Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Master on Formal Methods in Computer Science and Engineering

Master's Thesis

Towards Verification of Higher-Order (Constraint) Logic Programs via Abstract Interpretation

Author: Marco Ciccalè Baztán
Advisor: Manuel V. Hermenegildo
Co-Advisor: Jose F. Morales

Madrid, July 2025

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster en Métodos Formales en Ingeniería Informática

Título: Towards Verification of Higher-Order (Constraint) Logic Programs via Abstract Interpretation

Julio de 2025

Autor: Marco Ciccalè Baztán
Tutor: Manuel V. Hermenegildo
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid
Cotutor: Jose F. Morales
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

To my parents, who never stopped believing in me.

Acknowledgments

First and foremost, I would like to thank my supervisors Manuel, Jose and Pedro for unconditionally sharing their deep technical knowledge, for their constant support, and—most importantly—for encouraging me to keep doing—what Manuel calls—"fun stuff." Their guidance and our discussions have made me genuinely excited about pursuing a Ph.D., and I truly look forward to what lies ahead.

I would also like to thank Jurjo for putting up with me throughout this work—despite being nearly 8,000 km away, and for being picky when it came to the definitions and proofs (someone had to care *a bit too much*).

A big thanks to all the researchers at IMDEA—especially to Paula, Daniela, Marcos, Jorge, Leo, Javi, Andoni…—for the lunches, coffee breaks, and laughs throughout this year.

Last but not least, I would like to thank my friends and family for their unconditional support, and my partner Aitana for encouraging me throughout this year and for wholeheartedly supporting my decision to continue my academic journey.

Anywhere can be paradise as long as you have the will to live. After all, you are alive, so you will always have the chance to be happy...

The End of Evangelion

Abstract

Higher-order constructs enable more expressive and concise code by allowing procedures to be parameterized by other procedures, resulting in more modular and maintainable code. Assertions are linguistic constructs for writing partial program specifications, which can then be verified either at compile time (*i.e.*, statically) or run time (*i.e.*, dynamically). In the case of higher-order programs, assertions provide descriptions of the higher-order arguments of procedures. In the context of (C)LP, the run-time verification of such higher-order assertions has received some attention. However, verification at compile time remains relatively unexplored.

We propose a novel approach for the *compile-time* verification of higher-order (C)LP programs with assertions that describe higher-order arguments. Although our presentation is based for concreteness on the Ciao assertion language, the approach is quite general and flexible, and we believe it can be applied to similar gradual approaches. Higher-order arguments are described using predicate properties, a special kind of properties that allow using the full power of the (Ciao) assertion language for such arguments.

We first present a refinement of both the syntax and semantics of these properties. Next, we introduce an abstract criterion to determine whether a predicate conforms to a predicate property at compile time, based on a semantic order relation to compare the definition of a predicate property and the partial specification of a predicate. We then propose a technique for dealing with these properties using an abstract interpretation-based static analyzer for programs with first-order assertions, by reducing predicate properties to first-order properties that are natively understood by such an analyzer. Finally, we report on a prototype implementation and study the effectiveness of the approach with various examples within the Ciao system.

Keywords: Higher-Order · Static Analysis · Assertions · Abstract Interpretation · (Constraint) Logic Programming.

Resumen

Las primitivas de orden superior permiten escribir código más expresivo y conciso ya que permiten escribir procedimientos parametrizados por otros procedimientos, resultando en código más modular y mantenible. Las aserciones son un tipo especial de primitivas que permiten expresar especificaciones parciales de programas, que luego pueden ser verificadas en tiempo de compilación (es decir, de forma estática) o en tiempo de ejecución (es decir, de forma dinámica). En el caso de los programas de orden superior, las aserciones describen los argumentos de orden superior de los procedimientos. En el contexto de la programación lógica (y con restricciones) ((C)LP), la verificación dinámica de las aserciones de orden superior ha recibido cierta atención. Sin embargo, la verificación estática sigue siendo un área relativamente poco explorada.

Proponemos un enfoque novedoso para la verificación estática de programas (C)LP de orden superior con aserciones que describen argumentos de orden superior. Aunque nuestra presentación se basa en el lenguaje de aserciones de Ciao, la técnica es bastante general y flexible, y creemos que puede aplicarse a enfoques graduales similares. Los argumentos de orden superior se describen utilizando propiedades de predicados, un tipo especial de propiedades que permiten emplear todo el poder de expresividad del lenguaje de aserciones (de Ciao) para dichos argumentos.

Primero, presentamos una mejora tanto de la sintaxis como de la semántica de estas propiedades. A continuación, introducimos un criterio abstracto para determinar si un predicado cumple con una propiedad de predicado en tiempo de compilación, basado en una relación de orden semántica que compara la definición de una propiedad de predicado con la especificación parcial de un predicado. Luego, proponemos una técnica para tratar estas propiedades utilizando un analizador estático basado en interpretación abstracta para programas con aserciones de primer orden, mediante la reducción de las propiedades de predicados a propiedades de primer orden que dicho analizador puede entender de forma nativa. Finalmente, presentamos una implementación prototipo y estudiamos la efectividad del enfoque con varios ejemplos dentro del sistema Ciao.

Palabras Clave: Orden Superior · Análisis Estático · Aserciones · Interpretación Abstracta · Programación Lógica con Restricciones.

Contents

1 Introduction **1**
- 1.1 Objective . . . 1
- 1.2 Related Work . . . 2
- 1.3 Structure of the Document . . . 2

2 Background **3**
- 2.1 Preliminary Definitions . . . 3
- 2.2 (Constraint) Logic Programs . . . 4
- 2.3 Abstract Interpretation . . . 7
- 2.4 Abstract Interpretation of (C)LP Programs . . . 10

3 The Ciao System **13**
- 3.1 The CiaoPP Program Pre-Processor . . . 14
- 3.2 Assertion Language . . . 15
- 3.3 Run-Time Checking of Traditional Assertions . . . 18
- 3.4 Compile-Time Checking of Traditional Assertions . . . 21

4 Specifying and Verifying Higher-Order Programs **23**
- 4.1 Predicate Properties . . . 23
- 4.2 Conformance to a Predicate Property . . . 25
  - 4.2.1 Conformance . . . 25
  - 4.2.2 Abstract Conformance . . . 27
- 4.3 Wrappers . . . 36
- 4.4 First-Order Representation of Predicate Properties . . . 38
- 4.5 Algorithm . . . 38

5 Implementation and Experiments **41**
- 5.1 Implementation . . . 41
- 5.2 Experiments . . . 42

6 Conclusions & Future Work **49**
- 6.1 Future Work . . . 50

Bibliography **53**

Chapter 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, maintainable and scalable code-bases.

(Constraint) logic programming languages like Prolog [1] and functional programming languages like Haskell [2] 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, and also include higher-order libraries. As an example, consider the query: "`?- P = even, filter(P, [7,4,2,9], L).`" which passes the term `even` as an argument to the higher-order library predicate `filter/3`, which will *apply* the `even` predicate to each element of the input list, selecting those that succeed, and resulting in `L = [4,2]`.

Assertions are linguistic constructs for writing partial program specifications, which can then be verified or used to detect deviations in program behavior with respect to such partial specifications. The *assertion-based approach* to program verification [3, 4] differs from other approaches such as strong type systems [5] 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 (and a precursor of) gradual typing in functional languages [6].

1.1 Objective

In this thesis we propose a novel approach for the *compile-time* verification of higher-order (C)LP programs with assertions that describe higher-order arguments via the *abstract interpretation* mathematical theory [7] for approximating the concrete semantics of a program.

Although our presentation is based for concreteness on (C)LP and the Ciao assertion language, the approach is quite general and flexible, and we believe it can be applied to similar gradual approaches and other programming languages/paradigms.

1.2 Related Work

The combination of higher-order and assertions in the (C)LP context was already explored by [8]. 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.*,[1] [9, 10, 11]) generally adheres to the strong typing model.

1.3 Structure of the Document

The rest of this thesis is organized as follows: Chapter 2 provides the necessary background to understand the context and technical foundation of the thesis. Chapter 3 provides a high-level view of the Ciao system, together with an extensive description of its assertion language and the approaches used to verify such assertions. Chapter 4 presents the approach, starting from its underlying theory, through some instrumental techniques for reducing the complexity of the verification problem at hand, and finally deriving the main algorithm of the approach. Chapter 5 reports on a prototype implementation of the approach which leverages the abstract domains and the abstract interpretation-based analysis engine both within the Ciao system. It also provides detailed examples that were not possible to formally verify up until this point. Finally, Chapter 6 contains our conclusions and some lines of future work.

[1] *exempli gratia*—for example

Chapter 2

Background

This thesis is intended to be self-contained, assuming only a basic familiarity with first-order logic and general programming skills. This chapter provides the reader with the necessary background to understand the context and technical foundations of this thesis. First, §2.1 provides some instrumental and well-known mathematical notions. Next, §2.2 introduces (constraint) logic programming ((C)LP), together with its formal notation and operational semantics extended to support higher-order constructs. Then, §2.3 summarizes the basic notions of abstract interpretation, and its application for building *safe-by-construction* static analysis tools. Finally, §2.4 presents the use of abstract interpretation for (C)LP programs.

2.1 Preliminary Definitions

We write $o \triangleq e$ for denoting that the mathematical object o is defined as the mathematical expression e. We define $\mathbb{B} \triangleq \{\mathbf{tt}, \mathbf{ff}\}$ as the set of truth values: *true* (**tt**) and *false* (**ff**), and $\mathbb{N}$ and $\mathbb{Z}$ as the set of natural and integer numbers, respectively.

Set Theory. Let A and B be two arbitrary sets.

Definition 2.1 (Union). We define the *union* of A and B as the set containing all elements that are in A *or* in B. Formally, $A \cup B \triangleq \{x \mid x \in A \vee x \in B\}$. ◊

Definition 2.2 (Intersection). We define the *intersection* of A and B as the set containing all elements that are in A *and* in B. Formally, $A \cap B \triangleq \{x \mid x \in A \wedge x \in B\}$. ◊

Definition 2.3 (Subset). We define A to be a *subset* of B, denoted $A \subseteq B$, iff every element of A is also an element of B. Formally, $A \subseteq B \Leftrightarrow \forall x \in A.\ x \in B$. ◊

Definition 2.4 (Proper Subset). We define A to be a *proper subset* of B, denoted $A \subset B$, iff A is a subset of B and A is not equal to B. Formally, $A \subset B \Leftrightarrow (A \subseteq B \wedge A \neq B)$. ◊

Definition 2.5 (Powerset). We define the powerset of A, denoted $\wp(A)$, as the set of all subsets of A. Formally, $\wp(A) \triangleq \{C \mid C \subseteq A\}$. ◊

Definition 2.6 (Cartesian Product). We define the *Cartesian product* of A and B, denoted $A \times B$, as the set of all ordered pairs—denoted (x, y), or x/y—with their components being elements of A and B, respectively. Formally, $A \times B \triangleq \{(x, y) \mid x \in A \wedge y \in B\}$. ◊

Definition 2.7 (Set Difference). We define the *set difference* of A and B, denoted $A \setminus B$, as the set of all elements in A that are not in B. Formally, $A \setminus B \triangleq \{x \mid x \in A \wedge x \notin B\}$. ◊

Definition 2.8 (Cardinality). We define the *cardinality* of a set A, denoted $|A|$, as the number of distinct elements in A. Also, A is said to be *finite* iff $|A| < \omega$ with $\omega \in \mathbb{N}$. ◊

2.2 (Constraint) Logic Programs

Alain Colmerauer set out to develop a system for human-machine communication based on logic. During his research, he came across Robert Kowalski and Donald Kuehner's work on SL-resolution [12], which encouraged him to invite Kowalski to visit Marseille on two occasions. Their collaboration led in 1972 to the first Natural Language application of what we now know as Prolog; and also, to the *logic programming* paradigm (LP), and to the basis of Prolog itself: a linear resolution system restricted to Horn clauses that could answer questions in the problem domain described by the clauses input [13].

A few years later, in 1977, Kowalski introduced the notion of "Algorithm = Logic + Control" as a way to conceptually divide algorithms in two different components: (1) the "Logic," which captures the problem specification as a set of rules; and (2) the "Control," which states how these rules are used to compute solutions. Thus, by definition, a logic program (*i.e.*,[1] a set of rules) can be interpreted either: (1) *declaratively* ("Logic"), as the problem specification, or (2) *procedurally* ("Control"), as the way in which resolution acts on the set of rules. The fact that in LP the "Logic" and "Control" components of an algorithm are the same set of rules under different interpretations, distinguishes LP from other paradigms.

In 1987, Joxan Jaffar and Jean-Louis Lassez introduced the *constraint logic programming* (CLP) paradigm, observing that LP could be seen as an instance of constraint solving over variables in the Herbrand domain [14]. CLP extends LP by allowing constraints over variables of additional domains (*e.g.*, the real numbers domain $\mathfrak{R}$) in the rules, in addition to the Herbrand domain.

Notation. *Variables* start with a capital letter. The *anonymous variable*, denoted _, is a variable that is distinct from every other variable. 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.*, $\bar{t}$ denotes a sequence of terms), and write $|(\bar{\cdot})|$ for representing

[1] *id est*—that is

the length of the sequence ($\bar{\cdot}$). An *atom* has the form $p(\bar{t})$ where p is an n-ary predicate symbol, and $\bar{t}$ are terms. The function $\mathrm{ar}(p)$ 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, *e.g.*, $X - Y > abs(Z)$. A *literal* is either an atom, a higher-order atom or a constraint. *Constants* are simply 0-ary symbols. *Negation* is encoded as finite failure (NAFF), supported through a program expansion. A *goal* is a finite sequence of literals. A *rule* has the form $H \leftarrow B$ 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*, 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 the renaming σ to a syntactic object L. The *definition* of an atom L in a program, $\mathrm{defn}(L)$, is the set of renamed program rules s.t.[2] each renamed rule has L as its head. We assume that all rule heads are *normalized*, *i.e.*, H is an atom of the form $p(\bar{v})$ where $\bar{v}$ 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.

A *predicate* (or *procedure* in the context of (C)LP), is a set of rules with the same head. In the examples we use Prolog syntax, where the usual implication ($\leftarrow$) is denoted by (`:-`), and the trivial premise of a rule (**tt**) is omitted. We refer to a predicate p by a normalized atom $p(\bar{v})$ or by p/n, where $n = \mathrm{ar}(p) = |\bar{v}|$. Projecting the constraint θ onto the variables of the syntactic object L is denoted as $\theta{\restriction_L}$. We denote *constraint entailment* by $\theta_1 \models \theta_2$.

