Assertion-based Debugging of Higher-Order (C)LP Programs

Nataliia Stulova 1  José F. Morales 1
1IMDEA Software Institute
{nataliia.stulova,josef.morales,manuel.hermenegildo}@imdea.org

Manuel V. Hermenegildo 1,2
2School of Computer Science
Technical University of Madrid (UPM)
manuel.hermenegildo@upm.es

Abstract
Higher-order constructs extend the expressiveness of first-order (Constraint) Logic Programming ((C)LP) both syntactically and semantically. At the same time assertions have been in use for some time in (C)LP systems helping programmers detect errors and validate programs. However, these assertion-based extensions to (C)LP have not been integrated well with higher-order to date. This paper contributes to filling this gap by extending the assertion-based approach to error detection and program validation to the higher-order context within (C)LP. We propose an extension of properties and assertions as used in (C)LP in order to be able to fully describe arguments that are predicates. The extension makes the full power of the assertion language available when describing higher-order arguments. We provide syntax and semantics for (higher-order) properties and assertions, as well as for programs which contain such assertions, including the notions of error and partial correctness. We also discuss several alternatives for performing run-time checking of such programs.

Categories and Subject Descriptors D.1.6 [Programming Techniques]: Logic Programming; D.2.4 [Software Engineering]: Software/Program Verification—assertion checkers, validation; D.3.3 [Programming Languages]: Language Constructs and Features—constraints; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—assertions, pre- and post-conditions, specification techniques; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—operational semantics.

Keywords Logic Programming, Constraint Logic Programming, Higher-order, Validation, Assertions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPDP ‘14, September 08–10, 2014, Canterbury, UK.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm . . . $15.00.
http://dx.doi.org/10.1145/nnnn.nnnn.nnnn

1. Introduction
Higher-order programming adds flexibility to the software development process. Within the (Constraint) Logic Programming ((C)LP) paradigm, Prolog has included higher-order constructs since the early days, and there have many other proposals for combining the first-order kernel of (C)LP with different higher-order constructs (see, e.g., [4, 8, 22, 29]). Many of these proposals are currently in use in different (C)LP systems and have been found very useful in programming practice, inheriting the well-known benefits of code reuse (templates), elegance, clarity, and modularization.

A number of extensions have also been proposed for (C)LP in order to enhance the process of error detection and program validation. In addition to the use of classical strong typing [15, 27], a number of other approaches have been proposed which are based on the dynamic and/or static checking of user-provided, optional assertions [3, 10, 13, 16, 19, 21, 23, 26]. In practice, different aspects of the model of [13, 26] have been incorporated in a number of widely-used (C)LP systems, such as Ciao, SWI, and XSB [14, 20, 28]. A similar evolution is represented by the soft/gradual typing-based approaches in functional programming and the contracts-based extensions in object-oriented programming [7, 17, 19, 29].

These two aspects, assertions and higher-order, are not independent. When higher-order constructs are introduced in the language it becomes necessary to describe properties of arguments of predicates that are themselves also predicates. While the combination of contracts and higher-order has received some attention in functional programming [9, 11], within (C)LP the combination of higher-order with the previously mentioned assertion-based approaches has received comparatively little attention to date. Current Prolog systems simply use basic atomic types (i.e., stating simply that the argument is a pred, callable, etc.) to describe predicate-bearing variables. The approach of [2] is oriented to meta programming. It allows describing meta-types but there is no notion of directionality (modes), and only a single pattern is allowed per predicate.

This paper contributes to filling the existing gap between higher-order and assertions in (C)LP. Our starting point is the Ciao assertion model [13, 26], since, as mentioned before, it has been adopted at least in part in a number of the most popular (C)LP systems. After some preliminaries and notation (Section 2) we start by extending the traditional notion of programs and derivations in order to deal with higher-order calls and recall and adapt the notions of first-
order conditional literals, assertions, program correctness, and run-time checking to this type of derivations (Section 4). This part allows us to revisit the traditional model in this new, higher-order context, while introducing a different formalization than the original one of \cite{26}. This formalization, which will be used throughout the paper, is more compact and gathers all assertion violations as opposed to just the first one, among other differences. We then define an extension of the properties used in assertions and of the assertions themselves to higher-order, and provide corresponding semantics and results (Section 3).

2. Preliminaries and Notation

We recall some concepts and notation from standard (C)LP theory. We denote by $VS$, $FS$, and $PS$ the set of variable, function, and predicate symbols, respectively. Variables start with a capital letter. Each $p \in PS$ and $f \in FS$ is associated to a natural number called its arity, written $ar(p)$ or $ar(f)$. The set of terms $TS$ is inductively defined as follows: $VS \subseteq TS$, if $f \in FS$ and $t_1, \ldots, t_n \in TS$ then $f(t_1, \ldots, t_n) \in TS$ where $ar(f) = n$. An atom has the form $p(t_1, \ldots, t_n)$ where $p \in PS$, $ar(p) = n$, and $t_1, \ldots, t_n \in TS$. A constraint is essentially 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. A goal is a finite sequence of literals. A rule is of the form $H :- B$ where $H$, the head, is an atom and $B$, the body, is a possibly empty finite sequence of literals. A constraint logic program, or program, is a finite set of rules.

We use $\sigma$ to represent a variable renaming and $\sigma(X)$ to represent the result of applying the renaming $\sigma$ to some syntactic object $X$ (a term, atom, literal, goal, etc.). The definition of an atom $A$ in a program, $def(A)$, is the set of variable renamings of the program rules such that each renaming has $A$ as a head and has distinct new local variables. We assume that all rule heads are normalized, i.e., $H$ is of the form $p(X_1, \ldots, X_n)$ where the $X_1, \ldots, X_n$ are distinct free variables. This is not restrictive since programs can always be normalized, and it facilitates the presentation. However, for conciseness in the examples we sometimes use non-normalized programs. Let $\exists \theta \phi$ be the constraint $\phi$ restricted to the variables of the syntactic object $L$. We denote constraint entailment by $\models$, so that $\theta_1 \models \theta_2$ denotes that $\theta_1$ entails $\theta_2$. In such case we say that $\theta_2$ is weaker than $\theta_1$.

For brevity, we will assume in the rest of the paper that we are dealing with a single program, so that all sets of rules, etc. refer to that implicit program and it is not necessary to refer to it explicitly in the notation.

2.1 Operational Semantics

The operational semantics of a program is given in terms of its “derivations”, which are sequences of reductions between “states”. A state $(G | \theta)$ consists of a goal $G$ and a constraint store (or store for short) $\theta$. We use $::$ to denote concatenation of sequences and we assume for simplicity that the underlying constraint solver is complete. We use $S \rightarrow S'$ to indicate that a reduction can be applied to state $S$ to obtain state $S'$. Also, $S \rightarrow^* S'$ indicates that there is a sequence of reduction steps from state $S$ to state $S'$. We denote by $D_{i[1]}$ the $i$-th state of the derivation. As a shorthand, given a non-empty derivation $D$, $D_{i[-1]}$ denotes the last state. A query is a pair $(L, \theta)$, where $L$ is a literal and $\theta$ a store, for which the (C)LP system starts a computation from state $(L | \theta)$. The set of all derivations from the query $Q$ is denoted $derivs(Q)$. The observational behavior of a program is given by its “answers” to queries. A finite derivation from a query $(L, \theta)$ is finished if the last state in the derivation cannot be reduced. Note that $derivs(Q)$ contains not only finished derivations but also all intermediate derivations from a query. A finished derivation from a query $(L, \theta)$ is successful if the last state is of the form $(\Box | \theta')$, where $\Box$ denotes the empty goal sequence. In that case, the constraint $\exists \theta \theta'$ is an answer to $S$. We denote by $answers(Q)$ the set of answers to a query $Q$. A finished derivation is failed if the last state is not of the form $(\Box | \theta')$. A query $Q$ finitely fails if $derivs(Q)$ is finite and contains no successful derivation.

