Technical report CLIP-1/2017.0

Term Hiding and its Impact on Run-time Check Simplification*

Nataliia Stulova[1,2] José F. Morales[1] Manuel V. Hermenegildo[1,2]

[1]IMDEA Software Institute

{nataliia.stulova, josef.morales, manuel.hermenegildo}@imdea.org

[2]Universidad Politécnica de Madrid (UPM), Madrid, Spain

manuel.hermenegildo@upm.es

17 May 2017

Abstract

One of the most attractive features of untyped languages for programmers is the flexibility in term creation and manipulation. However, with such power comes the responsibility of ensuring correctness of operations. A solution is adding run-time checks to the program via assertions, but this can introduce overheads that are in many cases impractical. While such overheads can be greatly reduced with static analysis, the gains depend strongly on the quality of the information inferred. Reusable libraries, i.e., library modules that are pre-compiled independently of the client, pose special challenges in this context.

We propose a relaxed form of atom-based module system (which hides only a selected set of functor symbols but still provides a strict mechanism to prevent breaking visibility rules across modules) that can enrich significantly the shape information that can be inferred in reusable modular programs. We also propose an improved run-time checking approach that takes advantage of the proposed mechanisms to achieve

*An extended abstract of this work is published as [1]

large reductions in overhead, closer to those of static languages even in the reusable-library context. While the approach is general and system-independent, we present it for concreteness in the context of the Ciao assertion language and combined static/dynamic checking framework. Our method maintains full expressiveness of the checks in this context. Contrary to other approaches it does not introduce the need to switch the language to (static) type systems, which is known to change the semantics in languages like Prolog. We also study the approach experimentally and evaluate the overhead reduction achieved in the run-time checks.

Keywords: Module Systems; Implementation; Run-time Checking; Assertion-based Debugging and Validation; Static Analysis.

1 Introduction

Modular programming has become widely adopted due to the benefits it provides in code reuse and structuring data flow between program components. A tightly related concept is the principle of *information hiding* that allows concealing the concrete implementation details behind a well-defined interface and thus allows for cleaner abstractions. In different programming languages these concepts are implemented in different ways, some examples being the encapsulation mechanism of classes adopted in OOP and opaque data types. In the (constraint) logic programming context, most mature language implementations incorporate module systems, which are either *predicate-based* (where predicate symbol visibility is controlled by the module import-export rules but functor symbols are public) or *atom-based* (where both predicate and functor symbol visibility is controlled by the module import-export rules).

In this work we propose a hybrid predicate-based module system that offers an optional hiding mechanism for selected functor symbols. The proposed module system is still strict in the sense that it disallows breaking predicate or term visibility rules by bypassing the module interfaces. These features allow programmers to restrict the visibility of some terms to the module where they are defined thus both hiding the concrete implementation details from other modules and providing guarantees that all data terms with such shapes may only be constructed by the predicates of that particular module.

Our motivation comes from the reusable library scenario, i.e., the case of analyzing, verifying, and compiling a library for general use, without access to the client code or analysis information on it. This includes for example the important case of servers accessed via remote procedure calls. One of the most attractive features of untyped languages for programmers is the flexibility they offer in term creation and manipulation. However, with such power comes the responsibility of ensuring correctness in the manipulation of data, and this is specially relevant when data can come from unknown clients. A popular solution for ensuring safety is to enhance the language with optional assertions that allow specifying correctness conditions both at the module boundaries and internally to modules. These assertions can be checked dynamically by adding run-time checks to the program, but this can also introduce overheads that are in many cases impractical. Such overheads can be greatly reduced with static analysis, but the gains then depend strongly on the quality of the analysis information inferred. Unfortunately, in the reusable library setting shape/type analyses are necessarily imprecise, since in this context the unknown clients can fake data that is really intended to be internal to the library. Ensuring safety then requires sanitizing input data with potentially expensive run-time checks.

In order to alleviate this problem, we present a technique that, using the combination of term hiding and the strict visibility rules in the module system, greatly improves static analysis by enhancing the inference of shape information. We also demonstrate experimentally how, thanks to the term creation safety guarantees provided by the module system, it is possible to reduce the run-time overhead for the calls across module boundaries by several orders of magnitude. The combination of these techniques and traditional static analysis brings improvements in the number and size of checks that allow providing guarantees and overheads that are similar to those of statically-typed approaches, but without imposing on programs the restrictions of being well typed.

For concreteness, we use in this work the relevant parts of the Ciao system [2]: the module system, the assertion language –which allows providing optional program specifications with various kinds of information, such as modes, (regular) types, or non-determinism–, and the verification framework, that combines static and dynamic checking. However, our results are general and can be applied to other languages.

2 Modular Programs with Hiding

This section provides preliminary concepts and introduces a simple definition of *modular* logic programs with *hiding* of symbols. We start by recalling some basic notation and the standard program semantics, following the formalization of [3]. An *atom* A is a syntactic construction of the form $f(t_1, \ldots, t_n)$ where f is a symbol of arity n and t_i are terms. A *constraint* is a conjunction of expressions built from predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). A *literal* is either an atom or a constraint. *Terms* are inductively defined as variable symbols or constructions of the form $f(t_1, \ldots, t_n)$ where

f is a symbol of arity n ($n \geq 0$) and t_i are terms. Note that we do not distinguish between predicate and uninterpreted function symbols. *Constants* are introduced as 0-ary symbols. A *goal* is a conjunction of literals. A *clause* is defined as $H \leftarrow B$, where H is an atom and B is a goal. A *definite program* is a finite set of clauses. We extend the simple notion of definite program to modular program as follows:

Definition 1 (Modular Program)**.** *A* modular program *is a definite program P together with a mapping* $\mathsf{mod}(\cdot)$ *which assigns for each symbol f a unique module symbol m. Let C be a clause $H \leftarrow B$,* $\mathsf{mod}(C) \triangleq \mathsf{mod}(H)$*. Let A be an atom*[1] *or a term of the form $f(\ldots)$, then* $\mathsf{mod}(A) \triangleq \mathsf{mod}(f)$*.*

Typically, programs in their *source form* are organized into a set of modules, each of them annotated with directives for declaring import and export lists declaring the visibility of local symbols. We assume that by default no module symbols are assigned to the symbols of a program in its source form. The compiler and the run-time system, applying a set of visibility rules, compose an *expanded* program as described in Def 1 from a set of program sources. Very briefly,[2] the expanded program from a module m produces a set of clauses such that:

- Each C has the form $H \leftarrow B$ where $\mathsf{mod}(H) = m$
- For each atom $A = f(\ldots)$ in goal B, either $\mathsf{mod}(H) = m$ or $\mathsf{mod}(H) = n$ with n an imported module in m and f exported from n.

As mentioned before, the main advantage of modular programming is that it allows safe local reasoning on modules, since two different modules are not allowed to contribute clauses to the same predicate.[3] The concept of module *interface* is instrumental in local reasoning as it allows to clearly mark the module boundaries. In what follows we assume the following definition of module interface:

Definition 2 (Module Interface)**.** *A* module interface *consists of its exported predicates and a set of valid* call substitutions *for those predicates. As in*

[1] In practice constraints are also located in modules. It is trivial to extend the formalization to include this, we do not write it explicitly for simplicity.

[2] For the sake of clarity we do not include additional rules to support `meta_predicates` or run-time module expansions. Again, these features can be added without altering the results and have been left out for simplicity of exposition.

[3] In practice, an exception are `multifile` predicates. However, since they need to be declared explicitly, local reasoning is still valid assuming conservative semantics (e.g., *topmost* abstract values) for them.

Section 3, valid calls *are described (and implemented) by disjunctions of conjunctions of properties (rather than with possibly infinite sets).*

Note that there are different approaches to the implementation of module systems and its formalization. One possibility is to carry explicitly in the formalization the module information by, e.g., making symbols be a pair. However, this requires significant changes in the semantics and the implementation that are unnecessary in the expansion-based approach that we use. Also, this approach reflects better what is actually done in the implementation, for the same simplicity and efficiency reasons.

The default rules to determine the module of a symbol depend on the specific dialect and system. Most systems use either a *predicate-* or an *atom*-based module system. Informally, in predicate-based module systems all symbols involved in terms are global and in atom-based module systems [4] all symbols are local except for constants and those that are explicitly exported. The Ciao module system [5] introduced a `hide` directive that declares a source symbol as local and non-exported, but this notion was not formalized. The following definition provides such a formalization:

Definition 3 (Predicate-based Module Systems with Hidden Symbols)**.** *For each source term in m, each hidden symbol is mapped into an expanded term f s.t.* $\mathsf{mod}(f) = m$ *and f cannot appear in any clause C except if* $\mathsf{mod}(C) = m$*. Each non-hidden symbol is mapped into an f s.t.* $\mathsf{mod}(f) = \mathsf{usr}$*, where* usr *is a distinguished module name for global* user terms *and is implicitly imported from any module.*

We recall the classic operational semantics of non-modular definite programs, given in terms of program *derivations*, which are sequences of *reductions* between *states*. The *definition* of an atom A in a program, $\mathsf{defn}(A)$, is the set of variable renamings of the program clauses s.t. each renaming has A as a head and has distinct new local variables. We use :: to denote concatenation of sequences. A *state* $\langle G \mid \theta \rangle$ consists of a goal sequence G and a constraint store (or *store* for short) θ. A *query* is a pair (L, θ), where L is a literal and θ a store, for which the (constraint) logic programming system starts a computation from state $\langle L \mid \theta \rangle$. We denote by $\mathsf{answers}(Q)$ the set of answers to a query Q.

Definition 4 (Operational Semantics (Non-modular))**.** *A state* $S = \langle L :: G \mid \theta \rangle$ *where L is a literal can be* reduced[4] *to a state S' as follows:*

1. $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \wedge L \rangle$ *if L is a constraint and $\theta \wedge L$ is satisfiable.*

[4] Assuming, for simplicity, that the underlying constraint solver is complete.

2. $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle B :: G \mid \theta \rangle$ *if* L *is an atom of the form* $f(t_1, \ldots, t_n)$*, for some clause* $(L \leftarrow B) \in \textsf{defn}(L)$.

We now provide the operational semantics of modular programs, which is equivalent to the non-modular semantics but keeps track of the module that originated each goal. To track calls across module boundaries we also introduce the notion of *clause end literal*, a marker of the form $\mathsf{ret}(H)$ where H stands for the head of the parent clause.