Example 2.1 (Higher-Order Logic Program). Consider the program on the left which defines the `b/2` predicate, relating the numeric representation of a binary digit to its textual representation. It also defines the `maplist/3` predicate, which relates a list with the result of applying a given predicate to each of the elements of the list.

```
1 b(0, z).   b(1, o).
2 maplist(_, [], []).
3 maplist(P, [C|Cs], [R|Rs]) :-
4     P(C,R),
5     maplist(P, Cs, Rs).
```

More concretely, `b/2` is defined as a set of two facts (line 1) that relate the numbers `0` and `1` to the constants `z` and `o` respectively. The predicate `maplist/3` is defined *recursively*, the base case (the fact in line 2) represents the case in which both lists are empty—note the anonymous variable stating that the first argument is not relevant. The recursive case (the rule in lines 3 to 5) specifies that if both lists are not empty (with heads `C` and `R`, respectively), then two conditions must hold: (1) the higher-order literal `P(C,R)`—representing a call to the provided predicate `P`—must succeed, and (2) the remaining elements of the lists (`Cs` and `Rs`, respectively) with `P` must also satisfy the `maplist/3` predicate. ◇

[2] such that

CONSTR	*L is a constraint*	
$\langle L :: G \mid \theta\rangle \rightsquigarrow \langle G \mid \theta \wedge L\rangle$		if $\theta \wedge L \not\models \mathbf{ff}$ (*i.e.*, $\theta \wedge L$ is *satisfiable*)
HO-ATOM	*L is a higher-order atom of the form* $X(\bar{t})$	
$\langle L :: G \mid \theta\rangle \rightsquigarrow \langle p(\bar{t}) :: G \mid \theta\rangle$		if $\exists p \in \mathcal{P}.\ \theta \models (X = p) \wedge \mathsf{ar}(p) = \lvert\bar{t}\rvert$
ATOM	*L is an atom of the form* $p(\bar{t})$, *and* $\exists(L \leftarrow B) \in \mathsf{defn}(L)$	
$\langle L :: G \mid \theta\rangle \rightsquigarrow \langle B :: G \mid \theta\rangle$		

Figure 2.1: Higher-order (C)LP: Reductions.

Operational Semantics of Higher-Order Programs. The operational semantics of a higher-order program 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 for short) θ. We denote sequence concatenation by (::), and the empty sequence by $\square$. We assume for simplicity that the underlying constraint solver is complete and that projection exists. We use $S \rightsquigarrow S'$ to indicate that a reduction 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'. Given a state $S = \langle L :: G \mid \theta\rangle$ where L is a literal, it can be *reduced* to a state S' by applying the reduction rules in Figure 2.1. Assume $S \rightsquigarrow^* S'$ where $S = \langle L :: G \mid \theta\rangle$ and $S' = \langle G \mid \theta'\rangle$, we refer to S as a *call state* for L, and S' as a *success state* for L.

A *query* 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 (intermediate and finished) derivations of the program $\mathcal{P}$ from the query Q (also a set of queries $\mathcal{Q}$) is denoted $\mathsf{derivs}(\mathcal{P}, Q)$. And, given a derivation D, its last step is denoted $D_{[-1]}$. We now provide several instrumental definitions related to the operational semantics of (higher-order) programs.

Definition 2.9 (Succesful or Failed Derivation). Given a finished derivation D, we say that D is *successful* (resp.[3] *failed*) iff $D_{[-1]}$ is (resp. is not) of the form $\langle \square \mid \theta'\rangle$. ◊

Definition 2.10 (Answer to a Query). Given a program $\mathcal{P}$ and a successful derivation D from a query $Q = (L, \theta)$ whose last state is of the form $\langle \square \mid \theta'\rangle$, we say that $\theta'\restriction_L$ is an *answer* to Q. We denote by $\mathsf{answers}(\mathcal{P}, Q)$ the set of answers of the program $\mathcal{P}$ for Q. ◊

Definition 2.11 (Failed Query). Given a program $\mathcal{P}$ and a query Q, we say that Q *finitely fails* iff $\mathsf{derivs}(\mathcal{P}, Q)$ is finite and contains no successful derivation. ◊

Definition 2.12 (Calling Context). Given a program $\mathcal{P}$, a predicate represented by a normalized atom *Pred*, and a set of queries $\mathcal{Q}$, we define the *calling context* $C(\mathit{Pred}, \mathcal{P}, \mathcal{Q})$ of *Pred* for $\mathcal{P}$ and $\mathcal{Q}$ as

$$\{\theta\restriction_{\mathit{Pred}} \mid \exists D \in \mathsf{derivs}(\mathcal{P}, \mathcal{Q}).\ D_{[-1]} = \langle \mathit{Pred} :: G \mid \theta\rangle\}. \qquad \diamond$$

[3] respectively

Definition 2.13 (Success Context). Given a program $\mathcal{P}$, a predicate represented by a normalized atom *Pred*, a store θ, and a set of queries $\mathcal{Q}$, we define the *success context* $\mathcal{S}(Pred, \theta, \mathcal{P}, \mathcal{Q})$ of *Pred* and θ for $\mathcal{P}$ and $\mathcal{Q}$ as

$$\{\theta' \restriction_{Pred} \mid \exists D \in \mathsf{derivs}(\mathcal{P}, \mathcal{Q}).\ \exists G.\ \langle Pred :: G \mid \theta \rangle \in D.\ D_{[-1]} = \langle G \mid \theta' \rangle\}. \qquad \diamond$$

2.3 Abstract Interpretation

Abstract interpretation [7, 15, 16] is a mathematical theory for reasoning about the execution of computer programs. It does so by soundly approximating their semantics. Abstract interpretation can be applied to many areas of computer science, including semantics design, proof methods, and static program analysis. In this thesis we focus on the use of abstract interpretation for constructing *sound-by-construction* static program analysis tools. These tools can extract properties about a program by interpreting it over an *abstract domain*, which serves as a *simplified* and *safe* approximation of the *concrete domain*, effectively approximating the actual semantics of the program.

Order Theory. We first recall the basics of order theory and define some notation.

Definition 2.14 (Partial Order). Given a set L and a relation $\sqsubseteq$ on L, we say that $\sqsubseteq$ is a *partial order* iff for all x, y, z in L, we have that it is: *reflexive* ($x \sqsubseteq x$), *antisymmetric* ($x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$), and *transitive* ($x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$). $\diamond$

Definition 2.15 (Partially Ordered Set). Given a set L and a partial order $\sqsubseteq$ on L, we say that the tuple $\langle L, \sqsubseteq \rangle$ is a *partially ordered set* (or *poset* for short). $\diamond$

Definition 2.16 (Ascending Chain Condition). We say that a poset $\langle L, \sqsubseteq \rangle$ satisfies the *ascending chain condition* iff it has no infinite ascending chain. $\diamond$

Definition 2.17 ((Least) Upper Bound). Let $\langle L, \sqsubseteq \rangle$ be a poset and $S \subseteq L$ be a subset. The subset S has: (1) an *upper bound* $u \in L$ iff $\forall x \in S.\ x \sqsubseteq u$, and (2) a *least upper bound* $\sqcup S \in L$ iff $\sqcup S$ is an upper bound of S smaller than any other upper bound of S. $\diamond$

Definition 2.18 ((Greatest) Lower Bound). Let $\langle L, \sqsubseteq \rangle$ be a poset and $S \subseteq L$ be a subset. The subset S has: (1) a *lower bound* $l \in L$ iff $\forall x \in S.\ l \sqsubseteq x$, and (2) a *greatest lower bound* $\sqcap S \in L$ iff $\sqcap S$ is a lower bound of S greater than any other lower bound of S. $\diamond$

The least upper (resp. greatest lower) bound $\sqcup\{x, y\}$ (resp. $\sqcap\{x, y\}$) is denoted using the *infix* notation $x \sqcup y$ (resp. $x \sqcap y$).

Definition 2.19 (Lattice). Given a poset $\langle L, \sqsubseteq \rangle$, we say that the tuple $\langle L, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ is a *lattice* iff for all x, y in L, we have that: (1) the *lub* $x \sqcup y$, (2) the *glb* $x \sqcap y$, (3) the *supremum* (or *top* for short) $\top$, and (4) the *infimum* (or *bottom* for short) belong to L. $\diamond$

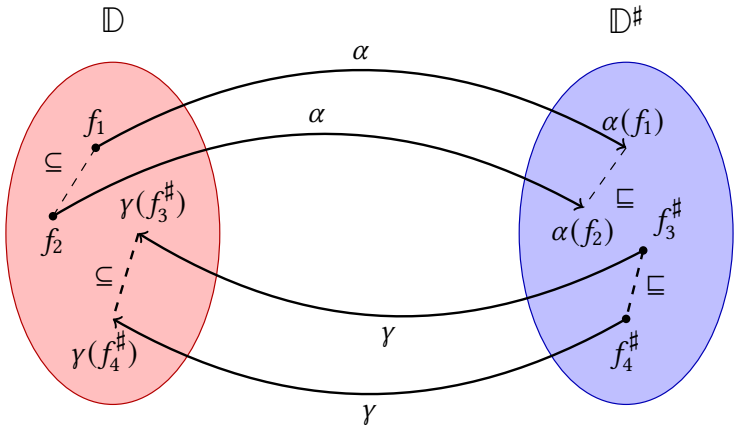


Figure 2.2: Illustration of a Galois connection.

Abstract Domains. Let us now formally introduce the notion of *concrete*, and *abstract* domains and their relation by recalling an instrumental definition from [16].

Definition 2.20 (Galois Connection). Given posets $\langle \mathbb{D}, \subseteq \rangle$ (the *concrete domain*) and $\langle \mathbb{D}^\sharp, \sqsubseteq \rangle$ (the *abstract domain*), the pair $\langle \alpha, \gamma \rangle$ of monotone functions $\alpha \in \mathbb{D} \to \mathbb{D}^\sharp$ (*abstraction*) and $\gamma \in \mathbb{D}^\sharp \to \mathbb{D}$ (*concretization*) is a *Galois connection* iff

$$\forall f \in \mathbb{D}.\ \forall f^\sharp \in \mathbb{D}^\sharp.\ \alpha(f) \sqsubseteq f^\sharp \Leftrightarrow f \subseteq \gamma(f^\sharp)$$

which we denote by

$$\langle \mathbb{D}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{D}^\sharp, \sqsubseteq \rangle \qquad \diamond$$

Figure 2.2 depicts an illustration of a Galois connection. Provided a Galois connection between a concrete domain $\langle \mathbb{D}, \subseteq \rangle$ and an abstract domain $\langle \mathbb{D}^\sharp, \sqsubseteq \rangle$ via $\langle \alpha, \gamma \rangle$, abstract interpretation *guarantees* that any analysis can be computed in a finite number of steps iff either $\langle \mathbb{D}^\sharp, \sqsubseteq \rangle$ satisfies the *ascending chain condition*, or a *widening* operator ∇ is used [7].

Example 2.2 (Abstracting Parity). Consider defining a simple analysis that determines the parity of the integer variables of a program. Let $\mathbb{E}$ and $\mathbb{O}$ denote the set of even and odd integer numbers, $\mathbb{P} \triangleq \{\bot, \mathit{even}, \mathit{odd}, \top\}$ be the set representing parity, and $\mathcal{P}$ be a program whose set of integer variables is denoted by $\mathbb{V}$. We define a concrete domain $\langle \mathbb{D}, \subseteq \rangle$, where

$$\mathbb{D} \triangleq \{f \mid f \in \mathbb{V} \to \wp(\mathbb{Z})\}$$

represents all possible mappings from each variable to sets of integers, and the ordering $\subseteq$ is the usual set inclusion relation lifted to apply pairwise over the second element of each pair, *i.e.*, over the set of integers assigned to each variable. Analogously to $\mathbb{D}$, we define the set

$$\mathbb{D}^\sharp \triangleq \{f^\sharp \mid f^\sharp \in \mathbb{V} \to \mathbb{P}\}$$

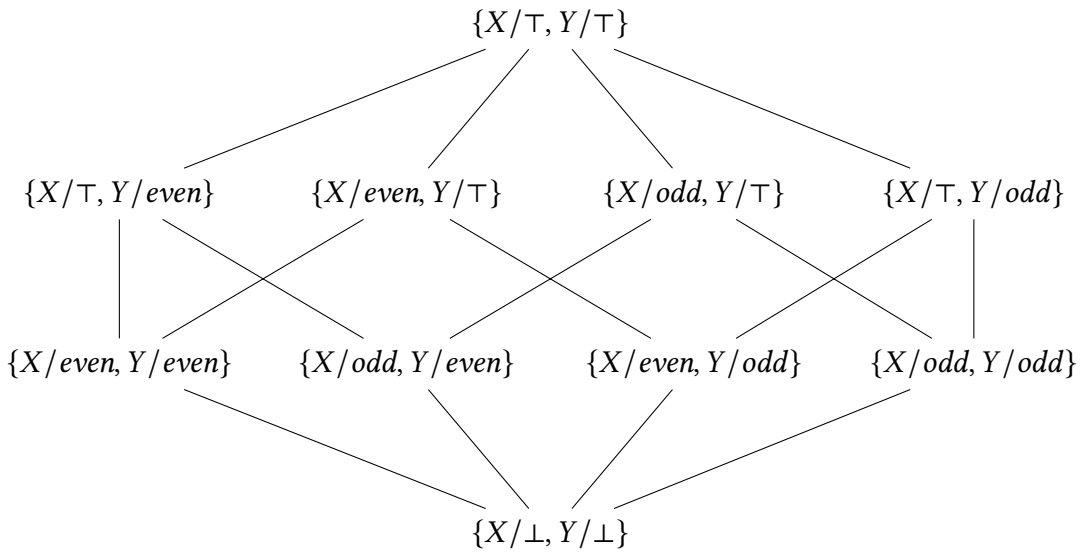


Figure 2.3: Parity abstract domain for a program $\mathcal{P}$ with integer variables X and Y.

as all possible mappings from each variable to their parity. Let us now define the concretization function $\gamma \in \mathbb{D}^\sharp \to \mathbb{D}$ as

$$\gamma(f^\sharp) \triangleq \left\{ \begin{array}{ll} X \mapsto \varnothing & \text{if } p^\sharp = \bot \\ X \mapsto \mathbb{E} & \text{if } p^\sharp = even \\ X \mapsto \mathbb{O} & \text{if } p^\sharp = odd \\ X \mapsto \mathbb{Z} & \text{otherwise} \end{array} \middle| \; (X \mapsto p^\sharp) \in f^\sharp \right\}$$

and the abstraction function $\alpha \in \mathbb{D} \to \mathbb{D}^\sharp$ as

$$\alpha(f) \triangleq \left\{ \begin{array}{ll} X \mapsto \bot & \text{if } p = \varnothing \\ X \mapsto even & \text{if } \forall n \in p.\ n \in \mathbb{E} \\ X \mapsto odd & \text{if } \forall n \in p.\ n \in \mathbb{O} \\ X \mapsto \top & \text{otherwise} \end{array} \middle| \; (X \mapsto p) \in f \right\}$$

From these definitions, we can now induce the order relation $\sqsubseteq \subseteq \mathbb{D}^\sharp \times \mathbb{D}^\sharp$ defined as

$$f_1^\sharp \sqsubseteq f_2^\sharp \Leftrightarrow \gamma(f_1^\sharp) \subseteq \gamma(f_2^\sharp)$$

thus allowing us to define the parity abstract domain $\langle \mathbb{D}^\sharp, \sqsubseteq \rangle$ which captures the property "an integer variable X in the program $\mathcal{P}$ is even or odd." The *Hasse* diagram in Figure 2.3 depicts the parity abstract domain $\langle \mathbb{D}^\sharp, \sqsubseteq \rangle$ for a program with two variables.

