Exploiting Term Hiding to Reduce Run-time Checking Overhead ⋆

Nataliia Stulova✉[1,2][0000−0002−6804−2253], José F. Morales[1][0000−0001−9782−8135], and Manuel V. Hermenegildo[1,2][0000−0002−7583−323X]

[1] IMDEA Software Institute, Madrid, Spain
[2] ETSI Informáticos, Universidad Politécnica de Madrid (UPM), Madrid, Spain
`{nataliia.stulova,josef.morales,manuel.hermenegildo}@imdea.org`

Abstract. One of the most attractive features of untyped languages is the flexibility in term creation and manipulation. However, with such power comes the responsibility of ensuring the correctness of these 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 static analysis can greatly reduce such overheads, 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 technique which takes advantage of module systems which can hide a selected set of functor symbols to significantly enrich the shape information that can be inferred for reusable libraries, as well as an improved run-time checking approach that leverages the proposed mechanisms to achieve 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 the full expressiveness of the assertion language in this context. In contrast to other approaches it does not introduce the need to switch the language to a (static) type system, 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: Logic Programming; Module Systems; Practicality of 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

⋆ Research partially funded by Spanish MINECO TIN2015-67522-C3-1-R grant *TRACES* and the Madrid M141047003 *N-GREENS* program. We also thank Dylan McDermott, Alan Mycroft, and the anonymous reviewers for their comments. An extended abstract on this work was published as [1].

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. Different programming languages implement these concepts in different ways, some examples being the encapsulation mechanism of classes in object-oriented programming and opaque data types. In the (constraint) logic programming context, most mature language implementations incorporate module systems, some of which allow programmers to restrict the visibility of some functor symbols to the module where they are defined, thus both hiding the concrete implementation details of terms from other modules and providing guarantees that only the predicates of that particular module can use those functor symbols as term constructors or matchers.

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 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, there are some common scenarios where shape/type analyses are necessarily imprecise. A motivational example is the case of reusable libraries, 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. Static analysis faces challenges in this context, since 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 techniques that, by exploiting term hiding and the strict visibility rules of the module system, can greatly improve the quality of the shape information inferred by static analysis and reduce the run-time overhead for the calls across module boundaries by several orders of magnitude. These techniques can result in improvements in the number and size of checks that allow bringing guarantees and overheads to levels close to those of statically-typed approaches, but without imposing on programs the restriction of being well-typed. For concreteness, we use in this work the relevant parts of the Ciao system [2], which pioneered the assertion-based, combined static+dynamic checking approach: the module system, the assertion language –which allows providing optional program specifications with various kinds of information, such as modes, shapes/types, non-determinism, etc.–, and the overall framework. However, our results are general and we believe they can be applied to many dynamic languages. In particular, we present a semantics for modular logic programs where the mapping of module symbols is abstract and implementation-agnostic, i.e., independent of the visibility rules of particular module systems.

2 Preliminaries

We first recall 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 the t_i are *terms*. 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 the t_i are terms. Note that we do not (yet) distinguish between predicate symbols and functors (uninterpreted function symbols), denoting the global set of *symbols* as FS. 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. *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 (the head) and B is a goal (the body). A *definite program* is a finite set of clauses. The *definition* of an atom A in a program, $\mathsf{cls}(A)$, is the set of program clauses whose head has the same predicate symbol and arity as A, renamed-apart. We assume that all clause heads are *normalized*, i.e., H is of the form $f(X_1, \ldots, X_n)$ where the $X_1, \ldots, X_n$ are distinct free variables.

We recall the classic operational semantics of (non-modular) definite programs, given in terms of program *derivations*, which are sequences of *reductions* between *states*. 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 $\langle L \mid \theta \rangle$. The set of all derivations from the query Q is denoted $\mathsf{derivs}(Q)$. A finite derivation from a query (L, θ) is *finished* if the last state in the derivation cannot be reduced, and it is *successful* if the last state is of the form $\langle \Box \mid \theta' \rangle$, where $\Box$ denotes the empty goal sequence. In that case, the constraint $\bar{\exists}_L \theta'$ (denoting the projection of θ onto the variables of L) is an *answer* to (L, θ). Else, the derivation is *failed*. We denote by $\mathsf{answers}(Q)$ the set of answers to a query Q.

