (list(X), list(Y)).

:- checked pred app1(Y,X,Z)
     : (list(Y), term(X), var(Z))
    => (list(Y), term(X), list(Z)).
```

*Here, the interface assertion (*calls*) for the* rev/2 *predicate remains active and generates run-time checks (i.e., calls into the module are sanitized). This contrasts with the situation in Example 1, where all assertions generate run-time checks.*

Run-time Check Instrumentation. We recall the definitional source transformation of [60], that introduces *wrapper* predicates that check calls and success assertions, and also groups all assertions for the same predicate together to produce optimized checks. Given a program, for every predicate p the transformation replaces all clauses $p(\bar{x}) \leftarrow body$ by $p'(\bar{x}) \leftarrow body$, where p' is a new predicate symbol, and inserts the wrapper clauses given by $\text{wrap}(p(\bar{x}), p')$:

$$\text{wrap}(p(\bar{x}), p') = \left\{ \begin{array}{l} p(\bar{x}) \text{ :- } p_C(\bar{x}, \bar{r}), p'(\bar{x}), p_S(\bar{x}, \bar{r}). \\ p_C(\bar{x}, \bar{r}) \text{ :- } ChecksC. \\ p_S(\bar{x}, \bar{r}) \text{ :- } ChecksS. \end{array} \right\}$$

Here *ChecksC* and *ChecksS* are the optimized compilation of pre- and postconditions $\bigvee_{i=1}^{n} Pre_i$ and $\bigwedge_{i=1}^{n}(Pre_i \rightarrow Post_i)$ respectively; and the additional *status* variables $\bar{r}$ are used to communicate the results of each Pre_i evaluation to the corresponding $(Pre_i \rightarrow Post_i)$ check, thus avoiding double evaluation of preconditions.

The compilation of checks for assertions emits a series of calls to a reify_check(P,Res) predicate, which accepts as the first argument a property P and unifies Res with 1 or 0, depending on whether the property check succeeds or not. The results of those reified checks are then combined and evaluated as Boolean algebra expressions using bitwise operations and the Prolog is/2 predicate. That is, the logical operators $(A \vee B)$, $(A \wedge B)$, and $(A \rightarrow B)$ used in encoding assertions are replaced by their bitwise logic counterparts R is A \/ B, R is A /\ B, R is (A # 1) \/ B, respectively.

Example 3 (Run-time Checks (a)). *The program transformation that introduces the run-time checking harness for the program fragment from Example 1 (assuming none of the assertions has been statically discharged by analysis) is essentially as follows:*

```
rev(A,B) :-
     revC(A,B,C),
     rev'(A,B),
     revS(A,B,C).

revC(A,B,E) :-
     reify_check(list(A),C),
     reify_check(var(B), D),
     E is C/\D,
     warn_if_false(E,'calls').

rev'([],[]).
rev'([X|Xs],Y) :-
     rev(Xs,Ys),
     app1(Ys,X,Y).

revS(A,B,E) :-
     reify_check(list(A),C),
     reify_check(list(B),D),
     F is C/\D,
     G is (E#1)\/F,
     warn_if_false(G,'success').


app1(A,B,C) :-
     app1C(A,B,C,D),
     app1'(A,B,C),
     app1S(A,B,C,D).

app1C(A,B,C,G) :-
     reify_check(list(A),D),
     reify_check(term(B),E),
     reify_check(var(C), F),
```

```
      G is D/\(E/\F),
      warn_if_false(G,'calls').

app1'([],X,[X]).
app1'([E|Y],X,[E|T]) :-
      app1(Y,X,T).

app1S(A,B,C,G) :-
      reify_check(list(A),D),
      reify_check(term(B),E),
      reify_check(list(C),F),
      H is D/\(E/\F),
      K is (G#1)\/H,
```

The `warn_if_false/2` *predicates raise run-time errors terminating program execution if their first argument is* `0`*, and succeed (with constant cost) otherwise. We will refer to this case as the* worst *performance case in programs with run-time checking.*

Example 4 (Run-time Checks (b)). *This example represents the run-time checking generated for the scenario of Example 2, i.e., after applying static analysis to simplify the assertions (see code below). Run-time checks are generated only for the interface calls of the* `rev/2` *predicate. Note that* `rev'/2` *here is a point separating calls to* `rev/2` *coming* outside *the module from the* internal *calls (now made through* `rev_i/2`*).*

```
rev(A,B) :-
      revC(A,B),
      rev'(A,B).

revC(A,B) :-
      reify_check(list(A),C),
      reify_check(var(A), D),
      E is C/\D,
      warn_if_false(E,'calls').

rev'(A,B) :-
      rev_i(A,B).

rev_i([],[]).
rev_i([X|Xs],Y) :-
      rev_i(Xs,Ys),
      app1(Ys,X,Y).
```

Note also that `app1/3` *is called directly (i.e., with no run-time checks). Clearly in this case there are fewer checks in the code and thus smaller overhead. We will refer to this case, where only interface checks remain, as the* base *performance case.*

3 STATIC COST ANALYSIS

Static cost analysis automatically infers information about the resources that will be used by program executions, without actually running the program with concrete data. Unlike profiling, static analysis can provide guarantees (upper and lower bounds) on the resource usage of all possible execution traces, given as functions on input data sizes. In this paper we build on the CiaoPP general cost analysis framework [11, 12, 46, 57], which is parametric with respect to *resources*, *programming languages*, and other aspects related to cost. It can be easily customized/instantiated by the user to infer a wide range of resources [46], including resolution steps, execution time, energy consumption, number of calls to a particular predicate, bits sent/received by an application over a socket, etc.

In order to perform such customization/instantiation, the Ciao assertion language is used [22, 46, 52]. For cost analysis it allows defining different resources and how basic components of a program (and library predicates) affect their use. Such assertions constitute the *cost model*. This model is taken (trusted) by the static analysis engine, that propagates it during an abstract interpretation of the program [57] through code segments, conditionals, loops, recursions, etc., mimicking the actual execution of the program with symbolic "abstract" data instead of concrete data. The engine is based on *abstract interpretation*, and defines the resource analysis itself as an *abstract domain* that is integrated into the PLAI abstract interpretation framework [44] of CiaoPP.

The engine infers cost functions (polynomial, exponential, logarithmic, etc.) for higher-level entities, such as procedures in the program. Such functions provide upper and lower bounds on resource usage that depend on input data sizes and possibly other (hardware) parameters that affect the particular resource. Typical size metrics include actual values of numbers, lengths of lists, term sizes (number of constant and function symbols), etc. [46, 57]. The analysis of recursive procedures sets up recurrence equations (cost relations), which are solved (possibly safely approximated), obtaining upper- and lower-bound (closed form) cost functions. To be parametric with respect to *programming languages*, CiaoPP differentiates between the *input language* (e.g., Ciao, Java source, Java bytecode, XC source, LLVM IR, or assembly) and the *intermediate program representation*, which is what the resource analysis actually operates on. Following [39] we use *Horn Clauses* as this intermediate program representation, which we refer to as the "HC IR." [2] A transformation is performed from each supported *input language* into the HC IR, which is then passed, along with Ciao assertions expressing the cost model (and possibly other trusted information), to the resource analysis engine mentioned above [57]. The setting up and solving of recurrence relations for inferring closed-form functions representing bounds on the sizes of output arguments and the resource usage of the predicates in the program are integrated into the PLAI framework as an abstract operation.

Example 5 (Static Cost Analysis Result). *The following assertion is part of the output of the resource usage analysis performed by CiaoPP for the* `rev/2` *predicate from Example 1:*