Definition 5 (Operational Semantics of Modular Programs). *We redefine the derivation semantics such that goal sequences are of the form* $(L, m) :: G$ *where* L *is a literal, and* m *is the module from which* L *was introduced. Then, a state* $S = \langle L :: G \mid \theta \rangle$ *can be reduced to a state* S' *as follows:*

1. $\langle (L, m) :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \wedge L \rangle$ *if* L *is a constraint and* $\theta \wedge L$ *is satisfiable.*
2. $\langle (L, m) :: G \mid \theta \rangle \rightsquigarrow \langle (B_1, n) :: \ldots :: (B_k, n) :: (\textsf{ret}(L), n) :: G \mid \theta \rangle$ *w.r.t. some clause* $(L \leftarrow B_1, \ldots, B_k) \in \textsf{defn}(L)$ *where* $\mathsf{mod}(L) = n$.
3. $\langle (L, m) :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \rangle$ *if* L *is a clause return literal* $\textsf{ret}(\_)$.

In order for reduction step 2 to succeed, the L *literal should either be defined in module* m *(and then* $n = m$*) or it should belong to the export list of module* n *and be in the import list of module* m.

3 Run-Time Checking of Modular Programs

Assertion Language. We assume that program specifications are provided by means of assertions: linguistic constructions that allow expressing properties of programs. For concreteness we will use the `pred` assertions of the Ciao assertion language [6, 7, 2], following the formalization of [8, 3]. The main intent behind the construction of a specification for a predicate using such `pred` assertions is to define the set of all admissible preconditions for this predicate, and for each such pre-condition in turn specify the respective post-condition. These pre- and post-conditions are formulas containing literals corresponding to predicates that are specially labeled as *properties*. Properties and the other predicates composing the program are written in the same language. This approach is motivated by the direct correspondence between the declarative and operational semantics of constraint logic programs. In what follows we refer to properties as *prop* literals. The same visibility rules apply to *prop* literals as to any other program literals. A set of assertions for a predicate, represented by atom L, is shown in the left part of Fig. 1. The Pre_i and $Post_i$

```
:- pred L :  Pre_1 => Post_1.
...
:- pred L :  Pre_n => Post_n.
```

$$C_i = \begin{cases} c_i.\mathsf{calls}(L, \bigvee_{j=1}^{n} Pre_j) & i = 0 \\ c_i.\mathsf{success}(L, Pre_i, Post_i) & i = 1..n \end{cases}$$

Figure 1: Correspondence between assertions and assertion conditions.

are conjunctions[5] of *prop* literals that refer to the variables of L. Such a set of assertions states that in any execution state $\langle (L, m) :: G \mid \theta \rangle$ 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 L succeeds, the corresponding $Post_i$ should hold upon success.

Given a predicate represented by a normalized atom L and the corresponding set of assertions for L, $\mathcal{A}(L) = \{A_1 \ldots A_n\}$, with A_i = "`:- pred` L `:` Pre_i `=>` $Post_i$`.`" such assertions are normalized into a set of *assertion conditions* for that predicate, denoted as $\mathcal{A}_C(L) = \{C_0, C_1, \ldots, C_n\}$, as shown in Fig. 1, right. The c_i are identifiers which are unique for each assertion condition. If there are no assertions associated with L then the corresponding set of assertion conditions is empty. The set of assertion conditions for a program is the union of the assertion conditions for each of the predicates in the program. The $\mathsf{calls}(L, \ldots)$ conditions encode the checks that ensure that the calls to the predicate represented by the L literal are within those admissible by the set of assertions, and we thus call them the *calls assertion conditions*. The $\mathsf{success}(L, Pre_i, Post_i)$ conditions encode the checks for compliance of the successes for particular sets of calls, and we thus call them the *success assertion conditions*.

Semantics with Assertions and Modules. We now present the operational semantics with assertions for modular programs, which checks whether assertion conditions hold or not while computing the derivations from a query in a modular program. The identifiers of the assertion conditions (the c_i) are used to keep track of any violated assertion conditions. The extended program state has the form $\langle G \mid \theta \mid \mathcal{E} \rangle$, where $\mathcal{E}$ denotes the set of identifiers for falsified assertion conditions and $|\mathcal{E}| \leq 1$. We will write identifiers in *negated* form when they appear in the error set. We also extend the clause return literal to the form $\mathsf{ret}(H, \mathcal{C})$, where $\mathcal{C}$ is the set of identifiers c_i of the assertion conditions that should be checked at that derivation point. A literal L *succeeds trivially* for θ in program P, denoted $\theta \Rightarrow_P L$, iff $\exists \theta' \in \mathsf{answers}(L, \theta)$ such that $\theta \models \theta'$. This notion captures the checking of properties and we will thus often refer to

[5] In the general case *Pre* and *Post* can be DNF formulas of *prop* literals but we limit them to conjunctions herein for simplicity of presentation.

this operation as "checking L in the context of θ."

Definition 6 (Operational Semantics for Modular Programs with Assertions)**.** *A state $S = \langle (L, m) :: G \mid \theta \mid \varnothing \rangle$, where L is a literal introduced from a clause in a module m, can be* reduced *to a state S', denoted $S \leadsto_{\mathcal{A}} S'$, as follows:*

1. *If L is a constraint then the new state is $S' = \langle G' \mid \theta' \mid \varnothing \rangle$ where G' and θ' are obtained in a same manner as in $\langle (L, m) :: G \mid \theta \rangle \leadsto \langle G' \mid \theta' \rangle$*

2. *If L is an atom and $\exists (L \leftarrow B_1, \ldots, B_k) \in \mathsf{defn}(L)$, then the new state is obtained as $S' = \langle (B_1, n) :: \ldots :: (B_k, n) :: (\mathsf{ret}(L, \mathcal{C}), n) :: G \mid \theta \mid \mathcal{E} \rangle$ where*

$$\mathcal{E} = \begin{cases} \{\bar{c}\} & \text{if } \exists\, c.\mathsf{calls}(L, Pre) \in \mathcal{A}_C(L) \ \wedge \ \theta \not\Rightarrow_P Pre \\ \varnothing & \text{otherwise} \end{cases}$$

   s.t. $\mathcal{C} = \{c_i \mid c_i.\mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A}_C(L) \ \wedge \ \theta \Rightarrow_P Pre_i\}$ and $\mathsf{mod}(L) = n$.

3. *If L is a clause return literal $\mathsf{ret}(\_, \mathcal{C})$, then $S' = \langle G \mid \theta \mid \mathcal{E} \rangle$ where*

$$\mathcal{E} = \begin{cases} \{\bar{c}\} & \text{if } \exists\, c \in \mathcal{C} \text{ s.t. } c.\mathsf{success}(L', \_, Post) \in \mathcal{A}_C(L') \ \wedge \ \theta \not\Rightarrow_P Post \\ \varnothing & \text{otherwise} \end{cases}$$

We will often refer to the operations in step 2 of this semantics as *checking the calls conditions* and to those in step 3 as *checking the success conditions.*

These two theorems from [8] carry over to the modular case:[6]

Theorem 1 (Correctness and Completeness Under Assertion Checking)**.** *Let P be the program, $\mathcal{Q}$ a set of all queries to it, and $\mathcal{A}$ the set of its assertions. Let $\mathsf{derivs}(\mathcal{Q})$ denote the set of derivations for a program from the set of queries in $\mathcal{Q}$, and $\mathsf{derivs}_{\mathcal{A}}(Q)$ denote the set of derivations using the semantics with assertions. Then for any tuple $(P, \mathcal{Q}, \mathcal{A})$ it holds that $\mathsf{derivs}(\mathcal{Q}) = \mathsf{derivs}_{\mathcal{A}}(\mathcal{Q})$ after filtering out check and ret literals and error sets.*

Theorem 2 (Run-time Error Detection)**.** *Let $\mathcal{E}(D)$ denote the error set of the last state of a derivation D. For any annotated program $(P, \mathcal{Q}, \mathcal{A})$, $C \in \mathcal{A}_C$ is false iff $\exists\, D \in \mathsf{derivs}_{\mathcal{A}}(\mathcal{Q})$ s.t. $\mathcal{E}(D) = \{\bar{c}\}$ where c is the identifier of C.*

[6]The proofs are trivial, and are not included for space reasons.

4 Shallow Run-Time Checking

Intuitively, *shallow run-time checking* is the usage of optimized –shallow– properties for calls conditions during run-time checks, that exploit certain characteristics of hidden symbols. These optimized, or *shallow* versions of properties are weakened forms that are semantically equivalent to the original ones in the context of the possible program executions, and are cheaper to execute (e.g., requiring asymptotically fewer steps).

Note that hidden functor symbols are essential to reason *compositionally* about the flow of data in a program composed of *reusable* libraries. This is analogous to the reasoning about the semantics of the predicates in a module, which requires the predicate symbols to be local.

In order to define formally shallow checking, as well as the algorithms to compute shallow versions of properties, we will characterize all possible terms that may exist outside a module m as its *escaping terms*, and introduce shallow properties as the *specialization* of the definition of these properties w.r.t. these escaping terms.

Escaping terms. We characterize all possible states outside a module m by defining a property that describes the *valid* values of the constraint store. Without hidden symbols, this property provides no information, since any module can construct any term. With hidden symbols, this is no longer true. We present some examples before introducing formally these concepts.

Example 1. Let `point/1` be a hidden symbol in a module that exports a single predicate `p/1` that constructs a term `point(1)`:

```
:- module(m1, [p/1]).
:- hide point/1.
p(A) :- A = point(B), B = 1.
```

There is no success substitution for `p/1` where variables can be bound to some `point(_)` more general than `point(1)`. The same applies to any possible substitution in any derivation in programs that are composed with this module. Without hiding, this is impossible to ensure (without client knowledge) since any module could define any `point(_)` terms.

Example 2. The following example *leaks* a `point(_)` term to module `m2`. Thus escaping terms include `point(_)`:

```
:- use_module(m2, [q/1]).
p(A) :- A = point(B), q(A), B = 1.
```

Example 3. The following example always passes a `point(1)` term (either by calling module `m2` or returning from `m1`).