3 An Abstract Approach to Modular Logic Programs

There have been several proposals to date for supporting modularity in logic programs, all of which are based on performing a partition of the set of program symbols into modules. As mentioned before, the two most widely adopted approaches are referred to as *predicate-based* and *atom-based* module systems. In predicate-based module systems all symbols involved in terms are global, i.e., they belong to a single global `user` module –a special module from which all modules import the symbols and to which all modules can add symbols. In atom-based module systems [4] only constants and explicitly exported symbols are global, while the rest of the symbols are local to their modules. Ciao [5] adopts a hybrid approach which is as in predicate-based systems but with the possibility of marking a selected set of symbols as local (we will use this model in the examples in Sec. 5). Despite the differences among these module systems, by performing module resolution applying the appropriate visibility rules, programs are reducible in all systems to a form that can be interpreted using the same Prolog-style semantics. We will use this property in order to abstract our results away from particular module systems and their symbol visibility rules. To this end we present a formalization of the "flattened" version of a modular program, where visibility is explicit and is thus independent of the visibility conventions of specific module systems. Let MS denote the set of all *module symbols*. The *flattened* form of a modular definite program is defined as follows:

Definition 1 (Modular Program). *A* modular program *is a pair* $(P, \mathsf{mod}(\cdot))$*, where* P *is a definite program and* $\mathsf{mod}(\cdot)$ *is a mapping that assigns for each symbol* $f \in \mathit{FS}$ *a unique module symbol* $m \in \mathit{MS}$*. Let* C *be a clause* $H \leftarrow B$ *in* P*,* $\mathsf{mod}(C) \triangleq \mathsf{mod}(H)$*. Let* A *be an atom*[3] *or a term of the form* $f(\ldots)$*. Then* $\mathsf{mod}(A) \triangleq \mathsf{mod}(f)$*.*

The $\mathsf{mod}(\cdot)$ mapping creates a partition of the clauses in the definite program P. We refer to each resulting equivalence class as a module, and represent it with the module symbol shared by all clauses in that class. The set of all symbols defined by a module m is $\mathsf{def}(m) = \{f | f \in \mathsf{FS}, \mathsf{mod}(f) = m, m \in \mathsf{MS}\}$.

Definition 2 (Interface of a Module). *The* interface *of a module* m *is given by the disjoint sets* $\mathsf{exp}(m)$ *and* $\mathsf{imp}(m)$*, s.t.* $\mathsf{exp}(m) \subseteq \mathsf{def}(m)$ *is the subset of the symbols defined in* m *that can appear in other modules, referred to as the* export list *of* m*, and* $\mathsf{imp}(m) = \{f | f \in \mathit{FS}, f \text{ is in symbols of } \mathsf{cls}(p), p \in \mathsf{def}(m)\} \setminus \mathsf{def}(m)$ *is a superset of symbols in the bodies of the predicates of* m*, that are not defined in* m*, referred to as the* import list *of* m*.*

To track calls across module boundaries we 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, as given in the following definition:

Definition 3 (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, as shown below. Then, a state* $S = \langle (L, m) :: G \mid \theta \rangle$ *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. $\langle (L, m) :: G \mid \theta \rangle \rightsquigarrow \langle (B_1, n) :: \ldots :: (B_k, n) :: (\mathsf{ret}(L), n) :: G \mid \theta \rangle$ *if* L *is an atom and* $\exists (L \leftarrow B_1, \ldots, B_k) \in \mathsf{cls}(L)$ *where* $\mathsf{mod}(L) = n$ *and it holds that* $(L \in \mathsf{def}(n) \wedge n = m) \bigvee (L \in \mathsf{exp}(n) \wedge L \in \mathsf{imp}(m) \wedge n \neq m)$*.*
3. $\langle (L, m) :: G \mid \theta \rangle \rightsquigarrow \langle G \mid \theta \rangle$ *if* L *is a clause return literal* $\mathsf{ret}(\_)$*.*

Basically, 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] 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.

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

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

Fig. 1. Correspondence between assertions and assertion conditions.

4 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]. Such `pred` assertions define the set of all admissible preconditions for a given predicate, and for each such pre-condition, a corresponding post-condition. These pre- and post-conditions are formulas containing literals defined by 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 these literals corresponding to properties as *prop* literals. The predicate symbols of *prop* literals are module-qualified in the same way as those of the other program literals.