```
:- true pred rev(X,Y)
     : (list(X), var(Y), length(X,L))
    => (list(X), list(Y),
        length(X,L), length(Y,L))
     + cost(exact(0.5*(L)**2+1.5*L+1),
            [steps]).
```

[2] Horn Clauses have been used successfully as intermediate representations for many different programming languages and compilation levels (e.g., bytecode, llvm-IR, or ISA), in a good number of other analysis and verification tools [2–4, 8, 9, 19, 20, 24–26, 32, 33, 38, 45].

It includes, in addition to the precondition (:Pre) and postcondition (=>Post) fields, a field for computational *properties (+Comp), in this case cost. The assertion uses the* `cost/2` *property for expressing the* `exact` *cost (first argument of the property) in terms of resolution* `steps` *(second argument) of any call to* `rev(X,Y)` *with the* `X` *bound to a list and* `Y` *a free variable. Such cost is given by the function* $0.5\,L^2 + 1.5\,L + 1$*, which depends on L, i.e., the length of the (input) argument* `X`*, and is the argument of the* `exact/1` *qualifier. It means that such function is both a lower and an upper bound on the cost of the specified call. This aspect of the assertion language (including the* `cost/2` *property) and our proposed extensions are discussed in Section 4.*

4 SPECIFYING, ANALYZING, AND VERIFYING RUN-TIME CHECKING OVERHEAD

Our approach to analysis and verification of run-time checking overhead consists of three basic components: using static cost analysis to infer upper and lower bounds on the cost of the program with and without the run-time checks; providing the programmer with a means for specifying the amount of overhead that is admissible; and comparing the inferred bounds to these specifications. The following three sections outline these components.

4.1 Computing the Run-time Checking Overhead (Ovhd)

The first step of our approach is to infer upper and lower bounds on the cost of the program with and without the run-time checks, using cost analysis. The inference of the bounds for the program without run-time checks was illustrated in Example 5. The following two examples illustrate the inference of bounds for the program with the run-time checks. They cover the two scenarios discussed previously, i.e., with and without the use of static analysis to remove run-time checks.

Example 6 (Cost with Run-time Checks (a)). *The code below is the result of cost analysis for the run-time checking harness of Example 3 for the* `rev/2` *predicate, together with a (stylized) version of the code analyzed, for reference. Note the change in the complexity order of* `rev/2` *from quadratic to cubic in L, the length of list* `A`*, which is most likely not admissible. The reason is that run-time checks are performed at each (recursive) call to* `app1/3`*, and they check the property* `list/1`*, for which the whole input list needs to be traversed. Thus, such run-time checks have linear complexity, and they are performed a linear number of times in* `app1/3`*, and hence, the complexity order of* `app1/3` *changes from linear to quadratic. Since* `rev/2` *calls* `app1/3` *for each element of the input list (i.e., a linear number of times), its complexity order changes from cuadratic to cubic. Note that the additional list traversals introduced by the run-time checks in the body of* `rev/2` *(which have linear complexity) do not affect the complexity order of* `rev/2` *because* `rev/2` *already called predicate* `app1/3` *that was linear. Such checks only increase the constant coefficients of the cost function for* `rev/2`*.*

```
:- true pred rev(A,B)
     : (list(A), var(B), length(A,L))
    => (list(A), list(B),
        length(A,L), length(B,L))
     + cost(exact(0.5*L**3+7*L**2+14.5*L+8),
            [steps]).
rev(A,B) :-
   revC(A,B,C),
   rev'(A,B),
   revS(A,B,C).

revC(A,B,C) :- list(A), var(B), bit_ops.

revS(A,B,C) :- list(A), list(B), bit_ops.

rev'([],[]).
rev'([X|Xs],Y) :-
   rev(Xs,Ys),
   app1(Ys,X,Y).

app1(A,B,C) :-
   app1C(A,B,C,D),
   app1'(A,B,C),
   app1S(A,B,C,D).

app1C(A,B,C,G) :- list(A), term(B), var(C), bit_ops.

app1S(A,B,C,G) :- list(A), term(B), list(C), bit_ops.

app1'([],X,[X]).
app1'([E|Y],X,[E|T]) :-
    app1(Y,X,T).
```

Example 7 (Cost with Run-time Checks (b)). *This example shows the result of cost analysis for the base instrumentation case of Example 4: although there are still some run-time checks present for the interface, the overall cost of the* `rev/2` *predicate remains quadratic, which is probably* admissible.

```
:- true pred rev(A,B)
    : (list(A), var(B), length(A,L))
   => (list(A), list(B),
       length(A,L), length(B,L))
    + cost(exact(0.5*L**2+2.5*L+7),
           [steps]).

rev(A,B)  :-
     revC(A,B),
     rev'(A,B).

revC(A,B) :- list(A), var(B), bit_ops.

rev'(A,B) :-
     rev_i(A,B).

rev_i([],[]).
rev_i([X|Xs],Y) :-
     rev_i(Xs,Ys),
     app1(Ys,X,Y).
```

4.2 Expressing the Admissible Run-time Checking Overhead (AOvhd)

We add now to our approach the possibility of expressing the admissible run-time checking overhead (AOvhd). This is done by means of an extension to the Ciao assertion language. As mentioned before, this language already allows expressing a wide range of properties, and this includes the properties related to resource usage.

Example 8 (Cost Specification). *For example in order to tell the system to check whether an upper bound on the cost, in terms of number of resolution steps, of a call* `p(A, B)` *with* `A` *instantiated to a natural number and* `B` *a free variable, is a function in* $O(\mathtt{A})$*, we can write the following assertion:*