Algorithm 1 ESCAPING_TERMS

```
function ESCAPING_TERMS(M)
    Def := usr(X)
    for all L exported from M do
        for all c.success(L, _, Post) ∈ 𝒜_C(L) do
            for all P ∈ LITNAMES(Post, vars(L)) do
                Def := Def ⊔ P(X)
    for all L imported from M do
        for all c.calls(L, Pre) ∈ 𝒜_C(L) do
            for all P ∈ LITNAMES(Pre, vars(L)) do
                Def := Def ⊔ P(X)
    return (esc_m(X) ← Def)
function LITNAMES(G, Args)
    return set of P such that A ∈ Args and G = (... ∧ P(A) ∧ ...)
```

```
p(A) :- A = point(B), B = 1, q(A).
```

Definition 7 (Escaping terms)**.** *Consider all states S in all derivations of any program that imports a given module m. $\mathsf{esc}_m(X)$ (escaping terms w.r.t. m) is the* smallest property *such that $\theta \Rightarrow_P \mathsf{esc}_m(X)$, for each $S = \langle (L, n) :: G \mid \theta \rangle$ with $n \neq m$, and variable X in the literal L.*

Let us denote by θ_i each of the θ in the definition above, and by $Vars_i$ the variables of L or H in $\mathsf{ret}(H)$. Then $\mathsf{esc}_m(X) \equiv \bigvee_i \bigvee_{V \in Vars_i} (X = V \wedge \theta_i)$. Note that $\mathsf{usr}(X)$ entails $\mathsf{esc}_m(X)$, where $\mathsf{usr}(X)$ is the property that describes all *user* terms.

Lemma 3 (Escaping at the boundaries)**.** *Consider all derivation steps $S_1 \rightsquigarrow S_2$ where $S_1 = \langle (L_1, m) :: \_ \mid \_ \rangle$ and $S_2 = \langle (L_2, n) :: \_ \mid \theta \rangle$ with $n \neq m$. That is, the derivation steps when calling a predicate at n from m (if L_1 is a literal) or when returning from m to module n (if L_1 is $\mathsf{ret}(\_)$). Let $\mathsf{esc}'_m(X)$ be the* smallest property *such that $\theta \Rightarrow_P \mathsf{esc}'_m(X)$ for each variable X in the literal L_2, and $\mathsf{usr}(X) \Rightarrow_P \mathsf{esc}'_m(X)$. Then $\mathsf{esc}'_m(X) \wedge \mathsf{usr}(X)$ is equivalent to $\mathsf{esc}_m(X)$.*

The lemma above states that it is enough to consider the states at the module boundaries to compute $\mathsf{esc}_m(X)$. This and other proofs can be found in A.

Algorithm 1 computes an over-approximation of the $\mathsf{esc}_m(X)$ property. The algorithm has two parts. First, it loops over the exported predicates in module m. For each exported predicate we use $Post$ from the success assertion conditions as a safe over-approximation of the constraints that can be introduced during the execution of the predicate. We compute the union ($\sqcup$, which is equivalent to $\vee$ but it may simplify the representation) of all properties that restrict any variable argument in $Post$. The second part of the algorithm performs the same operation on all the properties specified in the Pre of the calls assertions conditions. This is a safe approximation of the constraints that can be *leaked* to other modules called from m.

Note that the algorithm can use analysis information to detect more precise calls to the imported predicates, as well as more precise successes of the exported predicates, than those specified in the assertion conditions present in the program.

Lemma 4 (Correctness of ESCAPING_TERMS). *The* ESCAPING_TERMS *algorithm computes a safe (over)approximation to* $\mathsf{esc}_m(X)$ *(when using the operational semantics with assertions).*

Shallow Properties. Shallow run-time checking consists in using *shallow* versions of properties in the run-time checks for the calls across module boundaries. Despite this could be added directly to the operational semantics, we will present it as a program transformation based on the generation of shallow versions of the properties.

Example 4. Assume that the set of escaping terms of m contains `point(1)` and it does not contain the more general `point(_)`. Consider the property `intpoint(point(X)) :- int(X)`. Checking `intpoint(A)` at any program point outside m must check first that `A` is instantiated to `point(X)` and that `X` is instantiated to an integer (`int(X)`). However, the escaping terms show that it is not possible for a variable to be bound to `point(X)` without `X=1`. Thus, the latter check is redundant. We can compute the optimized – or *shallow* – version of `intpoint/1` in the context of all execution points external to m as `intpoint(point(_))`.

Let SPEC(L, Pre) generate a specialized version L' of predicate L w.r.t. calls given by Pre (see [9]). It holds that for all θ, $\theta \Rightarrow_P L$ iff $\theta \wedge Pre \Rightarrow_P L'$.

Definition 8 (Shallow property). *The shallow version of a property* $L(X)$ *w.r.t. module* m *is denoted as* $L(X)^{\#}$*, and computed as* SPEC($L(X), Q(X)$)*, where* $Q(X)$ *is a (safe) approximation of the escaping terms of* m *(*ESCAPING_TERMS(m)*).*

Algorithm 2 computes the optimized version of a module interface using shallow checks. It first introduces wrappers for the exported predicates, i.e., predicates `p(X) :- p'(X)`, renaming all internal occurrences of `p` by `p'`. Then

Algorithm 2 Shallow_Interface

function Shallow_Interface(M)
  Let M' be M with wrappers for exported predicates
   (to differentiate internal from external calls)
  Let $Q(X) :=$ Escaping_Terms(M')
  for all L exported from M **do**
    for all $c.\mathsf{calls}(L, Pre) \in \mathcal{A}_C(L)$ **do**
      Update $\mathcal{A}_C(L)$ with $c.\mathsf{calls}(L, Pre^{\#})$
    for all $c.\mathsf{success}(L, Pre, Post) \in \mathcal{A}_C(L)$ **do**
      Update $\mathcal{A}_C(L)$ with $c.\mathsf{success}(L, Pre^{\#}, Post)$
  return M'

it computes an approximation $Q(X)$ of the escaping terms of M. Finally, it updates all Pre in calls and success assertion conditions, for all exported predicates, with its shallow version $Pre^{\#}$. We compute the shallow version of a conjunction of literals $Pre = \bigwedge_i L_i$ as $Pre^{\#} = \bigwedge_i L_i^{\#}$.

Theorem 5 (Correctness of Shallow_Interface)**.** *Replacing a module m in a larger program by its shallow version does not alter the operational semantics.*

Discussion about precision. The presence of any *top* properties in the calls or success assertion conditions will propagate to the end in the Escaping_Terms algorithm (see Algorithm 1). For a significant class of programs, this is not a problem as soon as we can provide or infer precise assertions which do not use this top element. Note that $\mathsf{usr}(X)$, since it has a void intersection with any hidden term, does not represent a problem. For example, many generic Prolog term manipulation predicates (e.g., `functor/3`) typically accept a *top* element in their calls conditions. We restrict these predicates to work only on usr (i.e., not hidden) symbols.[7] More sophisticated solutions, that are outside the scope of this paper include: producing monolithic libraries (creating versions of the imported modules and using abstract interpretation to obtain more precise assertion conditions); or disabling shallow checking (e.g., with a dynamic flag) until the execution exits the context of m (which is correct except for the case when terms are dynamically asserted).

Multi-library scenarios. Recall that properties can be exported and used in assertions from other modules. The shallow version of properties in m are safe

[7]This can be implemented very efficiently with a simple bit check on the atom properties and does not impact the execution.

Table 1: Benchmark metrics.

Name	LOC	Size (KB)	Assertions	# Hidden
`AVL-tree`	147	16.7	20	2
`B-tree`	240	22.1	18	3
`Binary tree`	58	8.3	6	2
`Heap`	139	15.1	12	3
`RB-tree`	678	121.8	20	4

to be used not only at the module boundaries but also in any other assertion check outside m. Computing the shallow optimization can be performed per-library, without strictly requiring intermodular analysis. However, in some cases intermodular analysis may improve the precision of escaping terms and allow more aggressive optimizations.

5 Experimental Results

We now study the effectiveness of the combination of term hiding and shallow checking in the reusable library context, i.e., in libraries that use (some) hidden terms in their data structures and offer an interface for clients to access/manipulate such terms. We study the four assertion checking modes of [3]: *Unsafe* (no assertions are checked), *Client-Safe* (checks are generated for the assertions at the client-library boundary), *Safe-RT* (checks are generated also for internal library assertions), and *Safe-CT+RT* (like *RT*, but analysis information is used to clear as many checks as possible at compile-time). We use the lightweight instrumentation scheme from [10] for generating the run-time checks from the assertions. For eliminating the run-time checks via static analysis we reuse the Ciao verification framework extension from [3]. We concentrate in this work on shape analysis (regular types).

In our experiments each benchmark is composed of a library and a client/driver. We have selected a set of Prolog libraries that implement tree-based data structures. Libraries `B-tree` and `binary tree` were taken from the Ciao sources; libraries `AVL-tree`, `RB-tree`, and `heap` were adapted from YAP, adding similar assertions to those of the Ciao libraries. Table 1 shows some statistics for these libraries: number of lines of code (LOC), size of the object file (Size KB), the number of assertions in the library specification considered (Assertions), and the number of hidden functors per library (# Hidden). In order to focus on the assertions of the library operations used in the benchmarks (where by an operation we mean the set of predicates implementing it) we do not count in the tables the assertions for library predicates not directly involved in

Table 2: Static analysis and checking time for benchmarks for the *Safe-CT+RT* mode.

Benchmark	Analysis time, ms					Assertions	
	prep	`shfr`	prep	`eterms`	total	checking, ms	unchecked
`AVL-tree`	2	10	2	31	45 (2%)	59 (2%)	2/20
`B-tree`	3	9	3	38	53 (2%)	90 (3%)	3/18
`Binary tree`	1	9	1	14	25 (2%)	33 (2%)	2/6
`Heap`	2	7	2	24	35 (2%)	71 (4%)	2/12
`RB-tree`	13	11	14	35	73 (3%)	298 (10%)	3/20

those operations. Library assertions contain unmoded regular types (see the appendices for a simple example). For each library we have created two drivers (clients) resulting in two benchmarks per library:

- A benchmark that has constant ($O(1)$) time complexity for the library operation and $O(N)$ time complexity for the respective run-time check (e.g., looking up the value stored at the root of a binary tree and checking on each lookup that the input term is a binary tree).[8] Here a significant speedup can be expected when using *shallow* run-time checks, since the checking time dominates execution time and the reduction due to shallow checking should be more noticeable.

- A benchmark that has non-constant ($O(log(N))$) complexity of the library operation and $O(N)$ complexity of the respective run-time check (e.g., inserting an element in a binary tree and checking on each insertion that the input term is a tree. Here a smaller speedup is expected when using *shallow* run-time checking.