3. First-order Assertions on Higher-order Derivations

3.1 Higher-order Programs and Derivations

We start by extending the definition of program, state reduction, and derivations in order to deal with the syntax and semantics of higher-order calls.

Definition 1. Higher-order programs are a generalization of constraint logic programs where:

- The set of literals $LS$ is extended to include higher-order literals $X(t_1, \ldots, t_n)$, where $X \in VS$ and the $t_i \in TS$.
- The set of terms $TS$ is extended so that $PS \subseteq TS$ (i.e., predicate symbols $p$ can be used as constants).

In the following we assume a simple semantics where when a call to a higher-order literal $X(t_1, \ldots, t_n)$ occurs, $X$ has to be constrained to a predicate symbol in the store.

Definition 2. A state $S = (L :: G | \theta)$ where $L$ is a literal can be reduced to a state $S'$, denoted $S \rightarrow S'$, as follows:

1. If $L$ is a constraint and $\theta \land L$ is satisfiable, then $S' = (G | \theta \land L)$.
2. If $L$ is an atom of the form $p(t_1, \ldots, t_n)$, for some rule $(L :: B) \in def(L)$, then $S' = (B :: G | \theta)$.
3. If $L$ is of the form $X(t_1, \ldots, t_n)$, then $S' = (G' | \theta)$ where either $G' = p(t_1, \ldots, t_n) :: G$ if $\exists p \in PS \land \theta \models (X = p) \land ar(p) = n$, or $G' = \epsilon_{\mininstcall}$ otherwise.

The concepts of answers and of finished and successful derivations carry over without change to this notion of higher-order derivations. The notion of (finitely) failed derivation is extended as follows:

Definition 3. A finished derivation from a query $(L, \theta)$ is failed iff its last state is not of the form $(\Box | \theta')$ or $(\epsilon_{\mininstcall} | \theta')$.

Finally, we introduce the concept of floundered derivations:

Definition 4. A finished derivation from a query $(L, \theta)$ is floundered iff its last state is of the form $(\epsilon_{\mininstcall} | \theta')$.

3.2 First-order Pred Assertions

Assertions are linguistic constructions for expressing properties of programs. They are used for detecting deviations of

\footnote{This is also the most frequent semantics in current systems. Other alternatives, such as residuation ( delays), predicate enumeration, etc. can also be used, requiring relatively straightforward adaptations of the model proposed.}
the program behavior (symptoms) with respect to such assertions, or to ensure that no such deviations exist (correctness). Herein, we will use the pred assertions of [23], given that they are the most frequently used assertions in practice, and they subsume the other assertion schemas in that language. Thus, in the following we will use simply the term assertion to refer to a pred assertion. Assertions allow specifying certain conditions on the constraint store that must hold at certain points of program derivations. In particular, they allow stating sets of preconditions and conditional postconditions for a given predicate. A set of assertions for a predicate is of the form:

\[
\vdash \text{pred Head : Pre}_i \Rightarrow \text{Post}_i.
\]

\[
\vdash \text{pred Head : Pre}_n \Rightarrow \text{Post}_n.
\]

where Head is a normalized atom that denotes the predicate that the assertions apply to, and the \(\text{Pre}_i\) and \(\text{Post}_i\) refer to the variables of Head. We assume that variables in assertions are renamed such that the Head atom is identical for all assertions for a given predicate. A set of assertions as above states that in any execution state (Head \(\vdash \theta\)) at least one of the \(\text{Pre}_i\) conditions should hold, and that, given the \(\text{Pre}_i,\text{Post}_i\) pair(s) where \(\text{Pre}_i\) holds, then, if Head succeeds, the corresponding \(\text{Post}_i\) should hold upon success. The following example illustrates the basic concepts involved:

**Example 1.** The procedure qsort(A,B) is the usual one that relates lists A and their sorted versions B. The following assertions:

\[
\vdash \text{pred qsort}(A,B) : \text{list}(A) \Rightarrow \text{sorted}(B), \text{list}(B).
\]

\[
\vdash \text{pred qsort}(A,B) : \text{list}(B) \Rightarrow \text{permutation}(B,A), \text{list}(A).
\]

state that (restrict the meaning of qsort to):

- qsort(A,B) should be called either with A constrained to a list or with B constrained to a list;
- if qsort(A,B) succeeds when called with A constrained to a list then on success B should be a sorted list; and
- if qsort(A,B) succeeds when called with B constrained to a list then on success A should be a list which is a permutation of B.

### 3.3 Conditions on the Constraint Store

The conditions on the constraint store used in assertions are specified by means of special literals (e.g., \(\text{list}(A)\), \(\text{sorted}(B)\), \(\text{list}(B)\), \(\text{permutation}(B,A)\) in the previous example) that we will herein call prop literals. More concretely, we assume the \(\text{Pre}_i\) and \(\text{Post}_i\) to be DNF formulas of such literals.

We also assume that for each prop literal \(L_p\) used in some assertion there exists a corresponding predicate \(p\) defining it. Then, we can define the meaning of prop literals as follows:

**Definition 5.** The meaning of a prop literal \(L_p\) defined by predicate \(p\), denoted \([L_p]\), is the set of constraints given by \(\text{answers}(L_p, \text{true})\).

Intuitively, the meaning of prop literals is the set of “weakest” constraints for which the literal holds.

**Example 2.** Prop literals \(\text{list}/1\) and \(\text{sorted}/1\) can be defined by:

\[
\text{list}([]) \cdot \text{list}([\_], \_L) \cdot \text{list}(\_L).
\]

\[
\text{sorted}([], \_L) \cdot \text{sorted}(\_L).
\]

Then, their meaning is given by \([\text{list}(A)]\) = \{A = [], A = \[B|C\] \& \text{list}(C)\} and \([\text{sorted}(A)]\) = \{A = [], A = \[B|A\] = \[B|C|D\] \& \text{sorted}(E)\}.

The following definition from [26] defines when the condition represented by a prop literal (defined by a program predicate) holds for a given store:

**Definition 6.** A prop literal \(L\) succeeds trivially for \(\theta\), denoted \(\theta \Rightarrow_p L\), if \(\exists \theta' \in \text{answers}(L, \theta)\) such that \(\theta' \models \theta\). A DNF formula of prop literals succeeds trivially for \(\theta\) if all of the prop literals of at least one conjunct of the formula succeeds trivially.

Intuitively, a prop literal \(L\) succeeds trivially if \(L\) succeeds for \(\theta\) without adding new “relevant” constraints to \(\theta\):

**Example 3.** Consider prop literals \(\text{list}(A)\) and \(\text{sorted}(B)\) and the predicate definitions of Example 2.