Example 1 (Property). The following property describes a sorted list:
```
sorted([]).  sorted([_]).  sorted([X,Y|L]) :- X =< Y, sorted([Y|L]).
```
i.e., $[\![sorted(A)]\!] = \{A = [], A = [B], A = [B, C|D] \wedge B \leq C \wedge E = [C|D] \wedge sorted(E)\}$.

The left part of Fig. 1 shows a set of assertions for a predicate (identified by a normalized atom $Head$). The Pre_i and $Post_i$ fields are conjunctions[4] of *prop* literals that refer to the variables of $Head$. Informally, such a set of assertions states that in any execution state $\langle (Head, m) :: G \mid \theta \rangle$ 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 the predicate succeeds, the corresponding $Post_i$ should hold upon success. We denote the set of assertions for a predicate represented by $Head$ by $\mathcal{A}(Head)$, and the set of all assertions in a program by $\mathcal{A}$.

In our formalization, rather than using the assertions for a predicate directly, we use instead a normalized form which we refer to as the set of *assertion conditions* for that predicate, denoted as $\mathcal{A}_C(Head) = \{C_0, C_1, \ldots, C_n\}$, as shown in Fig. 1, right. The c_i are identifiers which are unique for each assertion condition. The $\mathsf{calls}(Head, \ldots)$ conditions encode the check that ensures that the calls to the predicate represented by the $Head$ literal are within those admissible by the set of assertions, and we thus call them the *calls assertion conditions*. The $\mathsf{success}(Head, Pre_i, Post_i)$ conditions encode the checks for compliance of the successes for particular sets of calls, and we thus call them the *success assertion*

[4] 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.

conditions. If there are no assertions associated with $Head$ then the corresponding set of assertion conditions is empty. The set of assertion conditions for a program, denoted $\mathcal{A}_C$ is the union of the assertion conditions for each of the predicates in the program, and is derived from the set $\mathcal{A}$ of all assertions in the program.

Semantics with Run-time Checking of 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 $\mathsf{err}(c)$ literal denotes a special goal that marks a derivation finished because of the violation of the assertion condition with identifier c. A finished derivation from a query (L, θ) is now *successful* if the last state is of the form $\langle \Box \mid \theta' \rangle$, *erroneous* if the last state is of the form $\langle \mathsf{err}(c) \mid \theta' \rangle$, or *failed* otherwise. The set of derivations for a program from its set of queries $\mathcal{Q}$ using the semantics with run-time checking of assertions is denoted by $\mathsf{rtc\text{-}derivs}(\mathcal{Q})$. 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 $\theta \models \theta'$. Intuitively, a literal L succeeds trivially if L succeeds for θ without adding new constraints to θ. This notion captures the checking of properties and we will thus often refer to this operation as "checking L in the context of θ." [5]

Definition 4 (Operational Semantics for Modular Programs with Run-time Checking). *A state $S = \langle (L, m) :: G \mid \theta \rangle$ can be* reduced *to a state S', denoted $S \rightsquigarrow_{\mathsf{rtc}} S'$, as follows:*

1. *If L is a constraint then $S' = \langle G \mid \theta \wedge L \rangle$ if $\theta \wedge L$ is satisfiable.*
2. *If L is an atom and $\exists (L \leftarrow B_1, \ldots, B_k) \in \mathsf{cls}(L)$, then the new state is*
$$S' = \begin{cases} \langle \mathsf{err}(c) \mid \theta \rangle & \textit{if } \exists\, c.\mathsf{calls}(L, Pre) \in \mathcal{A}_C(L) \ \wedge\ \theta \not\Rightarrow_P Pre \\ \langle (B_1, n) :: \ldots :: (B_k, n) :: (\mathsf{ret}(L, \mathcal{C}), n) :: G \mid \theta \rangle & \textit{otherwise} \end{cases}$$
s.t. $\mathcal{C} = \{c_i \mid c_i.\mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A}_C(L) \ \wedge\ \theta \Rightarrow_P Pre_i\}$ where $\mathsf{mod}(L) = n$ *and it holds that* $(L \in \mathsf{def}(n) \wedge n = m) \bigvee (L \in \mathsf{exp}(n) \wedge L \in \mathsf{imp}(m) \wedge n \neq m)$
3. *If L is a clause return literal $\mathsf{ret}(\_, \mathcal{C})$, then*
$$S' = \begin{cases} \langle \mathsf{err}(c) \mid \theta \rangle & \textit{if } \exists\, c \in \mathcal{C} \textit{ s.t. } c.\mathsf{success}(L', \_, Post) \in \mathcal{A}_C(L') \ \wedge\ \theta \not\Rightarrow_P Post \\ \langle G \mid \theta \rangle & \textit{otherwise} \end{cases}$$

