TPLP: Page 1–19.

doi:10.1017/S147106842510015X

*Hiord♯: An Approach to the Specification and Verification of Higher-Order (C)LP Programs**

MARCO CICCALÈ, DANIEL JURJO-RIVAS and JOSE F. MORALES
Universidad Politécnica de Madrid (UPM), IMDEA Software Institute, Madrid, Spain
(*e-mails:* m.ciccale@alumnos.upm.es, marco.ciccale@imdea.org, daniel.jurjo@alumnos.upm.es, daniel.jurjo@imdea.org, josefrancisco.morales@upm.es, josef.morales@imdea.org)

PEDRO LÓPEZ-GARCÍA
Spanish Council for Scientific Research, IMDEA Software Institute, Madrid, Spain
(*e-mails:* pedro.lopez@csic.es, pedro.lopez@imdea.org)

MANUEL V. HERMENEGILDO
Universidad Politécnica de Madrid (UPM), IMDEA Software Institute, Madrid, Spain
(*e-mail:* manuel.hermenegildo@upm.es, manuel.hermenegildo@imdea.org)

submitted 22 July 2025; revised 22 July 2025; accepted 27 July 2025

Abstract

Higher-order constructs enable more expressive and concise code by allowing procedures to be parameterized by other procedures. Assertions allow expressing partial program specifications, which can be verified either at compile time (statically) or run time (dynamically). In higher-order programs, assertions can also describe higher-order arguments. While in the context of (constraint) logic programming ((C)LP), run-time verification of higher-order assertions has received some attention, compile-time verification remains relatively unexplored. We propose a novel approach for statically verifying higher-order (C)LP programs with higher-order assertions. Although we use the Ciao assertion language for illustration, our approach is quite general, and we believe is applicable to similar contexts. Higher-order arguments are described using predicate properties – a special kind of property which exploits the (Ciao) assertion language. We refine the syntax and semantics of these properties and introduce an abstract criterion to determine conformance to a predicate property at compile time, based on a semantic order relation comparing the predicate property with the predicate assertions. We then show how to handle these properties using an abstract interpretation-based static analyzer for programs with first-order assertions by reducing predicate properties to first-order properties. Finally, we report on a prototype implementation and evaluate it through various examples within the Ciao system.

KEYWORDS: higher-order, static analysis, assertions, abstract interpretation, (constraint) logic programming

* Partially funded by MICIU projects CEX2024-001471-M *María de Maeztu* and TED2021-132464B-I00 *PRODIGY*, as well as by the Tezos foundation. We would also like to thank the anonymous reviewers for their very useful and constructive feedback.

1 Introduction

Abstraction is a fundamental principle in computer science often used for managing complexity. Higher-order constructs are a form of abstraction that enables writing code that is more concise and expressive by allowing procedures to be parameterized by other procedures, resulting in more modular and maintainable code. (Constraint) logic programming languages like Prolog (Körner *et al.* 2022) and functional programming languages like Haskell (Marlow 2010) have included different forms of higher-order since their early days, and languages from other programming paradigms like Java or C++ have adopted them later on. In particular, Prolog systems allow defining higher-order predicates and making higher-order calls. *For example*, the query: `?- filter(even,[7,4,9],L)`, passes the term `even` as an argument to the higher-order predicate `filter/3`, which *applies* the `even/1` predicate to each element of the input list, selecting those that succeed, yielding `L = [4]`. Assertions are linguistic constructs for writing partial program specifications, which can then be verified or used to detect deviations in program behavior w.r.t. such specifications. The *assertion-based approach* to program verification (Hermenegildo *et al.* 1999; Puebla *et al.* 2000b; Sanchez-Ordaz *et al.* 2021) differs from other approaches such as strong type systems (Cardelli 1989) in that assertions are optional and can include properties that are undecidable at compile time, and thus some checking may need to be relegated to run time. Hence, the assertion-based approach is closer to gradual typing in functional languages (Siek and Taha 2006). The combination of higher-order predicates and assertions in the (C)LP context was already explored by Stulova *et al.* (2014). This work introduced the notion of *predicate properties*, a special kind of properties that allow using the full power of the (Ciao) assertion language for describing the higher-order arguments of procedures. This work also proposed an operational semantics for *dynamically* checking higher-order (C)LP programs annotated with such higher-order assertions. However, the *static* verification of programs with higher-order assertions was not addressed in that work, and remains relatively unexplored since other related work in (C)LP that supports higher order (*e.g.* Miller (1991); Hill and Lloyd (1994); Somogyi *et al.* (1996)) generally adheres to the strong typing model. In this work we propose a novel approach for the compile-time verification of higher-order (C)LP programs with assertions describing higher-order arguments. We present a refinement of both the syntax and the semantics of predicate properties (§3). Next, we define an abstract criterion to determine whether a predicate conforms to a predicate property at compile time, based on a semantic order relation between the definition of a predicate property and the partial specification of a predicate (§4.1). Then, we introduce an approach for "casting" predicate usage in a program analysis-friendly manner that enhances and complements the proposed abstract criterion (§4.2). We also propose a technique for dealing with these properties using an abstract interpretation-based static analyzer for programs with first-order assertions, by representing predicate properties as first-order properties that are natively understood by such an analyzer (§4.3). Finally, we present a prototype implementation of these techniques and study its application to a number of examples (§5). For concreteness, we use in our presentation the Ciao (Hermenegildo *et al.* 2012) assertion language, and make use of its CiaoPP preprocessor (Hermenegildo *et al.* 2005), that combines both static and dynamic analysis. However, we believe the approach is quite general and flexible, and can be applied, at least conceptually, to similar gradual approaches.

2 Preliminaries and notation

Variables start with a capital letter. The set of terms is inductively defined as follows: (1) variables are terms (2) if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. We use the overbar notation $(\bar{\cdot})$ to denote a finite sequence of elements (*e.g.* $\bar{t} \equiv t_1, \ldots, t_n$), and write $|(\bar{\cdot})|$ for representing its length. An *atom* has the form $p(\bar{t})$ where p is an n-ary predicate symbol, and $\bar{t}$ are terms. The function $\mathsf{ar}(p)$ denotes the arity of a predicate p. A *higher-order atom* has the form $X(\bar{t})$ where X is a variable and $\bar{t}$ are terms. (Note that variables are *not* allowed in the function symbol position of terms, only in literals). A *constraint* is a conjunction of expressions built from predefined predicates whose arguments are constructed using predefined functions and variables, *for example*, $X - Y > abs(Z)$. A *literal* is either an atom, a higher-order atom, or a constraint. *Negation* is encoded as finite failure, supported through a program expansion. A *goal* is a finite sequence of literals. A *rule* has the form $H \texttt{ :- } B$ where H, the *head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *higher-order constraint logic program*, or *higher-order program* P is a finite set of rules. We use σ to represent a variable renaming, and $\sigma(L)$ or $L\sigma$ to represent the result of applying σ to a syntactic object L. The *definition* of an atom L in a program, $\mathsf{defn}(L)$, is the set of renamed program rules s.t. each renamed rule has L as its head. We assume that all rule heads are *normalized*, *that is*, H is an atom of the form $p(\bar{v})$ where $\bar{v}$ are distinct variables. Let $\exists_L \theta$ denote the projection of the constraint θ onto the variables of L. We denote *constraint entailment* by $\theta_1 \models \theta_2$.