- Assume that \(\theta = (A = f)\). Since \(\forall \theta' \in [\text{list}(A)] : \theta' \neq \theta\), as we would expect, \(\theta \not\Rightarrow_p \text{list}(A)\).
- Assume now that \(\theta = (A = [\_|X|])\). Though \(A\) is compatible with a list, it is not actually a (nil terminated) list. Again in this case \(\forall \theta' \in [\text{list}(A)] : \theta' \neq \theta\) and thus again \(\theta \not\Rightarrow_p \text{list}(A)\). The intuition behind this is that we cannot guarantee that \(A\) is actually a list given \(\theta\), since a possible instance of \(A\) in \(\theta\) is \(A = [\_|f]\), which is clearly not a list.
- Finally, assume that \(\theta = (A = [B]|B = 1)\). In such case \(\exists \theta' = (A = [B|C] \& \theta = c)\) such that \(\theta' \models \theta\) and \(\exists \theta = (B = 1)\) such that \((c \& \theta' \neq \text{false}) \& (\theta' \& c = \theta)\). Thus, in this last case \(\theta \Rightarrow_p \text{list}(A)\).

This means that we are considering prop literals as instantiation checks [12, 23], they are true iff the variables they check for are at least as constrained as their predicate definition requires.

**Definition 7.** A prop literal \(L\) is a test iff \(\forall \theta\) either \(\theta \Rightarrow_p L\) or \((L, \theta)\) finitely fails.

### 3.4 First-order Assertion Conditions and their Semantics

We represent the different checks on the constraint store imposed by a set of assertions as a set of assertion conditions as follows.

**Definition 8.** Given a predicate represented by a normalized atom Head, if the corresponding set of assertions is \(A = \{A_1, ..., A_n\}\), with \(A_i \Leftarrow \vdash \text{pred Head} : \text{Pre}_i \Rightarrow \text{Post}_i\) for the set of assertion conditions for Head is \(\{C_0, C_1, ..., C_n\}\), with:

\[
C_i = \begin{cases} 
\text{calls}(\text{Head}, \bigvee_{i=1}^n \text{Pre}_i) & i = 0 \\
\text{success}(\text{Head}, \text{Pre}_i, \text{Post}_i) & i = 1..n 
\end{cases}
\]

If there are no assertions associated with Head then the corresponding set of conditions is empty. The set of assertion conditions for a program is the union of the assertion conditions for each of the predicates in the program. Also, given a single assertion \(A_i\), we define its corresponding set of assertion conditions as \(\{C_0, C_i\}\) (this will be useful in defining the status of an assertion).

The \(\text{calls}(\text{Head},...)\) conditions encode the checks that the calls to the predicate represented by Head are within those admissible by the set of assertions, and we thus call them the \textit{calls assertion conditions}. The \(\text{success}(\text{Head}, \text{Pre}_i, \text{Post}_i)\) conditions encode the checks for compliance of the successes for particular sets of calls, and we thus call them the \textit{success assertion conditions}.
Example 4. The assertion conditions corresponding to the predicate assertions for \texttt{qsort} in Example 7 are as follows:

\begin{itemize}
  \item \texttt{calls(qsort(A, B), list(A), list(B))}
  \item \texttt{success(qsort(A, B), list(A), sorted(B), list(B))}
  \item \texttt{success(qsort(A, B), list(B), (permutation(B, A), list(A)))}
\end{itemize}

In order to define the semantics of assertion conditions, we introduce the auxiliary partial functions \texttt{prestep} and \texttt{step} as follows:

\begin{align*}
\texttt{prestep}(L_a, D) &= (\theta, \sigma) \equiv D_{[-1]} = (L \triangleright G \mid \theta) \land \exists \sigma L = \sigma(L_a) \\
\texttt{step}(L_a, D) &= (\theta, \sigma, \theta') \equiv D_{[-1]} = (G \mid \theta') \land \exists \sigma L = \sigma(L_a) \\
&\quad \land \exists D_{[0]} = (L \triangleright G \mid \theta')
\end{align*}

Given a derivation whose current state is a call to \(L_a\) (normalized atom), the \texttt{prestep} function returns the substitution \(\sigma\) for \(L_a\), and the constraint store \(\theta\) at the predicate \texttt{call} (i.e., just before the literal is reduced). Given a derivation whose current state corresponds exactly to the return from a call to \(L_a\), the \texttt{step} function returns the substitution \(\sigma\) for \(L_a\), the constraint store \(\theta\) at the call to \(L_a\), and the constraint store \(\theta'\) at \(L_a\)'s \texttt{success} (i.e., just after all literals introduced from the body of \(L_a\) have been fully reduced).

Using these functions, the semantics of our \texttt{calls} and \texttt{success} assertion conditions are given by the following definition:

\textbf{Definition 9.} Given a \texttt{calls} or \texttt{success} assertion condition \(C\), the valuation of \(C\) on a derivation \(D\), denoted \texttt{solve}(C, D)\ is defined as follows:

\begin{align*}
\texttt{solve(calls}(L_a, \texttt{Pre}, D), D) &\equiv \texttt{prestep}(L_a, D) = (\theta, \sigma) \\
&\implies (\theta \Rightarrow \texttt{Pre}(\theta)) \\
\texttt{solve(success}(L_a, \texttt{Pre}, \texttt{Post}, D), D) &\equiv \texttt{step}(L_a, D) = (\theta, \sigma, \theta') \\
&\implies ((\theta \Rightarrow \texttt{Pre}(\theta)) \\
&\implies (\theta' \Rightarrow \texttt{Post}(\theta)))
\end{align*}

where \(L_a\) is a normalized atom.

\section{3.5 Status of Assertions and Partial Correctness}

As mentioned before, the intended use of our assertions is to perform debugging with respect to partial correctness, i.e., to ensure that the program does not produce unexpected results for valid (“expected”) queries. Thus, we extend our notion of program to include assertions and valid queries.

\textbf{Definition 10.} An annotated program is a tuple \((P, Q, A)\) where \(P\) is a (higher-order) constraint logic program (as defined in Section 3.3), \(Q\) is a set of valid queries, and \(A\) is a set of assertions. As before, \(A_C\) denotes the set of calls and success assertion conditions derived from \(A\).

In the context of annotated programs we extend \texttt{derivs} to operate on the set of valid queries as follows: \(\texttt{derivs}(Q) = \bigcup_{Q \in Q} \texttt{derivs}(Q)\). We now provide several simple definitions which will be instrumental:

\textbf{Definition 11.} Given the set of queries \(Q\), the assertion condition \(C\) can be either checked or false, as follows:

\begin{align*}
\texttt{checked}(C) &\equiv \forall D \in \texttt{derivs}(Q) \cdot \texttt{solve}(C, D) \\
\texttt{false}(C) &\equiv \exists D \in \texttt{derivs}(Q) \cdot \neg \texttt{solve}(C, D)
\end{align*}

\section{3.6 Operational Semantics for Higher-order Programs with First-order Assertions}

We now provide an operational semantics which checks whether assertion conditions hold or not while computing the (possibly higher-order) derivations from a query.

\textbf{Definition 14.} Given the atom \(L_a\) and the set of assertion conditions \(A_C, A_C^n(L_a)\) denotes the set of labeled assertion condition instances for \(L_a\) of the form \(c \# C_a\), such that \(3C \in A_C, C = \text{calls}(L, \text{Pre})\) (or \(C = \text{success}(L, \text{Pre}, \text{Post})\)), \(\sigma\) is a renaming s.t. \(L = \sigma(L_a), C_a = \text{calls}(L_a, \sigma(\text{Pre}))\) (or \(C_a = \text{success}(L_a, \sigma(\text{Pre}), \sigma(\text{Post}))\)), and \(c\) is an identifier that is unique for each \(C_a\).