Theorem 1 below on the correctness of the operational semantics with run-time checking can be straightforwardly adapted from [8].[6] The completeness of

[5] Note that even if several assertion conditions may be violated at the same time, we consider only the first one of them. The ordering is only imposed by the implementation and does not affect the semantics.

[6] The formal definition of the equivalence relation on derivations, as well as proofs for the theorems and lemmas can be found in Appendix A of the extended version of this paper available from CoRR at `https://arxiv.org/abs/1705.06662` (v3).

this operational semantics as presented in Theorem 2 below can only be proved for *partial* program derivations, as the new semantics introduces the $\mathsf{err}()$ literal that directly replaces the goal sequence of a state in which a violation of an assertion condition occurs.

Theorem 1 (Correctness Under Assertion Checking). *For any tuple* $(P, \mathcal{Q}, \mathcal{A})$ *it holds that* $\forall D' \in \mathsf{rtc\text{-}derivs}(\mathcal{Q})\ \exists D \in \mathsf{derivs}(\mathcal{Q})$ *s.t.* D' *is equivalent to* D *(including partial derivations).*

Theorem 2 (Partial Completeness Under Assertion Checking). *For any tuple* $(P, \mathcal{Q}, \mathcal{A})$ *it holds that* $\forall D = (S_1, \ldots, S_k, S_{k+1}, \ldots, S_n) \in \mathsf{derivs}(\mathcal{Q})\ \exists D' \in \mathsf{rtc\text{-}derivs}(\mathcal{Q})$ *s.t.* D' *is equivalent to* D *or* $(S_1, \ldots, S_k, \langle \mathsf{err}(c) \mid \_ \rangle)$.

5 Shallow Run-Time Checking

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.[7] Our purpose herein is to study how in systems where the visibility of function symbols can be controlled, similar reasoning can be performed at the level of terms, and in particular how such reasoning can be applied to reducing the overhead of run-time checks. We will refer to these reduced checks as *shallow* run-time checks, which we will formally define later in this section. We start by recalling how in cases where the visibility of terms function symbols can be controlled, this reasoning is impossible without global (inter-modular) program analysis, using the following example:

Example 1. Consider a module $m1$ that exports a single predicate `p/1` that constructs `point/1` terms:

```
:- module(m1, [p/1, r/0]).     % m1 declared, p/1 and r/0 are exported
p(A) :- A = point(B), B = 1.   % A = user:point(1)
:- use_module(m2,[q/1]).       % import q/1 from a module m2
r :- X = point(2), q(X).       % X = user:point(2)
```

Here, we want to reason about the terms that can appear during program execution at several specific program points: (a) before we call `p/1` (point at which execution enters module `m1`); (b) when the call to `p/1` succeeds (point at which execution leaves the module); and (c) before we call `q/1` (point at which execution enters another module). When we exit the module at points (b) and (c) we know that in any `point(X)` constructed in $m1$ either $X = 1$ or $X = 2$. However, when we enter module $m1$ at point (a) `A` could have been bound by the calling module to any term including, e.g., `point([4,2])`, `point(2)`, `point(a)`, `point(1)`, etc., since the use of the `point/1` functor is not restricted.

Now we will consider the case where the visibility of terms can be controlled. We start by defining the following notion:

[7] In practice, an exception is `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.

Definition 5 (Hidden Functors of a Module). *The set of hidden functors of a module is the set of functors that appear in the module that are local and non-exported.*

Example 2. In this example we mark instead the `point/1` symbol as hidden. We use Ciao module system notation [5], where all function symbols belong to `user`, unless marked with a `:- hide f/N` declaration. Such symbols are hidden, i.e., local and not exported.[8]

```
:- module(m1, [p/1, r/0]).
:- hide point/1.              % point/0 is restricted to m1
p(A) :- A = point(B), B = 1.  % m1:point(1), not user:point(1)
:- use_module(m2,[q/1]).
r :- X = point(2), q(X).      % m1:point(2) escapes through call to q/1
```

Let us consider the same program points as in Example 1. When we exit the module, we can infer the same results, but with `m1:point/1` instead of `user:point/1`. Now, if we see the `m1:point(X)` term at point (a) we know that it has been constructed in $m1$, and the `X` has to be bound to either 1 or 2, because the code that can create bindings for X is only located in $m1$ (and the `point/1` terms are passed outside the module at points (b) and (c)).

