Towards Run-time Checks Simplification via Term Hiding*†

Nataliia Stulova[1], José F. Morales[2], and Manuel V. Hermenegildo[3]

1 **IMDEA Software Institute, Madrid, Spain and Universidad Politécnica de Madrid (UPM), Madrid, Spain**
nataliia.stulova@imdea.org
2 **IMDEA Software Institute, Madrid, Spain**
josef.morales@imdea.org
3 **IMDEA Software Institute, Madrid, Spain and Universidad Politécnica de Madrid (UPM), Madrid, Spain**
manuel.hermenegildo@upm.es

Abstract

Flexibility in term creation and manipulation in dynamic languages can threaten safety of data processing. To counter this issue expensive run-time checks are often added to programs to ensure safety of operations, yet they incur impractically high overheads. While such overheads can be greatly reduced with static analysis, the gains depend strongly on the quality of the information inferred.

We propose a technique for improving term shape inference during static program analysis, that exploits term visibility rules of the underlying module system. We also describe an improved run-time checking approach that takes advantage of the proposed mechanisms to achieve large reductions in overhead. While the approach is general and system-independent, we present and evaluate it for concreteness in the context of the Ciao assertion language and combined static/dynamic checking framework. One of its benefits is that 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, while allowing for reductions in overhead closer to those of static languages.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.3 Studies of Program Constructs, F.3.2 Semantics of Programming Languages

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

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.9

1 Brief Overview

Modular programming has become widely adopted due to the benefits it provides in code reuse and for 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

* In [2] we provide full details on this work.
† This research has been partially funded by Spanish MINECO project TIN2015-67522-C3-1-R *TRACES*, and Madrid Region program M141047003 *N-GREENS*.

programming languages these concepts are implemented in different ways, some examples being the encapsulation mechanism of classes adopted in object-oriented programming 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).

We propose a hybrid predicate-based module system [2] that offers an optional hiding mechanism for selected functor symbols, providing more fine-grained term visibility control. The proposed module system is still strict in the sense that it disallows breaking predicate or term visibility rules by bypassing the module interfaces. The hiding mechanism allow programmers to restrict the visibility of some terms to the module where they are defined, thus both making the concrete implementation details opaque to 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.

The need for mechanisms for controlling term visibility is in particular prominent in the context of assuring safety of data access and manipulation in untyped programming languages. 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 public module interface and for the private internal module routines [1]. 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 reduce the checking cost, we present a technique that, using the combination of term hiding and the strict visibility rules in the module system, enhances the inference of shape information during static program analysis. By restricting some functors to the scope of a module it becomes possible to reason statically about whether the data shapes that are built with these functors are *hidden* and *visible* to the other modules with respect to the module interface. We will further refer to all possible terms that may exist outside a module m as its *escaping terms*. In [2] we provide an algorithm to compute an over-approximation of the set of all escaping terms from a module for a given set of functor hiding declarations.

▶ **Example 1.** Let `point/1` be a hidden functor in a module `m1` that exports a single predicate `p/1` which constructs a term `point(1)`:

```
:- module(m1, [p/1]).          % module interface
:- hide point/1.               % hidden functor
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 term hiding, this is impossible to ensure (without client knowledge) since any module could define any `point(_)` terms (e.g., `point([_,_])`, `point(coord(_,_,_))`). In this simplified example `point(1)` is the *escaping* term of module `m1`.

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. The information about escaping terms obtained by the static analysis can then be used to replace the original run-time checks with their optimized versions while preserving the safety guarantees the original checks provide. 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). Shallow run-time checking consists in using *shallow* versions of properties in the run-time checks for the calls across module boundaries.

▶ **Example 2.** 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(_))`.

Note that since the argument(s) inside, e.g., the first level of a term can be arbitrarily large the savings from this technique can also be unbounded. In our work we show experimentally that for practical programs and settings, 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. Together, the combination of these techniques with traditional static analysis brings improvements in the number and cost of the run-time checks that allow providing equivalent guarantees to those of statically-typed approaches, at similar run-time cost, 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 [1]: 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.

References

1 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. CoRR abs/1102.5497 [cs.PL]. doi:10.1017/S1471068411000457.

2 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Term Hiding and its Impact on Run-time Check Simplification. Technical Report CLIP-1/2017.0, The CLIP Lab, May 2017. CoRR abs/1705.06662 [cs.PL].