```
:- check pred p(A, B)
     : (nat(A), var(B))
     + cost(o_ub(A), [steps, std]).
```

The first argument of the `cost/2` *property is a cost function, which in turn appears as the argument of a qualifier expressing the kind of approximation. In this case, the qualifier* `o_ub/1` *represents the complexity order of an upper bound function (i.e., the "big O"). Other qualifiers include* `ub/1` *(an upper-bound cost function, not just a complexity order),* `lb/1` *(a lower-bound cost function), and* `band/2` *(a cost band given by both a lower and upper bound). The second argument of the* `cost/2` *property is a list of qualifiers (identifiers). The first identifier expresses the resource, i.e., the cost metric used. The value* `steps` *represents the* number of resolution steps. *The second argument expresses the particular kind of cost used. The value* `std` *represents the* standard cost *(the value by default if it is omitted), the value* `acc` *the accumulated cost [37], etc.*

We introduce the possibility of writing assertions that are universally quantified over the predicate domain (i.e., that are applicable to all calls to all predicates in a program), which is particularly useful in our application. As an example, the following assertion:

```
:- check pred *
     + is_det.
```

states that all predicates in the program should be deterministic, i.e., produce at most one answer. An issue that appears in this context is that different predicates can have different numbers and types of arguments. To solve this problem we introduce a way to express symbolic complexity orders without requiring the specification of details about the arguments on which cost functions depend nor the size metric used, by means of symbols (identifiers) without arguments, such as `constant`, `linear`, `quadratic`, `exponential`, `logarithmic`, etc. For example, in order to extend the assertion in Example 8 to all possible predicate calls in a program (independently of the number and type of arguments), we can write:

```
:- check pred *
     + cost(so_ub(linear), [steps]).
```

In the context of the previous extensions, our objective is expressing and specifying limits on how the complexity/cost changes when run-time checks are performed, i.e., expressing and specifying limits on the run-time checking overhead. To this end we propose different ways to quantify this overhead. Let $C_p(\bar{n})$ represent the standard cost function of predicate p without any run-time checks and $C_{p\_rtc}(\bar{n})$ the cost function for the transformed/instrumented version of p that performs run-time checks, p_rtc. A good indicator of the relative overhead is the ratio:

$$\frac{C_{p\_rtc}(\bar{n})}{C_p(\bar{n})}$$

We introduce the qualifier `rtc_ratio` to express this type of ratios. For example, the assertion:

```
:- check pred p(A, B)
     : (nat(A), var(B))
     + cost(so_ub(linear),
            [steps, rtc_ratio]).
```

expresses that `p/2` should be called with the first argument bound to a natural number and the second one a variable, and the relative overhead introduced by run-time checking in the calls to `p/2` (the ratio between the cost of the predicate with and without run-time checks) should be at most a linear function. Similarly, using the universal quantification over predicates, the following assertion:

```
:- check pred *
     + cost(so_ub(linear),
            [steps, rtc_ratio]).
```

expresses that, for all predicates in the program, the ratio between the cost of the predicate with and without run-time checks should be at most a linear function.

4.3 Verifying the Admissible Run-time Checking Overhead (AOvhd)

We now turn to the third component of our approach: *verifying the admissible run-time checking overhead (AOvhd).* To this end, we leverage the general framework for resource usage analysis and verification of [35, 36], and adapt it for our purposes, using the assertions introduced in Section 4.2. The *verification* process compares 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) that may come from the specifications or from the analysis results. The possible outcomes of this process are the following:

(1) The status of the original (specification) assertion (i.e., `check`) is changed to `checked` (resp. `false`), meaning that the assertion is correct (resp. incorrect) for all input data meeting the precondition of the assertion,
(2) the assertion is "split" into two or three assertions with different status (`checked`, `false`, or `check`) whose preconditions include a conjunct expressing that the size of the input data belongs to the interval(s) for which the assertion is correct (status `checked`), incorrect (status `false`), or the tool is not able to determine whether the assertion is correct or incorrect (status `check`), or
(3) in the worst case, the assertion remains with status `check`, meaning that the tool is not able to prove nor to disprove (any part of) it.

In our case, the specifications express a band for the AOvhd, defined by a lower- and an upper-bound cost function (or complexity orders). If the lower (resp. upper) bound is omitted, then the lower (resp. upper) limit of the band is assumed to be zero (resp. ∞).

This implies that we need to perform some adaptations with respect to the verification of resource usage specifications for predicates described in [35, 36]. Assume for example that the user wants the system to check the following assertion:

```
:- check pred p(A, B)
     : (nat(A), var(B))
     + cost(ub(2*A), [steps, rtc_ratio]).
```

which expresses that the ratio defined in Section 4.2 (with $\bar{n}$ = A) $\frac{C_{p\_rtc}(\bar{n})}{C_p(\bar{n})}$ must be in the band $[0, 2{*}A]$ for a given predicate p. The approach in [35, 36] uses static analysis to infer both lower and upper bounds on $C_p(\bar{n})$, denoted $C_p^l(\bar{n})$ and $C_p^u(\bar{n})$ respectively. In addition, in our application, the static analysis needs to infer, both lower and upper bounds on $C_{p\_rtc}(\bar{n})$, denoted $C_{p\_rtc}^l(\bar{n})$ and $C_{p\_rtc}^u(\bar{n})$, and use all of these bounds to compute bounds on the ratio. A lower (resp. upper) bound on the ratio is given by $\frac{C_{p\_rtc}^l(\bar{n})}{C_p^u(\bar{n})}$ (resp. $\frac{C_{p\_rtc}^u(\bar{n})}{C_p^l(\bar{n})}$). Both bounds define an inferred (safely approximated) band for the actual ratio, which is compared with the (intended) ratio given in the specification (the band $[0, 2 * A]$) to produce the verification outcome as explained above.

4.4 Using the Accumulated Cost for Detecting Hot Spots

So far, we have used the standard notion of cost in the examples for simplicity. However, in our approach we also use the *accumulated cost* [37], inferred by CiaoPP, to detect which of the run-time check predicates (properties) have a higher impact on the overall run-time checking overhead, and are thus promising targets for optimization (or removal, if some reduction in safety guarantees is allowed). We leave the detailed description of the use of *accumulated cost* (enabled by our general analysis framework [37]) for future work, and just give the main idea and an example in this paper. The *accumulated cost* is based on the notion of *cost centers*, which in our approach are predicates to which execution costs are assigned during the execution of a program. The programmer can declare which predicates will be cost centers. Consider again a predicate p, and its instrumented version p_rtc that performs run-time checks, and let $C_p(\bar{n})$ and $C_{p\_rtc}(\bar{n})$ be their corresponding standard cost functions. Let ck represent a run-time check predicate (e.g., `list/1`, `num/1`, `var/1`, etc.). Let $\diamond_{p\_rtc}$ be the set of run-time check predicates used by p_rtc. Assume that we declare that the set of cost centers to be used by the analysis, $\diamond$, is $\diamond_{p\_rtc} \cup \{p\_rtc\}$. In this case, the cost of a (single) call to p_rtc *accumulated in* cost center ck, denoted $C_{p\_rtc}^{ck}(\bar{n})$, expresses how much of the standard cost $C_{p\_rtc}(\bar{n})$ is attributed to run-time check ck predicate (taking into account all the generated calls to ck). The ck predicate with the highest $C_{p\_rtc}^{ck}(\bar{n})$ is a hot spot, and thus, its optimization can be more profitable to reduce the overall run-time checking overhead. The predicate ck with the highest $C_{p\_rtc}^{ck}(\bar{n})$ is not necessarily the most costly by itself, i.e., the one with the highest standard cost. For example, a high $C_{p\_rtc}^{ck}(\bar{n})$ can be caused because ck is called very often. We create a ranking of run-time check predicates according to their accumulated cost. This can help in deciding which assertions and properties to simplify/optimize first to meet an overhead target.

Since p_rtc is declared as a cost center, the overall, absolute run-time checking overhead ($C_{p\_rtc}(\bar{n}) - C_p(\bar{n})$) can be computed as

$$\sum_{ck \in \diamond_{p\_rtc}} C_{p\_rtc}^{ck}(\bar{n}) \tag{1}$$

In addition, we can compute the standard cost of p_rtc as

$$C_{p\_rtc}(\bar{n}) = \sum_{q \in \diamond} C_{p\_rtc}^{q}(\bar{n}) \tag{2}$$

and the standard cost of p as

$$C_p(\bar{n}) = C_{p\_rtc}^{p\_rtc}(\bar{n}) \tag{3}$$

Thus, we only need to infer accumulated costs and combine them to both detect hot spots and compute the `rtc_ratio` described in Section 4.2.

Example 9 (Detecting hot spots). *Let* `app1_rtc/3` *denote the instrumented version for run-time checking of predicate* `app1/3` *in Example 1. The following table shows the cost centers automatically declared by the system, which are the predicate* `app1_rtc/3` *itself and the run-time checking properties it uses (first column), as well as the accumulated costs of a call to* `app1_rtc(A,B,_)` *in each of those cost centers, where l_X represents the length of list X (second column):*

***Cost center** (ck)*	$C_{app1\_rtc}^{ck}(l_A, l_B)$
`app1_rtc/3`	$l_A + 1$
`list/1`	$3 \times (l_A - 1)^2 + 6 \times (l_A + 1) \times (l_B + 1)$ $+8 \times (l_A + 1) - 12$
`var/1`	$l_A + 1$
`bit_ops/1`	$3 \times (l_A + 1)$

With these results, applying the formulas 1, 2 and 3, we obtain the following costs:

$$\begin{aligned} C_{app1\_rtc}(l_A, l_B) - C_{app1}(l_A, l_B) &= 3 \times (l_A - 1)^2 \\ &\quad + 6 \times (l_A + 1) \times (l_B + 1) \\ &\quad + 12 \times (l_A + 1) - 12 \\ C_{app1\_rtc}(l_A, l_B) &= 3 \times (l_A - 1)^2 \\ &\quad + 6 \times (l_A + 1) \times (l_B + 1) \\ &\quad + 13 \times (l_A + 1) - 12 \\ C_{app1}(l_A, l_B) &= l_A + 1 \end{aligned}$$

It is clear that the hot spot *is the* `list/1` *property, which is responsible for the change in complexity order of the instrumented version* `app1_rtc/3` *from linear to quadratic.*

Accumulated cost analysis can also be used to stop the verification process as soon as one run-time check predicate is found whose accumulated cost violates the specified admissible overhead. If the complexity order of the instrumented version of predicate p for run-time checking, p_rtc, increases that of p, then it is caused by some run-time check predicate ck, which can be detected by comparing accumulated costs. This is formalized as follows:

Theorem 1. *If $C_p = O(f)$ and $C_{p\_rtc} = O(f')$ then: $f < f'$ if and only if there is a run-time check predicate $ck \in \diamond_{p\_rtc}$ such that $C^{ck}_{p\_rtc} = O(g)$ and $f < g$.*

This theorem has some useful implications. For example, assume that the programmer writes an assertion stating that $\frac{f'}{f} \leq 1$.

Then, as soon as the analysis finds a run-time check predicate $ck \in \diamond_{p\_rtc}$, such that $C^{ck}_{p\_rtc} = O(g)$, and $\frac{g}{f} > 1$, we can say that the assertion does not hold. In addition, we can say that ck is a hot spot, responsible for p_rtc not meeting the admissible overhead (although there can be other run-time check predicates that are also responsible for it). In this case, some action must be taken for reducing the (complexity order of) the cost of p_rtc accumulated in ck, i.e., for reducing the overall impact of ck on the (standard) cost of p_rtc. The detailed diagnosis of hot spots and actions that can be taken for making p_rtc meet the admissible overhead are topics for future work.

Note that besides using the accumulated cost, more generally, we can use *static profiling*, i.e., the static inference of the kinds of information that are usually obtained at run-time by profilers by using the framework described in [37].

5 IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We have implemented a prototype of our approach by modifying the Ciao system, and in particular CiaoPP's abstract interpretation-based resource usage analysis and CiaoPP's libraries implementing different components for static and dynamic verification (run-time checking transformation, function comparison, etc.).

Table 1 contains a list of the benchmarks that we have used in our experiments.[3] Each benchmark has assertions with properties related to shapes, instantiation state, variable freeness, and variable sharing, as well as in some cases more complex properties such as, for example, sortedness. The benchmarks and assertions were chosen to be simple enough to have easily understandable costs but at the same time produce interesting cost functions and overhead ratios.