As mentioned before, these considerations will allow us to use an optimized form of checking that we refer to as *shallow checking*. In order to formalize this notion, we start by defining all possible terms that may exist outside a module m as its *escaping terms*. We will also introduce the notion of *shallow properties* as the specialization of the definition of these properties w.r.t. these escaping terms, and we will present algorithms to compute such shallow versions of properties.

Definition 6 (Visible Terms at a State). *A property that represents all terms that are visible in a state $S = \langle (L, \_) :: G \mid \theta \rangle$ of some derivation $D \in \mathsf{rtc\text{-}derivs}(\mathcal{Q})$ for a tuple $(P, \mathcal{Q}, \mathcal{A})$ is $\mathsf{vis}_S(X) \equiv \bigvee_{V \in \mathsf{Vars}_L} (X = V \wedge \theta)$ where Vars_L denotes the set of variables of literal L.*

Definition 7 (Escaping terms). *Consider all states S in all derivations $D \in \mathsf{rtc\text{-}derivs}(\mathcal{Q})$ of any tuple $(P, \mathcal{Q}, \mathcal{A})$ where P imports a given module m. A property that represents escaping terms w.r.t. m is $\mathsf{esc}_m(X) \equiv \bigvee \mathsf{vis}_S(X)$ for each $S = \langle (\_, n) :: \_ \mid \_ \rangle$ with $n \neq m$.*

The set of all public symbols to which a variable X can be bound is denoted as $\mathsf{usr}(X) = \{X | \mathsf{mod}(X) = \mathtt{user}\}$. The following lemma states that it is enough to consider the states at the module boundaries to compute $\mathsf{esc}_m(X)$:

Lemma 1 (Escaping at the Boundaries). *Consider all derivation steps $S_1 \rightsquigarrow_{\mathsf{rtc}} S_2$ 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 *(i.e., the property with the smallest model) 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 $\mathsf{esc}_{m'}(X) \vee \mathsf{usr}(X)$ is equivalent to $\mathsf{esc}_m(X)$.*

Algorithm 1 ESCAPING_TERMS

```
1: function ESCAPING_TERMS($M$)
2:     $Def := \mathsf{usr}(X)$
3:     for all $L$ exported from $M$ do
4:         for all $c.\mathsf{success}(L, \_, Post) \in \mathcal{A}_C(L)$ do
5:             for all $P \in \textsc{LitNames}(Post, vars(L))$ do
6:                 $Def := Def \sqcup P(X)$
7:     for all $L$ imported from $M$ do
8:         for all $c.\mathsf{calls}(L, Pre) \in \mathcal{A}_C(L)$ do
9:             for all $P \in \textsc{LitNames}(Pre, vars(L))$ do
10:                $Def := Def \sqcup P(X)$
11:    return $(\mathsf{esc}_m(X) \leftarrow Def)$
12: function LITNAMES($G, Args$)
13:    return set of $P$ such that $A \in Args$ and $G = (\ldots \wedge P(A) \wedge \ldots)$
```

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 can sometimes 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 2 (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. While this notion 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, since this also provides a direct implementation path.

Example 3. 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

[8] Note that this can be achieved in other systems: e.g., in XSB [4] it can be done with a `:- local/1` declaration, combined with not exporting the symbol.