2.1 Operational semantics of higher-order programs

The operational semantics of a higher-order program P is given in terms of its *derivations*, which are sequences of *reductions* between *states*. A state $\langle G \mid \theta \rangle$ consists of a goal G, and a constraint store (or store) θ. We denote sequence concatenation by (::). We assume for simplicity that the underlying constraint solver is complete and projection exists. We use $S \rightsquigarrow S'$ to indicate that a reduction step can be applied to state S to obtain state S'. Naturally, $S \rightsquigarrow^* S'$ indicates that there is a sequence of reduction steps from S to S'. A state $S = \langle L :: G \mid \theta \rangle$ where L is a literal, is *reduced* to a state S' as follows:

1. If L is a constraint and $\theta \wedge L$ is satisfiable, then $S' = \langle G \mid \theta \wedge L \rangle$.
2. If L is an atom and $\exists (L \texttt{ :- } B) \in \mathsf{defn}(L)$, then $S' = \langle B :: G \mid \theta \rangle$.
3. If L is a higher-order atom of the form $X(\bar{t})$, then $S' = \langle p(\bar{t}) :: G \mid \theta \rangle$ given that $\exists p \in P.\, \theta \models (X = p) \wedge \mathsf{ar}(p) = |\bar{t}|$.

Let L be an atom, $S = \langle L :: G \mid \theta \rangle$, $S' = \langle G \mid \theta' \rangle$, and suppose $S \rightsquigarrow^* S'$. We refer to S as a *call state* for L, and S' as a *success state* for L. A *query* Q is a pair (L, θ), where L is a literal and θ a store for which the (C)LP system starts a computation from state $\langle L \mid \theta \rangle$. The set of all derivations of P from a query Q is denoted $\mathsf{derivs}(P, Q)$, and this notation is naturally extended to a set of queries $\mathcal{Q}$. Let $D_{[-1]}$ denote the last state of any derivation D. A finite derivation from a query Q is *finished* if the last state in the derivation cannot be reduced. A finished derivation from a query Q is *successful* if the last state is of the form $\langle \square \mid \theta' \rangle$, where $\square$ denotes the empty goal sequence. In that case,

the constraint $\bar{\exists}_L \theta'$ is an *answer* to Q. We denote by $\mathsf{answers}(P, Q)$ the set of answers of P to a query Q. A finished derivation is *failed* if the last state is not of the form $\langle \square \mid \theta \rangle$. A query Q *finitely fails* if all derivations in $\mathsf{derivs}(P, Q)$ are finished and have failed.

2.2 Property formulas

Conditions on the constraint store are stated as *property formulas*. A property formula is a DNF formula of *property literals*. A property literal is a literal corresponding to a special kind of predicates called *properties*. Properties are typically defined in the source language, in the same way as ordinary predicates but marked accordingly, and are required to meet certain conditions (Hermenegildo *et al.* 1999; Puebla *et al.* 2000b). In particular, they are normally required to be checkable at run time but not necessarily decidable at compile time, where they are safely approximated.[1]

Example 2.1 (Properties).
The following program defines the properties `list/1` *("being a list") and* `prefix/2` *("being a prefix of a list"):*

```
1 :- prop list/1. list([]).
2 list([_|Xs]) :- list(Xs).
3 :- prop prefix/2. prefix([],Ys) :- list(Ys).
4 prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).
```

The property formula `(list(Xs),list(Ys),prefix(Xs,Ys))` *states that* `Xs` *and* `Ys` *should be lists, and that* `Xs` *should be a prefix of* `Ys`*. This formula contains three property literals corresponding to the* `list/1` *and* `prefix/2` *properties.*

We now recall an instrumental definition about properties from Puebla *et al.* (2000b):

Definition 2.1 (Succeeds trivially).
A property literal L succeeds trivially for θ in a program P, denoted $\theta \Rightarrow_P L$, iff $\exists \theta' \in \mathsf{answers}(P, (L, \theta)).\, \theta \models \theta'$. A property formula succeeds trivially for θ if all of the property literals of at least one conjunct of the formula succeeds trivially.

Intuitively, a property literal (or formula) succeeds trivially if it succeeds for θ in P without adding new "relevant" constraints to θ. *For example*, `list(X)` checks "`X` being a list."

2.3 Traditional assertions

Assertions are syntactic objects for expressing properties of programs that must be satisfied at program execution. We recall the herein relevant parts of the assertion schema of Puebla *et al.* (2000a). *Traditional* (or *first-order*) *predicate* (or *pred*) assertions have the following syntax: "`:- pred` *Pred* `:` *Pre* `=>` *Post*`.`," where *Pred* is a normalized atom representing a predicate, and *Pre* and *Post* are property formulas. They express that all calls to *Pred must* satisfy the precondition *Pre*, and, if such calls succeed, the postcondition *Post must* be satisfied. If there are several *pred* assertions, the *Pre* field of at least one of them must be satisfied.

[1] Ciao assertions can also include *global properties*, which may not always be checkable at run time (*e.g.* termination), but we focus for brevity on the described types of assertions and properties.

Example 2.2 (Assertions).
The following assertions for the `take/3` *predicate relating a list and its prefix:*

```
:- pred take(N,Xs,Ys) : (int(N), list(Xs)) => prefix(Ys,Xs).
:- pred take(N,Xs,Ys) : (list(Xs), prefix(Ys,Xs)) => int(N).
```

restrict the meaning of `take/3` *as follows:*

- `take(N,Xs,Ys)` *must be called with* `Xs` *bound to a list, and either* `N` *bound to an integer or* `Ys` *bound to a prefix of* `Xs`.
- *If* `take(N,Xs,Ys)` *succeeds when called with* `N` *bound to an integer and* `Xs` *bound to a list, then* `Ys` *must be bound to a prefix of* `Xs`.
- *If* `take(N,Xs,Ys)` *succeeds when called with* `Xs` *bound to a list and* `Ys` *bound to a prefix of* `Xs`, *then* `N` *must be bound to an integer.*

We represent checks on the store by a set of assertions with a set of *assertion conditions*.

Definition 2.2 (Assertion conditions).
Given a predicate represented by a normalized atom Pred, and its corresponding set of assertions $\{A_1, \ldots, A_n\}$ *with* $A_i =$ "`:- pred` *Pred* `:` Pre_i `=>` $Post_i$`.`," *the set of assertion conditions for Pred is* $\{C_0, C_1, \ldots, C_n\}$ *with*

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

Condition C_0 *encodes the checks that ensure that all calls to the predicate represented by Pred are within those admissible by the set of assertions; we refer to it as the calls assertion condition. Conditions* $C_1, \ldots, C_n$ *encode the checks for compliance of the successes for particular sets of calls, and we call them the success assertion conditions.*

From this point on, we denote by $\mathcal{A}$ both the set of assertions of the program and, interchangeably, its associated set of assertion conditions. Also, for a normalized atom *Pred*, $\mathcal{A}(Pred)$ denotes only the assertions of $\mathcal{A}$ associated to the predicate *Pred*.

Example 2.3 (Assertion conditions).
The set of assertion conditions for the set of pred assertions in Example 2.2 *is:*

```
calls(take(N,Xs,Ys), (int(N), list(Xs)) ∨ (list(Xs), prefix(Ys,Xs)))
success(take(N,Xs,Ys), (int(N), list(Xs)), prefix(Ys,Xs))
success(take(N,Xs,Ys), (list(Xs), prefix(Ys,Xs)), int(N))
```

2.4 Operational semantics of higher-order programs with traditional assertions

This operational semantics checks whether assertion conditions hold or not while computing the derivations from a query, halting the derivation as soon as an assertion condition is violated. For identifying a possible assertion condition violation, every assertion condition C is related to a unique label ℓ via a mapping $\mathsf{label}(C) = \ell$. States of derivations are

now of the form $\langle G \mid \theta \mid \mathcal{E} \rangle$, where $\mathcal{E}$ denotes the set of labels for falsified assertion conditions (with $|\mathcal{E}| \leqslant 1$); while such set is unnecessary if execution halts upon an assertion condition violation, we include it to keep the semantics presented in this paper close to that of previous work. A finished derivation from a query $Q = (L, \theta)$ is now *successful* if the last state is of the form $\langle \square \mid \theta' \mid \varnothing \rangle$, *failed* if the last state is of the form $\langle L' \mid \theta' \mid \varnothing \rangle$, and *erroneous* if the last state is of the form $\langle L' \mid \theta' \mid \{\ell\} \rangle$. We also extend the set of literals with syntactic objects of the form $\mathsf{check}(L, \ell)$ where L is a literal and ℓ is a label for an assertion condition, which we call *check literals*. Thus, a *literal* is now a constraint, an atom, a higher-order atom, or a check literal. We now recall the notion of *Semantics with Assertions* from Stulova *et al.* (2018), which we adapt to support higher-order atoms. A state $S = \langle L :: G \mid \theta \mid \varnothing \rangle$, can be *reduced* to a state S', denoted $S \rightsquigarrow_{\mathcal{A}} S'$, as follows:

1. If L is a constraint or a higher-order atom, then $S' = \langle G' \mid \theta' \mid \varnothing \rangle$, with G' and θ' obtained as in the operational semantics without assertions: $\langle L :: G \mid \theta \rangle \rightsquigarrow \langle G' \mid \theta' \rangle$.
2. If L is an atom and $\exists (L \texttt{ :- } B) \in \mathsf{defn}(L)$, then

$$S' = \begin{cases} \langle G \mid \theta \mid \{\ell\} \rangle & \text{if } \exists C = \mathsf{calls}(L, Pre) \in \mathcal{A}.\ \mathsf{label}(C) = \ell \wedge \theta \not\Rightarrow_P Pre \\ \langle B :: PostC :: G \mid \theta \mid \varnothing \rangle & \text{otherwise} \end{cases}$$

   where $PostC = \mathsf{check}(L, \ell_1) :: \ldots :: \mathsf{check}(L, \ell_n)$ includes all the checks $\mathsf{check}(L, \ell_i)$ such that $\ell_i = \mathsf{label}(C_i)$, with $C_i = \mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A} \wedge \theta \Rightarrow_P Pre_i$.
3. If L is a check literal $\mathsf{check}(L', \ell)$, then

$$S' = \begin{cases} \langle G \mid \theta \mid \{\ell\} \rangle & \text{if } \exists C = \mathsf{success}(L, \_, Post) \in \mathcal{A}.\ \mathsf{label}(C) = \ell \wedge \theta \not\Rightarrow_P Post \\ \langle G \mid \theta \mid \varnothing \rangle & \text{otherwise} \end{cases}$$

The set of derivations for a program P with assertions $\mathcal{A}$ from a set of queries $\mathcal{Q}$ using the semantics with assertions is denoted $\mathsf{derivs}_{\mathcal{A}}(P, \mathcal{Q})$. Given a predicate represented by a normalized atom L, a store θ, and a set of queries $\mathcal{Q}$, we define the *success context* $\mathcal{S}_{\mathcal{A}}(L, \theta, P, \mathcal{Q})$ of L and θ for P and $\mathcal{Q}$ as $\{\bar{\exists}_L \theta' \mid \exists D \in \mathsf{derivs}_{\mathcal{A}}(P, \mathcal{Q}).\ \exists G.\ \langle L :: G \mid \theta \rangle \in D.\ D_{[-1]} = \langle G \mid \theta' \rangle\}$. Intuitively, the *success context* of a predicate p with its assertions is the set of stores of the success states of p obtained using the semantics above.

2.5 Static program analysis by abstract interpretation

Abstract interpretation (Cousot and Cousot 1977) is a mathematical framework for constructing sound, static program analysis tools. These tools extract properties about a program by interpreting it over a special *abstract domain* ($D^\sharp$), whose elements are finite representations of (possibly infinite) sets of actual constraints in the *concrete domain* (D). Elements of the concrete and abstract domains are related by two functions: *abstraction* ($\alpha : D \to D^\sharp$) and *concretization* ($\gamma : D^\sharp \to D$). Provided certain conditions on $D^\sharp$, α, and γ, abstract interpretation guarantees soundness and termination of the analysis.

2.6 Goal-dependent abstract interpretation

We use, for concreteness, *goal-dependent abstract interpretation*, in particular the PLAI algorithm (Muthukumar and Hermenegildo 1992). This technique takes a program P,

an abstract domain $D^\sharp$, and a set of initial abstract queries $\mathcal{Q}^\sharp$, describing all the possible initial concrete queries to P. An abstract query $Q^\sharp$ is a pair (L, λ), where L is an atom and $\lambda \in D^\sharp$ an abstraction of a set of concrete initial program states (*e.g.* constraint stores). A set of abstract queries $\mathcal{Q}^\sharp$ represents a set of concrete queries defined as $\gamma(\mathcal{Q}^\sharp) = \{(L, \theta) \mid (L, \lambda) \in \mathcal{Q}^\sharp \wedge \theta \in \gamma(\lambda)\}$. The algorithm computes a set of triples $\{\langle L_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle L_n, \lambda_n^c, \lambda_n^s\rangle\}$ where L_i is an atom, and λ_i^c and λ_i^s are abstractions approximating the set of all call and success states for L_i, respectively, for all occurrences of literal L_i in all possible derivations of P from $\gamma(\mathcal{Q}^\sharp)$. Higher-order atoms are supported by reducing them to first-order calls when the called predicate can be determined by the analysis, or making conservative assumptions otherwise. For the rest of the paper, we assume that the abstract interpretation of a program P for the set of initial abstract queries $\mathcal{Q}^\sharp$, denoted by $[\![P]\!]^\sharp_{\mathcal{Q}^\sharp}$, works with an implicit abstract domain $D^\sharp$, which safely approximates the concrete values and operations. Although not strictly required, $D^\sharp$ has a lattice structure with a bottom-most element $\bot$, meet ($\sqcap$), join ($\sqcup$), and less than ($\sqsubseteq$) operators. As usual, $\bot$ denotes the abstraction s.t. $\gamma(\bot) = \varnothing$.

2.7 Compile-time verification of (first-order) assertions

In addition to generating the results mentioned above, the analyzer also checks any (first-order) assertions in the program by safely approximating the property formulas of such assertions, and comparing them against the analysis results ($[\![P]\!]^\sharp_{\mathcal{Q}^\sharp}$) using the abstract operators.[2] The verification result is reported as changes in the status and transformations of the assertions: **`checked`** if the properties are satisfied; **`false`** if some property is proved not to hold; or **`check`** if neither of the first two can be determined, in which case run-time checks will be inserted into the program to ensure run-time safety. The verification process yields an assignment of a value **`checked`**, **`false`**, or **`check`** to each assertion in $\mathcal{A}$, denoted $\mathsf{acheck}(\mathcal{A}, [\![P]\!]^\sharp_{\mathcal{Q}^\sharp})$.

3 Specifying higher-order programs: predicate properties

In higher-order (C)LP, variables can be bound to predicate symbols that are later invoked. This naturally gives rise to the need for expressing conditions on these predicates that must hold during program execution. To this end, *predicate properties* were introduced in Stulova *et al.* (2014), which we revise and refine here.[3] A predicate property is defined as a set of *anonymous assertions*. Anonymous assertions generalize traditional assertions by allowing the predicate symbol in the *Pred* field to act as a placeholder.