All experiments were run on a MacBook Pro with 2,6 GHz Intel Core i5 processor, 8GB RAM, and under the Mac OS X 10.12.3 operating system.

Static Analysis. Table 2 presents the detailed compile-time analysis and checking times for the *Safe-CT+RT* mode. Numbers in parentheses indicate the percentage of the total compilation time spent on analysis, which stays reasonably low even in the most complicated case (13% for the `RB-tree` library). Nevertheless, the analysis was able to discharge most of the assertions in our benchmarks, leaving always only 2-3 assertions unchecked (i.e., that will need run-time checks), for the predicates of the library operations being benchmarked.

[8] In the case of the `heap` library we used the `size/2` operation since heap size is stored in a term that wraps the root node of the heap tree.

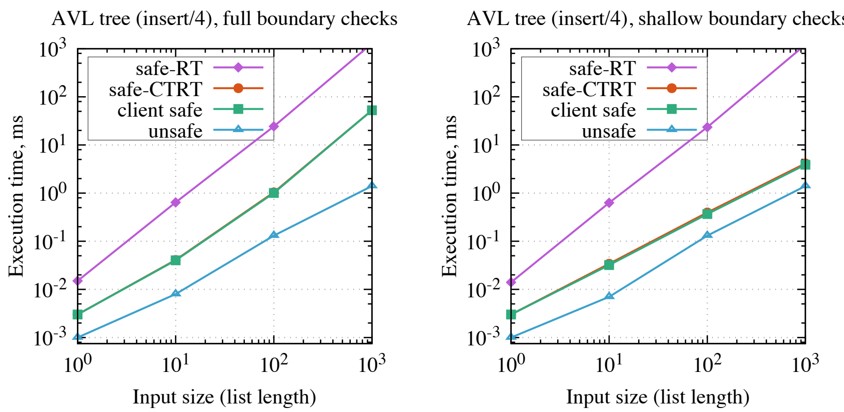

Figure 2: Run times in different checking modes, `AVL-tree` library, $O(log(N))$ operation.

Run-time Checking. After the static preprocessing phase we have divided our libraries into two groups:

- Libraries where the only unchecked assertions left are the ones for the boundary calls (`AVL-tree`, `heap`, and `binary tree`).[9]
- Libraries with also some unchecked assertions for internal calls (`B-tree` and `RB-tree`).

We present run time plots for one library of each group. Since the unchecked assertions in the second group correspond to internal calls of the $O(log(N))$ operation benchmark, we only show here a set of plots of the $O(1)$ operation benchmark for one library, as these plots are very similar across all benchmarks. The remaining plots can be found in C.

Fig. 2 illustrates the overhead reductions from using the shallow run-time checks in the `AVL-tree` benchmark for the $O(N)$ *insert* operation benchmark. This is also the best case that can be achieved for this kind of operations, since in the *Safe-CT+RT* mode all inner assertions are discharged statically. Fig. 3 shows the overhead reductions from using the shallow checks in the `B-tree` benchmark for the $O(N)$ *insert* operation benchmark. In contrast with the previous case, here the overhead reductions achieved by employing shallow checks are dominated by the total check cost. However, in both cases we can observe that the use of shallow checks in the *Client-Safe* checking mode results in *constant* run-time overhead, compared to the growing overhead in *Unsafe*

[9]Note that due to our reusable library scenario the analysis of the libraries is performed without any knowledge of the client and thus the library interface checks must always remain.

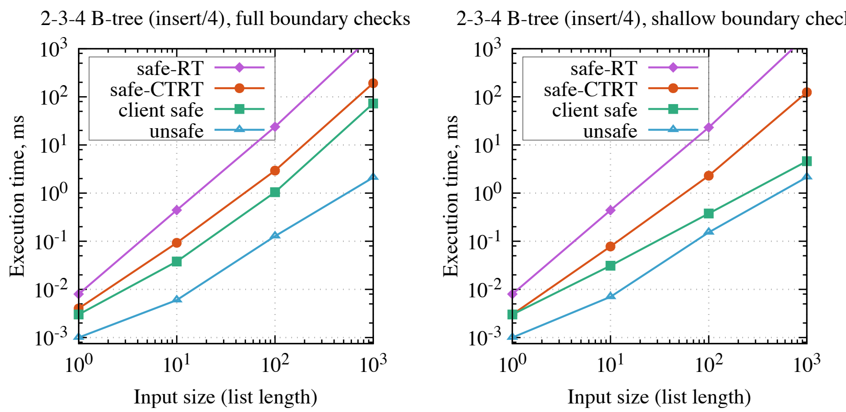

Figure 3: Run times in different checking modes, `B-tree` library, $O(log(N))$ operation.

(i.e., no checks) mode (Fig. 2 and Fig. 3, right). Fig. 4 presents the overhead reductions in run-time checking resulting from the use of the shallow checks in the `AVL-tree` benchmark for the $O(1)$ *peek* operation benchmark on the root. As we can see, using shallow checks allows us to obtain constant overhead on the boundary checks for such cheap operations in all execution modes but *Safe-RT*. In summary, the shallow checking technique seems quite effective in reducing the shape-related run-time checking overheads for the reusable-library scenario.

6 Related Work

Modularity. The topic of modules and logic programming has received considerable attention, dating back to [11, 12, 13] and resulting in standardization attempts for ISO-Prolog [14]. Currently, most mature Prolog implementations adopt some flavor of a module system, *predicate-based* in SWI [15], SICStus [16], YAP [17], ECLiPSe [18], and *atom-based* in XSB [4]. The Ciao approach [5], while theoretically compatible with that of XSB, has until now been closer to a predicate-based module system. Some previous research in the comparative advantages of atom-based module systems can be found in [19]. Our proposal can be seen as a way to achieve the benefits of an atom-based module system with a small effort in systems with predicate-based module systems.

Parallels with Static Typing. While traditionally Prolog is untyped, there have been some proposals for integrating it with type systems, starting with the work of [20]. Several typed Prolog-based systems have been proposed, notable examples being Mercury [21], Gödel [22], and Visual Prolog [23]. An

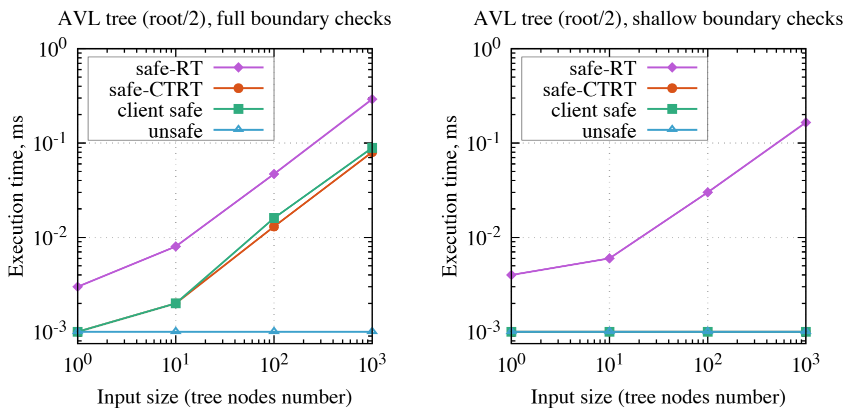

Figure 4: Run times in different checking modes, `AVL-tree` library, $O(1)$ operation.

approach for combining typed and untyped Prolog modules has been proposed in [24]. A conceptually similar approach in the world of functional programming is embodied in *gradual typing* [25, 26]. The Ciao model offers an (earlier) alternative (closer to *soft typing* [27]) based on abstract interpretation that is arguably more general and flexible than the above (assertions can contain any abstract property) –see [28] for a discussion of this topic.

Run-time Checking Optimization. Prohibitively high run-time overhead is common in systems that combine static and dynamic checking [26]. The impact of global static analysis in reducing in run-time checking overhead has been studied in [3]. A complementary approach is improving the instrumentation of the checks and combining it with run-time data caching [29, 10] or limiting the points at which the tests are performed [30]. While these optimizations can bring significant reductions in overhead, it still remains dependent on the size of the terms that are being checked. We have shown herein that even in the challenging context of calls across module boundaries it is sometimes possible to achieve constant run-time overhead.

7 Conclusions

We have described a lightweight modification of a predicate-based module system to support term hiding and explored the optimizations that can be achieved with this technique in the context of combined compile-time/run-time verification. We have studied the challenging case of reusable libraries, i.e., library modules that are pre-compiled independently of the client. We have shown that with our approach the shape information that can be inferred in such reusable modular programs can be enriched significantly and large reductions in overhead can be achieved, closer to those of static languages even in this reusable-library context. Our approach does not require switching to a strongly-typed language, which is not natural in languages like Prolog. An additional advantage of term hiding is that it is less intrusive than the alternative atom-based approach (e.g., it requires few changes in Prolog libraries) and thus we believe our approach can be easily incorporated into traditional, predicate-based Prolog systems.

References

[1] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Towards Run-time Checks Simplification via Term Hiding (extended abstract). In *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017)*, August 2017.

[2] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497.

[3] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Reducing the Overhead of Assertion Run-time Checks via static analysis. In *18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16)*, pages 90–103. ACM Press, September 2016.

[4] Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP*, 12(1-2):157–187, 2012.

[5] D. Cabeza and M. V. Hermenegildo. A New Module System for Prolog. In *International Conference CL 2000*, volume 1861 of *LNAI*, pages 131–148. Springer-Verlag, July 2000.

[6] M. V. 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.

[7] G. Puebla, F. Bueno, and M. V. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Proc. of LOPSTR'99*, LNCS 1817, pages 273–292. Springer-Verlag, March 2000.

[8] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Assertion-based Debugging of Higher-Order (C)LP Programs. In *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*. ACM Press, September 2014.

[9] G. Puebla, E. Albert, and M. V. Hermenegildo. Abstract Interpretation with Specialized Definitions. In *SAS'06*, number 4134 in LNCS, pages 107–126. Springer-Verlag, 2006.

[10] N. Stulova, J. F. Morales, and M. V. Hermenegildo. 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):726–741, September 2015.

[11] D.S. Warren and W. Chen. Formal semantics of a theory of modules. Technical report 87/11, SUNY at Stony Brook, 1987.

[12] W. Chen. A theory of modules based on second-order logic. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 24–33, San Francisco, 1987.

[13] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, pages 79–108, 1989.

[14] ISO. *PROLOG. ISO/IEC DIS 13211-2 — Part 2: Modules*, 2000.

[15] J. Wielemaker. *The SWI-Prolog User's Manual 5.9.9*, 2010.