Algorithm 2 SHALLOW_INTERFACE

```
1: function SHALLOW_INTERFACE($M$)
2:     Let $M'$ be $M$ with wrappers for exported predicates
3:       (to differentiate internal from external calls)
4:     Let $Q(X) := $ ESCAPING_TERMS($M'$)
5:     for all $L$ exported from $M$ do
6:         for all $c.\mathsf{calls}(L, Pre) \in \mathcal{A}_C(L)$ do
7:             Update $\mathcal{A}_C(L)$ with $c.\mathsf{calls}(L, Pre^{\#})$
8:         for all $c.\mathsf{success}(L, Pre, Post) \in \mathcal{A}_C(L)$ do
9:             Update $\mathcal{A}_C(L)$ with $c.\mathsf{success}(L, Pre^{\#}, Post)$
10:    return $M'$
```

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. the 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 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 their 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 3 (Correctness of SHALLOW_INTERFACE). *Replacing a module m in a larger program by its shallow version does not alter the (run-time checking) 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 long 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 `user` (i.e., not hidden) symbols.[9] 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 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.

6 Experimental Results

We explore 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 library assertions are checked), *Client-Safe* (checks are generated only for the assertions of the predicates exported by the library, assertions for the internal library predicates are not checked), *Safe-RT* (checks are generated from assertions both for internal and exported library predicates), 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 program assertions. For eliminating the run-time checks via static analysis we reuse the Ciao verification framework, including the extensions from [3]. We concentrate in these experiments 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 (Pred Assertions), and the number of hidden functors per library (# Hidden Symbols).

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 those operations. Library assertions contain instantiation (moded)

[9] 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)	Pred Assertions	# Hidden Symbols
`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

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

regular types.[10] For each library we have created two drivers (clients) resulting in two experiments per library. In the first one the library operation has constant ($O(1)$) time complexity and the respective run-time check has $O(N)$ time complexity (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). Here a major speedup is expected when using *shallow* run-time checks, since the checking time dominates operation execution time and the reduction due to shallow checking should be more noticeable. In the second one the library operation has non-constant ($O(log(N))$) complexity and the respective run-time check $O(N)$ complexity (e.g., inserting an element in a binary tree and checking on each insertion that the input term is a tree). Here obviously a smaller speedup is to be expected with *shallow* checking. All experiments were run on a MacBook Pro, 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.

Run-time Checking After the static preprocessing phase we have divided our libraries into two groups: (a) libraries where the only unchecked assertions left

[10] A simple example of assertions, escaping terms, and shallow checks, as well as full plots for all benchmarks can be found in Appendices B-C of the extended version of this paper available from CoRR at `https://arxiv.org/abs/1705.06662` (v3).

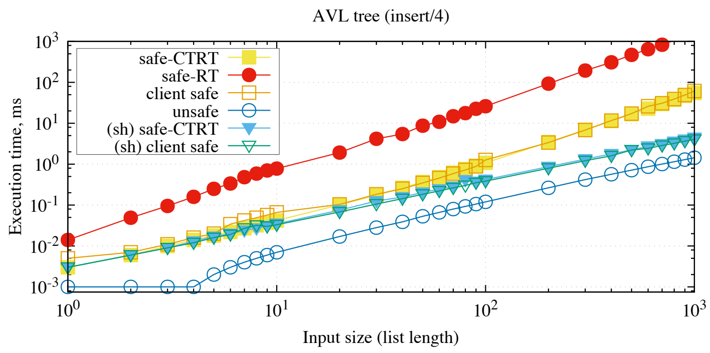

Fig. 2. Run times in different checking modes, `AVL-tree` library, $O(log(N))$ operation.

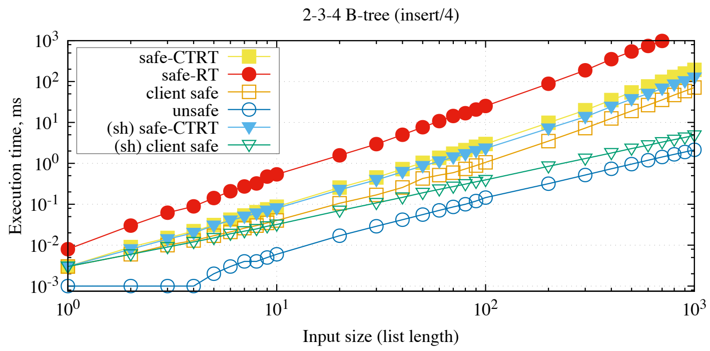

Fig. 3. Run times in different checking modes, `B-tree` library, $O(log(N))$ operation.

are the ones for the boundary calls (`AVL-tree`, `heap`, and `binary tree`),[11] and (b) libraries with also some unchecked assertions for internal calls (`B-tree` and `RB-tree`). We present run time plots[12] for one library of each group. Since the unchecked assertions in the second group correspond to internal calls of the $O(log(N))$ operation experiment, we only show here a set of plots of the $O(1)$ operation experiment for one library, as these plots are very similar across all benchmarks.