Definition 3.1 (Anonymous assertion).
An anonymous assertion $^\circ A$ *is an assertion whose Pred field is of the form* $\_(\bar{v})$, *where* $\bar{v}$ *are free, distinct variables, and* `_` *is a placeholder for a predicate symbol.*[4] *Instantiating* `_`

[2] We refer the reader to Puebla *et al.* (2000b); Sanchez-Ordaz *et al.* (2021) for the technical details on this subject.
[3] We propose a more compact syntax here that avoids having to use a named variable for the anonymous predicate symbol (as in Stulova *et al.* (2014)) and takes advantage of functional notation (`:=`).
[4] We also use for compactness "`_`" as anonymous functor, a syntactic extension from the Ciao `hiord` package (Cabeza *et al.* 2004), but double quotes `''` can also be used to stay within ISO-Prolog syntax.

with a specific predicate symbol p produces a traditional assertion for p derived from the anonymous assertion $\mathcal{A}$*, denoted* $\mathcal{A}|_p$*.*

Example 3.1 (Anonymous assertion).
Let $\mathcal{A}$ *be the anonymous assertion: "*`:- pred _(X,Y) : int(X) => int(Y).`*" Then,* $\mathcal{A}|_p$ *is the traditional assertion: "*`:- pred p(X,Y) : int(X) => int(Y).`*" obtained by instantiating the anonymous assertion* $\mathcal{A}$ *with the predicate symbol* `p`*.*

Definition 3.2 (Predicate property).
A predicate property Π *is a set of anonymous assertions* $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$*. Its syntax is: "*$\Pi$ `:= { :- pred`$\_(\bar{v})$ `:` Pre_1 `=>` $Post_1$ `.` $\ldots$ `:- pred`$\_(\bar{v})$ `:` Pre_n `=>` $Post_n$`. }.`*". The function* `ar`(Π) *denotes the arity of the predicates for which all of the anonymous assertions in* Π *express a property. Instantiating* Π *with a specific predicate symbol p produces a set of traditional assertions for p, denoted* $\Pi|_p = \{\mathcal{A}_1|_p, \ldots, \mathcal{A}_n|_p\}$*.*

We use Π to refer to both a set of anonymous assertions and, interchangeably, the corresponding set of anonymous assertion conditions; extending *instantiation* accordingly.

Example 3.2 (Predicate property).
The following program defines the predicate property `int_op` *("being a predicate that behaves as an integer nondeterministic binary operator"), and a higher-order assertion:*

```
1 int_op := { :- pred _(X,Y,Z) : (int(X), int(Y)) => int(Z). }.
2 :- pred eval(A,B,Op,R) : (int(A), int(B), int_op(Op)) => int(R).
```

The predicate property literal `int_op (Op)` *states that* `Op` *should be a 3-ary predicate s.t., if called with its first two arguments bound to integers, then its third argument should be bound to an integer upon success. The higher-order assertion for the* `eval/4` *predicate states that it must be called with its first two arguments bound to integers and its third argument bound to a predicate that* conforms *to property* `int_op`*, and that if any such call succeeds, then its fourth argument should be bound to an integer.*

4 Verifying higher-order programs

Once established how to specify higher-order programs using predicate properties, we now concentrate on how to verify such programs. We first recall some instrumental definitions from Puebla *et al.* (2000b) for reasoning about abstractions of property formulas. For the rest of the discussion, let P be a program and F a property formula defined in P.

Definition 4.1 (Trivial success set).
We define the trivial success set of F *as* $F^{\natural} = \{\bar{\exists}_F\theta \mid \theta \Rightarrow_P F\}$*.*

Example 4.1 (Trivial success set).
Let $F =$ `list(L)`*, both* $\theta_1 =$ `{L = [1,2]}` *and* $\theta_2 =$ `{L = [1,X]}` *are in* $F^{\natural}$*, but* $\theta_3 =$ `{L = [1|_]}` *is not, since a call to* `(L = [1|_], list(L))` *would further instantiate the second argument of* `[1|_]`*. The trivial success set* $F^{\natural}$ *of* F *captures the notion that the* `list(L)` *property formula requires* `L` *to be instantiated to (the structure of) a list.*

Definition 4.2 (Abstract trivial success subset).
An abstraction is an abstract trivial success subset of F*, denoted* $F^{\sharp-}$*, iff* $\gamma(F^{\sharp-}) \subseteq F^{\natural}$*.*

Definition 4.3 (Abstract trivial success superset).
An abstraction is an abstract trivial success superset of F, *denoted* $F^{\sharp+}$, *iff* $\gamma(F^{\sharp+}) \supseteq F^\natural$.

Intuitively, $F^{\sharp-}$ and $F^{\sharp+}$ are, respectively, a safe under- and over-approximation of the trivial success set $F^\natural$ of the property formula F, and they can always be computed at compile-time by choosing the closest element in the abstract domain.

4.1 Conformance to a predicate property

When we provide a partial specification for a higher-order argument X of a higher-order predicate using a predicate property Π, we are describing requirements that the predicates that X may be bound to must meet. We refer to a predicate p behaving correctly w.r.t. Π as: p *conforming* to Π. We will now formalize this notion, with the goal of being able to safely approximate the set of predicates that X can be bound to without violating Π.

Definition 4.4 (Covered predicate).
Given $C = \mathsf{calls}(\_(\bar{v}), {}^\circ Pre) \in \Pi$ *and* ${}^\circ C = \mathsf{calls}(p(\bar{v}), Pre) \in \mathcal{A}$, *we say that* p *can be covered with* Π *iff* ${}^\circ Pre^\natural \subseteq Pre^\natural$.

Intuitively, p can be covered with Π if the set of admissible calls to p is a superset of the set of admissible calls described by Π.

Definition 4.5 (Redundance).
Under the same conditions as in Definition 4.4, *given that* p *can be covered with* Π, *we define the set of assertion conditions* $\mathcal{A}'$ *as follows:*

$$\mathcal{A}' = \{\mathsf{calls}(p(\bar{v}), Pre \wedge {}^\circ Pre)\} \cup (\mathcal{A} \setminus \{C\}) \cup (\Pi \setminus \{{}^\circ C\})|_p$$

Given a sequence of literals G, *let* $\mathcal{U}(G)$ *denote the result of removing all check literals from* G. *We extend* $\mathcal{U}$ *to derivations so that* $\mathcal{U}(D)$ *denotes the derivation resulting from transforming any (extended) state* $\langle G \mid \theta \mid \mathcal{E}\rangle$ *in* D *into the state* $\langle G' \mid \theta\rangle$, *where* $G' = \mathcal{U}(G)$.
Let Q_p *be a query to* p. *We say that* Π *is redundant for* p *under* Q_p *iff*

$$\forall D' \in \mathsf{derivs}_{\mathcal{A}'}(P, Q_p).\ D'_{[-1]} = \langle G' \mid \theta \mid \{\ell'\}\rangle,$$

and

$$\forall D \in \mathsf{derivs}_{\mathcal{A}}(P, Q_p).\ \mathcal{U}(D) = \mathcal{U}(D'),$$

it holds that $D_{[-1]} \rightsquigarrow^*_{\mathcal{A}} \langle G \mid \theta \mid \{\ell\}\rangle$ *through a derivation that reduces only check literals (if any at all),*[5] *where* ℓ *(resp.,* ℓ'*) is the label for a* calls *or* success *assertion condition in* $\mathcal{A}(p(\bar{v}))$ *(resp.,* $\mathcal{A}'(p(\bar{v}))$*).*

Intuitively, a predicate property Π is redundant for a predicate p under a query Q_p to p iff augmenting the original set of assertion conditions ($\mathcal{A}$) with that of Π ($\mathcal{A}'$) does not introduce new run-time check errors in any derivation starting from Q_p.[6]

[5] Note that this implies $\mathcal{U}(G) = \mathcal{U}(G')$.
[6] Note that, for the purposes of determining conformance, the assertions for the predicates in the program can be provided by the user, inferred by analysis, or a combination of both.