As stated throughout the paper, our objective is to exploit static cost analysis to obtain guarantees on program performance and detect cases where adding run-time checks introduces overhead that is not admissible. To this end, we have considered the code instrumentation scenarios discussed previously, i.e. (cf. Examples 3 and 4):

performance	static checking	run-time checking instr.
Original	no	no (off)
Worst	no	yes (full)
Base	*eterms + shfr*	yes (opt)

and we have performed for each benchmark and each scenario run-time checking overhead analysis and verification, following the proposed approach. The optimization in the opt case consists in statically proving some of the properties appearing in the assertions, using different static analyses and using this information to eliminate the checks that are proved to always succeed, as in Example 2. In our experiments we apply this to two classes of properties. The first one is the *state of instantiation of variables*, i.e., which variables are bound to ground terms, or unbound, and, if they are unbound, the *sharing (aliasing) patterns*, i.e., which variables point to each other ("share"). This is a property that can appear in assertions (typically stating that a variable is independent of others) but, more importantly, it is also very important to track grounding information ("strong update"), to ensure the correctness and precision of the state of instantiation information. These properties are approximated using the *sharing and freeness (shfr)* domain [43, 44]. The second class of properties we will be using refers to the shapes of the data structures constructed by the program in memory. To this end we use the *eterms* [66] abstract domain which infers safely these shapes as regular trees. The inferred abstractions are useful for simplifying properties referring to the types/shapes of arguments in assertions.

Regarding the cost analysis, the resource inferred in these experiments is the *number of resolution steps* (i.e., each clause body is assumed to have unitary cost). While in practice other resources can be of interest (time, memory, energy, etc.), the number of resolution steps is a good abstraction for our purposes and the techniques carry over straightforwardly to the other resources. The times in the tables are given in milliseconds. The experiments were performed on a MacBook Pro with 2.5GHz Intel Core i5 CPU, 10 GB 1333 MHz DDR3 memory, running macOS Sierra 10.2.6.

Tables 2 and 3 show the results that our prototype obtains for the different benchmarks. In Table 2 we group the benchmarks for which the analysis is able to infer the exact cost function, while in Table 3 we have the benchmarks for which the analysis infers a safe upper-bound of their actual resource consumption. The analysis also infers lower bounds, but we do not show them and concentrate instead on the upper bounds for conciseness. Note that in those cases where the analysis infers exact bounds (Table 2), the inferred lower and upper bounds are of course the same. Column **Bench.** shows the name of the entry predicate for each benchmark. Column **RTC** indicates the scenario, as defined before, i.e., no run-time checks (off); full run-time checks (full); or only those left after optimizing via static verification (opt).

Column **Bound Inferred** shows the resource usage functions inferred by our resource analysis, for each of the cases. These functions depend on the input data sizes of the entry predicate (as before, l_X represents the length of list X). In order to measure the precision of the functions inferred, in Column **%D** we show the average deviation of the bounds obtained by evaluating the functions in Column **Bound Inferred**, with respect to the costs measured with dynamic profiling. The input data for dynamic profiling was selected to exhibit worst case executions. In those cases where the inferred bounds are exact, the deviation is always 0.0%. In Column **Ovhd** we show the relative run-time checking overhead as the ratio (`rtc_ratio`) between the complexity order of the cost of the instrumented code (for full or opt), and the complexity order of the cost corresponding to the original code (off). Finally, in Column $\mathbf{T_A(ms)}$ we list the *cost* analysis time for each of the three cases.[4]

[3] Sources and additional information available at http://cliplab.org/papers/rtchecks-cost/.

[4] This time does not include the static analysis and verification time in the opt case, performed with the *eterms+shfr* domains, since the process of simplifying at compile-time the assertions is orthogonal to this paper. Recent experiments and results on this topic can be found in [61].

Table 1: Description of the benchmarks.

`app1(A,B,_)`	list concatenation
`oins(E,L,_)`	insertion into an ordered list
`mmtx(A,B,_)`	matrix multiplication
`nrev(L,_)`	list reversal
`ldiff(A,B,_)`	2 lists difference
`sift(A,_)`	sieve of Eratosthenes
`pfxsum(A,_)`	sum of prefixes of a list of numbers
`bsts(N,T)`	membership checks in a binary search tree

Table 2: Experimental results (benchmarks for which analysis infers exact cost functions).

Bench.	RTC	Bound Inferred	%D	T_A(ms)	Ovhd	Verif.
	off	$l_A + 1$	0.0	98.13		
`app1(A,B,_)`	full	$3 \cdot {l_A}^2 + 6 \cdot l_A \cdot l_B + 16 \cdot l_A + 6 \cdot l_B + 13$	0.0	521.18	$l_A + l_B$	false
	opt	$3 \cdot l_A + 2 \cdot l_B + 8$	0.0	311.98	$\frac{l_B}{l_A} + 1$	false
	off	$\frac{1}{2} \cdot {l_L}^2 + \frac{3}{2} \cdot l_L + 1$	0.0	218.15		
`nrev(L,_)`	full	$\frac{1}{2} \cdot {l_L}^3 + \frac{17}{2} \cdot {l_L}^2 + 21 \cdot l_L + 11$	0.0	885.08	l_L	false
	opt	$\frac{1}{2} \cdot {l_L}^2 + \frac{5}{2} \cdot l_L + 7$	0.0	756.82	1	checked
	off	$\frac{1}{2} \cdot {l_A}^2 + \frac{3}{2} \cdot l_A + 1$	0.0	255.55		
`sift(A,_)`	full	$\frac{2}{3} \cdot {l_A}^3 + 7 \cdot {l_A}^2 + \frac{49}{3} \cdot l_A + 10$	0.0	980.63	l_A	false
	opt	$\frac{1}{2} \cdot {l_A}^2 + \frac{7}{2} \cdot l_A + 5$	0.0	521.65	1	checked
	off	$l_A + 2$	0.0	146.98		
`pfxsum(A,_)`	full	$2 \cdot {l_A}^2 + 15 \cdot l_A + 20$	0.0	749.94	l_A	false
	opt	$3 \cdot l_A + 7$	0.0	469.71	1	checked

From the results shown in Column **Ovhd** we see that the analysis correctly detects that the full run-time checking versions of the benchmarks (full case) are asymptotically worse than the original program, showing for example a linear asymptotic ratio (run-time checking overhead) for `oins/3`, or even exponential for `bsts/2`. In the case of `app/3`, we can see that the asymptotic relative overhead is linear, but the instrumented versions become dependent on the size of both arguments, while originally the cost was only depending on the size of the first list (though probably it is still worthwhile performing the checks since a list check on the second argument should have been performed anyway in the code). On the other hand, for all the benchmarks except for `app/3` and `bsts/2`, the resulting asymptotic relative overhead of the optimized run-time checking version (opt case), is null, i.e., Ovhd = 1.