Fig. 2 illustrates the overhead reductions from using the shallow run-time checks in the `AVL-tree` benchmark for the $O(N)$ *insert* operation experiment. 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 $O(log(N))$ *insert* operation experiment. In contrast with the previous case, here the overhead reductions achieved by employing shallow checks are dominated by the total check cost, and while the overhead reduction is obvious

[11] 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.
[12] The current measurements depend on the C `getrusage()` function, that on Mac OS has microsecond resolution.

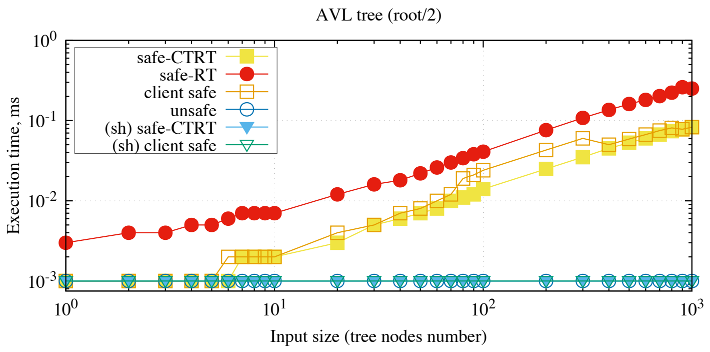

Fig. 4. Run times in different checking modes, `AVL-tree` library, $O(1)$ operation.

in the *Client-Safe* mode, it is not significant in the *Safe-CT+RT* mode where some internal assertion was being checked.

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 experiment 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.

7 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], and ECLiPSe [18], and *atom-based* in XSB [4]. As mentioned before, Ciao [2,5] uses a hybrid approach, which behaves by default as in predicate-based systems but with the possibility of marking a selected set of symbols as hidden, making it essentially compatible with that of XSB. Some previous research in the comparative advantages of atom-based module systems can be found in [19].

Parallels with Static Typing and Contracts While traditionally Prolog is untyped, there have been some proposals for integrating it with type systems, starting with [20]. Several strongly-typed Prolog-based systems have been proposed, notable examples being Mercury [21], Gödel [22], and Visual Prolog [23]. An approach for combining typed and untyped Prolog modules has been proposed in [24]. A conceptually similar approach in the world of functional programming is *gradual typing* [25,26]. The Ciao model offers an (earlier) alternative, closer to *soft typing* [27], but based on safe approximations and abstract interpretation, thus providing a more general and flexible approach than the previous work, since assertions can contain any abstract property –see [28] for a discussion of this topic. This approach has recently also been applied in a number of contract-based systems [29,30,31], for which we believe our techniques can be relevant.

Run-time Checking Optimization High run-time overhead is a common problem in systems that include dynamic checking [26]. The impact of global static analysis in reducing 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 [32,10] or limiting the points at which tests are performed [33]. While these optimizations can bring significant reductions in overhead, it still remains dependent on the size of the terms being checked. We have shown herein that even in the challenging context of calls across open module boundaries it is sometimes possible to achieve constant run-time overhead.

8 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 can be enriched significantly and large reductions in overhead can be achieved. The overheads achieved are closer to those of static languages, even in the reusable-library context, without requiring switching to strong typing, which is less natural in Prolog-style languages, where there is a difference between error and failure/backtracking.

References

1. Stulova, N., Morales, J., Hermenegildo, M.: Towards Run-time Checks Simplification via Term Hiding (extended abstract). In: Tech. Comm. of ICLP'17, OASIcs (2017)
2. Hermenegildo, M., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. TPLP **12**(1–2) (2012) 219–252
3. Stulova, N., Morales, J., Hermenegildo, M.: Reducing the Overhead of Assertion Run-time Checks via Static Analysis. In: PPDP'16, ACM (Sept. 2016) 90–103
4. Swift, T., Warren, D.S.: XSB: Extending Prolog with Tabled Logic Programming. TPLP **12**(1-2) (2012) 157–187
5. Cabeza, D., Hermenegildo, M.V.: A New Module System for Prolog. In: Int'l. Conf. on Computational Logic. Volume 1861 of LNAI., Springer (July 2000) 131–148
6. Hermenegildo, M.V., Puebla, G., Bueno, F.: Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In: The Logic Programming Paradigm. Springer (1999) 161–192
7. Puebla, G., Bueno, F., Hermenegildo, M.: Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In: LOPSTR'99. LNCS 1817, Springer-Verlag (March 2000) 273–292
8. Stulova, N., Morales, J., Hermenegildo, M.: Assertion-based Debugging of Higher-Order (C)LP Programs. In: PPDP'14, ACM (Sept. 2014)