Definition 4.6 (Conformance).
Let $\mathcal{Q}_p$ be the set of all possible queries to p. A predicate p conforms to Π, *denoted* $p \prec \Pi$, *iff* $\forall Q_p \in \mathcal{Q}_p.\, \Pi$ *is redundant for p under Q_p. Conversely, p does not conform to* Π, *denoted* $p \not\prec \Pi$, *iff* $\exists Q_p \in \mathcal{Q}_p.\, \Pi$ *is not redundant for p under Q_p.*

To prove that a predicate conforms to a predicate property, all possible derivations from all possible queries to that predicate have to be considered, which is often not feasible in practice. To this end, we introduce the notion of *abstract conformance* as a compile-time conformance criterion. Abstract conformance safely approximates the notion of conformance by comparing the assertion conditions of a predicate and those of a predicate property under the order relation of an abstract domain. We denote by $(\prec^{\sharp -})$ the notion of *strong* abstract conformance, and by $(\prec^{\sharp +})$ that of *weak* abstract conformance. *That is*, an under- and over-approximation of abstract conformance, respectively. Intuitively, strong abstract conformance captures only predicates known to conform, while weak abstract conformance also includes those for which conformance is unknown. Thus, the negation of weak abstract conformance captures the predicates that are known not to conform.

Definition 4.7 (Abstract conformance on "calls").
Let Pre be the precondition of the calls *assertion condition for p in $\mathcal{A}$, and $°C$ be an anonymous* calls *assertion condition* $\mathsf{calls}(\_(\bar{v}), °Pre)$. *Then:*

$$p \prec^{\sharp -} °C \Leftrightarrow (Pre^{\sharp +} \sqsubseteq °Pre^{\sharp -}) \wedge (Pre^{\sharp -} \sqsupseteq °Pre^{\sharp +})$$
$$p \not\prec^{\sharp +} °C \Leftrightarrow Pre^{\sharp +} \sqcap °Pre^{\sharp +} = \bot$$

Definition 4.8 (Abstract conformance on "success").
Let $\mathcal{A}$ be the set of assertion conditions for p, and $°C$ be an anonymous success *assertion condition* $\mathsf{success}(\_(\bar{v}), °Pre, °Post)$. *Then:*

$$p \prec^{\sharp -} °C \Leftrightarrow \exists S \subset \mathcal{A}, ({Pre^{\sharp -}}_{\sqcup} \sqsupseteq °Pre^{\sharp +}) \wedge ({Post^{\sharp +}}_{\sqcup} \sqsubseteq °Post^{\sharp -}),$$
$$\text{where } \begin{cases} {Pre^{\sharp -}}_{\sqcup} & = \sqcup\{Pre^{\sharp -} \mid \mathsf{success}(p(\bar{v}), Pre, \_) \in S\} \\ {Post^{\sharp +}}_{\sqcup} & = \sqcup\{Post^{\sharp +} \mid \mathsf{success}(p(\bar{v}), \_, Post) \in S\} \end{cases}$$
$$p \not\prec^{\sharp +} °C \Leftrightarrow \exists\, \mathsf{success}(p(\bar{v}), Pre, Post) \in \mathcal{A}, (Pre^{\sharp +} \sqsubseteq °Pre^{\sharp -}) \wedge$$
$$\wedge\, (Post^{\sharp +} \sqcap °Post^{\sharp +} = \bot) \wedge \exists \theta \in Pre^{\natural}.\, \mathcal{S}_{\mathcal{A}}(p(\bar{v}), \theta, P, \gamma({\mathcal{Q}^{\sharp}}_p)) \neq \varnothing$$

where ${\mathcal{Q}^{\sharp}}_p$ is the set of abstract queries s.t. $\gamma({\mathcal{Q}^{\sharp}}_p)$ is a superset of the set of all valid queries to p described by the calls *assertion condition of p in $\mathcal{A}$.*

Definition 4.9 (Abstract conformance).
We define abstract conformance to a predicate property as follows:

$$p \prec^{\sharp -} \Pi \Leftrightarrow \forall\, °C \in \Pi_C.\, p \prec^{\sharp -} °C$$
$$p \not\prec^{\sharp +} \Pi \Leftrightarrow \exists\, °C \in \Pi_C.\, p \not\prec^{\sharp +} °C$$

Note that abstract conformance is computed by first computing the abstract trivial success subsets or supersets of the involved property formulas, and then applying the

(a)

```
p_nat_nat := { :- pred _(X,Y) : nat(X) => nat(Y). }.
```

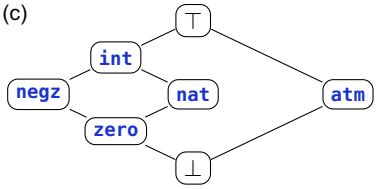
Predicate property.

(b)

```
:- pred  n2n(X,Y) :  nat(X) =>  nat(Y). % A1
:- pred  a2n(X,Y) :  atm(X) =>  nat(Y). % A2
:- pred  i2z(X,Y) :  int(X) => zero(Y). % A3
:- pred  z2i(X,Y) : zero(X) =>  int(Y). % A4
:- pred nz2n(X,Y) : negz(X) =>  nat(Y). % A5
```

Assertions.

(c)



Abstract domain lattice.

Fig. 1. Example case analysis on a predicate property and assertions.

operators of the abstract domain. For abstract domains which may lose precision with their ($\sqcup$) abstract operator, more advanced techniques for leveraging multiple abstractions become necessary, *for example*, *covering* (Debray *et al.* 1997). We now relate the notions of conformance and abstract conformance.

Theorem 4.1.
Let p be a predicate, Π *be a predicate property:* $p \prec^{\sharp -} \Pi \Rightarrow p \prec \Pi$, *and* $p \not\prec^{\sharp +} \Pi \Rightarrow p \not\prec \Pi$.

Proof.
The proofs proceed by contradiction and direct proof, respectively, using Defs. 4.2 to 4.8 and some basic set manipulation. Detailed proofs can be found in Appendix A. □

Example 4.2 (Abstract conformance).
Consider determining conformance to the predicate property in Figure 1*a – which, for simplicity, we will interchangeably refer to as* Π *for the rest of the example – given the assertions* $\mathcal{A} = \{A_1, \ldots, A_5\}$ *in* Figure 1*b. (Notice that the property formulas of both* Π *and* $\mathcal{A}$ *include elements of the abstract domain represented by the lattice in* Figure 1*c). Their corresponding sets of assertion conditions are:*

$$\Pi = \{\mathsf{calls}(\_(\mathtt{X},\mathtt{Y}),\ \mathtt{nat}(\mathtt{X})),\ \mathsf{success}(\_(\mathtt{X},\mathtt{Y}),\ \mathtt{nat}(\mathtt{X}),\ \mathtt{nat}(\mathtt{Y}))\}$$
$$\mathcal{A} = \{\mathsf{calls}(Pred_i, Pre_i),\ \mathsf{success}(Pred_i, Pre_i, Post_i) \mid A_i \in \mathcal{A}\}$$

We aim to determine which predicates partially specified by $\mathcal{A}$ *abstractly conform to the predicate property* Π. *For each predicate* $Pred_i$ *and its associated* calls *and* success *assertion conditions, we: (1) determine abstract conformance to the anonymous* calls *condition of* Π; *(2) determine abstract conformance to the anonymous* success *condition of* Π; *and (3) determine abstract conformance to* Π:

Tables 1 and 2 summarize the abstract conformance analysis between the calls *and* success *assertion conditions of each predicate and those of* Π, *respectively. Specifically, for* calls *(in* Table 1*), we compare the preconditions and apply* Definition 4.7; *for* success *(in* Table 2*), we compare both pre- and post-conditions and apply* Definition 4.8.

As a summary, the only predicate that definitely conforms to `p_nat_nat` *is* n2n/2, *since both of its assertion conditions conform to* Π.