We can now define a Galois connection

$$\langle \mathbb{D}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{D}^\sharp, \sqsubseteq \rangle$$

which follows directly from the definitions of α and γ. ◊

2.4 Abstract Interpretation of (C)LP Programs

In the context of (C)LP, it is common for elements of the abstract domain to be finite representations of the elements of the concrete domain, which are (possibly infinite) sets of actual constraints. As usual, these two domains are related via the abstraction and concretization monotone functions, which together form a Galois connection between both domains. We use, for concreteness, *goal-dependent abstract interpretation.* In particular the PLAI algorithm, based on an efficient fixpoint computation [17]. PLAI takes as input a program $\mathcal{P}$, an abstract domain, and a set of initial abstract queries $Q^\sharp$ describing all the possible initial concrete queries to $\mathcal{P}$. As a result, PLAI yields an (abstract domain-dependent) abstraction of the concrete semantics of the program $\mathcal{P}$. Intuitively, due to monotonicity, more specific initial abstract queries yield more precise analysis results.

An abstract query $Q^\sharp$ is a pair (L, λ), where L is an atom and λ is an element of the abstract domain representing a set of concrete initial program states (*e.g.*, constraint stores). A set of abstract queries $Q^\sharp$ represents a set of concrete queries

$$\gamma(Q^\sharp) \triangleq \{(L, \theta) \mid (L, \lambda) \in Q^\sharp \ \wedge \ \theta \in \gamma(\lambda)\}$$

The goal-dependent abstract interpretation of a program $\mathcal{P}$ for the set of initial abstract queries $Q^\sharp$ (computed by PLAI) is denoted $[\![\mathcal{P}]\!]^\sharp_{Q^\sharp}$. More concretely, PLAI yields a set of analysis triples

$$\{\langle Pred_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle Pred_n, \lambda_n^c, \lambda_n^s\rangle\}$$

where $Pred_i$ is a normalized atom representing a predicate, and λ_i^c and λ_i^s are abstractions that safely approximate the calling and success context of $Pred_i$, respectively, from the set of queries $\gamma(Q^\sharp)$. That is, given any analysis triple $\langle Pred, \lambda^c, \lambda^s\rangle \in [\![\mathcal{P}]\!]^\sharp_{Q^\sharp}$,

$$\gamma(\lambda^c) \supseteq C(Pred, \mathcal{P}, \gamma(Q^\sharp))$$
$$\gamma(\lambda^s) \supseteq \cup\{S(Pred, \theta, \mathcal{P}, \gamma(Q^\sharp)) \mid \theta \in \gamma(Q^\sharp)\}$$

The operational semantics for higher-order programs is supported by reducing higher-order calls to first-order calls when the called predicate can be determined by the analysis, or making conservative assumptions otherwise.

Without loss of generality, from this point onwards we assume: (1) a concrete domain $\langle \wp(\mathbb{D}), \subseteq, \cup, \cap, \mathbb{D}, \varnothing\rangle$ with a lattice structure whose set has a powerset shape, and (2) an abstract domain $\langle \mathbb{D}^\sharp, \sqsubseteq, \sqcup, \sqcap, \top, \bot\rangle$ with a lattice structure which safely approximates the concrete values and operations.

More concretely, the order relation of the abstract domain ($\sqsubseteq$) is induced by the order relation of the concrete domain ($\subseteq$, the set inclusion) and the abstraction function α. Similarly, the *join* ($\sqcup$) and *meet* ($\sqcap$) operators mimic those of the concrete domain in a precise sense. Formally, given any three abstractions $\lambda_1, \lambda_2, \lambda_3 \in \mathbb{D}^\sharp$:

$$\lambda_1 \sqsubseteq \lambda_2 \Leftrightarrow \gamma(\lambda_1) \subseteq \gamma(\lambda_2) \tag{2.1}$$
$$\lambda_1 \sqcup \lambda_2 = \lambda_3 \Leftrightarrow \gamma(\lambda_1) \cup \gamma(\lambda_2) = \gamma(\lambda_3) \tag{2.2}$$
$$\lambda_1 \sqcap \lambda_2 = \lambda_3 \Leftrightarrow \gamma(\lambda_1) \cap \gamma(\lambda_2) = \gamma(\lambda_3) \tag{2.3}$$

Typically, the ($\sqcup$) and ($\sqcap$) operations yield *safe approximations* of their concrete counterparts ($\cup$) and ($\cap$). However, without loss of generality and to keep the presentation as clear as possible, we assume that the abstract operators do not *lose precision*, *i.e.,*

$$\gamma(\lambda_1 \sqcup \lambda_2) = \gamma(\lambda_1) \cup \gamma(\lambda_2) \tag{2.4}$$

$$\gamma(\lambda_1 \sqcap \lambda_2) = \gamma(\lambda_1) \cap \gamma(\lambda_2) \tag{2.5}$$

We will explicitly indicate the points where more involved techniques would be needed to handle situations in which this is not the case.

The abstraction $\bot$ represents unreachable code (*i.e.*, $\gamma(\bot) = \varnothing$), and the abstraction $\top$ represents the most general abstraction (*i.e.*, $\gamma(\top) = \mathbb{D}$).

Chapter 3

The Ciao System

The Ciao system [18] (see Figure 3.1 for a high-level view) is a modern, general-purpose and multi-paradigm programming language with an advanced programming environment. The design philosophy behind the system is to provide a combination of programming language and development tools that together help programmers produce trustworthy, scalable and maintainable code in less time and with less effort. The Ciao system addresses these points from two different approaches: *verification* and *testing*. Verification consists on the automatic or interactive construction of formal proofs about a piece of code adhering to a given specification. Testing consists in executing a piece of code for concrete test cases checking that its behavior is the expected one. The Ciao system introduced a development model and workflow [4, 3, 19] that seamlessly integrates both verification and testing. In §3.1 we present a high-level view of the CiaoPP program pre-processor, the main tool behind the verification and testing phases of the aforementioned workflow.

As part of CiaoPP, one of the fundamental building blocks of this workflow are program *assertions*, which are presented in §3.2. Assertions serve as both specifications for static analysis and as run-time check generators, unifying run-time verification and unit testing with static verification and static debugging. Assertions are optional, and they can be checked at run or compile time, to be presented in §3.3 and §3.4 respectively. The model considers the situations in which some of the assertions may not be checkable at compile time, and will generate (run-time) tests for them when possible.

This model represents an alternative approach for writing safe programs without relying on full static typing, specially valuable for dynamic languages like Prolog where flexibility is not sacrificed for correctness. The intention is to combine the best elements of static and dynamic languages approaches [20], and is an antecedent to the now popular *gradual-* and *hybrid-typing* approaches [21, 6, 22].

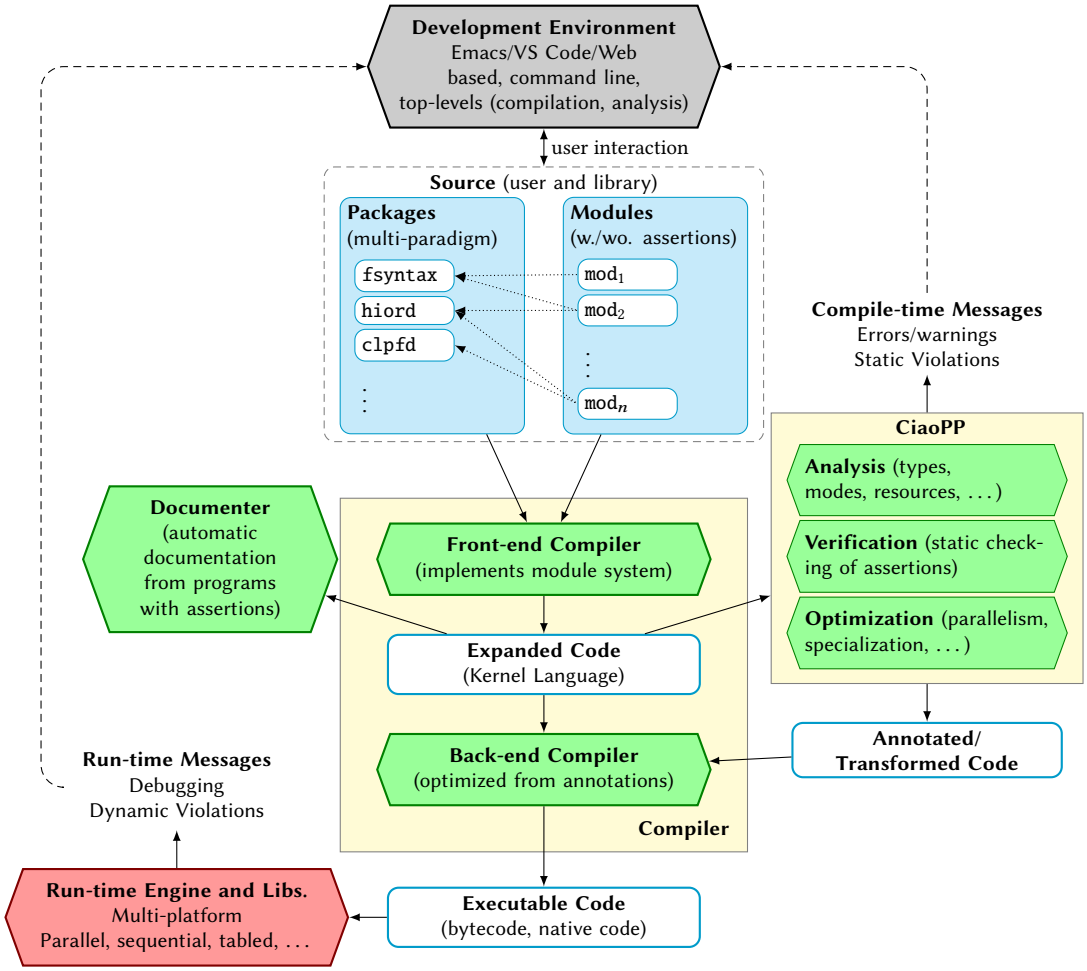


Figure 3.1: High-level view of the Ciao system.

3.1 The CiaoPP Program Pre-Processor

The CiaoPP program pre-processor [23, 17, 24, 3, 4, 19] (see Figure 3.2 for a high-level view) is the abstract interpretation-based program pre-processor of the Ciao system that can be used to perform debugging, analysis, and source-to-source transformation of programs. It can be applied to (Ciao) Prolog programs, and to programs written in other low- or high-level programming languages by transforming such programs to the Horn clause-based intermediate representation used by CiaoPP. More concretely, CiaoPP can perform the following tasks:

- Inference of *properties* at a predicate and literal level, such as *types*, *modes* and other variable instantiation properties, *non-failure*, *determinacy*, *upper bounds* in the computational cost and *size* of terms in the program, etc.
- Static debugging and verification of programs w.r.t.[1] an assertion-based (partial) specification.

[1] with respect to

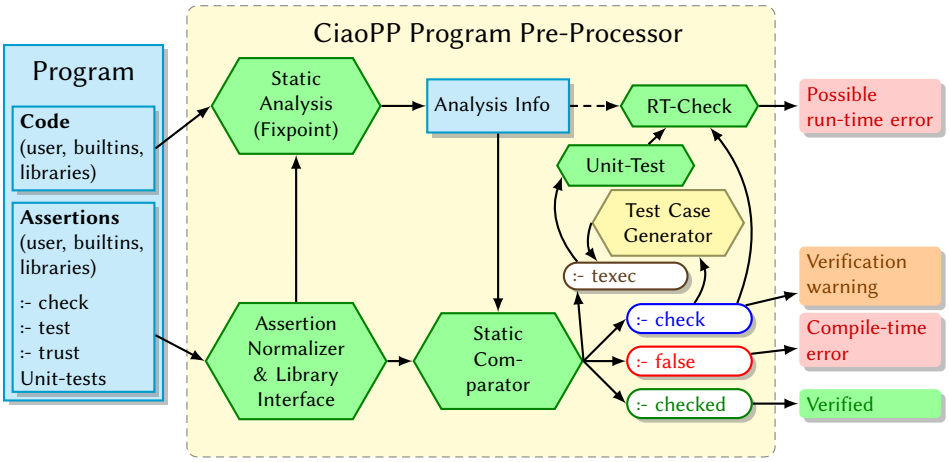


Figure 3.2: High-level view of the CiaoPP program pre-processor.

- Source-to-source program transformations, such as *specialization*, *slicing*, *partial evaluation*, and *parallelization*. It can also generate *run-time tests* for properties that could not be proved to hold statically, ensuring that the program will run safely by dynamically checking such properties.
- Construction of *abstract models* of programs that acts as a certificate of the correctness of the code, *i.e.*, an abstraction-carrying code approach to mobile code safety [25].

All the aforementioned tasks rely on the properties inferred at compile time by the PLAI fixpoint computation algorithm introduced in §2.4.

3.2 Assertion Language

We now introduce the assertion language of the Ciao system.