In the case of `bsts/2`, the overhead is still exponential because the type analysis is not able to statically prove the property *binary search tree*. Thus, it is still necessary to traverse the input binary tree at run-time in order to verify it. However, the optimized version traverses the input tree only once, while the full version traverses it on each call, which is reflected in the resulting cost function. In any case, note that the exponential functions are on the depth of the tree d_T, not on the number of nodes. Analogously, in `oins/3` the static analysis is not able to prove the *sorted* property for the input list, although in that case the complexity order does not change for the optimized version, only the constant coefficients of the cost function are increased. We have included optimized versions of these two cases (marking them with $*$) to show the change in the overhead if the properties involved were verified; however, the *eterms+shfr* domains used cannot prove these complex properties.

Column **Verif.** shows the result of verification (i.e., `checked`/`false`/`check`) assuming a global assertion for all predicates in all the benchmarks stating that the relative run-time checking overhead should not be larger than 1 (Ovhd $\leq$ 1). Finally, Column $\mathbf{T_A}$**(ms)** shows that the analysis time is ≈ 4 times slower on versions with full instrumentation, and ≈ 2 times slower on versions instrumented with run-time checks after static analysis, respectively, but in any case all analysis times are small.

We believe that these results are encouraging and strongly suggest that our approach can provide information that can help the programmer understand statically, at the algorithmic level whether

Table 3: Experimental results (rest of the benchmarks; we show the upper bounds).

Bench.	RTC	Bound Inferred	%D	T_A(ms)	Ovhd	Verif.
oins(E,L,_)	off	$l_L + 2$	0.09	142.55		
	full	$3 \cdot (l_L + 1)^2 + 10 \cdot L_L + 11$	99.93	917.39	l_L	false
	opt*	$3 \cdot l_L + 6$	50.14	340.15	1	checked
mmtx(A,B,_)	off	$r_A \cdot c_A \cdot c_B + 3 \cdot r_A \cdot c_B + 2 \cdot r_A - 2 \cdot c_B$	7.58	460.21		
	full	$4 \cdot {r_A}^2 c_A \cdot c_B + 4 \cdot {r_A}^2 \cdot c_A + 4 \cdot {r_A}^2 \cdot c_B + 4 \cdot {r_A}^2 + r_A \cdot {c_A}^2 \cdot c_B + 4 \cdot r_A \cdot {c_A}^2 + 2 \cdot r_A \cdot c_A \cdot {c_B}^2 + 11 \cdot r_A \cdot c_A \cdot c_B + 20 \cdot r_A \cdot c_A + 15 \cdot r_A + 7$	0.0	1682.54	$N^\dagger$	false
	opt	$r_A \cdot c_A \cdot c_B + 2 \cdot c_A \cdot c_B + 2 \cdot r_A \cdot c_A + 4 \cdot r_A \cdot c_A + 6 \cdot r_A + 2 \cdot c_A + 11$	0.0	1120.23	1	checked
ldiff(A,B,_)	off	$l_A \cdot l_B + 2 \cdot l_A + 1$	2.06	786.22		
	full	${l_A}^2 + 3 \cdot l_A \cdot l_B + 13 \cdot l_A + 2 \cdot l_B + 10$	0.27	1769.22	$\frac{l_A}{l_B} + 1$	false
	opt	$l_A \cdot l_B + 5 \cdot l_A + 2 \cdot l_B + 6$	0.0	1226.15	1	checked
bsts(N,T)	off	$d_T + 3$	0.1	714.83		
	full	$3 \cdot 2^{(d_T+2)} + \frac{3}{2} \cdot d_T^2 + \frac{27}{2} \cdot d_T + 20$	1.19	438.72	$\frac{2^{d_T}}{d_T}$	false
	opt*	$3 \cdot 2^{(d_T+1)} + 4 \cdot d_T + 14$	4.01	245.09	$\frac{2^{d_T}}{d_T}$	false

$^\dagger N = max(r_A, c_A, c_B)$

the overheads introduced by the run-time checking required by the assertions in the program are acceptable or not.

6 CONCLUSIONS

We have proposed a method that uses static analysis to infer bounds on the overhead that run-time checking introduces in programs. The bounds are functions parameterized by input data sizes. Unlike profiling, this approach can provide guarantees for all possible execution traces, and allows assessing how the overhead grows as the size of the input grows. We have also extended the Ciao assertion verification framework to express "admissible" overheads, and statically and automatically check whether the instrumented program conforms with such specifications. Our experimental evaluation suggests that our method is feasible and also promising in providing bounds that help the programmer understand at the algorithmic level the overheads introduced by the run-time checking required for the assertions in the program, in different scenarios, such as performing full run-time checking or checking only the module interfaces.

Since our static analysis is compositional, there are no theoretical limits to the size of programs it can be applied to. Our approach incorporates a mechanism, the trust assertions of Ciao, that allows the programmer to provide the cost of any predicate for which the analysis infers an imprecise result, so that the imprecision does not propagate to the rest of the code. This is of course a burden, but it is obviously less work than the alternative without the tool, i.e., having to reason about every predicate. The approach is in any case useful even for small programs, since it can uncover (changes in) costs that are not immediately obvious even in such programs (see Example 6).

In general, the user should reason about the cost of the run-time checking performed by the program in the same way as about that of the rest of the code. Our tool addresses both of these tasks. Note that, since both tasks are undecidable, the best it can do is compute *safe* approximations. Since our tool cannot possibly solve the problem completely, its objective is instead to assist the programmer in these two tasks in a formally correct way. Again, the underlying argument is that it will always be better to have this tool take care of a good part of both tasks, rather than having to do everything by hand.

We believe that the application of static cost analysis for estimating the impact of run-time checks on program cost and complexity is an important contribution of this paper, and that an interesting synergy emerges from this combination. The use of run-time checks is unavoidable in many situations where it is not feasible to verify statically a given property and it is still necessary to guarantee that no incorrect execution is allowed. In this scenario our approach allows the programmer to annotate the program with pre- and post-conditions, but additionally with conditions about the admissible impact of run-time checking, in such a way that some alerts and guarantees can be received statically regarding the final performance of the program. We believe that this is essential for making design decisions, specially regarding performance-correctness trade-offs.

Note also that a useful aspect of our approach is that a change in an implementation of a predicate with the same interface but introducing an undesirable cost can be detected through an assertion violation.

Finally, as argued in the introduction and in the context of the discussion of Horn clauses as intermediate representation (and illustrated by our previous work with Java, Java bytecode, or XC), although we have presented our proposal for concreteness in the context of the Ciao system and applied it to logic programs, we believe the approach is general and can be applied directly to other languages and systems.

ACKNOWLEDGMENTS

Research partially funded by EU FP7 agreement no 318337, *ENTRA*, Spanish MINECO TIN2015-67522-C3-1-R *TRACES* project, and the Madrid M141047003 *N-GREENS* program.

A (C)LP NOTATION USED IN THE PAPER

We recall below some common (C)LP notation used throughout the paper:

- variable names start with a capital letter: `L`, `Xs`;
- predicate and functor names start with a lower-case letters: `app1C`, `rev`, `warn_if_false`;
- each predicate and functor symbol has a number associated with it, called *arity*, that denotes the number of arguments of that symbol. E.g., the notation `app1/3` means that the predicate `app1` and 3 arguments;
- `[X|Xs]` denotes a list with head `X` and tail `Xs`.

System properties appearing in the examples:

- `term(X)`: `X` is any program term (variable, constant, number, structure, etc.);
- `var(X)`: `X` is a free variable;
- `nat(X)`: `X` is a natural number;
- `list(X)`: `X` is a list (see property definition below);
- `length(L,N)`: list `L` has `N` elements.