Table 1. *Abs. conf. on "calls" example with* $^{\circ}Pre = \texttt{nat(X)}$

$Pred_i$	$Pred_i$	Relation w. $^{\circ}Pre$	Abs. Conf.
n2n(X,Y)	nat(X)	nat(X) $=$ nat(X)	yes
a2n(X,Y)	atm(X)	atm(X) $\sqcap$ nat(X) $= \bot$	no
i2z(X,Y)	int(X)	int(X) $\sqsupseteq$ nat(X)	maybe
z2i(X,Y)	zero(X)	zero(X) $\sqsubseteq$ nat(X)	maybe
nz2n(X,Y)	negz(X)	negz(X) $\sqcap$ nat(X) $\neq \bot$ $\wedge$negz(X) $\not\sqsubseteq$ nat(X) $\wedge$negz(X) $\not\sqsupseteq$ nat(X)	maybe

Table 2. *Abs. conf. on "success" example with* $^{\circ}Pre = \texttt{nat(X)}$ *and* $^{\circ}Post = \texttt{nat(Y)}$

$Pred_i$	$Pred_i$	Relation w. $^{\circ}Pre$	$Post_i$	Relation w. $^{\circ}Post$	Abs. Conf.
n2n(X,Y)	nat(X)	nat(X) $=$ nat(X)	nat(Y)	nat(Y) $=$ nat(Y)	yes
a2n(X,Y)	atm(X)	atm(X) $\sqcap$ nat(X) $= \bot$	nat(Y)	nat(Y) $=$ nat(Y)	maybe
i2z(X,Y)	int(X)	int(X) $\sqsupseteq$ nat(X)	zero(Y)	zero(Y) $\sqsubseteq$ nat(Y)	yes
z2i(X,Y)	zero(X)	zero(X) $\sqsubseteq$ nat(X)	int(Y)	int(Y) $\sqsupseteq$ nat(Y)	maybe
nz2n(X,Y)	negz(X)	negz(X) $\sqcap$ nat(X) $\neq \bot$ $\wedge$negz(X) $\not\sqsubseteq$ nat(X) $\wedge$negz(X) $\not\sqsupseteq$ nat(X)	nat(Y)	nat(Y) $=$ nat(Y)	maybe

4.2 Wrappers

Consider a predicate p and a predicate property Π s.t. p can be covered by Π. From Definition 4.4 we know that given their respective preconditions *Pre* and $^{\circ}Pre$, $Pre^{\natural} \supseteq {}^{\circ}Pre^{\natural}$. Thus, according to Definition 4.7, p *may* abstractly conform to Π ($p \prec^{\sharp+} \Pi$), since *Pre* may describe *more* admissible call states for p than $^{\circ}Pre$, which can lead to omitting some run-time check errors that would be raised by $^{\circ}Pre$.

Example 4.3 (Weak abstract conformance).
Consider a query `?- foo(even)` *to the following program.*

```
1 p_nat := { :- pred _(N) : nat(N). }. % 1-ary predicates for naturals.
2
3 :- pred even(N) : int(N).            :- pred foo(P) : p_nat(P).
4 even(N) :-                           foo(P) :- P(10).  % (1)
5     integer(N), 0 is N mod 2.        foo(P) :- P(-10). % (2)
```

Take a derivation of such query that starts by reducing to the body of the first clause (1)*: no* calls *assertion condition violation is expected since all predicates that conform to* `p_nat` *must accept all natural numbers on calls. Now, take a derivation that reduces to the body of the second clause* (2)*: a* calls *assertion condition violation is expected since all predicates that conform to* `p_nat` *must raise an error for any input different from a natural number. However, in this particular case, no error is raised, since the predicate* `even/1` *accepts any integer on calls. Moreover, if we had:*

```
6 p_neg := { :- pred _(N) : neg(N). }. % 1-ary predicates for negatives.
7
8 :- pred bar(P) : p_neg(P).
9 bar(P) :- P(-4).
```

then a clause like `foo(P) :- bar(P).` *would be problematic. For this clause, looking at the assertion of* `foo(P)`*, the predicate in* `P` *is required to conform to* **`p_nat`***, which would be an error when calling* `bar(P)` *since* **`p_neg`** *is disjoint from* **`p_nat`***. However, if we consider the particular case in which* `P` = `even`*, it may not. So, according to* Definition 4.7*, we could only conclude that* `even/1` $\prec^{\sharp+}$ **`p_nat`**, *that is,* `even/1` *may abstractly conform.*

In the example above, we motivate the need for such a restrictive condition for abstract conformance on *calls* (see Definition 4.7). However, we may want to use predicates whose set of admissible calls is greater than that of a predicate property, but without unexpected behavior. To this end, we propose a technique to restrict the set of admissible calls of a predicate p described by *Pre* to match that of $^\circ$*Pre* in a program analysis-friendly manner. This restriction is implemented using *wrappers*. A wrapper for p with Π is simply a new predicate $w(\bar{v})$ `:-` $p(\bar{v})$ with an assertion "**`:- pred`** $w(\bar{v})$ `:` $^\circ$*Pre*`.`" (note that fields of *pred* assertions, in this case the postcondition, can be omitted, equivalently to *true*). A wrapper for p with Π also makes explicit the intention of creating a Π-tailored version of p. Additionally, wrappers can also be used to alleviate the process of determining abstract conformance on *calls* (particularly useful in the implementation), since the wrapper would syntactically (and thus, semantically) match the precondition of the predicate property.

Example 4.4 (Wrapper).
As a follow-up of the previous example, consider wrapping `even/1` *with* **`p_nat`***:*

```
10 :- pred even_nat(N) : nat(N).
11 even_nat(N) :- even(N).
```

Intuitively, `even_nat/1` *conforms to* **`p_nat`***, and the analyzer can now infer that the clause* `foo(P) :- bar(P).` *should raise an error since* `even_nat/1` *only accepts naturals.*

4.2.1 Rationale for explicit wrappers

The design of *Hiord*$^\sharp$ follows the philosophy behind the Ciao system (Hermenegildo *et al.* 2012), which extends Prolog with static and dynamic assertion checking (among other modular extensions) without altering its untyped nature. We also considered some alternative solutions to the problem at hand, such as *tainting* each predicate passed as an argument annotated with a *predicate property*, and restricting its future use in all internal (recursive) calls. However, this approach would have required modifying the standard Prolog semantics regarding higher-order calls. The use of *wrappers* allows us to simulate this behavior without altering the underlying semantics.

4.3 First-order representation of predicate properties

As mentioned in §2, the abstract interpretation-based static analyzer can *infer* properties about higher-order programs, and also *verify* first-order assertions. However, here we

Algorithm 1 [*Hiord*$^\sharp$]: Verify a higher-order program with higher-order assertions

Input: Program: P, assertions: $\mathcal{A}$, abstract queries: $\mathcal{Q}^\sharp$
Output: Verified status (`checked`/`false`/`check`) for the assertions $\mathcal{A}$ of P: $\mathcal{V}$

1: $R \leftarrow \varnothing$
2: **repeat** ▷ *start fixpoint computation*
3: $R' \leftarrow R$ ▷ *save state to check fixpoint convergence*
4: **for all** predicate property $\Pi \in P$ **do**
5: $R \leftarrow R \cup \{\pi^-(p) \mid p \in P \wedge p \prec^{\sharp-} \Pi\} \cup \{\pi^+(p) \mid p \in P \wedge p \prec^{\sharp+} \Pi\}$ ▷ *regtypes*
6: **end for**
7: **until** $R = R'$ ▷ *fixpoint reached*
8: $\mathcal{V} \leftarrow \mathsf{acheck}(\mathcal{A}, [\![P \cup R]\!]^\sharp_{\mathcal{Q}^\sharp})$ ▷ *first-order assertion checking process*

obviously need to deal with predicate properties in assertions. Usually, for a new type of property, a new abstract domain is needed. As an alternative approach, we herein propose representing predicate properties as first-order properties of a kind which can be natively supported by the analyzer, thus allowing us to leverage existing and mature abstract domains. More concretely, we propose representing predicate properties as *regular types*, a special kind of properties (and thus defined as predicates) that are used to describe the *shape* of a term. Intuitively, such types will capture sets of predicate names. For example, given the predicate property `pp`, we can represent that the predicates `p`, and `q` strongly, and `r` weakly conforms to `pp` as the following regular types: $\texttt{pp}^-\texttt{/1} = \{\texttt{pp}^-\texttt{(p)}, \texttt{pp}^-\texttt{(q)}\}$ and $\texttt{pp}^+\texttt{/1} = \{\texttt{pp}^+\texttt{(p)}, \texttt{pp}^+\texttt{(q)}, \texttt{pp}^+\texttt{(r)}\}$.[7] Formally, given a predicate property Π, we define two associated regular types: π^- `/1` and π^+ `/1`, that capture the set of predicates that *strongly* and *weakly* abstractly conform to Π as follows: $\pi^-\texttt{/1} = \{\pi^-(p) \mid p \in P \wedge p \prec^{\sharp-} \Pi\}$, and $\pi^+\texttt{/1} = \{\pi^+(p) \mid p \in P \wedge p \prec^{\sharp+} \Pi\}$. By definition, π^- `/1` is a *subtype* of π^+ `/1`. These regular types reduce the compile-time checking of higher-order assertions to that of first-order assertions. Regular types can be abstracted and inferred by several abstract domains; for concreteness we use `eterms` (Vaucheret and Bueno 2002).