[16] Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog User's Manual*, 4.1.1 edition, December 2009.

[17] Vítor Santos Costa, Luís Damas, and Ricardo Rocha. The YAP Prolog System. *Theory and Practice of Logic Programming*, 2011. http://arxiv.org/abs/1102.3896v1.

[18] Cisco Systems. *ECLIPSE User Manual*, 2006.

[19] Rémy Haemmerlé and François Fages. Modules for Prolog Revisited. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2006.

[20] A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[21] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, October 1996.

[22] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, 1994.

[23] Prolog Development Center. Visual Prolog.

[24] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. Towards Typed Prolog. In Enrico Pontelli and María M. García de la Banda, editors, *International Conference on Logic Programming*, number 5366 in LNCS, pages 693–697. Springer Verlag, December 2008.

[25] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *In Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[26] A. Takikawa, D. Feltey, B. Greenman, M.S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *POPL 2016*, pages 456–468, January 2016.

[27] Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI 1991)*, pages 278–292. SIGPLAN, ACM, 1991.

[28] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. The Ciao Approach to the Dynamic vs. Static Language Dilemma. In *Proc. Int'l. WS on Scripts to Programs, STOP'11*. ACM, 2011.

[29] E. Koukoutos and V. Kuncak. Checking Data Structure Properties Orders of Magnitude Faster. In *Runtime Verification*, volume 8734 of *LNCS*, pages 263–268. Springer, 2014.

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

A Main Proofs

A.1 Proof of Lemma 3

Proof. Let $\mathsf{esc}_m(X) \equiv \bigvee_i \bigvee_{V \in \mathit{Vars}_i}(X = V \wedge \theta_i)$ and $\mathsf{esc}'_m(X) \equiv \bigvee_i \bigvee_{V \in \mathit{Vars}'_i}(X = V \wedge \theta'_i)$. From the definitions, it can be seen that the set of all θ'_i (at the boundaries, before and after m) is a subset of all θ_i (outside m). The rest of the θ_i correspond to states not preceded by a literal from m. For such states $\bigvee_{V \in \mathit{Vars}_i}(X = V \wedge \theta_i)$ must be: 1) covered by $\mathsf{usr}(X)$ (and thus $\mathsf{esc}'_m(X)$); or 2) contain some $X = f(\ldots)$ with f hidden in m. Since f cannot appear in literals from $n \neq m$ then it must have come from some $\theta_b \wedge \theta_o$, where θ_b is some ancestor at the boundaries (already covered), θ_o is a conjunction of constraints introduced outside m (with cannot contain f), and thus it is more specific and also covered by $\mathsf{esc}'_m(X)$). □

A.2 Proof of Lemma 4

Proof. Let $Q(X) = \textsc{Escaping\_Terms}(m)$, we will show that $Q(X)$ over-approximates $\mathsf{esc}_m(X)$. Since $\mathsf{esc}_m(X)$ is equivalent to $\mathsf{esc}'_m(X)$ (Lemma 3), it is enough to consider the derivation steps at the boundaries. That is, $S_1 \leadsto S_2$ where $S_1 = \langle (L_1, m) :: \_ \mid \_ \rangle$ and $S_2 = \langle (L_2, n) :: \_ \mid \theta \rangle$ with $n \neq m$. If L_1 is a literal (not $\mathsf{ret}(\_)$) then it corresponds to the case of calling an imported predicate. The operational semantics ensure that $\theta \Rightarrow_P \mathit{Pre}$ and thus $Q(X)$ over-approximates this case. If L_2 is $\mathsf{ret}(\_)$ then it corresponds to the case of returning from m. The operational semantics ensure that $\theta \Rightarrow_P \mathit{Post}$ and thus $Q(X)$ also over-approximates this case. □

A.3 Proof of Theorem 5

Proof. By definition, the transformation only affects the checks for $\mathit{Pre} = (\bigwedge_i L_i(X_i))$ conjunctions in assertion conditions of exported predicates in m. These checks correspond to the derivation steps $S_1 \leadsto S_2$ where $S_1 = \langle (\_, n) :: G \mid \theta \rangle$ and $S_2 = \langle (\_, m) :: G \mid \_ \rangle$ with $n \neq m$. Let $Q(X)$ be obtained from $\textsc{Escaping\_Terms}(m)$. The shallow version $Pre^{\#} = (\bigwedge_i L_i(X_i))^{\#} = (\bigwedge_i \textsc{Spec}(L_i(X_i), Q(X_i)))$ (Definition 8). By Definition 7 it holds that $\theta \Rightarrow_P (\bigwedge_i \mathsf{esc}_m(X_i))$. By Lemma 4 it holds that $\theta \Rightarrow_P (\bigwedge_i Q(X_i))$. By correctness of $\textsc{Spec}$, since θ entails each $Q(X_i)$, then the full and specialized versions of L_i can be interchanged. $\square$

B Example: Computation of Escaping Terms and Shallow Checks (Code from the `binary tree` Library)

The code excerpt below below contains the declarations for hiding locally the `binary tree` library functors, the exported `insert/3` predicate, its assertion, and the definitions of the regular types used in this assertion:

```
:- hide(empty/0).
:- hide(tree/3).

:- regtype val_key/1.
val_key(X) :- int(X).

:- regtype val_tree/1.
val_tree(empty).
val_tree(tree(LC,X,RC)) :- val_tree(LC), val_key(X), val_tree(RC).

:- pred insert(K,T0,T1)  : val_key(K), val_tree(T0), term(T1)
                        => val_key(K), val_tree(T0), val_tree(T1).

insert(X,empty,tree(empty,X,empty)).
insert(X,tree(LC,X,RC),tree(LC,X,RC)).
insert(X,tree(LC,Y,RC),tree(LC_p,Y,RC)) :- X < Y,insert(X,LC,LC_p).
insert(X,tree(LC,Y,RC),tree(LC,Y,RC_p)) :- X > Y,insert(X,RC,RC_p).
```

The assertion conditions for the `insert/3` predicate are:

$$c_0.\mathsf{calls}(insert(K,T0,T1), val\_key(K), val\_tree(T0), term(T1))$$
$$c_1.\mathsf{success}(insert(K,T0,T1), (val\_key(K), val\_tree(T0), term(T1)), (val\_key(K), val\_tree(T0), val\_tree(T1)))$$

Denoting the module that contains the library source code as bt, the set of *escaping terms* computed by Algorithm 1 can be represented as the following regular type:

```
esc_bt(bt:empty).
esc_bt(bt:tree(L,K,R)) :- val_tree(L), val_key(K), val_tree(R).
esc_bt(X) :- usr(X).
```

where $usr(\_)$ is a property that denotes user terms. The explicit module qualification bt: is used only to clarify that `empty/0` and `tree/3` are local to module bt and not user functors. The resulting *shallow interface* produced by Algorithm 2 is:

```
:- regtype val_key/1.
val_key(X) :- int(X).

:- regtype val_tree/1.
val_tree(empty).
val_tree(tree(LC,X,RC)) :- val_tree(LC), val_key(X), val_tree(RC).

:- pred insert(K,T0,T1)  : val_key(K), val_tree#(T0), term(T1).
                        => val_key(K), val_tree(T0),  val_tree(T1).
insert(K,T0,T1) :- insert'(K,T0,T1).

:- pred insert'(K,T0,T1)  : val_key(K), val_tree(T0), term(T1)
                         => val_key(K), val_tree(T0), val_tree(T1).
... clauses of insert'/3 ...
```

where the `val_tree`$^{\#}$ property can be materialized as:

```
val_tree#(empty).
val_tree#(tree(_,_,_)).
```

The run-time checks instrumentation then can use the shallow `val_tree`$^{\#}$`/1` property in the checks for the calls across module boundaries and the original `val_tree/1` property for the calls inside the *bt* module.

C Additional Plots

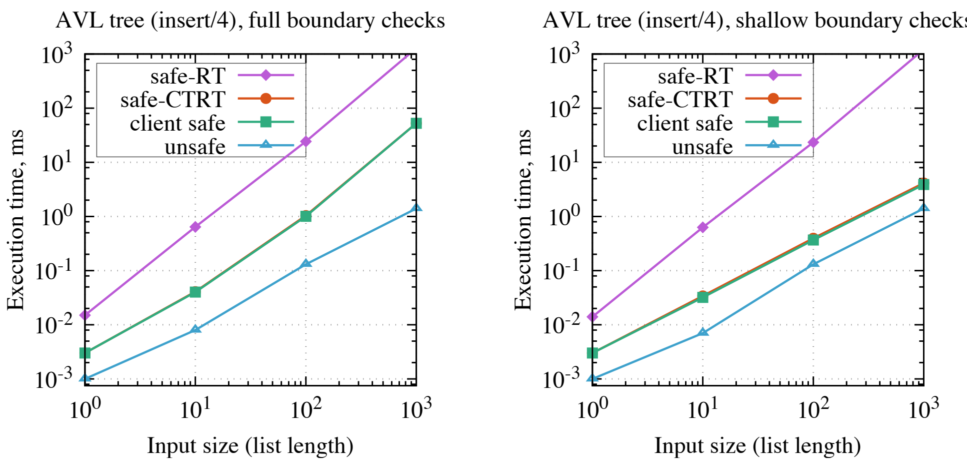

Figure 5: Run times for the `AVL-tree` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

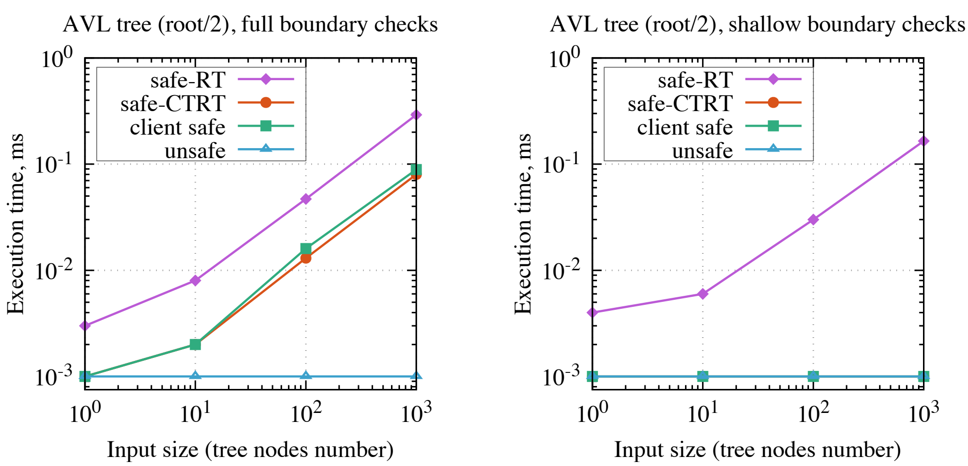

Figure 6: Run times for the `AVL-tree` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity

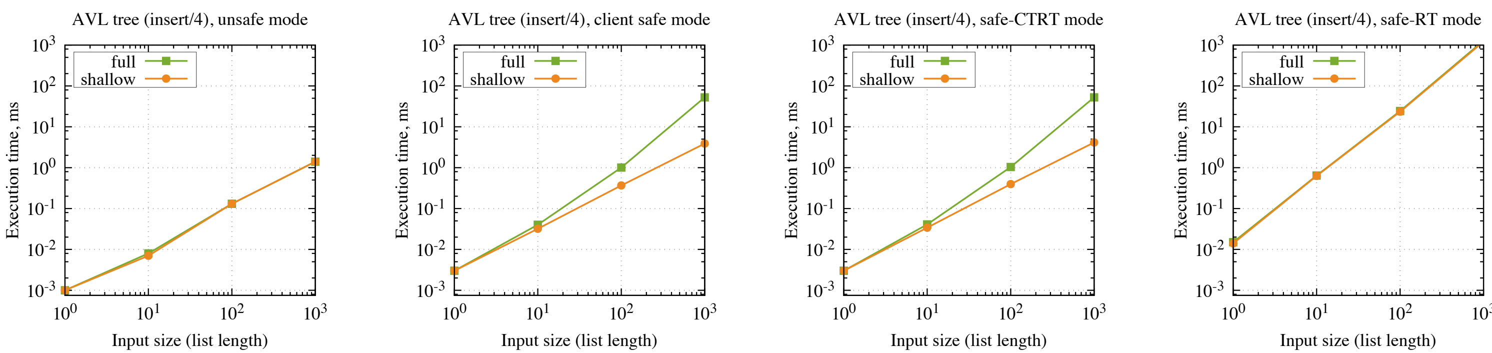

Figure 7: Run times for the `AVL-tree` benchmark per execution mode, $O(log(N))$ operation + $O(N)$ check complexity

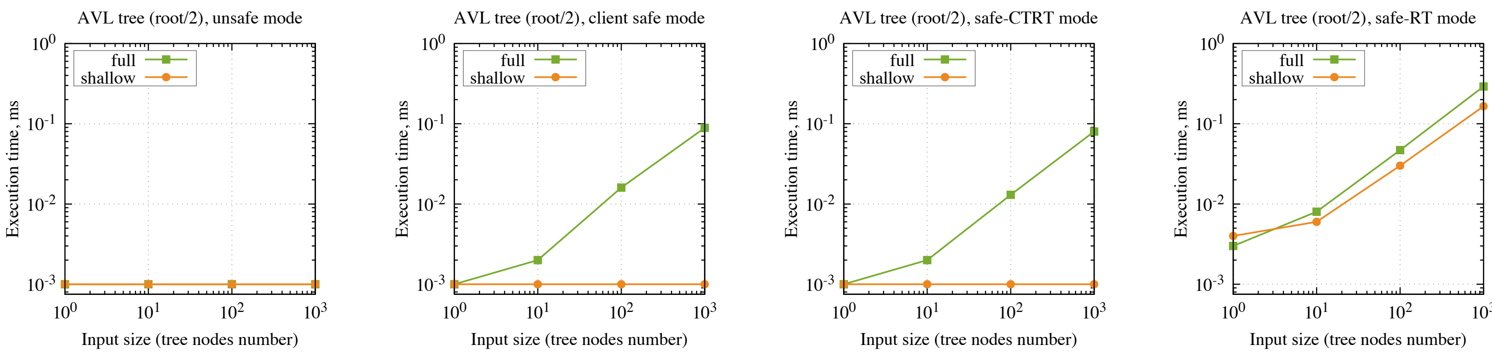

Figure 8: Run times for the `AVL-tree` benchmark per execution mode, $O(1)$ operation + $O(N)$ check complexity

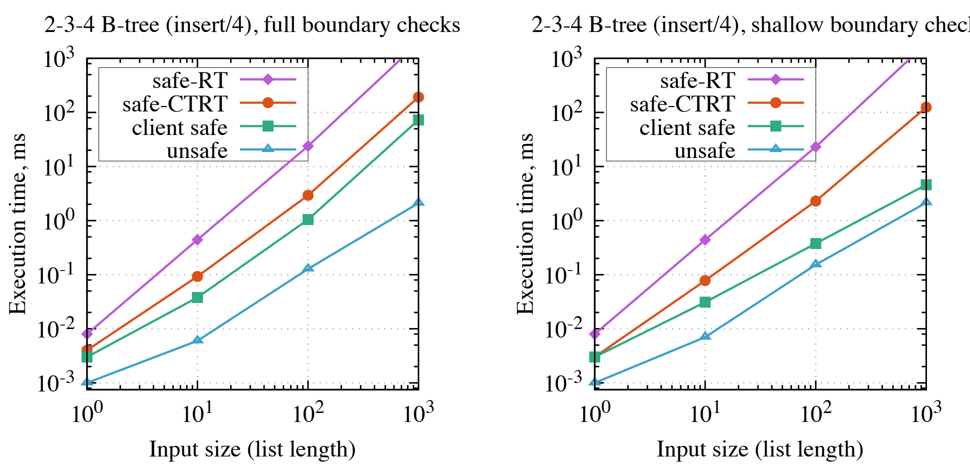

Figure 9: Run times for the `2-3-4 B-tree` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

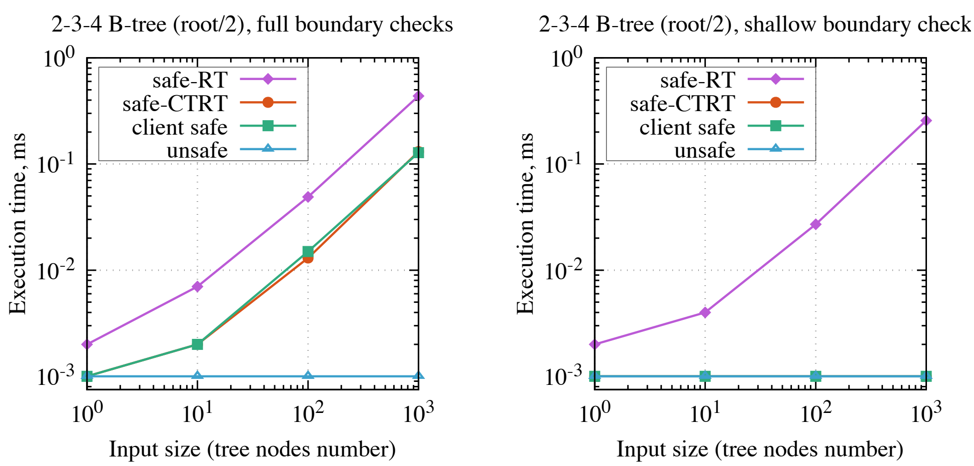

Figure 10: Run times for the `2-3-4 B-tree` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity

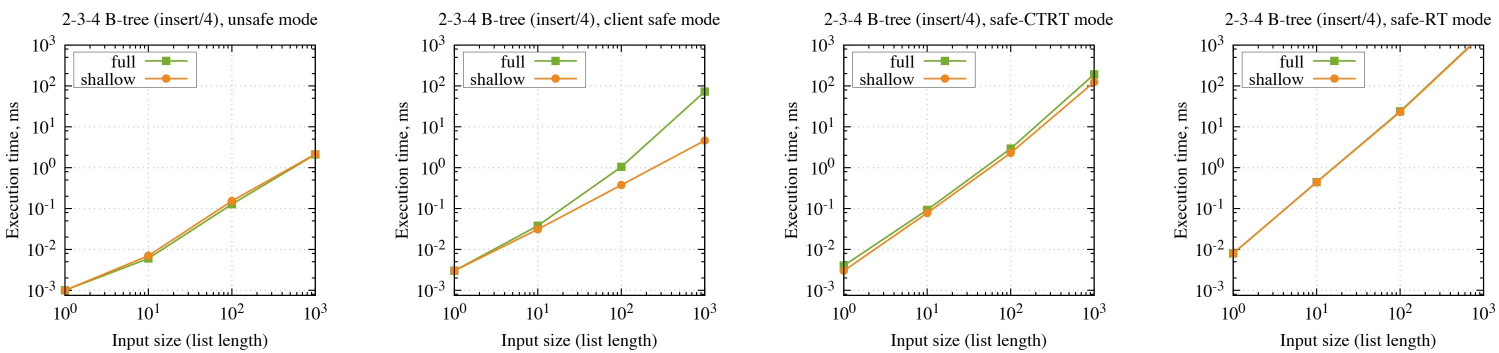

Figure 11: Run times for the `2-3-4 B-tree` benchmark per execution mode, $O(log(N))$ operation + $O(N)$ check complexity

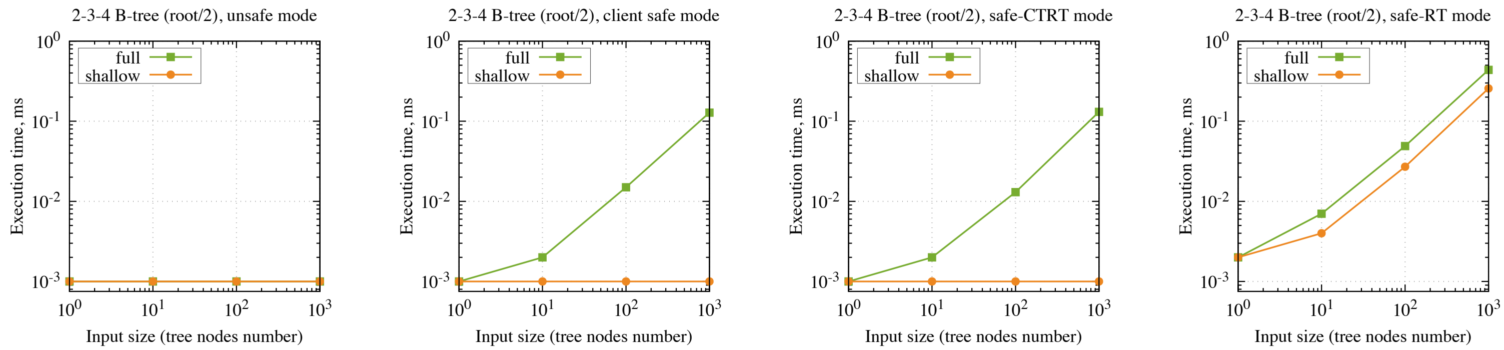

Figure 12: Run times for the `2-3-4 B-tree` benchmark per execution mode, $O(1)$ operation + $O(N)$ check complexity