```
1 list([]). % empty list
2 list([Head|Tail]) :- list(Tail).
```

Arithmetic expressions appearing in the examples:

- `A /\ B` : integer bitwise AND;
- `A \/ B` : integer bitwise OR;
- `A # B` : integer bitwise exclusive OR (XOR);
- `(A#1)\/B`: integer bitwise implication ($A \rightarrow B \Leftrightarrow \neg A \vee B$).

REFERENCES

[1] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2012. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)* 413, 1 (2012), 142–159. https://doi.org/10.1016/j.tcs.2011.07.009

[2] Gourinath Banda and John P. Gallagher. 2009. Analysis of Linear Hybrid Systems in CLP. In *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008 (Lecture Notes in Computer Science)*, Michael Hanus (Ed.), Vol. 5438. Springer, 55–70.

[3] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (Lecture Notes in Computer Science)*, Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte (Eds.), Vol. 9300. Springer, 24–51. https://doi.org/10.1007/978-3-319-23534-9_2

[4] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *SAS (LNCS)*, Francesco Logozzo and Manuel Fähndrich (Eds.), Vol. 7935. Springer, 105–125.

[5] J. Boye, W. Drabent, and J. Małuszyński. 1997. Declarative Diagnosis of Constraint Programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*. U. of Linköping Press, Linköping, Sweden, 123–141.

[6] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. V. Hermenegildo, J. Maluszynski, and G. Puebla. 1997. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*. U. of Linköping Press, Linköping, Sweden, 155–170. ftp://cliplab.org/pub/papers/aadebug_discipldeliv.ps.gz

[7] Robert Cartwright and Mike Fagan. 1991. Soft Typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI 1991)*. ACM, New York, NY, USA, 278–292.

[8] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2014. VeriMAP: A Tool for Verifying Programs through Transformations. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science)*, Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 568–574. https://doi.org/10.1007/978-3-642-54862-8_47

[9] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008 (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340.

[10] S. K. Debray and N. W. Lin. 1993. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems* 15, 5 (November 1993), 826–875.

[11] S. K. Debray, N.-W. Lin, and M. V. Hermenegildo. 1990. Task Granularity Analysis in Logic Programs. In *Proc. 1990 ACM Conf. on Programming Language Design and Implementation (PLDI)*. ACM Press, 174–188.

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

[13] Christos Dimoulas and Matthias Felleisen. 2011. On Contract Satisfaction in a Higher-Order World. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 5, Article 16 (Nov. 2011), 29 pages. https://doi.org/10.1145/2039346.2039348

[14] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. 1988. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*. 573–581.

[15] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. 1989. Algorithmic Debugging with Assertions. In *Meta-programming in Logic Programming*, H. Abramson and M.H.Rogers (Eds.). MIT Press, 501–522.

[16] Manuel Fähndrich and Francesco Logozzo. 2011. Static Contract Checking with Abstract Interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software, FoVeOOS'10 (Lecture Notes in Computer Science)*, Vol. 6528. Springer-Verlag, Berlin, Heidelberg, 10–30. http://dl.acm.org/citation.cfm?id=1949303.1949305

[17] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 48–59. https://doi.org/10.1145/581478.581484

[18] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. 2009. Profile-guided Static Typing for Dynamic Scripting Languages. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 283–300. https://doi.org/10.1145/1640089.1640110

[19] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution). In *TACAS (LNCS)*, Cormac Flanagan and Barbara König (Eds.), Vol. 7214. Springer, 549–551.

[20] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[21] M. Hanus. 2017. Combining Static and Dynamic Contract Checking for Curry. *CoRR* abs/1709.04816 (2017).

[22] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. 2012. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming* 12, 1–2 (January 2012), 219–252. https://doi.org/doi:

10.1017/S1471068411000457 http://arxiv.org/abs/1102.5497.

[23] M. V. Hermenegildo, G. Puebla, and F. Bueno. 1999. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren (Eds.). Springer-Verlag, 161–192.

[24] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. 2012. A Verification Toolkit for Numerical Transition Systems - Tool Paper. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings (Lecture Notes in Computer Science)*, Dimitra Giannakopoulou and Dominique Méry (Eds.), Vol. 7436. Springer, 247–251. https://doi.org/10.1007/978-3-642-32759-9_21

[25] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. 2012. TRACER: A Symbolic Execution Tool for Verification. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. Springer, 758–766. https://doi.org/10.1007/978-3-642-31424-7_61

[26] B. Kafle, J. P. Gallagher, and J. F. Morales. 2016. RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer, 261–268. https://doi.org/10.1007/978-3-319-41528-4_14

[27] Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. 2018. Refinement Types for Ruby. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'18)*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, Cham, 269–290. https://doi.org/10.1007/978-3-319-73721-8_13

[28] Emmanouil Koukoutos and Viktor Kuncak. 2014. Checking Data Structure Properties Orders of Magnitude Faster. In *Runtime Verification*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.). Lecture Notes in Computer Science, Vol. 8734. Springer International Publishing, 263–268. https://doi.org/10.1007/978-3-319-11164-3_22

[29] Claude Laï. 2000. Assertions with Constraints for CLP Debugging. In *Analysis and Visualization Tools for Constraint Programming (Lecture Notes in Computer Science)*, Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski (Eds.), Vol. 1870. Springer, 109–120.

[30] Leslie Lamport and Lawrence C. Paulson. 1999. Should Your Specification Language be Typed? *ACM Transactions on Programming Languages and Systems* 21, 3 (May 1999), 502–526.

[31] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. 2007. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.* 19, 2 (2007), 159–189.

[32] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. 2016. Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR. In *Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers*, M. Van Eekelen and U. Dal Lago (Eds.). Lecture Notes in Computer Science, Vol. 9964. Springer, 81–100. https://doi.org/10.1007/978-3-319-46559-3_5 arXiv:1511.01413

[33] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder. 2014. Energy Consumption Analysis of Programs based on XMOS ISA-Level Models. In *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers (Lecture Notes in Computer Science)*, Gopal Gupta and Ricardo Peña (Eds.), Vol. 8901. Springer, 72–90. https://doi.org/10.1007/978-3-319-14125-1_5