4.4 *Hiord*$^\sharp$ *algorithm*

We now present *Hiord*$^\sharp$, the core algorithm for the compile-time verification of a higher-order program P with higher-order assertions $\mathcal{A}$ (Algorithm 1). First, it initializes a set of rules R, and it computes the regular type representations of each predicate property Π in P, *that is*, the π^- and π^+ predicates respectively (lines 4 to 6). This computation is performed by directly applying Definitions 4.7 to 4.9 using the operators of the abstract domain, and (implicitly) extending the program P with R. Since predicate properties can include predicate property literals from other predicate properties – *that is*, *dependencies* among predicate properties – lines 4 to 6 are repeated until a *fixpoint* is reached (lines 2 and 7). Next, it computes the abstract interpretation of P, augmented with the regular

[7] Or, using Ciao's functional notation: "`:- regtype pp+/1. pp+ := p | q | r.`."

type representations of every predicate property, for the set of abstract queries $\mathcal{Q}^\sharp$ (line 8). Finally, it performs the compile-time verification of the set of (now first-order) assertions $\mathcal{A}$ w.r.t. the static analysis results, where predicate properties are now treated as standard regular types (line 8). As the result of the algorithm, we obtain the verified status of each assertion of $\mathcal{A}$, where each assertion can be discharged (**checked**), disproved (**false**) and an error flagged, or left in **check** status, and subject to run-time checks, as in Stulova *et al.* (2014). We argue that, despite the inherent complexity of the verification problem in hand, the proposed concepts make the compile-time checking algorithm clear and concise; and, more importantly, easily implementable using a first-order assertion checker.

5 Implementation and experiments

To demonstrate the potential of our approach, we have implemented a prototype of the *Hiord*$^\sharp$ technique as part of the Ciao system. It implements Algorithm 1 and uses CiaoPP, the Ciao program preprocessor, with the eterms abstract domain. We ran experiments on a set of small but representative higher-order programs that were not possible to verify until this point. We illustrate below our experiments with a selection of these programs.

5.1 A synthetic benchmark

We started by defining a test case comprising a predicate property using an anonymous *pred* assertion and 25 predicates, each with a *pred* assertion, designed to exhaustively cover all possible orderings between the pre- and post-conditions of the predicate property and of each predicate. We then ran *Hiord*$^\sharp$, obtaining the correct results that 2 predicates *definitely did conform* and 7 predicates *definitely did not conform*, with 16 predicates left where no definite conclusion could be reached.

5.2 Higher-order list utilities

We defined various partially specified higher-order utility predicates specialized for working with lists of a particular type **t**, *for example*, **t(X) :- num(X)**. For example, consider the **t_cmp** predicate property defined below:

```
1 | t_cmp := { :- pred _(X,Y) : (t(X), t(Y)). }.
```

which describes *comparator* predicates of elements of type **t**, that we then use in the higher-order assertion for a comparator-parameterizable *quicksort* implementation:

```
2 | :- pred qsort(Xs,P,Ys) : (list(t,Xs), t_cmp(P)) => list(t,Ys).
```

and where the `partition/4` predicate includes a call `P(X,Y)`. The analysis is able to propagate the **t_cmp** property on `P` to that point, and if in `P(X,Y)`, `X` is inferred to be bound to, for example, `a`, an error is statically captured by *Hiord*$^\sharp$. Consider the following comparators:

```
3 | :- pred lex(X,Y) : (term(X), term(Y)).
4 | lex(X,Y) :- X @< Y.
```

```
5 | :- pred lex_t(X,Y) : (t(X), t(Y)).
6 | lex_t(X,Y) :- lex(X,Y).
```

For a query `?-qsort(Xs,lex,Ys)`, *Hiord*$^\sharp$ reports a warning on *calls* since `lex/2` $\prec^{\sharp+}$ **`t_cmp`** (it weakly abstractly conforms). Intuitively, `lex/2` is not definitely conformant since it will not raise a run-time check error when called with a term that is not of type **`t`**, introducing unexpected behavior. For a query `?-qsort(Xs,lex_t,Ys)` *Hiord*$^\sharp$ proves that it behaves correctly w.r.t. its higher-order assertion, since `lex_t/2` $\prec^{\sharp-}$ **`t_cmp`**.

Additionally, consider a predicate property which represents a parameterizable sorter of lists of elements of type **`t`**, defined "in terms of" the **`t_cmp`** predicate property:

```
7 | t_sort := { :- pred _(Xs,C,Ys) : (list(t,Xs), t_cmp(C)) => list(t,Ys). }.
```

For determining that `qsort/3` $\prec^{\sharp-}$ **`t_sort`**, *Hiord*$^\sharp$ would need to perform an additional iteration of the fixpoint computation after the one above, that is, after computing the predicates that weakly or strongly abstractly conform to the **`t_cmp`** predicate property.

5.3. HTTP *server*

Consider the following schematic HTTP server, parameterized by a predicate that must be able to handle four REST operations. We use regular types for representing requests and responses, and a predicate property for representing handlers:

```
1 | handler := { :- pred _(Rq,Rs) : req(Rq) => res(Rs). }.
2 |
3 | :- regtype req/1. req := 'DELETE' | 'GET' | 'POST' | 'PUT'.
4 | :- regtype res/1. res := 'OK' | 'CREATED' | 'BAD_REQUEST' | 'NOT_FOUND'.
```

and we add the following higher-order assertion to the server predicate:

```
5 | :- pred server(H,Rq,Rs) : (handler(H), req(Rq)) => res(Rs).
```

Hiord$^\sharp$ detects that the predicate `h/2` does not definitely conform to **`handler`** (`h/2` $\prec^{\sharp+}$ **`handler`**) due to one its clauses:

```
6 | h('PUT', Rs) :- ..., Rs = 'BAD_REQ'. % Bug, should be 'BAD_REQUEST'
```

5.4. Dutch national flag

This problem involves sorting a list of red, white, or blue elements, such that elements of the same color are grouped together in a specified order (typically red, then white, then blue). However, we want to generalize the solution by allowing the user to provide a comparator that, given two elements, yields their comparison. We first define regular types to represent colored elements and the result of their comparison:

```
1 | :- regtype rwb/1. rwb := r | w | b.
```

```
2 | :- regtype lge/1. lge := < | > | = .
```

Next, we define a **`dutch_cmp`** predicate property describing comparators between **`rwb`** elements; and provide a higher-order assertion to the `dutch_flag/3` higher-order predicate:

```
3 | dutch_cmp := { :- pred _(X,R,Y) : (rwb(X), rwb(Y)) => lge(R). }.
4 | :- pred dutch_flag(C,Xs,Ys) : (dutch_cmp(C), list(rwb,Xs)) => list(rwb,Ys).
```