Property Formulas. Conditions on the constraint store are stated as *property formulas*. A property formula is simply a disjunctive normal form (DNF) formula of *property literals*. A property (or prop) 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 [26, 4]. In particular, properties are normally required to be checkable at run time but not necessarily decidable at compile time, where they are safely approximated.[2]

Example 3.1 (Properties & Property Literals). The program in Listing 1 defines the properties `list/1` ("being a list") and `prefix/2` ("being a prefix of a list"). The property

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

```
:- prop list/1.
list([]).
list([_|Xs]) :- list(Xs).

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

Listing 1: `list_props` program.

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 two definitions from [4] for reasoning about property formulas:

Definition 3.1 (Succeeds Trivially). A property literal L *succeeds trivially* for a constraint store θ in a program $\mathcal{P}$, denoted $\theta \mapsto L$, iff

$$\exists \theta' \in \mathsf{answers}(\mathcal{P}, (L, \theta)).\ \theta \models \theta'$$

Also, a property formula F succeeds trivially for a constraint store θ in a program $\mathcal{P}$ if all of the property literals of at least one conjunct of F succeed trivially. ◊

Intuitively, a property literal (or formula) succeeds trivially if it succeeds for θ without adding new "relevant" constraints to θ.

Example 3.2 (Succeeds Trivially). Let `list(L)` be a property formula, and $\theta_1 = \{$`L = [1,2]`$\}$, $\theta_2 = \{$`L = [1,X]`$\}$, $\theta_3 = \{$`L = [1|Ls]`$\}$ be three constraint stores. The property formula `list(L)` succeeds trivially both for θ_1 and θ_2, since a call to `list(L)` with either of them would succeed without adding new constraints to the store. However, `list(L)` does not succeed trivially for θ_3, since a call `list(L)` with θ_3 would further constraint the term `[1|Ls]` by adding the constraint `Ls = []` to θ_3. Intuitively, the property formula `list(L)` captures the notion that `L` is required to be *instantiated* to a list. ◊

This means that property literals are considered as *instantiation checks*: they are *true* iff the variables they check are at least as constrained as their predicate definition requires.

Traditional Assertions. Assertions are syntactic objects for (partially) specifying procedures by means of expressing properties that must be satisfied at program execution. They are primarily used for detecting deviations in a procedure's behavior w.r.t. its specification. We now recall the assertion schema of [27] that is relevant herein. *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 pre-condition *Pre*, and, if such calls succeed, the post-condition *Post must* be satisfied. If there are several *pred*

```
1 :- pred take(N, Xs, Ys) : (nnegint(N), list(Xs)) => prefix(Ys, Xs).
2 :- pred take(N, Xs, Ys) : (list(Xs), prefix(Ys, Xs)) => nnegint(N).
3 take(0, _, []).
4 take(N, [X|Xs], [X|Ys]) :-
5     N  #> 0,
6     N1 #= N-1,
7     take(N1, Xs, Ys).
```

Listing 2: `take` program.

assertions, then the *Pre* field of at least one of them must be satisfied (see later). We call this kind of assertions *traditional assertions* or *first-order assertions* interchangeably.

Example 3.3 (Assertions). Consider the program in Listing 2. The `take(N,Xs,Ys)` predicate relates a list `Xs` and its prefix `Ys` of length `N` by leveraging the `clpfd` finite domain constraints package for reversible arithmetic.

Consider also the *pred* assertions (partially) specifying the `take(N,Xs,Ys)` predicate using the `list/1` and `prefix/2` properties defined in the `list_props` program shown in Listing 1, and the `nnegint/1` built-in property ("being a non-negative integer"). These assertions restrict the meaning of `take(N,Xs,Ys)` as follows:

- `take(N,Xs,Ys)` must be called with `Xs` bound to a list, and either `N` bound to a non-negative integer or `Ys` bound to a prefix of `Xs`.
- if `take(N,Xs,Ys)` succeeds when called with `N` bound to a non-negative integer and `Xs` bound to a list, then `Ys` must be bound to a prefix of `Xs` on success.
- 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 a non-negative integer on success. ◊

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

Definition 3.2 (Set of 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

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

where $\mathsf{calls}(Pred, \bigvee_{j=1}^{n} Pre_j)$, the *calls assertion condition*, encodes the check that ensure that all the calls to the predicate represented by *Pred* are within those admissible by the set of assertions. And $\mathsf{success}(Pred, Pre_i, Post_i)$, the *success assertion conditions*, encode the checks for compliance of the successes for particular sets of calls. ◊

From this point onwards, 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 represented by *Pred*; and if no assertions are provided for a predicate, we implicitly assume the most general assertion conditions calls(*Pred*, **tt**) and success(*Pred*, **tt**, **tt**).

Example 3.4 (Set of Assertion Conditions). The assertion conditions for the *pred* assertions of the `take(N,Xs,Ys)` predicate defined in the `take` program in Listing 2 are:

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

3.3 Run-Time Checking of Traditional Assertions

We now introduce an extension of the operational semantics of higher-order programs presented in §2.2 that 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. Every assertion condition C is related to a unique label ℓ via a mapping $\text{label}(C) = \ell$ for identifying a possible violation. 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 condition instances, and $|\mathcal{E}| \leqslant 1$.[3] We also extend the set of literals with (instrumental) literals of the form $\text{check}(Pred, \ell)$ where *Pred* is a normalized atom representing a predicate, 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 [28], which we slightly adapt to support higher-order atoms. Under this semantics, given a state $S = \langle L :: G \mid \theta \mid \varnothing \rangle$ where L is a literal, it can be *reduced* to a state S', denoted $S \rightsquigarrow_{\mathcal{A}} S'$, by applying the reduction rules in Figure 3.3. The set of all (intermediate and finished) derivations for the program $\mathcal{P}$ with assertions $\mathcal{A}$ from the query Q (also a set of queries $\mathcal{Q}$) using the semantics with assertions is denoted $\text{derivs}_{\mathcal{A}}(\mathcal{P}, Q)$.

Assertion conditions are checked by directly applying Definition 3.1 in rules **ATOM**$_{\mathcal{A}}$ and **CHECK**$_{\mathcal{A}}$. More concretely, when applying the **ATOM**$_{\mathcal{A}}$ rule to an atom of the form $p(\bar{t})$, we first check its associated calls assertion condition (if any), before reducing it to its body followed by the *PostC* sequence. And, when applying the **CHECK**$_{\mathcal{A}}$ rule to a check literal $\text{check}(Pred, \ell)$, we are simply checking the success assertion condition labeled with ℓ, which is associated with the predicate represented by *Pred*. In both cases, if an assertion condition is violated during the derivation of a set of queries $\mathcal{Q}$, we say that the assertion condition is `false` for $\mathcal{Q}$, and such derivations *finish* in an state whose $\mathcal{E}$ set is not empty.

[3] While an $\mathcal{E}$ set is unnecessary if execution halts upon an assertion condition violation, we include it to keep the semantics presented in this thesis close to that of previous work.

CONSTR$_{\mathcal{A}}$	*L is a constraint*	
$\langle L :: G \mid \theta \mid \varnothing\rangle \rightsquigarrow_{\mathcal{A}} \langle G \mid \theta \wedge L \mid \varnothing\rangle$		if $\theta \wedge L \not\models \mathbf{ff}$ (*i.e.*, $\theta \wedge L$ is *satisfiable*)
HO-ATOM$_{\mathcal{A}}$	*L is a higher-order literal of the form $X(\bar{t})$*	
$\langle L :: G \mid \theta \mid \varnothing\rangle \rightsquigarrow_{\mathcal{A}} \langle p(\bar{t}) :: G \mid \theta \mid \varnothing\rangle$		if $\exists p \in \mathcal{P}.\ \theta \models (X = p) \wedge \mathsf{ar}(p) = \lvert\bar{t}\rvert$
ATOM$_{\mathcal{A}}$	*L is an atom of the form $p(\bar{t})$, and $\exists(L \leftarrow B) \in \mathsf{defn}(L)$*	
$\langle L :: G \mid \theta \mid \varnothing\rangle \rightsquigarrow_{\mathcal{A}}$	$\langle G \mid \theta \mid \{\ell\}\rangle$	if $\exists C = \mathsf{calls}(L, Pre) \in \mathcal{A}.\ \mathsf{label}(C) = \ell \wedge \theta \not\Rightarrow Pre$
	$\langle B :: PostC :: G \mid \theta \mid \varnothing\rangle$	otherwise,
CHECK$_{\mathcal{A}}$	*L is a check literal* $\mathsf{check}(L', \ell)$	
$\langle L :: G \mid \theta \mid \varnothing\rangle \rightsquigarrow_{\mathcal{A}}$	$\langle G \mid \theta \mid \{\ell\}\rangle$	if $\exists C = \mathsf{success}(L', \_, Post) \in \mathcal{A}.$ $\mathsf{label}(C) = \ell \ \wedge\ \theta \not\Rightarrow Post$
	$\langle G \mid \theta \mid \varnothing\rangle$	otherwise

where *PostC* is the sequence of check literals $\mathsf{check}(L, \ell_1) :: \ldots :: \mathsf{check}(L, \ell_n)$ such that each $\ell_i = \mathsf{label}(C_i)$, where $C_i = \mathsf{success}(L, Pre_i, Post_i) \in \mathcal{A}$, and $\theta \Rightarrow Pre_i$

Figure 3.3: Higher-order (C)LP with assertions: Reductions.

Definition 3.3 (Erroneous Derivation). Given a finished derivation D, we say that D is *erroneous* iff the last state in D is of the form $\langle G' \mid \theta' \mid \{\ell\}\rangle$. ◊

Otherwise, if an assertion condition is *never* violated in any derivation from $\mathcal{Q}$, we say that the assertion condition is checked for $\mathcal{Q}$. Additionally, we extend Definitions 2.12 and 2.13 to derivations with assertions.

Definition 3.4 (Calling Context with Assertions). Given a program $\mathcal{P}$ and its set of assertions $\mathcal{A}$, a predicate represented by a normalized atom *Pred*, and a set of queries $\mathcal{Q}$, we define the *calling context* $\mathcal{C}_{\mathcal{A}}(Pred, \mathcal{P}, \mathcal{Q})$ of *Pred* for $\mathcal{P}$ and $\mathcal{Q}$ as

$$\{\theta{\restriction_{Pred}} \mid \exists D \in \mathsf{derivs}_{\mathcal{A}}(\mathcal{P}, \mathcal{Q}).\ D_{[-1]} = \langle Pred :: G \mid \theta\rangle\}.$$ ◊

Definition 3.5 (Success Context with Assertions). Given a program $\mathcal{P}$ and its set of assertions $\mathcal{A}$, a predicate represented by a normalized atom *Pred*, a store θ, and a set of queries $\mathcal{Q}$, we define the *success context* $\mathcal{S}_{\mathcal{A}}(Pred, \theta, \mathcal{P}, \mathcal{Q})$ of *Pred* and θ for $\mathcal{P}$ and $\mathcal{Q}$ as

$$\{\theta'{\restriction_{Pred}} \mid \exists D \in \mathsf{derivs}_{\mathcal{A}}(\mathcal{P}, \mathcal{Q}).\ \exists G.\ \langle Pred :: G \mid \theta\rangle \in D.\ D_{[-1]} = \langle G \mid \theta'\rangle\}.$$ ◊

Remark 3.1. Naturally, given a program $\mathcal{P}$ with assertions $\mathcal{A}$, and a set of queries $\gamma(\mathcal{Q}^\sharp)$; for any predicate in $\mathcal{P}$ represented by the normalized atom *Pred*, its calling and success

contexts *without* assertions are *supersets* of those *with* assertions:

$$C(Pred, \mathcal{P}, \mathcal{Q}) \supseteq C_{\mathcal{A}}(Pred, \mathcal{P}, \mathcal{Q})$$
$$\mathcal{S}(Pred, \theta, \mathcal{P}, \mathcal{Q}) \supseteq \mathcal{S}_{\mathcal{A}}(Pred, \theta, \mathcal{P}, \mathcal{Q})$$

since $\text{derivs}_{\mathcal{A}}(\mathcal{P}, \mathcal{Q}) \subseteq \text{derivs}(\mathcal{P}, \mathcal{Q})$ by the definition of the semantics without and with assertions shown in Figures 2.1 and 3.3 respectively.

Thus, it follows that given the analysis triple $\langle Pred, \lambda^c, \lambda^s \rangle \in [\![\mathcal{P}]\!]^{\sharp}_{Q^{\sharp}}$,