In order to keep track of the violated assertion conditions, we introduce an extended program state of the form \((G \mid \theta \mid E)\), where \(E\) denotes the set of identifiers for falsified assertion condition instances. We also extend the set of literals with syntactic objects of the form \texttt{check}(c) where \(c\) is an identifier for an assertion condition instance, which we call check literals. Thus, a literal is now a constraint, an atom, a higher-order literal, or a check literal.

\textbf{Definition 15.} A state \(S = (L :: G \mid \theta \mid E)\), where \(L\) is a literal can be reduced to a state \(S'\), denoted \(S \Rightarrow_A S'\), as follows:

\begin{enumerate}
  \item If \(L\) is a constraint or \(L = X(t_1, \ldots, t_n)\), then \(S' = (G' \mid \theta' \mid E')\) where \(G'\) and \(\theta'\) are obtained in a same manner as in \((L :: G \mid \theta) \Rightarrow (G' \mid \theta')\).
  \item If \(L\) is an atom and \(3(L :: B) \in \text{defn}(L)\), then \(S' = (B :: \text{Post}C :: G \mid \theta \mid E')\) where:
  \begin{itemize}
    \item Post\(C\) is the sequence \texttt{check}(c_1) \ldots \texttt{check}(c_n) including all the checks \texttt{check}(c_i) such that \(c_i \# \text{success}(L, \text{Pre}, \text{Post}_i) \in A_C^n(L) \land \theta \Rightarrow \text{Pre}_{c_i}\)
  \end{itemize}
\end{enumerate}

\textsuperscript{3}While check literals are simply instrumental here, note that they are also directly useful for supporting program point assertions (which are basically check literals that appear in the body of rules). However, for simplicity we do not discuss program point assertions in this paper.
• and either $E' = E \cup \{\varepsilon\}$ if $\exists c \#\text{calls}(L, \text{Pre}) \in A_C^p(L)$
  s.t. $\theta \neq P$ Pre or $E' = \varepsilon$ otherwise.

3. If $L$ is a check literal $\text{check}(c)$, then $S' = \langle G \mid \theta \mid E' \rangle$
  where either $E' = E \cup \{\varepsilon\}$ if $\not\exists\text{success}(L, \text{Post}) \in A_C^p(L) \land \theta \neq P$ Post or $E' = \varepsilon$ otherwise.

Note that the order in which the PostC check literals are selected is irrelevant.

The set of derivations for a program from its set of queries $Q$ using the semantics with assertions is denoted $\text{deriv}_A(Q)$.

**Definition 16.** The set of error-erased derivations from $\neg A$ is obtained by a syntactic rewriting ($\neg$) which removes states that begin by a check literal, check literals from goals, and the error set. It is recursively defined as follows:

$$\{D_1, \ldots, D_n\}^\circ = \{D_1^\circ, \ldots, D_n^\circ\}$$

\[(S_1, \ldots, S_m, S_{m+1})^\circ = \begin{cases} (S_1, \ldots, S_m)^\circ & \text{if } S_{m+1} = (\text{check}(\_)) \Rightarrow \_ \Rightarrow \_ \\ (S_1, \ldots, S_m)^\circ \cup ((S_{m+1})^\circ) & \text{otherwise} \end{cases} \]

\[(G \mid \theta \mid E)^\circ = (G^\circ \mid \theta) \]

\[(L :: G)^\circ = \begin{cases} G^\circ & \text{if } L = \text{check}(\_) \\ L :: (G^\circ) & \text{otherwise} \end{cases} \]

$\square^\circ = \square$

where $\parallel$ stands for sequence concatenation.

**Theorem 1** (Correctness and Completeness Under Assertion Checking). For any annotated program $(P, Q, A)$, given $D = \text{deriv}(Q)$ and $D' = \text{deriv}_A(Q)$, it holds that $D$ and $D'$ are equivalent after filtering out check literals and error sets (formally defined as $D = (D')^\circ$ in appendix Def. 16).

Proof. We will prove $D = (D')^\circ$ by showing that $D \subseteq (D')^\circ$ and $D' \supseteq (D')^\circ$.

• ($\subseteq$) For all $D \in D$ exists $D' \in D'$ so that $D = (D')^\circ$.

• ($\supseteq$) For all $D' \in D'$, $D = (D')^\circ \in D$.

We will prove each case:

- ($\subseteq$) Let $D = (S_1, \ldots, S_n)$, $S_i = (L_i \mid \theta_i)$, for some $Q = (L_1, \theta_1) \in Q$ and $S_i \neg\neg S_{i+1}$. Proof by induction on the length $n$ of $D$:
  - Base case ($n = 1$). Let $S_i = (L_1 \mid \theta_1 \mid \theta)$. It holds that $(S_i)^\circ = (L_1 \mid \theta_1 \mid \theta)^\circ = (L_1 \mid \theta_1) = S_i$ (since $L_1$ does not contain any check literal). Thus, $(D')^\circ = (S_i)^\circ = (S_i) = D$.
  - Inductive case (show $n + 1$ assuming $n$ holds). For each $D_2 = (S_1, \ldots, S_n, S_{n+1})$ there exists $D_2 = (S_1, \ldots, S_m, S_{m+1})$ such that $(D_2)^\circ = D_2$. Given the induction hypothesis it is enough to show that for each $S_0 \neg\neg S_{n+1}$ there exists $S_0 \neg\neg A S_{n+1}$ such that $(S_{n+1})^\circ = S_{n+1}$. According to $\neg A$ (see Def. 15), $L'_{n+1}$ and $\theta'_{n+1}$ are obtained in the same way than in $\neg\neg$ (see Def. 2) except for the introduction of check literals. Since all check literals are removed in error-erased states, it follows that $(S_{n+1})^\circ = S_{n+1}$.

- ($\supseteq$) Let $D' = (S_1', \ldots, S_n')$, $S_i' = (L_i' \mid \theta_i' \mid E_i)$, for some $Q = (L_1', \theta_1') \in Q$ and $S_i' \neg\neg A S_{i+1}$. Proof by induction on the length $m$ of $D'$:
  - Base case ($m = 1$). It holds that $(S_1')^\circ = S_1$ (showed in base case for $\subseteq$). Then $(D')^\circ = D \in D$.
  - Inductive case (show $m + 1$ assuming $m$ holds). We want to show that given $D_2' = (S_1', \ldots, S_m', S_{m+1})$, $(D_2')^\circ = D_2' \in D$. Given the induction hypothesis it is enough to show that for each $S_m \neg\neg A S_{m+1}$ there exists $S_m \neg\neg S_{m+1}$ such that $(S_{m+1})^\circ = (S_{m+1})^\circ$ (so that $(S_1, \ldots, S_n, S_{n+1}) \in D)$ or $S_m = (S_{m+1})^\circ$. (D_2 \in D)$. According to cases of Def. 15:
    - If $L''_m$ begins with a check literal then $(L''_m)^\circ = (L''_m)^\circ$. Thus $(S_{m+1})^\circ = (S_m) = S_m$.
    - Otherwise, it holds that $(S_{m+1})^\circ = S_{m+1}$ using the same reasoning than in the inductive case for $\subseteq$.