Assume that we are given the implementation of `dutch_flag/3` and we need to provide a comparator `cmp/3` which conforms to **`dutch_cmp`**. Consider a first implementation attempt:

```
6 cmp(red,=,  red). cmp(white,=,white). cmp(blue,=, blue).
7 cmp(red,<,white). cmp(white,>,  red). cmp(blue,>,  red).
8 cmp(red,<, blue). cmp(white,<, blue). cmp(blue,>,white).
```

```
1 :- regtype rt1/1.
2 rt1 := red | blue
3      | white.
```

When determining its conformance to **`dutch_cmp`**, *Hiord*$^\sharp$ finds that `cmp/3` $\not\prec^{\sharp+}$ **`dutch_cmp`**, since CiaoPP infers the regular type **`rt1`** for the elements to compare, and **`rt1`** $\sqcap$ **`rwb`** $= \bot$ in eterms. We proceed by correcting it, but we accidentally mistype some of the `r` elements for `o` elements in lines 7 and 8; and CiaoPP infers the following assertion and regular type:

```
6 :- pred cmp(X,R,Y) : (rt2(X), rwb(Y)) => lge(R).
7 cmp(o,=,r). cmp(w,=,w). cmp(b,=,b).
8 cmp(o,<,w). cmp(w,>,r). cmp(b,>,r).
9 cmp(r,<,b). cmp(w,<,b). cmp(b,>,w).
```

```
1 :- regtype rt2/1.
2 rt2 := r | b
3      | w | o.
```

However, *Hiord*$^\sharp$ now reports that `cmp/3` $\prec^{\sharp+}$ **`dutch_cmp`**, since it would not raise a run-time check error when called with an `o` on its first argument. Formally, **`(rt2(X), rwb(Y))`** $\sqsupseteq$ **`(rwb(X), rwb(Y))`** in eterms. In an attempt at improving the precision of the ordering, we refine `cmp/3` to yield more informative results on the order relation between elements:

```
6 :- pred cmp(X,R,Y) : (rwb(X), rwb(Y)) => lgLGe(R).
7 cmp(r, =,r). cmp(w,=,w). cmp(b, =,b).
8 cmp(r, <,w). cmp(w,>,r). cmp(b,>>,r).
9 cmp(r,<<,b). cmp(w,<,b). cmp(b, >,w).
```

```
10 :- regtype lgLGe/1.
11 lgLGe :=  < |  >
12        | << | >>
13        |  = .
```

In particular, we introduce **`<<`** and **`>>`** to reflect that `X` is "much lower" than `Y`, and vice-versa; and define a new regular type and assertion. However, *Hiord*$^\sharp$ still reports that `cmp/3` $\prec^{\sharp+}$ **`dutch_cmp`**, since `cmp/3` may yield comparison results that are not reflected in **`lge`**. Formally **`lgLGe(R)`** $\sqsupseteq$ `lge(R)` in eterms. Finally, we develop the following comparator:

```
6 :- pred cmp(X, R, Y) : (rwb(X), rwb(Y)) => lge(R).
7 cmp(r,=,r). cmp(w,=,w). cmp(b,=,b).
8 cmp(r,<,w). cmp(w,>,r). cmp(b,>,r).
9 cmp(r,<,b). cmp(w,<,b). cmp(b,>,w).
```

And *Hiord*$^\sharp$ proves that `cmp/3` $\prec^{\sharp-}$ **`dutch_cmp`**, since it behaves exactly as expected.

6 Conclusions

We have presented *Hiord*$^\sharp$, a novel approach for the compile-time verification of higher-order (C)LP programs with higher-order assertions. We started by refining both the syntax and semantics of predicate properties. Then, we introduced an abstract criterion to determine whether a predicate conforms with a predicate property at compile time. We also motivated and explained a *wrapper*-based technique for "casting" predicate usage in a program analysis-friendly manner that enhances and complements the proposed abstract criterion. We then proposed a technique for dealing with these properties using an abstract interpretation-based static analyzer for programs with first-order assertions. Finally, we reported on a prototype implementation and studied the effectiveness of the approach with various examples within the Ciao system. We believe our proposal

constitutes a practical approach to closing the existing gap in the verification at compile time of higher-order assertions; and that it is quite general and flexible, and can be applied, at least conceptually, to other similar gradual approaches.

Competing interests

The authors declare none.

Supplementary material

Supplementary material for this paper can be found at: https://doi.org/10.1017/S147106842510015X.

References

Cabeza, D., Hermenegildo, M. and Lipton, J. 2004. Hiord: A Type-free higher-order logic programming language with predicate abstraction. In *ASIAN'04*, Springer-Verlag, Vol. 3321 in *LNCS*, 93–108.

Cardelli, L. 1989. Typeful programming. In *Formal Description of Programming Concepts*, 1989, IFIP State-of-the-Art Reports, E. J. Neuhold and M. Paul, Eds. Springer, 431–498.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, ACM Press, 238–252.

Debray, S., Lopez-Garcia, P. and Hermenegildo, M. 1997. Non-failure analysis for logic programs. In *1997 International Conference on Logic Programming*, MIT Press, Cambridge, MA, 48–62.Cambridge, MA

Hermenegildo, M., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming* 12, 219–252.

Hermenegildo, M., Puebla, G. and Bueno, F. 1999. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In *The Logic Programming Paradigm: A 25–Year Perspective*, Springer-Verlag, 161–192.

Hermenegildo, M., Puebla, G., Bueno, F. and Garcia, P. L. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and The Ciao System Preprocessor). *Science of Computer Programming* 58, 1-2, 115–140.

Hill, P. and Lloyd, J. 1994. *The Goedel Programming Language*. MIT Press, Cambridge MA.

Körner, P., Leuschel, M., Barbosa, J., Santos-Costa, V., Dahl, V., Hermenegildo, M. V., Morales, J. F., Wielemaker, J., Diaz, D., Abreu, S. and Ciatto, G. 2022. Fifty years of prolog and beyond. *Theory and Practice of Logic Programming* 22, 6, 776–858, 20th Anniversary Special Issue.

Marlow, S. 2010. *Haskell 2010 language report*. Haskell Language Committee.

Miller, D. 1991. A logic programming language with λ-abstraction, function variables, and simple unification. *Journal of Logic and Computation* 1, 4, 497–536.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming* 13, 2-3, 315–347.

Puebla, G., Bueno, F. and Hermenegildo, M. 2000a. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming*, Springer-Verlag, Vol. 1870 in LNCS, 23–61.

Puebla, G., Bueno, F. and Hermenegildo, M. 2000b. Combined static and dynamic assertion-based debugging of constraint logic programs. In *Proc. of LOPSTR'99*, Springer-Verlag, Vol. 1817 LNCS, 273–292.

Sanchez-Ordaz, M., Garcia-Contreras, I., Perez-Carrasco, V., Morales, J. F., Lopez-Garcia, P. and Hermenegildo, M. 2021. VeriFly: On-the-fly assertion checking via incrementality. *Theory and Practice of Logic Programming* 21, 6, 768–784.

Siek, J. G. and Taha, W. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 81–92. University of Chicago Department of Computer Science.

Somogyi, Z., Henderson, F. and Conway, T. 1996. The execution algorithm of mercury: An efficient purely declarative logic programming language. *Journal of Logic Programming* 29, 1-3, 17–64.

Stulova, N., Morales, J. and Hermenegildo, M. 2014. Assertion-based debugging of higher-order (C)LP programs. In PPDP'14, ACM, 225–235.

Stulova, N., Morales, J. F. and Hermenegildo, M. 2018. Some trade-offs in reducing the overhead of assertion run-time checks via static analysis. *Science of Computer Programming* 155, 3–26.

Vaucheret, C. and Bueno, F. 2002. More precise yet efficient type inference for logic programs. In LNCS *SAS'02*, Springer, Vol. 2477 in LNCS, 102–116.