[34] Francesco Logozzo et al. [n. d.]. Clousot. http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx.

[35] P. López-García, L. Darmawan, and F. Bueno. 2010. A Framework for Verification and Debugging of Resource Usage Properties. In *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10) (Leibniz International Proceedings in Informatics (LIPIcs))*, M. V. Hermenegildo and T. Schaub (Eds.), Vol. 7. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 104–113.

[36] P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. V. Hermenegildo. 2012. Interval-Based Resource Usage Verification: Formalization and Prototype. In *Foundational and Practical Aspects of Resource Analysis. Second International Workshop FOPARA 2011, Revised Selected Papers*, R. Pe na, M.V. Eekelen, and O. Shkaravska (Eds.). Lecture Notes in Computer Science, Vol. 7177. Springer-Verlag, 54–71. https://doi.org/10.1007/978-3-642-32495-6_4

[37] P. Lopez-Garcia, M. Klemen, U. Liqat, and M. V. Hermenegildo. 2016. A General Framework for Static Profiling of Parametric Resource Usage. *Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue* 16, 5-6 (October 2016), 849–865. https://doi.org/10.1017/S1471068416000442

[38] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to FLIX: a Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 194–208. https://doi.org/10.1145/2908080.2908096

[39] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. 2007. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007) (Lecture Notes in Computer Science)*. Springer-Verlag, 154–168.

[40] E. Mera, P. López-García, and M. V. Hermenegildo. 2009. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th Int'l. Conference on Logic Programming (ICLP'09) (LNCS)*, Vol. 5649. Springer-Verlag, 281–295.

[41] E. Mera, T. Trigo, P. López-García, and M. V. Hermenegildo. 2011. Profiling for Run-Time Checking of Computational Properties and Performance Debugging. In *Practical Aspects of Declarative Languages (PADL'11) (Lecture Notes in Computer Science)*, Vol. 6539. Springer-Verlag, 38–53.

[42] MSR. [n. d.]. Code Contracts. http://research.microsoft.com/en-us/projects/contracts/.

[43] K. Muthukumar and M. Hermenegildo. 1991. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*. MIT Press, 49–63.

[44] K. Muthukumar and M. Hermenegildo. 1992. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming* 13, 2/3 (July 1992), 315–347.

[45] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. 2009. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09) (Electronic Notes in Theoretical Computer Science)*, Vol. 253. Elsevier - North Holland, 65–82.

[46] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. 2007. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07) (Lecture Notes in Computer Science)*, Vol. 4670. Springer.

[47] Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 139–152. https://doi.org/10.1145/2628136.2628156

[48] Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2017. Higher-order Symbolic Execution for Contract Verification and Refutation. *Journal of Functional Programming* 27, 3 (January 2017). https://doi.org/10.1017/S0956796816000216

[49] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for java. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. 201–212. https://doi.org/10.1145/1390630.1390656

[50] P. Pietrzak, J. Correas, G. Puebla, and M. V. Hermenegildo. 2006. Context-Sensitive Multivariant Assertion Checking in Modular Programs. In *13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06) (LNCS)*. Springer-Verlag, 392–406.

[51] Á. Rebón Portillo, K. Hammond, H-W. Loidl, and P. Vasconcelos. 2002. Cost Analysis Using Automatic Size and Time Inference. In *Proceedings of the International Workshop on Implementation of Functional Languages (Lecture Notes in Computer Science)*, Vol. 2670. Springer-Verlag, Madrid, Spain, 232–247.

[52] G. Puebla, F. Bueno, and M. V. Hermenegildo. 2000. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, P. Deransart, M. V. Hermenegildo, and J. Maluszynski (Eds.). Number 1870 in LNCS. Springer-Verlag, 23–61.

[53] G. Puebla, F. Bueno, and M. V. Hermenegildo. 2000. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99) (LNCS)*. Springer-Verlag, 273–292.

[54] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 167–180. https://doi.org/10.1145/2676726.2676971

[55] Brianna M. Ren and Jeffrey S. Foster. 2016. Just-in-time Static Type Checking for Dynamic Languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 462–476. https://doi.org/10.1145/2908080.2908127

[56] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. 2008. Towards Typed Prolog. In *International Conference on Logic Programming (LNCS)*, Enrico Pontelli and María M. García de la Banda (Eds.). Springer Verlag, 693–697.

[57] A. Serrano, P. Lopez-Garcia, and M. V. Hermenegildo. 2014. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue* 14, 4-5 (2014), 739–754. https://doi.org/10.1017/S147106841400057X

[58] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. 81–92.

[59] Vincent St-Amour, Leif Andersen, and Matthias Felleisen. 2015. Feature-Specific Profiling. In *Proceedings of the 24th International Conference on Compiler Construction*, Björn Franke (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–68.

[60] N. Stulova, J. F. Morales, and M. V. Hermenegildo. 2015. Practical Run-time Checking via Unobtrusive Property Caching. *Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue* 15, 04-05 (September 2015), 726–741. https://doi.org/10.1017/S1471068415000344 http://arxiv.org/abs/1507.05986.

[61] N. Stulova, J. F. Morales, and M. V. Hermenegildo. 2018. Some Trade-offs in Reducing the Overhead of Assertion Run-time Checks via Static Analysis. *Science of Computer Programming* 155 (April 2018), 3–26. https://doi.org/10.1016/j.scico.2017.12.006 Selected and Extended papers from the 2016 International Symposium on Principles and Practice of Declarative Programming.

[62] Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. 2015. Towards Practical Gradual Typing. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 4–27. https://doi.org/10.4230/LIPIcs.ECOOP.2015.4

[63] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 456–468. https://doi.org/10.1145/2837614.2837630

[64] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008), San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 395–406. https://doi.org/10.1145/1328438.1328486

[65] Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-order symbolic execution via contracts. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 537–554. https://doi.org/10.1145/2384616.2384655

[66] C. Vaucheret and F. Bueno. 2002. More Precise yet Efficient Type Inference for Logic Programs. In *9th International Static Analysis Symposium (SAS'02) (Lecture Notes in Computer Science)*, Vol. 2477. Springer-Verlag, 102–116.

[67] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161