$$\gamma(\lambda^c) \supseteq C_{\mathcal{A}}(Pred, \mathcal{P}, \gamma(Q^{\sharp}))$$
$$\gamma(\lambda^s) \supseteq \cup\{\mathcal{S}_{\mathcal{A}}(Pred, \theta, \mathcal{P}, \gamma(Q^{\sharp})) \mid \theta \in \gamma(Q^{\sharp})\}$$ ◊

We now define the notion of run-time checking of assertions during the derivation of a program from a set of queries.

Definition 3.6 (Run-Time Checking of Assertions). Given a set of queries $\mathcal{Q}$, we say that an assertion is `checked` (resp. `false`) for $\mathcal{Q}$ if *all* (resp. *any*) of the corresponding assertion conditions are `checked` (resp. `false`) for $\mathcal{Q}$. ◊

Example 3.5 (Run-Time Checking of Assertions). Consider deriving a query

$$Q = (\texttt{P(N,Xs,Ys)}, \theta), \text{ with } \theta = \{\texttt{P} = \texttt{take}, \texttt{N} = \texttt{-1}, \texttt{Xs} = \texttt{[1,2]}\}$$

to the `take` program in Listing 2 while checking its set of assertions $\mathcal{A}$ using the operational semantics with assertions in Figure 3.3. Given Q, we start the derivation from the state $\langle \texttt{P(N,Xs,Ys)} \mid \theta \mid \varnothing \rangle$.

$$\langle \texttt{P(N,Xs,Ys)} \mid \theta \mid \varnothing \rangle \rightsquigarrow_{\mathcal{A}} \langle \texttt{take(N,Xs,Ys)} \mid \theta \mid \varnothing \rangle \quad \text{⟅applying HO-ATOM}_{\mathcal{A}}\text{⟆}$$
$$\rightsquigarrow_{\mathcal{A}} \langle \texttt{take(N,Xs,Ys)} \mid \theta \mid \{\ell\} \rangle \quad \text{⟅applying ATOM}_{\mathcal{A}}\text{⟆}$$

where $\ell = \text{label}(\text{calls}(\texttt{take(N,Xs,Ys)}, Pre))$—see Example 3.4 for the full condition.

Notice that the derivation ends in an erroneous state since `N` is not a non-negative integer ($(\texttt{N} = \texttt{-1}) \in \theta$), and thus the calls assertion condition associated to the `take(N,Xs,Ys)` predicate does not succeed trivially for θ ($\theta \not\Rightarrow Pre$). Then, we can conclude that the calls assertion condition labeled with ℓ is `false`. ◊

Intuitively, we can conclude that an assertion is `false` just by finding a derivation step in which one of the assertion conditions of such assertion is violated. However, for concluding that an assertion is `checked`, we need to prove that its set of assertion conditions is `checked` for all possible derivations from a set of queries, which is often not feasible in practice. This is why static analysis is often used for this purpose.

3.4 Compile-Time Checking of Traditional Assertions

For the static verification of traditional assertions, the abstract interpretation $[\![\mathcal{P}]\!]^{\sharp}_{Q^{\sharp}}$ of a program $\mathcal{P}$ for a set of abstract queries $Q^{\sharp}$ is *compared* (in the abstract domain) against the property formulas in each assertion. Given the nature of abstract interpretation (and static analyses in general), the `check` assertion status is now considered. This status represents that an assertion could not be statically proved to be `checked` or `false`, and that it is subject to its run-time checking.

Since properties are defined in the source language, some of the expressible properties may be *undecidable*, or not exactly representable in an abstract domain (*e.g.*, the `prefix/2` property defined in Listing 1). However, it is always possible to safely under- or over-approximate them in any abstract domain. We now recall some essential definitions from [4] for reasoning about property formulas at compile time.

Definition 3.7 (Trivial Success Set of a Property Formula). Given a property formula F, we define the *trivial success set* of F, denoted $F^{\natural}$, as

$$F^{\natural} \triangleq \{\theta\restriction_F \mid \theta \rightmapsto F\} \qquad \diamond$$

Intuitively, the trivial success set $F^{\natural}$ of a property formula F is the (possibly infinite) set of constraint stores for which the formula succeeds trivially (see Definition 3.1). Thus, $F^{\natural}$ belongs to the set of the concrete domain $\wp(\mathbb{D})$. Since this object is only used in the context of the compile-time checking via abstract interpretation, we use the $\natural$ symbol notation (as a dual of the $\sharp$ symbol notation for "things" living in the abstract domain).

Lemma 3.1. Given two property formulas F_1 and F_2, $(F_1 \wedge F_2)^{\natural} = F_1^{\natural} \cap F_2^{\natural}$. $\diamond$

Proof. We proceed by direct proof. Assume $(F_1 \wedge F_2)^{\natural} \neq \varnothing$, take any $\theta \in (F_1 \wedge F_2)^{\natural}$. By Definition 3.7, $\theta \rightmapsto (F_1 \wedge F_2)$, and from Definition 3.1, $\theta \rightmapsto F_1 \wedge \theta \rightmapsto F_1$. Hence, by Definition 3.7, $\theta \in F_1^{\natural} \wedge \theta \in F_2^{\natural}$, and from Definition 2.2, we conclude $\theta \in (F_1^{\natural} \cap F_2^{\natural})$. Assume now $(F_1 \wedge F_2)^{\natural} = \varnothing$, then $\forall \theta \in F_1^{\natural}.\ \theta \notin F_2^{\natural}$; hence $F_1^{\natural} \cap F_2^{\natural} = \varnothing$. $\square$

Definition 3.8 (Abstract Trivial Success Subset of a Property Formula). Given a property formula F, an abstraction is an *abstract trivial success subset* of F, denoted $F^{\sharp-}$, iff

$$\gamma(F^{\sharp-}) \subseteq F^{\natural} \qquad \diamond$$

Definition 3.9 (Abstract Trivial Success Superset of a Property Formula). Given a property formula F, an abstraction is an *abstract trivial success superset* of F, denoted $F^{\sharp+}$, iff

$$F^{\natural} \subseteq \gamma(F^{\sharp+}) \qquad \diamond$$

Intuitively, the abstract trivial success subset $F^{\sharp-}$ or superset $F^{\sharp+}$ of a property formula F safely under- and over-approximate the trivial success set $F^{\natural}$ of F, respectively. Thus, $F^{\sharp-}, F^{\sharp+} \in \mathbb{D}^{\sharp}$, and $\gamma(F^{\sharp-}) \subseteq F^{\natural} \subseteq \gamma(F^{\sharp+})$.

These approximations can be computed by PLAI, and they are *always* computable by choosing the closest element in the abstract domain. Intuitively, $\bot$ and $\top$ are, respectively, an under- and over-approximation of any property formula. Based in these abstractions, the following definitions from [29] present sufficient conditions for the compile-time checking of first-order assertions.

Definition 3.10 (Compile-Time Checking of Calls Assertion Condition). Given the abstract interpretation $[\![\mathcal{P}]\!]^\sharp_{Q^\sharp}$ of a program $\mathcal{P}$ for a set of abstract queries $Q^\sharp$, we say that a calls assertion condition calls$(Pred, Pre)$ is

$$\texttt{checked} \Leftrightarrow \forall \langle Pred, \lambda^c, \lambda^s \rangle \in [\![\mathcal{P}]\!]^\sharp_{Q^\sharp}.\ \lambda^c \sqsubseteq Pre^{\sharp -}$$
$$\texttt{false} \Leftrightarrow \forall \langle Pred, \lambda^c, \lambda^s \rangle \in [\![\mathcal{P}]\!]^\sharp_{Q^\sharp}.\ \lambda^c \sqcap Pre^{\sharp +} = \bot \qquad \diamond$$

Definition 3.11 (Compile-Time Checking of Success Assertion Condition). Given the abstract interpretation $[\![\mathcal{P}]\!]^\sharp_{Q^\sharp}$ of a program $\mathcal{P}$ for a set of abstract queries $Q^\sharp$, we say that a success assertion condition success$(Pred, Pre, Post)$ is

$$\texttt{checked} \Leftrightarrow \forall \langle Pred, \lambda^c, \lambda^s \rangle \in [\![\mathcal{P}]\!]^\sharp_{Q^\sharp}.\ (\lambda^c \sqcap Pre^{\sharp +} = \bot) \vee (\lambda^s \sqsubseteq Post^{\sharp -})$$
$$\texttt{false} \Leftrightarrow \forall \langle Pred, \lambda^c, \lambda^s \rangle \in [\![\mathcal{P}]\!]^\sharp_{Q^\sharp}.\ (\lambda^c \sqsubseteq Pre^{\sharp -}) \wedge (\lambda^s \sqcap Post^{\sharp +} = \bot) \qquad \diamond$$

Chapter 4

Specifying and Verifying Higher-Order Programs

In higher-order (C)LP, variables can be bound to predicate symbols that are later invoked. This naturally gives rise to the need of expressing conditions on these predicates that must hold during program execution—much like how properties are used for expressing conditions on first-order terms. This chapter presents the main contribution of this thesis. First, in §4.1 we revisit the notion of *predicate properties* first presented in [8], and we provide a refinement of both the syntax and semantics of these properties. Subsequently, in §4.2 we introduce the notion of *conformance* to a predicate property in the setting of the concrete semantics, *i.e.*, a predicate "behaving correctly w.r.t." a predicate property. Together with its abstract counterpart, *abstract conformance*, an alternative to conformance which is statically computable, *i.e.*, at compile-time. Next, in §4.3 we present a programming technique for creating predicate property-tailored predicates for simplifying the computation of abstract conformance, among other advantages to be presented later. Afterwards, in §4.4 we introduce a reduction from predicate properties to first-order properties, as a technique for reasoning about higher-order properties using a first-order static analyzer. Finally, in §4.5 we present a high-level algorithm for the compile-time verification of a program with higher-order predicates and assertions using all the concepts presented in the previous sections.

4.1 Predicate Properties

Predicate properties were already introduced in [8], which we revise and refine here.[1] 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. This placeholder can be *instantiated* with a predicate symbol p,

[1] We propose a more compact syntax here that avoids having to use a named variable for the anonymous predicate symbol (as in [8]) and takes advantage of functional notation (`:=`).

producing a traditional assertion for p. We now provide the formal definitions and some examples.

Definition 4.1 (Anonymous Assertion). An *anonymous assertion* $\mathcal{A}$ 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.[2] Instantiating _ with a specific predicate symbol p produces a traditional assertion for p derived from the anonymous assertion, denoted $\mathcal{A}|_p$. ◊

Example 4.1 (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 4.2 (Predicate Property). A *predicate property* Π is defined as a set of anonymous assertions $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$. Its syntax is:

$$
\begin{array}{l}
\Pi \texttt{ := \{} \\
\quad \texttt{:- pred } \_(\bar{v}) \texttt{ : } \mathit{Pre}_1 \texttt{ => } \mathit{Post}_1\texttt{.} \\
\quad \ldots \\
\quad \texttt{:- pred } \_(\bar{v}) \texttt{ : } \mathit{Pre}_n \texttt{ => } \mathit{Post}_n\texttt{.} \\
\texttt{\}.}
\end{array}
$$

and the function $\mathrm{ar}(\Pi)$ 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 $\Pi|_p = \{\mathcal{A}_1|_p, \ldots, \mathcal{A}_n|_p\}$. ◊

Similarly to our use of $\mathcal{A}$ to denote a set of assertions and its associated set of assertion conditions, we use Π to refer to both a set of anonymous assertions and, interchangeably, the corresponding set of anonymous assertion conditions. In this way, the instantiation of a predicate property Π to a specific predicate symbol is naturally extended to its associated set of anonymous assertion conditions.

Example 4.2 (Predicate Property). Consider the program in Listing 3, where we define the predicate property `int_op` ("being a predicate that behaves as an integer nondeterministic binary operator"), and (partially) specify the higher-order predicate `eval/4` using the `int_op` predicate property.

The predicate property literal `int_op(IntOp)` states that `IntOp` should be bound to a 3-ary predicate such that, if it is called with its first two arguments bound to integers, then its third argument should be bound to an integer upon success. The assertion for the `eval/4` predicate states that it must be called with its first two arguments bound

[2] We also use for compactness "_" as anonymous functor, a syntactic extension from the Ciao `hiord` package [30], but double quotes `''` can also be used to stay within ISO-Prolog syntax.