9. Puebla, G., Albert, E., Hermenegildo, M.V.: Abstract Interpretation with Specialized Definitions. In: SAS'06. Number 4134 in LNCS, Springer (2006) 107–126
10. Stulova, N., Morales, J., Hermenegildo, M.: Practical Run-time Checking via Unobtrusive Property Caching. TPLP **15**(04-05) (Sept. 2015) 726–741
11. Warren, D., Chen, W.: Formal Semantics of a Theory of Modules. Technical report 87/11, SUNY at Stony Brook (1987)
12. Chen, W.: A Theory of Modules Based on Second-Order Logic. In: 4th IEEE Int'l. Symposium on Logic Programming, San Francisco (1987) 24–33
13. Miller, D.: A Logical Analysis of Modules in Logic Programming. Journal of Logic Programming (1989) 79–108
14. ISO: PROLOG. ISO/IEC DIS 13211-2 — Part 2: Modules. (2000)
15. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. TPLP **12**(1-2) (2012) 67–96
16. Carlsson, M., Mildner, P.: SICStus Prolog – the first 25 years. TPLP **12**(1-2) (2012) 35–66
17. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog system. TPLP **12**(1-2) (2012) 5–34
18. Schimpf, J., Shen, K.: ECLiPSe – from LP to CLP. TPLP **12**(1-2) (2012) 127–156
19. Haemmerlé, R., Fages, F.: Modules for Prolog Revisited. In: Proc. of ICLP'06. (2006) 41–55
20. Mycroft, A., O'Keefe, R.: A Polymorphic Type System for Prolog. Artificial Intelligence **23** (1984) 295–307
21. Somogyi, Z., Henderson, F., Conway, T.: The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. Journal of Logic Programming **29**(1–3) (October 1996) 17–64
22. Hill, P., Lloyd, J.: The Goedel Programming Language. MIT Press (1994)
23. Prolog Development Center: Visual Prolog
24. Schrijvers, T., Santos Costa, V., Wielemaker, J., Demoen, B.: Towards Typed Prolog. In: ICLP'08. Number 5366 in LNCS, Springer (December 2008) 693–697
25. Siek, J.G., Taha, W.: Gradual Typing for Functional Languages. In: Scheme and Functional Programming Workshop. (2006) 81–92
26. Takikawa, A., Feltey, D., Greenman, B., New, M., Vitek, J., Felleisen, M.: Is Sound Gradual Typing Dead? In: POPL 2016. (January 2016) 456–468
27. Cartwright, R., Fagan, M.: Soft Typing. In: PLDI'91, SIGPLAN, ACM (1991) 278–292
28. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: The Ciao Approach to the Dynamic vs. Static Language Dilemma. In: Proc. Int'l. WS on Scripts to Programs, STOP'11, ACM (2011)
29. Fähndrich, M., Logozzo, F.: Static Contract Checking with Abstract Interpretation. In: Int'l. Conf. on Formal Verification of Object-oriented Software, FoVeOOS'10. Volume 6528 of LNCS., Springer (2011) 10–30
30. Tobin-Hochstadt, S., Van Horn, D.: Higher-Order Symbolic Execution via Contracts. In: OOPSLA, ACM (2012) 537–554
31. Nguyen, P., Horn, D.V.: Relatively Complete Counterexamples for Higher-Order Programs. In: PLDI'15, ACM (2015) 446–456
32. Koukoutos, E., Kuncak, V.: Checking Data Structure Properties Orders of Magnitude Faster. In: Runtime Verification. Volume 8734 of LNCS. Springer (2014) 263–268
33. Mera, E., López-García, P., Hermenegildo, M.V.: Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In: ICLP'09. Volume 5649 of LNCS., Springer-Verlag (July 2009) 281–295