This result implies that the semantics with assertions can also be used to obtain all answers to the original query. Furthermore, the following theorem guarantees that we can use the proposed operational semantics for annotated programs in order to detect (all) violations of assertions:

**Definition 17.** Let $\mathcal{E}(D)$ denote the error set of the last state of derivation $D$, $D_{[1]} = (\_ \mid \_ \mid \_ \mid E)$. The run-time valuation of an assertion condition $C$ on a derivation $D$ is given by:

$$\text{rtsolve}(C, D) \equiv \forall c. C', \sigma. L (c\#C' \in A_C^p(L) \land \sigma(C) = C') \Rightarrow \mathcal{E}(D) \not\Rightarrow c$$

I.e., condition $\text{rtsolve}(C, D)$ is valid if none of the possible instances of the assertion condition $C$ are in the error set for derivation $D$.

**Theorem 2** (Run-time Error Detection). For any annotated program $(P, Q, A)$, $C \in A_C$ is false iff $\exists D \in \text{deriv}_A(Q)$ s.t. $\neg\text{rtsolve}(C, D)$.

Proof. $A \in A_C$ is false

$\iff$ from Def. 12 and Def. 3 $\exists (C_1, C_2)$ assertion conditions s.t. $\text{false}(C_1) \lor \text{false}(C_2)$, where $C_1 = \text{calls}(L, \text{Pre})$ and $C_2 = \text{success}(L, \text{Post}, \text{Pre})$ correspond to $A$. Let us first prove $\neg\text{rtsolve}(C_1, D)$, and then $\neg\text{rtsolve}(C_2, D)$.

false$(C_1)$$

$\iff$ from Def. 11 $\exists D \in \text{deriv}(Q)$ s.t. $\neg\text{solve}(C_1, D)$

$\iff$ from Def. 9 $\exists \text{prestep}(L, D) = (\theta, \sigma) \land \theta \not\Rightarrow P \sigma(\text{Pre})$

$\iff$ from Def. 15 $\exists S \neg\neg A S'$

where

$$S = (L :: G \mid \theta \mid E) \exists c\#\text{calls}(L, \text{Pre}) \in A_C^p(L)$$

$S' = (\_ \mid \theta \mid \varepsilon') \land \varepsilon' = \varepsilon \cup \{c\}$

$\iff$ from Def. 17 $\neg\text{rtsolve}(C_1, D)$

false$(C_2)$$

$\iff$ from Def. 11 $\exists D \in \text{deriv}(Q)$ s.t. $\neg\text{solve}(C_2, D)$

$\iff$ from Def. 9 $\exists \text{step}(L, D) = (\theta, \sigma, \theta') \land \theta \not\Rightarrow P \sigma(\text{Pre}) \land \theta' \not\Rightarrow P \sigma(\text{Post})$

$\iff$ from Def. 15 $\exists S \neg\neg A S''$

where

$$S = (L :: G \mid \theta \mid \varepsilon) \land \varepsilon \exists c\#\text{success}(L, \text{Pre}, \text{Post}) \in A_C^p(L) \land \theta \not\Rightarrow P \text{ Pre}$$

$S' = (\check{\text{check}}(c) :: G \mid \theta' \mid \varepsilon') \land \theta' \not\Rightarrow P \text{ Post}$

$S'' = (\_ \mid \varepsilon') \land \varepsilon' = \varepsilon \cup \{c\}$

$\iff$ from Def. 17 $\neg\text{rtsolve}(C_2, D)$
4. Higher-order Assertions on Higher-order Derivations

Once we have established basic results for the case of first-order assertions in the context of higher-order derivations, we extend the notion of assertion itself to the higher-order case. The motivation is that in the higher-order context terms can be bound to predicates and our aim is to also be able to state and check properties of such predicates.

4.1 Anonymous Assertions

We start by generalizing the notion of assertion to include anonymous assertions: assertions where the predicate symbol is a variable from VS, which can be instantiated to any suitable predicate symbol from PS to produce non-anonymous assertions. An anonymous assertion is an expression of the form “:- pred \( L : \text{Pre} \Rightarrow \text{Post} \)”, where \( L \) is of the form \( X(V_1, \ldots, V_m) \) and \( \text{Pre} \) and \( \text{Post} \) are DNF formulas of prop literals.

Example 5. The anonymous assertion:

“:- pred \( X(A, B) : \text{list}(A) \Rightarrow \text{list}(B) \).”

states that any predicate \( p \in P \) that \( X \) is constrained to should be of arity 2, it should be called with its first argument instantiated to a list, and if it succeeds, then its second argument should be also a list on success.

We now introduce predprops, which gather a number of anonymous assertions in order to fully describe variables containing higher-order terms (predicate symbols), similarly to how prop literals describe conditions for variables containing first-order terms.

Definition 18. Given \( \text{Pre} \) and \( \text{Post} \), conjunctions of prop literals, a predprop \( pp(X) \) is an expression of the form:

\[
\begin{align*}
\text{pp}(X) & \equiv \text{:- pred } X(V_1, \ldots, V_m) : \text{Pre}_1 \Rightarrow \text{Post}_1, \\
& \quad \ldots \text{:- pred } X(V_1, \ldots, V_m) : \text{Pre}_n \Rightarrow \text{Post}_n.
\end{align*}
\]

Definition 19. The corresponding set of anonymous assertion conditions for the predprop \( pp(X) \) is defined as

\[ A_C[pp(X)] = \{ C_i[X] \mid i = 0..n \} \]

where:

\[ C_i[X] = \begin{cases} 
\text{calls}(X(V_1, \ldots, V_m), \text{Pre}) & i = 0 \\
\text{success}(X(V_1, \ldots, V_m), \text{Pre}_i, \text{Post}_i) & i = 1..n 
\end{cases} \]

The variable \( X \) can be instantiated to a particular predicate symbol \( q \in PS \) to produce a set of non-anonymous assertion conditions \( A_C[pp(p)] \) for \( q \).

Example 6. Consider defining a comparator (Cmp) predprop that describes predicates of arity 3 which can be used to compare numerical values:

\[
\text{comparator(Cmp)} \{ \begin{align*}
\text{:- pred Cmp}(X,Y,\text{Res}) : \text{int}(X), \text{int}(Y) \Rightarrow \text{between}(-1,1,\text{Res}). \\
\text{:- pred Cmp}(X,Y,\text{Res}) : \text{flt}(X), \text{flt}(Y) \Rightarrow \text{between}(-1,1,\text{Res}).
\end{align*}
\}
\]

4.2 Operational Semantics for Higher-order Programs with Higher-order Assertions

We now discuss several alternative operational semantics for higher-order programs with higher-order assertions. In all cases the aim of the semantics is to check whether assertions with predprops hold or not during the computation of the derivations from a query.

4.2.1 Checking with Static predprops

According to Definition 20, a predprop literal \( pp(X) \) denotes the subset of predicates for which all the associated assertions are checked. When that set of assertions can be statically computed, then \( \theta \Rightarrow P \text{ Cond} \) can be used for both prop and predprop Cond literals, and the operational semantics is identical to the one for the higher-order programs and regular assertions.