```
int_op := { :- pred _(X, Y, Z) : (int(X), int(Y)) => int(Z). }.

:- pred eval(A, B, IntOp, R) : (int(A), int(B), int_op(IntOp)) => int(R).
```

Listing 3: `int_op` predicate property.

to integers and its third argument bound to a predicate that *conforms* to the `int_op` predicate property, and that if any such call succeeds, then its third argument should be bound to an integer. ◊

Once we have established how to specify higher-order programs using predicate properties, we will now concentrate on how to verify such programs.

4.2 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. As mentioned before, we refer to a predicate p behaving correctly w.r.t. Π as p *conforming* to Π. In this section we refine and formalize this notion of conformance. Our overall objective is to be able to safely approximate the set of predicates that X can be bound to without violating the conditions imposed by the predicate property Π.

It is important to note that, for the purposes of determining conformance, the assertions of the predicates in the program can be provided by the user, inferred by static analysis, or a combination of both. Throughout the rest of the discussion, let $\mathcal{P}$ be a program, p be a predicate s.t. $p \in \mathcal{P}$, Π be a predicate property, and $\mathcal{A}$ be the set of assertion conditions of $\mathcal{P}$.

4.2.1 Conformance

Definition 4.3 (Covered Predicate). Given the calls assertion condition $\mathrm{calls}(p(\bar{v}), Pre) \in \mathcal{A}$, and the anonymous calls assertion condition $\mathrm{calls}(\_(\bar{v}), {}^{\circ}Pre) \in \Pi$ associated to Π. We say that the predicate p can be *covered* with the predicate property Π iff

$$ {}^{\circ}Pre^{\natural} \subseteq Pre^{\natural} $$ ◊

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

Lemma 4.1. Under the same conditions as in Definition 4.3,

$$ {}^{\circ}Pre^{\sharp+} \sqsubseteq Pre^{\sharp-} \Rightarrow p \text{ can be covered with } \Pi $$ ◊

Proof. The proof proceeds by assuming ${}^{\circ}Pre^{\sharp+} \sqsubseteq Pre^{\sharp-}$ and showing ${}^{\circ}Pre^{\natural} \subseteq Pre^{\natural}$.

$$
\begin{aligned}
& {}^{\circ}Pre^{\sharp+} \sqsubseteq Pre^{\sharp-} \\
\Leftrightarrow\ & \gamma({}^{\circ}Pre^{\sharp+}) \subseteq \gamma(Pre^{\sharp-}) && \text{⟅(2.1)⟆} \\
\Leftrightarrow\ & {}^{\circ}Pre^{\natural} \subseteq \gamma({}^{\circ}Pre^{\sharp+}) \subseteq \gamma(Pre^{\sharp-}) \subseteq Pre^{\natural} && \text{⟅Defs. 3.8 and 3.9⟆} \\
\Leftrightarrow\ & {}^{\circ}Pre^{\natural} \subseteq Pre^{\natural} && \text{⟅Def. 2.14⟆}
\end{aligned}
$$

□

Definition 4.4 (Redundance). Under the same conditions as in Definition 4.3, given that the predicate p can be covered with the predicate property Π, we define the set of assertion conditions $\mathcal{A}'$ as

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

where $C = \mathsf{calls}(p(\bar{v}), Pre) \in \mathcal{A}$ and ${}^{\circ}C = \mathsf{calls}(\_(\bar{v}), {}^{\circ}Pre) \in \Pi$.

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 all states $\langle G \mid \theta \mid \mathcal{E} \rangle$ 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

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

and

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

it holds that $D_{[-1]} \leadsto^*_{\mathcal{A}} \langle G \mid \theta \mid \{\ell\} \rangle$ through a derivation that reduces only check literals (if any at all),[3] 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 $p \in \mathcal{P}$ under a query Q_p to p iff augmenting the original set of assertion conditions ($\mathcal{A}$) with that of Π instantiated to p ($\mathcal{A}'$) does not introduce new run-time check errors in any derivation of $\mathcal{P}$ starting from Q_p. That is, for any derivation D' of $\mathcal{P}$ with $\mathcal{A}'$ from Q_p that ends in an erroneous state, and for any derivation D of $\mathcal{P}$ with $\mathcal{A}$ from Q_p that is equivalent—modulo check literals—to D', then D ends in an erroneous state after 0 to n check literal-reduction steps.

Definition 4.5 (Conformance). Let $\mathcal{P}$ be a program, p be a predicate s.t. $p \in \mathcal{P}$, Π be a predicate property, and $\mathcal{Q}_p$ be the set of all possible queries to p. We say that p *conforms* to Π, denoted $p \prec \Pi$ iff

$$\forall Q_p \in \mathcal{Q}_p.\ \Pi \textit{ is redundant for } p \textit{ under } Q_p$$

Conversely, we say that p *does not conform to* Π, denoted $p \not\prec \Pi$ iff

$$\exists Q_p \in \mathcal{Q}_p.\ \Pi \textit{ is not redundant for } p \textit{ under } Q_p$$ ◊

[3] Note that this implies $\mathcal{U}(G) = \mathcal{U}(G')$.

4.2.2 Abstract Conformance

Intuitively, 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, denoted $\not\prec^{\sharp+}$, captures the predicates that are known not to conform.

We now provide sufficient conditions for determining (non-)abstract conformance.

Definition 4.6 (Abstract Conformance on "Calls"). Let Pre be the pre-condition of the *calls* assertion condition for p in $\mathcal{A}$, and $\complement C$ be an anonymous calls assertion condition $\mathsf{calls}(\_(\bar{v}), \complement Pre)$. We say that the predicate p *abstractly conforms on calls* to the anonymous assertion condition $\complement C$, denoted $p \prec^{\sharp-} \complement C$, iff

$$(Pre^{\sharp+} \sqsubseteq \complement Pre^{\sharp-}) \wedge (Pre^{\sharp-} \sqsupseteq \complement Pre^{\sharp+})$$

Conversely, we say that p *does not abstractly conform on calls* to $\complement C$, denoted $p \not\prec^{\sharp+} \complement C$, iff

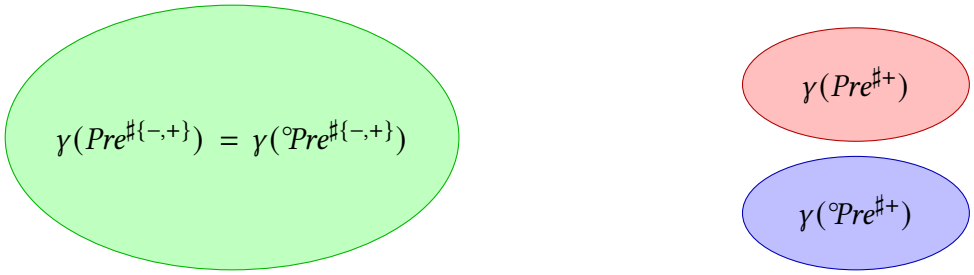
$$Pre^{\sharp+} \sqcap \complement Pre^{\sharp+} = \bot \qquad \diamond$$

Intuitively, when determining *abstract conformance* to an anonymous calls condition, we need both pre-conditions to be (essentially) equivalent. That is, they both succeed for the same call states. This is to avoid unexpected behavior, either by getting *more* (or *less*) run-time check errors than expected.

Although this condition seems—and is, indeed—hard to satisfy, we later provide an example that illustrates the need for such a strict condition; which also serves as motivation for the programming technique based in *wrappers* presented in §4.3 as a way of alleviating this condition and turning it into a trivial check

Conversely, when determining *non abstract conformance* to an an anonymous calls condition, we require that their pre-conditions describe *completely disjoint* call patterns. This way we are sure that regardless of the concrete semantics of the predicate, it is—for sure—not conformant to the anonymous calls assertion condition.

An illustration of the conditions above is depicted in Figure 4.1.



(a) Abstract conformance on *calls*.

(b) Non abstract conformance on *calls*.

Figure 4.1: Venn diagram representations of the abstract conformance on *calls* conditions (Definition 4.6). More concretely, the set inclusion relation between the concretizations of the property formulas of the involved calls assertion conditions.

Definition 4.7 (Abstract Conformance on "Success"). Let $\mathcal{C}$ be an anonymous success assertion condition success$(_(\bar{v}), {}^{\circ}Pre, {}^{\circ}Post)$. We say that the predicate p *abstractly conforms on success* to the anonymous assertion condition $\mathcal{C}$, denoted $p \prec^{\sharp-} \mathcal{C}$, iff

$$\exists S \subset \mathcal{A}.\ (Pre^{\sharp-}_{\sqcup} \sqsupseteq {}^{\circ}Pre^{\sharp+}) \wedge (Post^{\sharp+}_{\sqcup} \sqsubseteq {}^{\circ}Post^{\sharp-})$$

where

$$Pre^{\sharp-}_{\sqcup} = \sqcup\{Pre^{\sharp-} \mid \text{success}(p(\bar{v}), Pre, \_) \in S\}$$
$$Post^{\sharp+}_{\sqcup} = \sqcup\{Post^{\sharp+} \mid \text{success}(p(\bar{v}), \_, Post) \in S\}$$

Conversely, we say that p *does not abstractly conform on success* to $\mathcal{C}$, denoted $p \not\prec^{\sharp+} \mathcal{C}$, iff

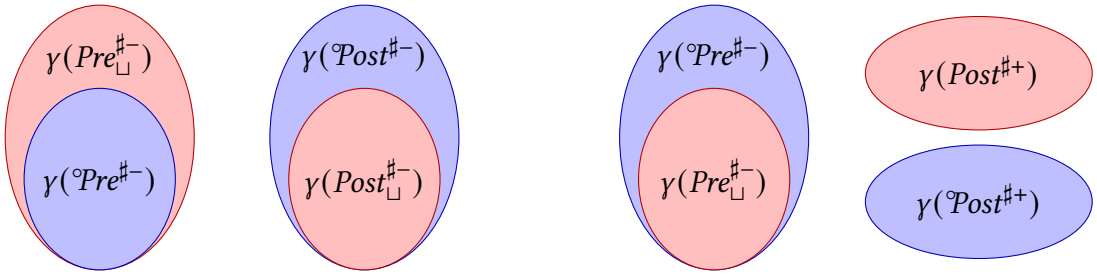
$$\exists\, \text{success}(p(\bar{v}), Pre, Post) \in \mathcal{A}.\ (Pre^{\sharp+} \sqsubseteq {}^{\circ}Pre^{\sharp-}) \wedge (Post^{\sharp+} \sqcap {}^{\circ}Post^{\sharp+} = \bot) \wedge$$
$$\wedge\, \exists \theta \in Pre^{\natural}.\ \mathcal{S}_{\mathcal{A}}(p(\bar{v}), \theta, \mathcal{P}, \gamma(Q^{\sharp}_{p})) \neq \varnothing$$

where $Q^{\sharp}_{p}$ is the set of abstract queries s.t. $\gamma(Q^{\sharp}_{p})$ is a *superset* of the set of valid queries to p described by p's calls assertion condition in $\mathcal{A}$. ◊

Intuitively, when determining *abstract conformance* to an anonymous success assertion condition, we look for a *subset* of the success assertion conditions of the predicate such that:

- *All* of their pre-conditions succeed for—at least—all the call states for which the anonymous success assertion condition pre-condition succeeds (first conjunct). That is, they are *more general*, *i.e.*, describing a broader set of call states for which the post-conditions are applicable.
- *All* of their post-conditions succeed for—at most—all the success states for which the anonymous success assertion condition post-condition succeeds (second conjunct). That is, they are *less general*, *i.e.*, describing a smaller set of success states for which the post-conditions succeed.[4]

[4]The reader may notice that these two concepts are—in a way—similar to the notion of *covariance* and



(a) Abstract conformance on *success*.

(b) Non abstract conformance on *success*.

Figure 4.2: Venn diagram representations of the abstract conformance on *success* conditions (Definition 4.7). More concretely, the set inclusion relation between the concretizations of the property formulas of the involved success assertion conditions.

As discussed in §3.4, for abstract domains which may lose precision with their ($\sqcup$) abstract operator, more advanced techniques for leveraging multiple abstractions become necessary. See, for instance, *covering* [32, 33] or *abstraction partitioning* [34, 35], which continue to be active research topics in the field of abstract interpretation.

Conversely, when determining *non abstract conformance* to an anonymous success assertion condition, we look for a success assertion condition of the predicate such that:

- Its pre-condition succeeds for—at most—all the call states for which the anonymous success assertion condition pre-condition succeeds (first conjunct).
- Its post-condition succeeds for a set of success states that is *disjoint* from that of the anonymous success assertion condition post-condition (second conjunct).

An illustration on the conditions above is depicted in Figure 4.2.

However, if you recall Definition 4.5, for determining *non conformance*, we need to find a counter-example: an *erroneous* derivation with the augmented set of assertions such that the corresponding derivation with the original set of assertions is *successful*. So, what if the predicate does *not* have *any* possible successful derivation? To make our argument more concrete, consider the following example.

Example 4.3 (The Need for a Success State). Given the program in Listing 4, we may be tempted to conclude that `p/1` *does not abstractly conform on success* to `p_nat_nat`, since `Y` would be bound to an integer upon success—violating the post-condition of `p/1`. However, consider the case in which the `opaque/1` predicate *always fails*, but that we cannot detect it in any way. For example, a library predicate, external call via a *foreign function interface*, etc. Then, according to Definition 4.5, `p/1` would *conform* to `p_nat_nat`, since every possible derivation of `p/1` would not proceed further than line 7, hence not raising a run-time check error. Although it would raise a run-time check error if execution reaches line 8. ◊

contravariance in the context of subtyping of function types [31]. However, since we are dealing with multiple *modes* (or *usages*) of a single predicate, it can be seen as a *generalization* of such notion.

```
% 2-ary predicates that, when called with `X` bound to an
% integer, `Y` is unified with an integer upon success.
p_nat_nat := { :- pred _(X, Y) : nat(X) => nat(Y). }.