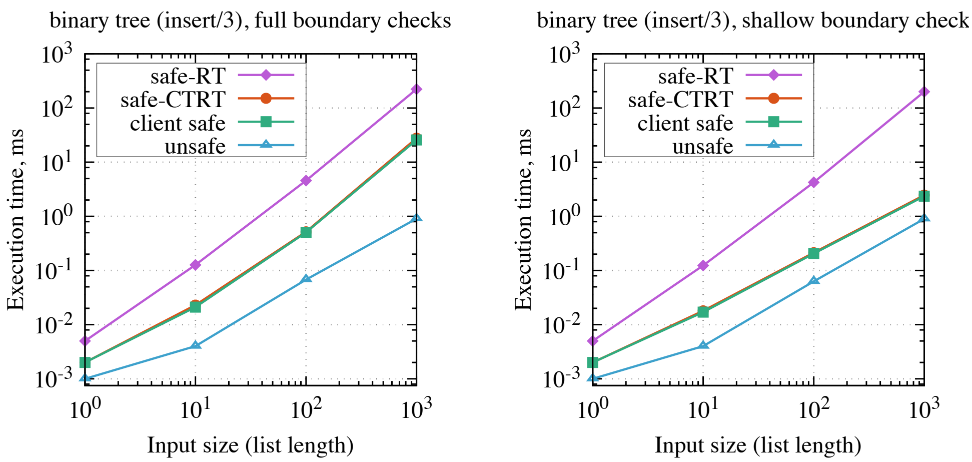

Figure 13: Run times for the `binary tree` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

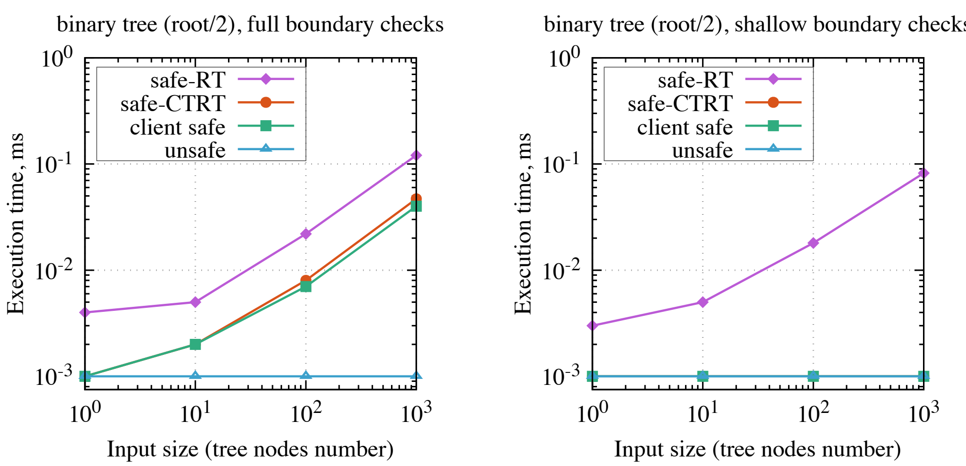

Figure 14: Run times for the `binary tree` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity

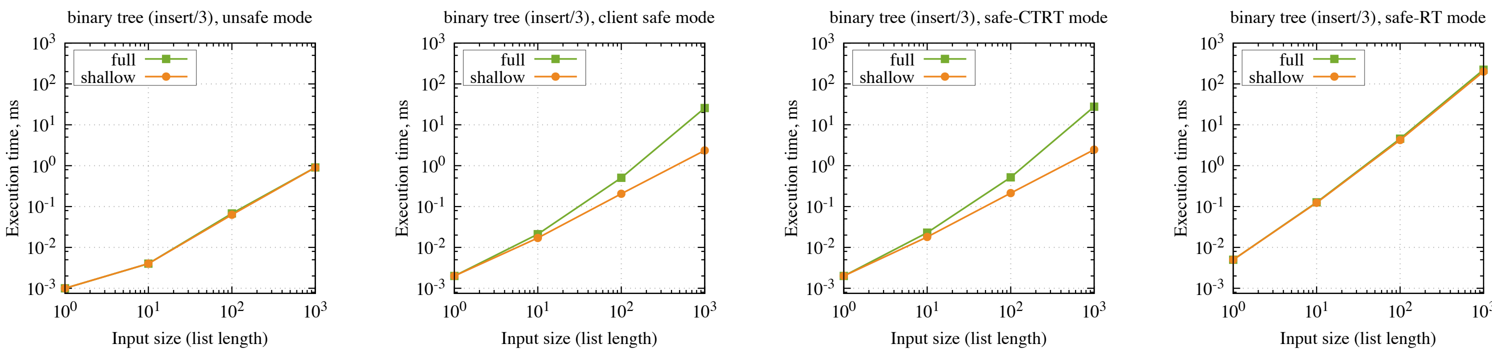

Figure 15: Run times for the `binary tree` benchmark per execution mode, $O(log(N))$ operation + $O(N)$ check complexity

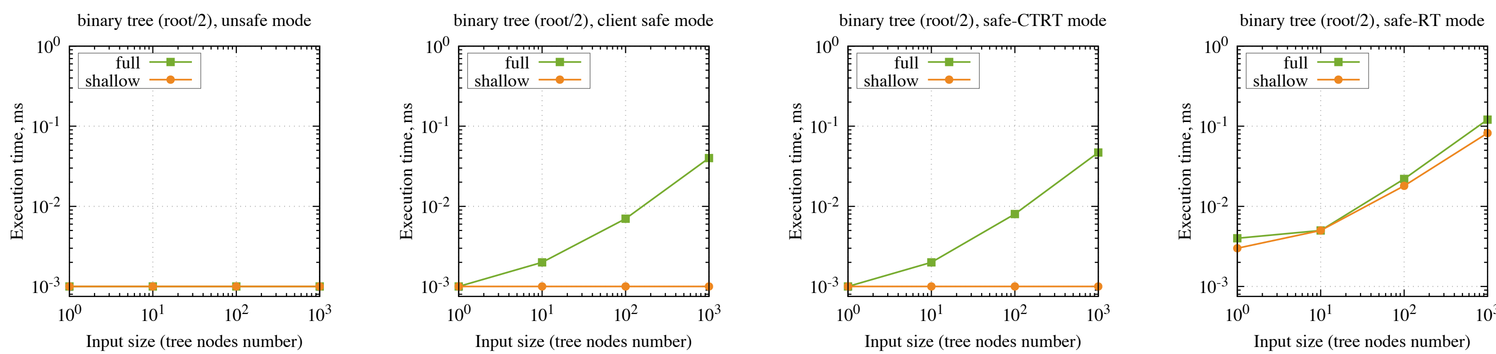

Figure 16: Run times for the `binary tree` benchmark per execution mode, $O(1)$ operation + $O(N)$ check complexity

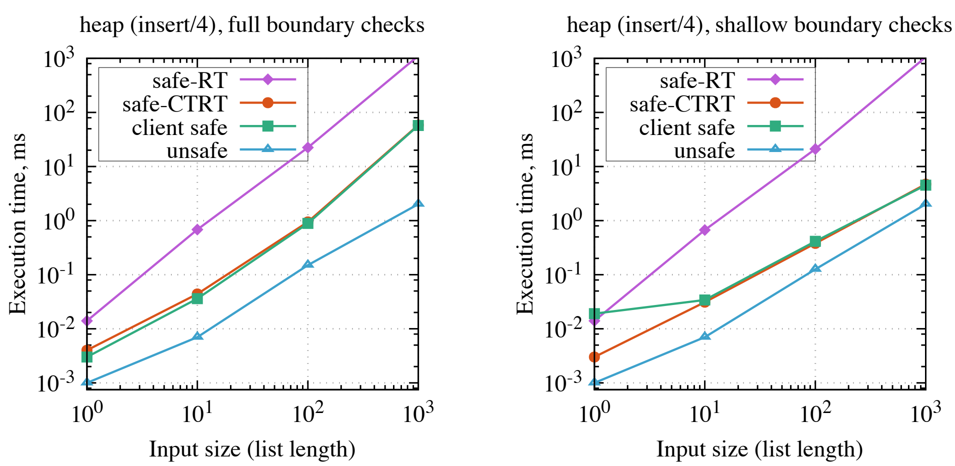

Figure 17: Run times for the `min-heap` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

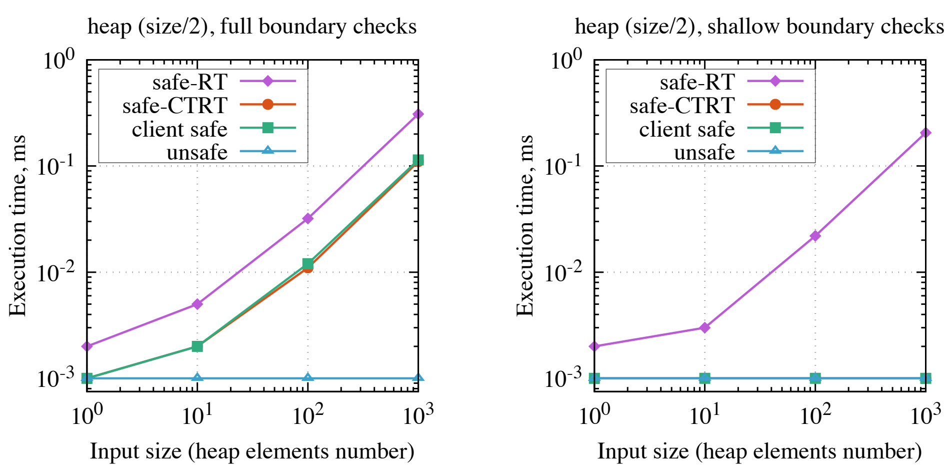

Figure 18: Run times for the `min-heap` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity

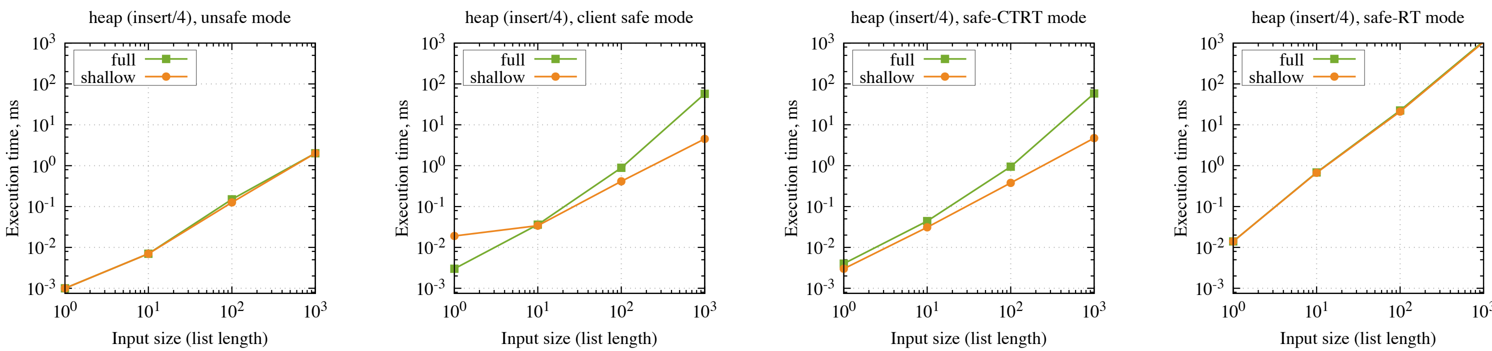

Figure 19: Run times for the `min-heap` benchmark per execution mode, $O(log(N))$ operation + $O(N)$ check complexity

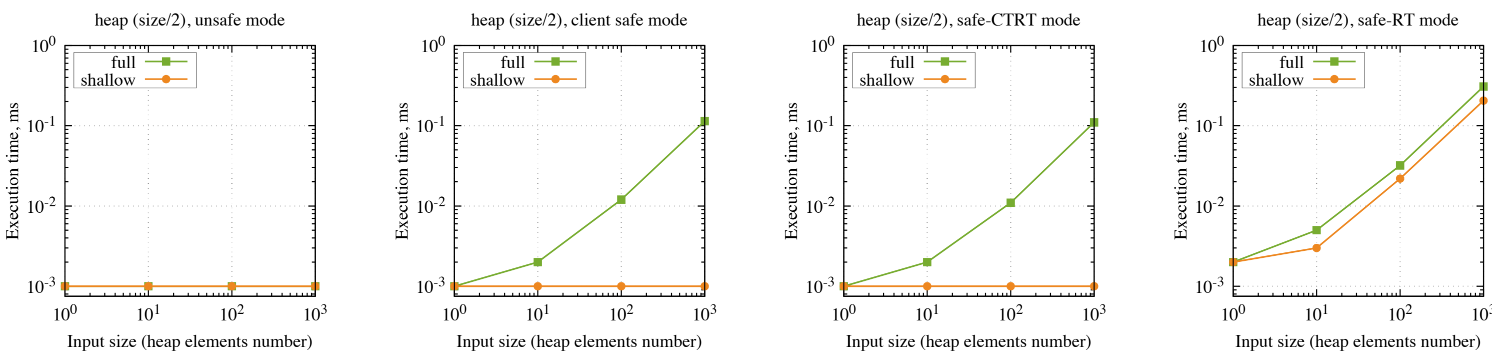

Figure 20: Run times for the `min-heap` benchmark per execution mode, $O(1)$ operation + $O(N)$ check complexity

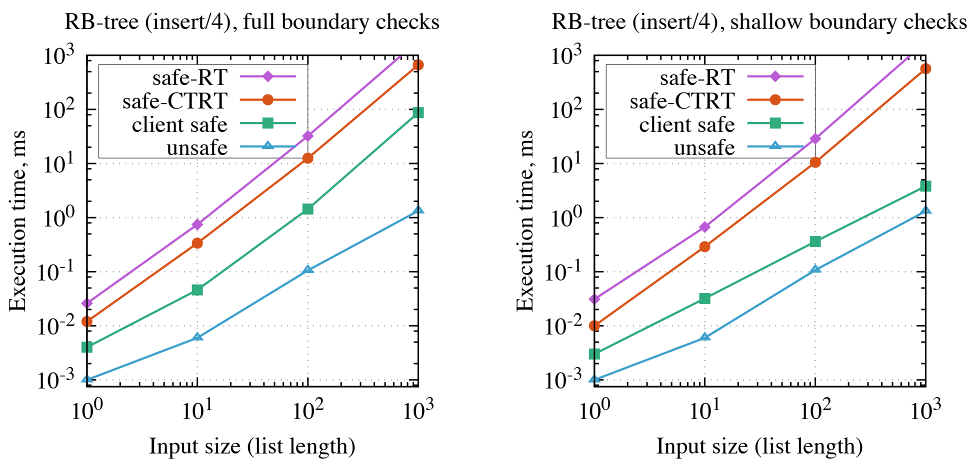

Figure 21: Run times for the `RB-tree` benchmark in different execution modes, $O(log(N))$ operation + $O(N)$ check complexity

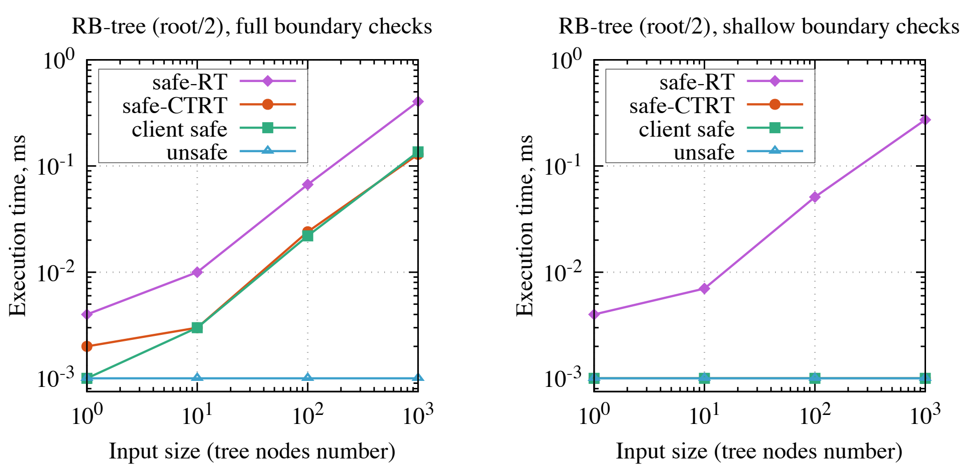

Figure 22: Run times for the `RB-tree` benchmark in different execution modes, $O(1)$ operation + $O(N)$ check complexity

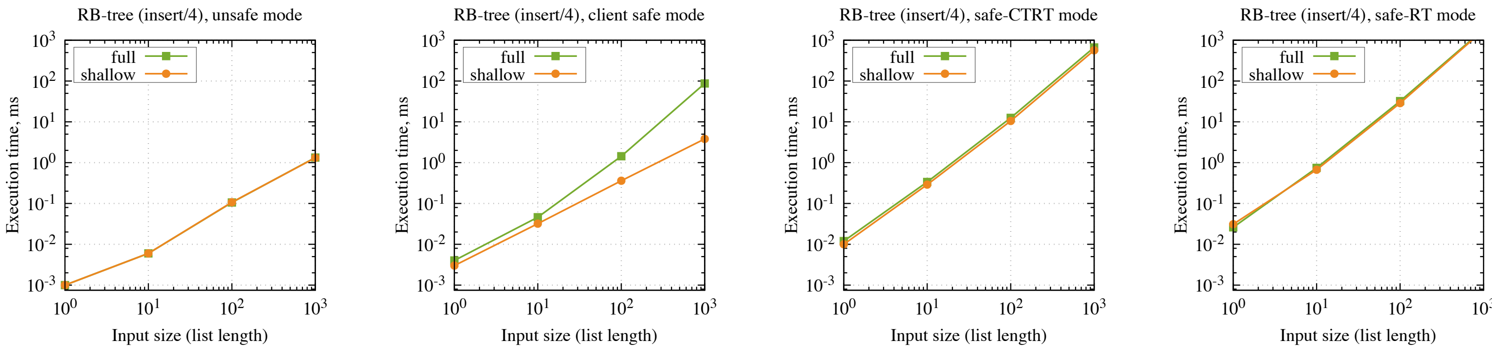

Figure 23: Run times for the `RB-tree` benchmark per execution mode, $O(log(N))$ operation + $O(N)$ check complexity

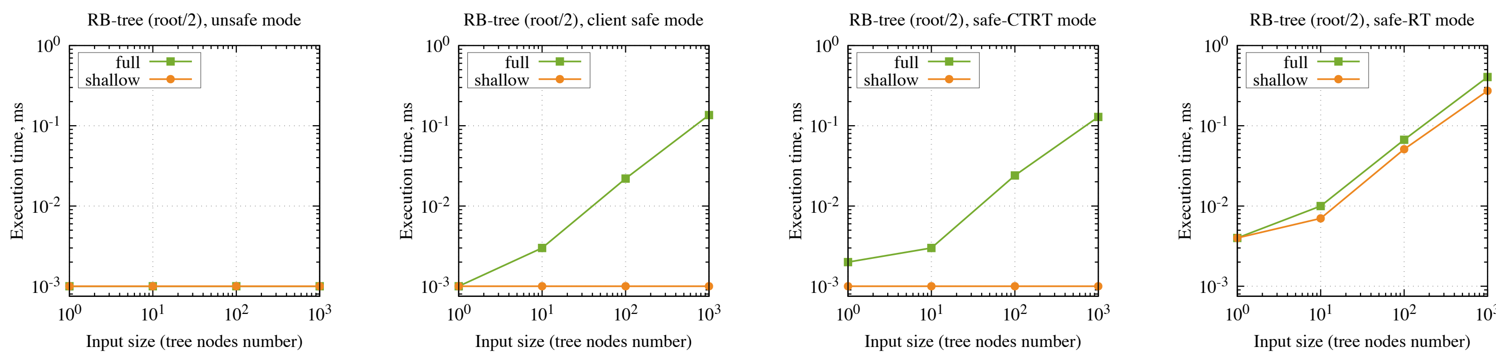

Figure 24: Run times for the `RB-tree` benchmark per execution mode, $O(1)$ operation + $O(N)$ check complexity