We will denote as \( S \Rightarrow A \ S' \) a reduction from a state \( S \) to a state \( S' \) under the semantics for higher-order derivations in programs with assertions that may contain higher-order properties, which are statically precomputed. Thus, state reductions are performed as follows:

\[
\begin{align*}
(G \mid \theta \mid E) \Rightarrow A (G' \mid \theta' \mid E') \\
(G \mid \theta \mid E') \Rightarrow A (G' \mid \theta' \mid E')
\end{align*}
\]

The meaning of each predprop, \( pp(X) \), can be inferred or checked (if given by the user) by static analysis.
In this semantics, given the program shown in Fig. 1 and the goal \( test_{c}(z,-2) \), assertions are detected to be false since \( \{ P = z \} \not\subseteq \text{neg}(P) \) and \( \{ P = z \} \not\subseteq \text{neg}(P) \).

4.2.2 Checking with Dynamic predprops

Given the difficulty in determining the meaning of \( \text{pp}(X) \) statically, we also propose a semantics with dynamic checking. We start with an over-approximation of each predprop \( \text{pp}(X) = \{ X = p \mid p \in \mathcal{PS} \} \) and incrementally remove predicate symbols, as violations of assertion conditions are detected:

- we can detect when some assertion condition instance is violated (Def. [14]);
- we need a way to obtain a set of assertion condition instances from predprops (anonymous assertion condition instances);

We do that by defining instantiations of anonymous assertion conditions for particular predicate symbols and the dependencies among those instances.

The following two definitions extend the notion of assertion condition instances from Def. [14] to the case of anonymous assertion conditions and higher-order literals:

**Definition 21.** Given a predprop \( \text{pp}(X) \) and a predicate symbol \( p \in \mathcal{PS} \), \( A[p] \) denotes the set of labelled hypothetical assertion conditions of the form \( h \# C_p \), such that \( C[X] \in A[p] \) (Def. [14]), \( L = X(V_1, \ldots, V_n) \), \( L_p = p(V_1, \ldots, V_n) \), \( C_p \) is defined as:

\[
C_p = \begin{cases} 
\text{calls}(L_p, \text{Pre}) & \text{if } C[X] = \text{calls}(L, \text{Pre}) \\
\text{success}(L_p, \text{Pre}, \text{Post}) & \text{if } C[X] = \text{success}(L, \text{Pre}, \text{Post}) 
\end{cases}
\]

and \( h \) is an identifier that is unique for each \( C_p \).

In this semantics we allow the assertion condition instances to be derived from the hypothetical assertion conditions in the same way, as in Def. [14]. However, the violation of such an instance has to be treated in a special way, as it does not signal the violation of its conditions, but instead of the corresponding predprop. For simplicity, we also introduce a special label \( h_0 \) to denote the assertion conditions that appeared originally in the program. The error set \( E \) in Def. [15] contained negated assertion condition instance identifiers. Now we extend this set with assertion dependency rules of the form \( \bigwedge \{ \forall \bar{c} \rightarrow \bar{c} \} \rightarrow \bar{c} \). The following definitions provide the description of how such dependencies are generated.

**Definition 22.** The simplification of a literal \( L \) w.r.t. \( \theta \) is defined as:

\[
simp(L, \theta) = \begin{cases} 
L & \text{if } L \text{ is a predprop} \\
\text{true} & \text{if } \theta \Rightarrow_p L \\
\text{false} & \text{if } \theta \not\Rightarrow_p L 
\end{cases}
\]

We extend this definition for a conjunction of literals.

**Definition 23.** Given the label \( c \) of an assertion condition instance and a formula of the form \( \text{Props} = \bigvee^n_{j=1}(\text{Prop}_{ij}) \), where \( \text{Prop}_{ij} \) is either a prop or predprop literal, the extension of \( A_C \) and \( \bar{E} \) for dynamic predprop checking, denoted as \( \text{ext}(A_C, c, \text{Props}) = (\Delta A_C, \Delta \bar{E}) \), is obtained as follows:

1. if \( \text{simpl}(\text{Props}, \theta) = \text{false} \), then \( \Delta A_C = \emptyset \) and \( \Delta \bar{E} = \{ c \} \);
2. otherwise: \( \Delta A_C = \bigcup^{n} i=1 A_C^i \), and \( \Delta \bar{E} = \bigcup^{n}_{i=1} (\bigvee_{h \in h_i} h \rightarrow c) \) where:
   \[
   A_C^i = \{ h \# C \in A_C^{\#}(\text{Prop}_{ij}) \mid 0 \leq j \leq m (i) \}
   \]
   \( \text{Prop}_{ij} = \text{pp}_i(X_{ij}), \text{pp}_i(X_{ij}) \) is a predprop and \( X_{ij} \) is bound to some \( q \in \mathcal{PS} \).

\[ H_i = \{ h \mid h \# C \in A_C^i \} \]

We will denote as \( S \sim_{H \Delta} S' \) a reduction from a state \( S \) to a state \( S' \) under the current semantics.

**Definition 24.** A state \( S = (L :: G \mid \emptyset \mid E) \), where \( L \) is a literal can be reduced to a state \( S' \), denoted \( S \sim_{H \Delta} S' \), as follows:

1. If \( L \) is a constraint or \( L = X(t_1, \ldots, t_n) \), then \( S' = (G' \mid 0' \mid E ) \) where \( G' \) and \( 0' \) are obtained in a same manner as in \( (L :: G \mid \emptyset \mid E) \);
2. If \( L \) is an atom and \( \exists(L :: B) \in \text{defn}(L) \), then for each \( c_i \# C_i \in A_C^\# (L) \):
   \[
   h_i = \begin{cases} 
   \text{true} & \text{if } C_i \text{ is a success of } L, Pre_i, \text{Post}_i \\
   \text{false} & \text{otherwise}
   \end{cases}
   \]
   and \( S' \) = \( (B :: PostC :: G' \mid 0 \mid E') \), where \( E' = E \cup \bigcup \{ \text{call} \rightarrow h \} \), \( \Delta A_C, \Delta \bar{E} = A_C \cup \Delta A_C, \) and \( PostC \) is the sequence \( \text{Post}C_1 \) = \( \cdots \) = \( \text{Post}C_n \) (simplifying true literals).

3. If \( L \) is a check literal \( \text{call} \) and \( c \# \text{success} (L', \text{Post}) \in A_C^\# (L') \), then \( S' = (G \mid 0 \mid E') \) where \( (\Delta \bar{E}, \Delta A_C) = \text{ext}(A_C, c, \text{Post}) \), \( E' = E \cup \Delta \bar{E} \) and \( A_C = A_C \cup \Delta A_C \).

Note that in this semantics we support more than one call assertion condition per predicate (as several predprops may be applied to the same predicate symbol). Also note that in general we cannot prove with dynamic checking that a predprop is true. So, as a safe approximation we treat preconditions in such success assertion conditions as false.

**Definition 25.** An assertion condition \( C \) is trivial if it is of the form \( \text{call} \) (\( \text{true} \)) or \( \text{success} \) (\( \text{false} \)). It is also assumed that for any predprop \( \text{pp}(X) \), \( A_C[\text{pp}(X)] \) does not contain trivial assertion conditions.

**Theorem 3** (Higher-order Run-Time Checking). For any annotated program \( (P, Q, A) \), if \( \exists D \in \text{deriv}_{H \Delta} (Q) \) s.t. \( \neg \text{rtsolve}(C, D) \Rightarrow C \in A_C \) is false.

\[ \text{Proof.} \text{In this proof we reflect the case when an assertion condition is falsified because of some of its preprops violation. To do so it is enough to show that at most one predprop was violated. Let us first prove the theorem for the simplified case when the falsified assertion condition is } C_i = \text{success}(L, Pre, \text{pp}(X)) \text{ and then for the case } C_i = \text{call}(L, \text{pp}(X)) \text{, where } \text{pp}(X) \text{ is a predprop. Without the loss of generality we assume that } A_C[\text{pp}(X)] \text{ has cardinality of } 1 \text{ (which is a case when } \text{pp}(X) \text{ consists of one anonymous assertion and one of the corresponding anonymous assertion conditions is trivial).} \]

\[ \neg \text{rtsolve}(C_i, D) \]
\[ \Rightarrow \text{From Def. [17]} \exists \bar{c}', C_i, \sigma, L (c' \# C_i \in A_C^\# (L)) \land (\sigma(C_i) = \text{true}) \]
\[ \Rightarrow \text{true} \]
\[ \Rightarrow \text{true} \]
Given the state \( \theta \) from Def. 17:
\[ Q \equiv \{ \text{test.c(n,X), true} \} \]
and the state \( \theta \) before the call to \( \text{check(c2)} \) and \( \text{check(c3)} \).

From Def. 24 and Th. 2, we know that \( \neg \text{check}(c_{\alpha}) \) and thus \( (X = q) \not\in \{ \text{predprop}(X) \} \) according to Def. 20.

From Def. 6 and \( \text{check}(c_{\alpha}) \) it must hold that \( \Delta \theta \) for the program in Fig. 1.

From Def. 24 and Th. 3, we know that \( \neg \text{check}(c_{\beta}) \) and thus \( (X = q) \not\in \{ \text{predprop}(X) \} \) according to Def. 20.

From Def. 6 and \( \text{check}(c_{\beta}) \) it must hold that \( \Delta \theta \) for the program in Fig. 1.

Let us trace finished derivations \( D^1, D^2 \) and \( D^3 \) from the queries \( Q_1 = \{ \text{test.c(n,X), true} \} \), \( Q_2 = \{ \text{test.c(c,X), true} \} \) and \( Q_3 = \{ \{ \text{test.s(1,P)}, \text{P}(-2) \}, \text{true} \} \), respectively, to the program in Fig. 1.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{G} & \Delta \theta & \Delta \xi & \Delta(\text{labeled instances + hypothetic } A_C) \\
\hline
\text{test.c(n,X)} & P = n & c_1 \rightarrow h_0 & c_1 \# \text{calls}(\text{test.c(n,X), nneg(n)} \vee \neg \text{neg(n)}) \\
& N = -1 & h_1 \land h_2 \rightarrow c_1 & h_1 \# \text{success}(n|Z), \text{true}, \text{nnegint}(Z)) \\
& X = N & & h_2 \# \text{success}(n|Z), \text{true}, \text{negint}(Z)) \\
\hline
\text{P(-1)} & Z = -1 & c_2 \rightarrow h_1 & c_2 \# \text{success}(n(-1), \text{true}, \text{nnegint}(-1)) \\
& & c_3 \rightarrow h_2 & c_3 \# \text{success}(n(-1), \text{true}, \text{negint}(-1)) \\
\hline
\end{array}
\]

Table 1. A derivation of the query \( \{ \text{test.c(n,X), true} \} \) to the program in Fig. 1.

In \( D^1 \) (see Tab. 1), we encounter two assertions for \( \text{test.c/2} \) with a predprop in each precondition and trivial postconditions. According to state reduction rules, \( \Delta A_C \) consists of calls assertion condition instance \( c_1 \) and two hypothetical assertion conditions \( h_1 \) and \( h_2 \). From the predprop \( \text{nneg} \) and \( \text{neg} \), and \( \Delta \xi = \{ c_1 \rightarrow h_0, h_1 \land h_2 \rightarrow c_1 \} \) in \( D^1 \) and current goal \( \text{P}(-1) \) (which is implicitly reduced as \( n(-1) \)), success assertion condition instances \( c_2 \) and \( c_3 \) are derived from the hypotheses \( h_1 \) and \( h_2 \). Consequently, two check literals, \( \text{check}(c_2) \) and \( \text{check}(c_3) \) are added to the goal sequence. In states \( D^3 \) and \( D^4 \) those literals are reduced, which results in adding \( c_2 \) and \( c_3 \) to \( \xi \) because \( \text{nnegint}(-1) \) property from the postcondition of \( c_2 \) is violated. This example shows that the mechanism of dependencies between assertion conditions allows to avoid “false negative” results in assertion checking.

The derivation \( D^2 \) is similar to \( D^1 \) (see Tab. 2). The difference is in \( D^4 \) state, when it becomes possible to infer \( \xi \vdash c_1 \) and thus to conclude that \( c_1 \not\in \{ \text{predprop}(X) \} \) and \( c_1 \not\in \{ \text{predprop}(X) \} \) and that both assertions for \( \text{test.c/2} \) are false for this query.

In \( D^3 \) (see Tab. 3), we encounter two assertions with a predprop in each postcondition. According to state reduction rules, \( \Delta A_C \) for this state consists of calls and success assertion condition instances, \( c_1 \) and \( c_2 \). \( \Delta \theta \equiv \{ h_0, c_1 \rightarrow h_0 \} \) for them. Also, a check literal \( \text{check}(c_1) \) is added to the goal sequence. After its reduction a hypothetical assertion condition \( c_2 \) derived from \( \text{nneg}(X) \) predprop, is added to \( A_C \) in \( D^3 \) and \( \xi \) is extended with a dependency rule \( \{ c_2 \rightarrow h_1 \} \). In state \( D^4 \) an assertion condition instance \( c_2 \) is obtained from \( h_2 \) and \( \Delta \theta = \{ c_2 \rightarrow h_2 \} \). Finally, in state \( D^5 \) it becomes possible to infer \( \xi \vdash c_1 \) and thus deduce that the corresponding assertion for \( \text{test.s/2} \) predicate is false because of the predprop \( \text{nneg}(X) \) violation.

5. Conclusions and Future Work

This paper contributes towards filling the gap between higher-order (C)LP programs and assertion-based extensions for error detection and program validation. To this end we have defined a new class of properties, “predicate properties” (predprops in short), and proposed a syntax and semantics for them. These new properties can be used in assertions for higher-order predicates to describe the properties of the higher-order arguments. We have also discussed several operational semantics for performing run-time checking of programs including predprops and provided correctness results.

Our predprop properties specify conditions for predicates that are independent of the usage context. This corresponds
in functional programming to the notion of tight contract satisfaction [9], and it contrasts with alternative approaches such as loose contract satisfaction [11]. In the latter, contracts are attached to higher-order arguments by implicit function wrappers. The scope of checking is local to the function evaluation. Although this is a reasonable and pragmatic solution, we believe that our approach is more general and more amenable for combination with static verification techniques. For example, avoiding wrappers allows us to remove function wrappers. The scope of checking is local to the function wrappers.

We have included the proposed predprop extensions in an experimental branch of the Ciao assertion language implementation. This has the immediate advantage, in addition to the enhanced checking, that it allows us to document higher-order programs in much more accurate way. We have also implemented several prototypes for operational semantics with dynamic predprop checking (see [5] for a minimalistic implementation), which we plan to integrate into the already existing assertion checking mechanisms for first-order assertions.

Table 2. A derivation of the query \((\text{test}\_c(c,X), \text{true})\) to the program in Fig. 1.

<table>
<thead>
<tr>
<th>G</th>
<th>(\Delta \theta)</th>
<th>(\Delta \xi)</th>
<th>(\Delta(\text{labeled instances} + \text{hypothetic } \mathcal{A}_C))</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_c(c,X)</td>
<td>(P = c)</td>
<td>(c_1 \rightarrow h_0)</td>
<td>(c_1 #\text{calls}(test_c(c,X), nnegint(c) \lor \neg neg(c)))</td>
</tr>
<tr>
<td></td>
<td>(N = a)</td>
<td>(h_2 \land h_3 \rightarrow c_1)</td>
<td>(h_2 #\text{success}(Z(c), \text{true), nnegint}(Z)))</td>
</tr>
<tr>
<td></td>
<td>(X = N)</td>
<td>(c_1 \rightarrow h_3)</td>
<td>(h_3 #\text{success}(Z(c), \text{true, nnegint}(Z)))</td>
</tr>
<tr>
<td>(P(a))</td>
<td>(Z = a)</td>
<td>(c_2 \rightarrow h_2)</td>
<td>(c_2 #\text{success}(a(c), \text{true, nnegint}(a)))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(c_3 \rightarrow h_3)</td>
<td>(c_3 #\text{success}(a(c), \text{true, nnegint}(a)))</td>
</tr>
<tr>
<td>check(c_2),</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check(c_3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>check(c_3)</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(\square)</td>
<td></td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. A finished derivation of the query \(((\text{test}\_s(1,P), P(-2)), \text{true})\) to the program in Fig. 1.

<table>
<thead>
<tr>
<th>G</th>
<th>(\Delta \theta)</th>
<th>(\Delta \xi)</th>
<th>(\Delta(\text{labeled instances} + \text{hypothetic } \mathcal{A}_C))</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_s(1,P)</td>
<td>(N = 1)</td>
<td>(c_0 \rightarrow h_0)</td>
<td>(c_0 #\text{calls}(test_s(1,P), nnegint(1) \lor \neg negint(1)))</td>
</tr>
<tr>
<td>(P(-2))</td>
<td>(P = z)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>check(c_1),</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P(-2))</td>
<td>check(c_1),</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(P(-2))</td>
<td>(h_2 \rightarrow c_1)</td>
<td>(h_2 #\text{success}(Z(c), \text{true, nnegint}(Z)))</td>
</tr>
<tr>
<td>check(c_2)</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>(\square)</td>
<td></td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

References


A. Minimalistic Sample Implementation

The following code (portable to most Prolog systems with minor changes) shows a minimalistic sample implementation (as an interpreter \texttt{intr/1}) of the operational semantics for dynamic predprop checking (Def. 24). Conciseness and simplicity has been favoured over efficiency. We assume that clauses, assertion conditions, and predprops have been parsed and stored in \texttt{cl/2}, \texttt{ac/1}, \texttt{pp/2} facts, respectively. The interpreter will throw an exception the first time that a failed program assertion is detected (see \texttt{ext/2 predicate}). E.g., \texttt{intr((test_s(1,P),P(1)))} is a valid query while \texttt{intr((test_s(1,P),P(-2)))} throws a failed assertion exception. Predicate \texttt{reset/0} must be called between \texttt{intr/1} queries to reset error status and temporary data. In the handler errors can be gathered (as in the semantics) or execution aborted.

```prolog
% Sample program data and properties
% negint/1 and nnegint/1 properties
eval_prop(negint(X)) :- integer(X), X < 0.
eval_prop(nnegint(X)) :- integer(X), X >= 0.

% predprops nneg/1 and neg/1
pp(nneg(P), ac(P(X), nneg_c1(P)#success(true,nnegint(X)))).
pp(neg(P), ac(P(X), neg_c1(P)#success(true, negint(X)))).

% assertion conditions and clauses for test_s/2
ac(test_s(N,_P), c1#calls((nnegint(N);negint(N)))).
ac(test_s(N,P), c2#success(nnegint(N), nneg(P))).
ac(test_s(N,P), c3#success(negint(N), neg(P))).
cl(test_s( 1,P), P = z).
cl(test_s(-1,P), P = n).

% Intepreter
:- dynamic hyp_ac/2. % hypothetical assertion condition
:- dynamic negac/1. % (negated) assertion dependency rule

% Get assertion conditions for the given literal A
get_acs(A, Acs) :- ( bagof(Ac, get_ac(A, Ac), Acs) -> true ; Acs = [] ).
get_ac(A, Ac) :- ( ac(A, Ac) ; hyp_ac(A, Ac) ).

pre([]) --> [].
pre([A|As]) --> pre(A), pre(As).
pre(_Id#success(_, _)) --> ( { simp0(Pre, true) } -> [Id] ; [] ).
post([], Acs).
post([Id|Ids], Acs) :- post(Id, Acs), post(Ids, Acs).
post(Id, Acs) :- member(Id#success(_,Post), Acs), Id = id0, !, ext(Post,Id).
post(_, _).

% Check/extend assertion conditions
ext(Props, Id) :-
  simp(Props, Props2), ext(Props2, Id),
  negac(A), atom(A)
  -> throw(failed_assertion(A)) ; true).
ext(true, _Id) :- !.
ext(false, Id) :- !, assertz((negac(Id) :- true)).
ext(Props, Id) :- acsubs(Props, Props2),
  assertz((negac(Id) :- Props2)).

% Add assertion dependency rules
acsubs((A,B), (A2;B2)) :- !,
  acsubs(A, A2), acsubs(B, B2).
acsubs((A ; B), (A2 , B2)) :- !,
  acsubs(A, A2), acsubs(B, B2).
acsubs(ac(L, Id#Ac), negac(Id)) :-
  ctog(L, L2), assertz(hyp_ac(L2, Id#Ac)).

% Condition simplification
simp(true, R) :- !, R = true.
simp((X,Y), R) :- !,
simp(X, Rx), simp(Y, Ry), or(Rx, Ry, R).
simp(X, R) :- pp(X, Ac), !, R = Ac.
simp(X, R) :- eval_prop(X), !, R = true.
simp(_, R) :- R = false.
simp0(true, R) :- !, R = true.
simp0((X,Y), R) :- !,
simp0(X, Rx), simp0(Y, Ry), and(Rx, Ry, R).
simp0(X, R) :- pp(X, Ac), !, R = Ac.
simp0(X, R) :- eval_prop(X), !, R = true.
simp0(_, R) :- R = false.

% Condition simplification for success preconditions
or(true, _, R) :- !, R = true.
or(false, X, R) :- !, R = X.
or(_, true, R) :- !, R = true.
or(_, false, R) :- !, R = X.
or(X, Y, (X,Y)).

and(false, _, R) :- !, R = false.
and(true, X, R) :- !, R = X.
and(_, false, R) :- !, R = false.
and(_, true, R) :- !, R = X.
and(X, Y, (X,Y)).
```

% From call(N,...) to N(...) where N is a predicate symbol
ctog(X, _) :- var(X), !, throw(inst_error).
cctog(X, X1) :-
  X =.. [call,N|Args],
  ( atom(N) -> true ; throw(inst_error) ),
  X1 =.. [N|Args].