:- pred p(X, Y) : nat(X) => atm(Y).
p(4, Y) :-
    opaque(4), % This call always fails.
    Y = 4.     % This unification is unreachable.
```

Listing 4: Example program invoking an opaque predicate which always fails.

To this end, we add the third conjunct in the definition of non abstract conformance on *success*, which encodes the existence of at least one successful state for the predicate with its set of assertions.

We now lift the notion of abstract conformance and the definitions presented above to predicate properties.

Definition 4.8 (Abstract Conformance). We say that the predicate p *abstractly conforms* to a predicate property Π, denoted $p \prec^{\sharp-} \Pi$, iff

$$\forall\, ^{\circ}C \in \Pi.\ p \prec^{\sharp-} {}^{\circ}C$$

Conversely, we say that p *does not abstractly conform* to Π, denoted $p \not\prec^{\sharp+} \Pi$, iff

$$\exists\, ^{\circ}C \in \Pi.\ p \not\prec^{\sharp+} {}^{\circ}C \qquad \diamond$$

We now relate the notions of conformance and abstract conformance.

Theorem 4.1. Let p be a predicate and Π be a predicate property. Then,

$$p \prec^{\sharp-} \Pi \Rightarrow p \prec \Pi \qquad (\prec^{\sharp-})$$
$$p \not\prec^{\sharp+} \Pi \Rightarrow p \not\prec \Pi \qquad (\not\prec^{\sharp+})$$

$\diamond$

Proof. Let $\mathcal{P}$ be a program, p be a predicate s.t. $p \in \mathcal{P}$, and Π be a predicate property. We will prove each case separately.

- ($\prec^{\sharp-}$). We proceed by contradiction. Assume that p abstractly conforms to Π:

  $$p \prec^{\sharp-} \Pi \tag{4.1}$$

  and that p does not conform to Π:

  $$p \not\prec \Pi \tag{4.2}$$

  First, by (4.1) and Lemma 4.1, we know that p can be *covered* with Π, hence

  $$^{\circ}Pre^{\natural} \subseteq Pre^{\natural} \tag{4.3}$$

From (4.2) we know that there exists a query $Q_p = (p(\bar{v}), \theta_0)$ to p s.t. Π is *not* redundant for p. Thus, by negating the redundance condition from Definition 4.4, we know that: $\exists D' \in \mathsf{derivs}_{\mathcal{A}'}(\mathcal{P}, Q_p)$. $D'_{[-1]} = \langle G' \mid \theta \mid \{\ell'\}\rangle$, and $\exists D \in \mathsf{derivs}_{\mathcal{A}}(\mathcal{P}, Q_p)$. $\mathcal{U}(D) = \mathcal{U}(D')$, s.t. $D_{[-1]} \leadsto^*_{\mathcal{A}} \langle G \mid \theta \mid \varnothing\rangle$ through any derivation that reduces only check literals; where $\mathcal{A}'$ is defined as in Definition 4.5.

We also know that both $\mathsf{derivs}_{\mathcal{A}'}(\mathcal{P}, Q_p)$ and $\mathsf{derivs}_{\mathcal{A}}(\mathcal{P}, Q_p)$ share the same initial state $S = \langle p(\bar{v}) \mid \theta_0 \mid \varnothing\rangle$ given that they are deriving the same query Q_p to p.

We now consider two cases: the error comes from a calls or success assertion condition in $\mathcal{A}'$.

Calls: The label ℓ' identifies a calls assertion condition C' in $\mathcal{A}'$—of the form $\mathsf{calls}(p(\bar{v}), Pre \wedge {}^\circ Pre)$ (from the definition of $\mathcal{A}'$)—which fails to be checked for θ; and the calls assertion condition C in $\mathcal{A}$—of the form $\mathsf{calls}(p(\bar{v}), Pre)$—is checked for θ.

On the one hand:

$$
\begin{aligned}
& C' \text{ fails to be checked for } \theta \\
\Leftrightarrow\ & \theta \not\models Pre \wedge {}^\circ Pre && \text{⟅Semantics in Fig. 3.3⟆} \\
\Leftrightarrow\ & \theta \notin (Pre \wedge {}^\circ Pre)^\natural && \text{⟅Def. 3.7⟆} \\
\Leftrightarrow\ & \theta \notin Pre^\natural \cap {}^\circ Pre^\natural && \text{⟅Lem. 3.1⟆} \\
\Leftrightarrow\ & \theta \notin {}^\circ Pre^\natural && \text{⟅(4.3)⟆}
\end{aligned}
$$

On the other hand:

$$
\begin{aligned}
& C \text{ is checked for } \theta \\
\Leftrightarrow\ & \theta \models Pre && \text{⟅Semantics in Fig. 3.3⟆} \\
\Leftrightarrow\ & \theta \in Pre^\natural && \text{⟅Def. 3.7⟆}
\end{aligned}
$$

Since $p \prec^{\sharp-} \Pi$ (4.1), we know from Definition 4.6 that:

$$
\begin{aligned}
& Pre^{\sharp+} \sqsubseteq {}^\circ Pre^{\sharp-} \\
\Leftrightarrow\ & \gamma(Pre^{\sharp+}) \subseteq \gamma({}^\circ Pre^{\sharp-}) && \text{⟅(2.1)⟆} \\
\Leftrightarrow\ & Pre^\natural \subseteq \gamma(Pre^{\sharp+}) \subseteq \gamma({}^\circ Pre^{\sharp-}) \subseteq {}^\circ Pre^\natural && \text{⟅Defs. 3.8 and 3.9⟆}
\end{aligned}
$$

Since $\theta \in Pre^\natural$ and $\subseteq$ is a partial order, it implies that $\theta \in {}^\circ Pre^\natural$, contradicting $\theta \notin {}^\circ Pre^\natural$. ↯

Success: The label ℓ' identifies a success assertion condition C' in $\mathcal{A}'$—of the form $\mathsf{success}(p(\bar{v}), {}^\circ Pre, {}^\circ Post)$—which fails to be checked for θ; and all success assertion conditions C_i in $\mathcal{A}$—of the form $\mathsf{success}(p(\bar{v}), Pre_i, Post_i)$ with $i \in 1..n$—is checked for θ.

On the one hand:

$$
\begin{array}{ll}
& C' \text{ fails to be checked for } \theta \\
\Leftrightarrow & (\theta_0 \Longmapsto {}^{\circ}\mathit{Pre}) \wedge (\theta \not\Longmapsto {}^{\circ}\mathit{Post}) & \text{(Semantics in Fig. 3.3)} \\
\Leftrightarrow & (\theta_0 \in {}^{\circ}\mathit{Pre}^{\natural}) \wedge (\theta \notin {}^{\circ}\mathit{Post}^{\natural}) & \text{(Def. 3.7)}
\end{array}
$$

Since $p \prec^{\sharp-} \Pi$ (4.1), we know from Definition 4.7 that there exists a proper subset of success assertion conditions $S \subset \mathcal{A}$ s.t.:

$$(\mathit{Pre}^{\sharp-}_{\sqcup} = \sqcup\{\mathit{Pre}^{\sharp-} \mid \mathsf{success}(p(\bar{v}), \mathit{Pre}, \_) \in S\}) \sqsupseteq {}^{\circ}\mathit{Pre}^{\sharp+} \tag{4.4}$$

and

$$(\mathit{Post}^{\sharp+}_{\sqcup} = \sqcup\{\mathit{Post}^{\sharp+} \mid \mathsf{success}(p(\bar{v}), \_, \mathit{Post}) \in S\}) \sqsubseteq {}^{\circ}\mathit{Post}^{\sharp-} \tag{4.5}$$

From (4.4) and (4.5), and Definitions 3.8 and 3.9, we obtain the following relations:

$$\cup\{\mathit{Pre}^{\natural} \mid \mathsf{success}(p(\bar{v}), \mathit{Pre}, \_) \in S\} \supseteq \gamma(\mathit{Pre}^{\sharp-}_{\sqcup}) \supseteq \gamma({}^{\circ}\mathit{Pre}^{\sharp+}) \supseteq {}^{\circ}\mathit{Pre}^{\natural} \tag{4.6}$$

and

$$\cup\{\mathit{Post}^{\natural} \mid \mathsf{success}(p(\bar{v}), \_, \mathit{Post}) \in S\} \subseteq \gamma(\mathit{Post}^{\sharp+}_{\sqcup}) \subseteq \gamma({}^{\circ}\mathit{Post}^{\sharp-}) \subseteq {}^{\circ}\mathit{Post}^{\natural} \tag{4.7}$$

Finally, since $\theta_0 \in {}^{\circ}\mathit{Pre}^{\natural}$:

$$
\begin{array}{ll}
& \theta_0 \in {}^{\circ}\mathit{Pre}^{\natural} \\
\Leftrightarrow & \theta_0 \in \cup\{\mathit{Pre}^{\natural} \mid \mathsf{success}(p(\bar{v}), \mathit{Pre}, \_) \in S\} & \text{((4.6))} \\
\Leftrightarrow & \exists (C = \mathsf{success}(p(\bar{v}), \mathit{Pre}, \mathit{Post})) \in S.\ \theta_0 \in \mathit{Pre}^{\natural} & \text{(Def. 2.1)} \\
\Leftrightarrow & \theta \Longmapsto \mathit{Post} & \text{(C checked for θ, hypothesis)} \\
\Leftrightarrow & \theta \in \mathit{Post}^{\natural} & \text{(Def. 3.7)} \\
\Leftrightarrow & \theta \in \cup\{\mathit{Post}^{\natural} \mid \mathsf{success}(p(\bar{v}), \_, \mathit{Post}) \in S\} & \text{(Def. 2.1)} \\
\Leftrightarrow & \theta \in {}^{\circ}\mathit{Post}^{\natural} & \text{((4.7))}
\end{array}
$$

which leads to a contradiction with our initial hypothesis $\theta \notin {}^{\circ}\mathit{Post}^{\natural}$. ↯

- ($\not\prec^{\sharp+}$). We proceed by direct proof. Since p does not abstractly conform to Π:

$$p \not\prec^{\sharp+} \Pi \tag{4.8}$$

we consider two cases: p does not abstractly conform to Π's anonymous calls assertion condition, or to some anonymous success assertion condition of Π.

Calls: Let $(\complement C = \text{calls}(\_(\bar{v}), \complement Pre)) \in \Pi$, we know that $p \not\prec^{\sharp+} \complement C$, then:

$$
\begin{aligned}
& p \not\prec^{\sharp+} \complement C \\
\Leftrightarrow\ & Pre^{\sharp+} \sqcap \complement Pre^{\sharp+} = \bot && \text{(Def. 4.6)} \\
\Leftrightarrow\ & Pre^{\natural} \cap \complement Pre^{\natural} = \varnothing && \text{(Def. 3.9)} \\
\Leftrightarrow\ & \forall \theta \in \complement Pre^{\natural}.\ \theta \notin Pre^{\natural} && \text{(Def. 2.2)} \\
\Rightarrow\ & \complement Pre^{\natural} \not\subset Pre^{\natural} && \text{(Def. 2.4)} \\
\Rightarrow\ & p \textit{ cannot be covered with } \Pi && \text{(Def. 4.3)} \\
\Rightarrow\ & p \not\prec \Pi && \text{(Defs. 4.4 and 4.5)}
\end{aligned}
$$

Success: Let $(\complement C = \text{success}(\_(\bar{v}), \complement Pre, \complement Post)) \in \Pi$, since $p \not\prec^{\sharp+} \complement C$, we know that there exists a success assertion condition $(C = \text{success}(p(\bar{v}), Pre, Post)) \in \mathcal{A}$ s.t. the following hold:

$$Pre^{\sharp+} \sqsubseteq \complement Pre^{\sharp-} \tag{4.9}$$

$$Post^{\sharp+} \sqcap \complement Post^{\sharp+} = \bot \tag{4.10}$$

$$\exists \theta_0 \in Pre^{\natural}.\ \mathcal{S}_{\mathcal{A}}(p(\bar{v}), \theta_0, \mathcal{P}, \gamma(Q_p^{\sharp})) \neq \varnothing \tag{4.11}$$

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

First, by Definitions 3.8 and 3.9, from (4.9) and (4.10) we obtain:

$$Pre^{\natural} \subseteq \complement Pre^{\natural} \tag{4.12}$$

$$Post^{\natural} \cap \complement Post^{\natural} = \varnothing \tag{4.13}$$

Next, from (4.11) and (4.12), it follows that $\theta_0 \in Pre^{\natural}$ and $\theta_0 \in \complement Pre^{\natural}$. Also, from (4.11) and Definition 3.5, we know that

$$\theta \in \mathcal{S}_{\mathcal{A}}(p(\bar{v}), \theta_0, \mathcal{P}, \gamma(Q_p^{\sharp}))$$

hence $\theta \in Post^{\natural}$. However, by (4.13) we know that $\theta \notin \complement Post^{\natural}$.

By Definition 3.7, this means that, on the one hand:

$$\theta_0 \mapsto Pre \wedge \theta \mapsto Post \tag{4.14}$$

And in the other hand:

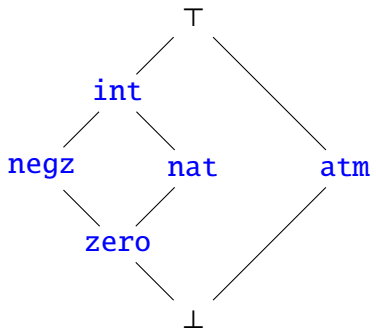
$$\theta_0 \mapsto \complement Pre \wedge \theta \not\mapsto \complement Post \tag{4.15}$$

The above conditions (4.14) and (4.15), together with the definition of the operational semantics with assertions in Figure 3.3, implies that there exists a query $Q_p = (p(\bar{v}), \theta_0)$ to p s.t. Π is *not* redundant for p. And, by the definition of *non conformance* (Definition 4.5), this implies that p *does not* conform to Π. □

```
% 2-ary predicates that, when called with `X` bound to an
% integer, `Y` is unified with an integer upon success.
p_nat_nat := { :- pred _(X, Y) : nat(X) => nat(Y). }. % Π

:- 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
```

(a) Predicate property and some assertions.



(b) Abstract domain lattice.

Figure 4.3: Example case analysis on a predicate property and assertions.

Example 4.4 (Abstract Conformance). Consider the source code in Figure 4.3a. We first define the predicate property `p_nat_nat`, which, for simplicity, we will interchangeably refer to as Π for the rest of the example. We then partially specify some predicates with assertions $\mathcal{A} = \{A_1, \ldots, A_5\}$. Notice that the property formulas of both Π and $\mathcal{A}$ include property literals of the properties (exactly) represented in the abstract domain whose (simplified) lattice is depicted in Figure 4.3b. The corresponding assertion conditions for Π and A_i are:

$$\Pi \equiv \{\texttt{calls(\_(X,Y), nat(X))}, \texttt{success(\_(X,Y), nat(X), nat(Y))}\}$$
$$A_1 \equiv \{\texttt{calls(n2n(X,Y), nat(X))}, \texttt{success(n2n(X,Y), nat(X), nat(Y))}\}$$
$$A_2 \equiv \{\texttt{calls(a2n(X,Y), atm(X))}, \texttt{success(a2n(X,Y), atm(X), nat(Y))}\}$$
$$A_3 \equiv \{\texttt{calls(i2z(X,Y), int(X))}, \texttt{success(i2z(X,Y), int(X), zero(Y))}\}$$
$$A_4 \equiv \{\texttt{calls(z2i(X,Y), zero(X))}, \texttt{success(z2i(X,Y), zero(X), int(Y))}\}$$
$$A_5 \equiv \{\texttt{calls(nz2n(X,Y), negz(X))}, \texttt{success(nz2n(X,Y), negz(X), nat(Y))}\}$$

Assume now that we would like to determine which of the predicates partially specified by assertions $\mathcal{A}$ abstractly conform to the predicate property Π. For each predicate and its corresponding pair of *calls* and *success* assertion conditions, we proceed as follows:

1. Determine abstract conformance to Π's anonymous calls condition;
2. Determine abstract conformance to Π's anonymous success condition; and,
3. Determine abstract conformance to Π.

Abstract Conformance on "Calls" to Π: Table 4.1 summarizes the abstract conformance analysis between the calls assertion condition of each predicate and Π's anony-

Table 4.1: Abs. conf. on "*calls*" example with $^{\circ}Pre = \texttt{nat(X)}$.

Pred	***Pre$_i$***	***Relation w. $^{\circ}$Pre***	***Abs. Conf.***
n2n(X,Y)	nat(X)	nat(X) = nat(X)	yes
a2n(X,Y)	atm(X)	atm(X) ⊓ nat(X) = ⊥	no
i2z(X,Y)	int(X)	int(X) ⊒ nat(X)	maybe
z2i(X,Y)	zero(X)	zero(X) ⊑ nat(X)	maybe
nz2n(X,Y)	negz(X)	negz(X) ⊓ nat(X) ≠ ⊥ ∧ negz(X) ⋢ nat(X) ∧ negz(X) ⋣ nat(X)	maybe

Table 4.2: Abs. conf. on "*success*" example with $^{\circ}Pre = \texttt{nat(X)}$, $^{\circ}Post = \texttt{nat(Y)}$.

Pred	***Pre$_i$***	***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) ⊓ nat(X) = ⊥	nat(Y)	nat(Y) = nat(Y)	maybe
i2n(X,Y)	int(X)	int(X) ⊒ nat(X)	zero(Y)	zero(Y) ⊑ nat(Y)	yes
z2i(X,Y)	zero(X)	zero(X) ⊑ nat(X)	int(Y)	int(Y) ⊒ nat(Y)	maybe
nz2n(X,Y)	negz(X)	negz(X) ⊓ nat(X) ≠ ⊥ ∧ negz(X) ⋢ nat(X) ∧ negz(X) ⋣ nat(X)	nat(Y)	nat(Y) = nat(Y)	maybe

mous calls assertion condition, by comparing their respective pre-conditions. We then apply Definition 4.6 to determine abstract conformance on *calls*.

Abstract Conformance on "Success" to Π: Table 4.2 summarizes the abstract conformance analysis between the success assertion condition of each predicate and Π's anonymous success assertion condition, by comparing their respective pre- and post-conditions. We then apply Definition 4.7 to determine abstract conformance on *success*.

Abstract Conformance to Π: As a summary, we have that the only predicate that definitely conforms to the p_nat_nat predicate property is n2n/2, since both of its assertion conditions conform to those of p_nat_nat. ◇

4.3 Wrappers

Consider a predicate p and a predicate property Π such that p can be covered by Π. From Definition 4.3 we know that given their respective pre-conditions *Pre* and $^{\circ}$*Pre* given by their calls assertion conditions, $Pre^{\natural} \supseteq {}^{\circ}Pre^{\natural}$. Thus, by Lemma 4.1 and according to Definition 4.6, p *may* abstractly conform to Π ($p \prec^{\sharp+} \Pi$).

The reason behind not concluding abstract conformance is that *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*. Intuitively, if that is the case for a query Q_p, p would not *conform* to Π according to Definition 4.5 since p would not be *redundant* for Π under Q_p (Definition 4.4). And this would mean that abstract conformance yields *unsound* results.

To make our argument more concrete consider the following example.

Example 4.5 (Weak Abstract Conformance). Consider a query (`foo(P)`, {`P = even`}) to the program:

```
1  % 1-ary predicates that must be called with a natural number.
2  p_nat := { :- pred _(N) : nat(N). }.
3
4  :- pred even(N) : int(N).
5  even(N) :-
6      integer(N),
7      0 is N mod 2.
8
9  :- pred foo(P) : p_nat(P).
10 foo(P) :- P(10).  % (1)
11 foo(P) :- P(-10). % (2)
```

Take a derivation of such query that starts by reducing to the body of the first clause (1)—line 10: 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)—line 11: 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 `even`/1 accepts any integer on calls.

Moreover, if we extended the program with a new clause for `foo`/1:

```
12 foo(P) :- bar(P). % (3)
13
14 % 1-ary predicates that must be called with a negative number.
15 p_neg := { :- pred _(N) : neg(N). }.
16
17 :- pred bar(P) : p_neg(P).
18 bar(P) :- P(-4).
```

then a clause like (3)—line 12—would be problematic. For this clause, looking at the assertion of `foo/1`, the predicate in `P` is required to conform to `p_nat`, which would raise a run-time check 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, since `even/1` actually accepts both natural and negative numbers.

So, as formalized in Definition 4.6, we could only conclude that `even/1` $\prec^{\sharp+}$ `p_nat`, *i.e.*, `even/1` *may* abstractly conform to the `p_nat` predicate property. ◊

In the example above, we motivate the need for such a restrictive condition for abstract conformance on calls (see Definition 4.6). However, we may want to use predicates whose set of admissible calls is greater than that of a predicate property, but without introducing 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 with a single clause $w(\bar{v}) \leftarrow p(\bar{v})$ with an assertion "`:- pred` $w(\bar{v})$ `:` $^{\circ}Pre$`.`" (note that fields of *pred* assertions, in this case the post-condition, can be omitted, equivalently to **tt**). 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 pre-condition of the predicate property.

Example 4.6 (Wrapper). As a follow-up of the previous example, consider *wrapping* the `even/1` predicate to restrict its set of admissible calls to natural numbers:

```
19 :- pred even_nat(N) : nat(N).
20 even_nat(N) :- even(N).
```

Intuitively, the wrapper `even_nat/1` for `even/1` with `p_nat` (trivially) conforms to `p_nat` (`even_nat/1` $\prec^{\sharp-}$ `p_nat`). Also, computing the condition for abstract conformance on calls of `even_nat/1` to `p_nat` becomes trivial, and the analyzer can now infer that the clause in line 12 should raise an error on calls (since `even_nat/1` no longer accepts negative numbers). ◊

Rationale for Explicit Wrappers. The design of this approach follows the philosophy behind the Ciao system [18], 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.

4.4 First-Order Representation of Predicate Properties

As mentioned in §2.1, the abstract interpretation-based static analyzer can *infer properties* about higher-order programs, and also *verify traditional (i.e., first-order) assertions*. However, here we 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/n`, and `q/n` strongly, and `r/n` weakly conform to `pp` as the following regular types:

$$\begin{aligned} \texttt{pp}^{-}\texttt{/1} &= \{\texttt{pp}^{-}\texttt{(p)}, \texttt{pp}^{-}\texttt{(q)}\}, \\ \texttt{pp}^{+}\texttt{/1} &= \{\texttt{pp}^{+}\texttt{(p)}, \texttt{pp}^{+}\texttt{(q)}, \texttt{pp}^{+}\texttt{(r)}\}. \end{aligned}$$

Or, using Ciao's functional notation:

```
:- regtype pp⁻/1.   pp⁻ := p | q.
:- regtype pp⁺/1.   pp⁺ := p | q | r.
```

Formally, given a predicate property Π, we define two associated regular types $\pi^{-}/1$ and $\pi^{+}/1$ that capture the set of predicates that *strongly* and *weakly* abstractly conform to Π as follows:

$$\begin{aligned} \pi^{-}/1 &= \{\pi^{-}(p) \mid p \in \mathcal{P} \wedge p \prec^{\sharp -} \Pi\}, \\ \pi^{+}/1 &= \{\pi^{+}(p) \mid p \in \mathcal{P} \wedge p \prec^{\sharp +} \Pi\}. \end{aligned}$$

Note that, by definition, $\pi^{-}/1$ is a *subtype* of $\pi^{+}/1$. These regular types reduce the compile-time checking of higher-order assertions to that of first-order assertions since they can be abstracted and inferred by several abstract domains; for concreteness we use the eterms abstract domain [36].

4.5 Algorithm

We now present the core algorithm for the compile-time verification of a higher-order program $\mathcal{P}$ with higher-order assertions $\mathcal{A}$ from a high-level point of view (Algorithm 1). We now describe each step:

1. First, it initializes a set of rules R, and it computes the regular type representations of each predicate property Π in $\mathcal{P}$, *i.e.*, the π^{-} and π^{+} predicates respectively

Algorithm 1: Verify a Higher-Order Program with Higher-Order Assertions

Input : Program: $\mathcal{P}$, Assertions: $\mathcal{A}$, Abstract Queries: $Q^\sharp$
Output: Verified Status (`checked`/`false`/`check`) for the Assertions $\mathcal{A}$ of $\mathcal{P}$: $\mathcal{V}$

1 $R \leftarrow \emptyset$
2 **repeat**
3 $\quad R' \leftarrow R$
4 $\quad$ **foreach** *Predicate Property* $\Pi \in \mathcal{P}$ **do**
5 $\quad\quad R \leftarrow R \cup \{\pi^-(p) \mid p \in \mathcal{P} \wedge p \prec^{\sharp-} \Pi\} \cup \{\pi^+(p) \mid p \in \mathcal{P} \wedge p \prec^{\sharp+} \Pi\}$
6 **until** $R = R'$
7 $\mathcal{V} \leftarrow \mathsf{acheck}(\mathcal{A}, [\![\mathcal{P} \cup R]\!]^\sharp_{Q^\sharp})$ *// first-order assertion checking process*

(lines 4 and 5). These computations are performed by directly applying Definitions 4.6 to 4.8 using the operators of the abstract domain and the first-order analyzer.

2. Since predicate properties can include predicate property literals from other predicate properties—*i.e.*, there can be *dependencies* among predicate properties—lines 4 and 5 are repeated until a *fixpoint* is reached (between lines 2 and 6).

3. Next, it computes the abstract interpretation of $\mathcal{P}$, augmented with the regular type representations of every predicate property, for the set of abstract queries $Q^\sharp$ (line 7).

4. 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 7).

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 [8].

We argue that, despite the inherent complexity of the verification problem in hand, the proposed concepts make the resulting compile-time checking algorithm clear and concise; and, more importantly, easily implementable using a first-order assertion checker.

Chapter 5

Implementation and Experiments

For demonstrating the potential of our approach, we have developed a prototype implementation in the form of a Ciao *bundle* named *Hiord*$^\sharp$, which is integrated into the Ciao/CiaoPP system. In §5.1 we describe how the approach is implemented using *program transformation* techniques and the *abstract domain* operations. Then, in §5.2 we present a set of experiments which serves as a representative set of programs with the most common used higher-order patterns that were not possible to verify until this point.

5.1 Implementation

The *Hiord*$^\sharp$ bundle has been implemented and integrated within the CiaoPP framework. It acts as a pre-processor for the input program, assertions, and predicate properties, producing a transformed program along with a set of (now first-order) assertions that can be analyzed and checked by CiaoPP. A high-level view of CiaoPP extended with *Hiord*$^\sharp$ is depicted in Figure 5.1.

Essentially, *Hiord*$^\sharp$ begins by normalizing the predicate properties to an internal representation. It then implements Algorithm 1, relying in PLAI to compute under- and over-approximations of the involved property formulas, and on the abstract domain operations for comparing such approximations. Finally, it asserts the grounded versions of predicate properties into the program as regular types (as explained in §4.4).

Focusing on the abstract conformance computation step, recall that the condition for determining non abstract conformance on success (presented in Definition 4.7) is a conjunction of three sub-conditions. While the first two conjuncts are reliably checkable at compile time using PLAI and the abstract domain operations, the third conjunct is, in general, *undecidable*. It can be approximated using significantly more computationally expensive techniques, such as non-failure analyses [32, 33], but these are not practical for general use, as the condition in question only affects a small number of corner cases.

For this reason, the implementation *omits* checking the third conjunct. This omission

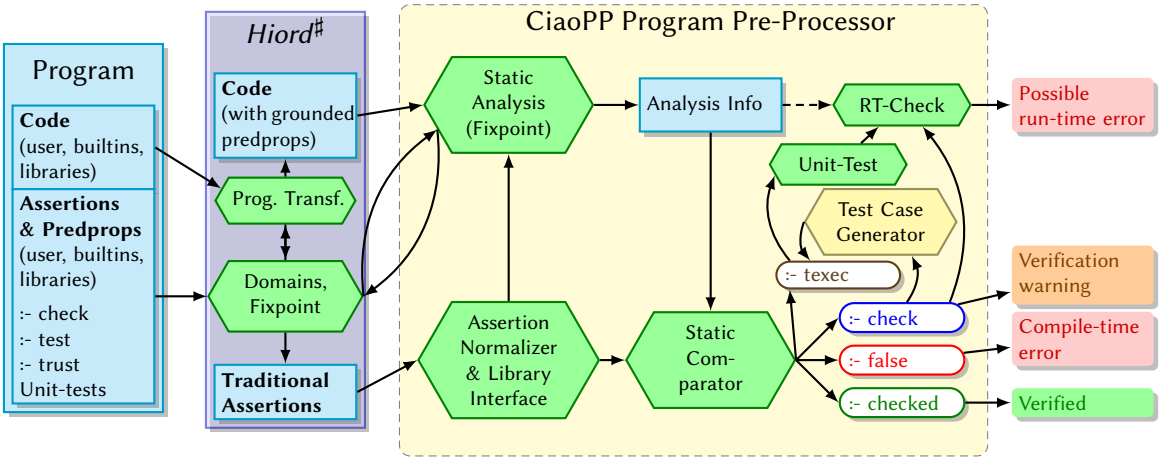


Figure 5.1: High-level view of the CiaoPP program pre-processor including *Hiord*♯.

makes *Hiord*♯ *conservative*: it may classify some predicates—typically corner cases such as the one in Example 4.3—as non abstractly conformant on success, even though they *may* in fact conform (in the concrete semantics setting). This simplification is *sound*, since the omitted conjunct is part of a *negative condition*; therefore, skipping it cannot cause non-conformant predicates to be classified as conformant—it may only lead to some predicates being conservatively rejected.

5.2 Experiments

We now provide a detailed description of the most relevant experiments that we ran.

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. This test case was explicitly designed to exhaustively cover all possible orderings between the *calls* pre-conditions of the predicate property and of each predicate, and likewise for the *success* pre- and post-conditions.

We defined the following regular types:

```
1 :- regtype p0/1.  :- regtype p1/1.  :- regtype p2/1.      :- regtype dj/1.
2 p0 := 0 | 1.      p1 := 0 | 1 | 2.  p2 := 0 | 1 | 2 | 3. dj := -1.
3
4 :- regtype q0/1.  :- regtype q1/1.  :- regtype q2/1.      :- regtype mx/1.
5 q0 := a | b.      q1 := a | b | c.  q2 := a | b | c | d. mx := a | 0.
```

whose order relation in the concrete domain is depicted in Figure 5.2; which then induces their corresponding order relation in the eterms abstract domain.

Then, we defined the following predicate property `f11`:

```
6 f11 := { :- pred _(X, Y) : p1(X) => p1(Y). }.
```

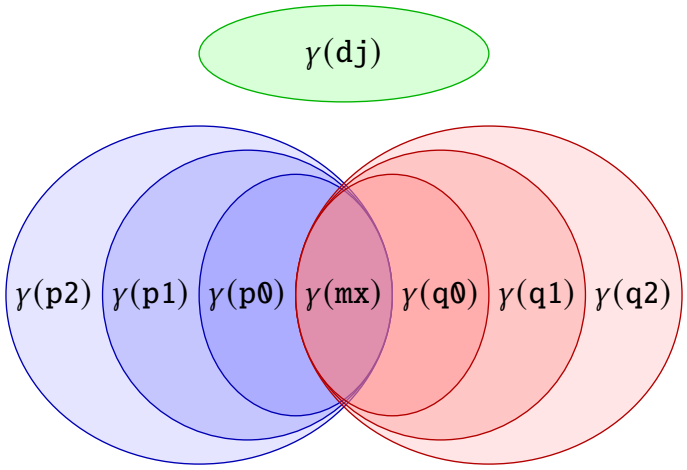


Figure 5.2: Venn diagram representation of the set inclusion relation between the concretizations of each regular type. Such relation then induces the order relation of the eterms abstract domain.

and tested conformance of predicates (partially) specified by assertions covering all possible orders. We now show an example of how each possible combination is encoded:

```
:- pred g00(X, Y) : p0(X) => q0(Y).
:- pred g01(X, Y) : p0(X) => q1(Y).
:- pred g02(X, Y) : p0(X) => q2(Y).
:- pred g0d(X, Y) : p0(X) => dj(Y).
:- pred g0m(X, Y) : p0(X) => mx(Y).
...
```

We then ran *Hiord*$^\sharp$ obtaining the correct results according to the abstract conformance definitions, reporting that from all tested combinations: 2 predicates *definitely conform*, 7 predicates *definitely not conform*, with 16 predicates left where no definite conclusion could be reached (at compile-time).

Higher-Order List Utilities. We defined various partially specified higher-order utility predicates specialized for working with lists of a particular type `t/1`, *e.g.*, `t := ~num`. For example, consider the `t_cmp` predicate property defined below:

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

which describes *comparator* predicates of elements of type `t/1`.

Next, we use such predicate property in the following higher-order assertion for a generic *quicksort* implementation that is parameterized by a comparator predicate `P`:

```
:- pred qsort(Xs, P, Ys) : (list(t, Xs), t_cmp(P)) => list(t, Ys).
qsort(Xs, P, Ys) :-
    qsort_(Xs, P, Ys, []).

qsort_([], _, R, R).
```

```
7  qsort_([X|L], P, R, R0) :-
8      partition(L, X, P, L1, L2),
9      qsort_(L2, P, R1, R0),
10     qsort_(L1, P, R, [X|R1]).
11
12 partition([], _, _, [], []).
13 partition([X|L], Y, P, [X|L1], L2) :-
14     P(X, Y), !,
15     partition(L, Y, P, L1, L2).
16 partition([X|L], Y, P, L1, [X|L2]) :-
17     partition(L, Y, P, L1, L2).
```

In the program point of the higher-order atom `P(X,Y)` in line 14, the static analysis is able to propagate the `t_cmp` predicate property to the predicate in `P` in that exact program point; and if in such program point `X` is inferred to be bound to, *e.g.*, an atom `a`, an error is indeed statically captured by *Hiord*$^\sharp$.

Consider the following comparator predicate:

```
18 :- pred lex(X, Y) : (term(X), term(Y)).
19 lex(X, Y) :- X @< Y.
```

For a query (`qsort(Xs, P, Ys)`, {`P` = `lex`}), *Hiord*$^\sharp$ reports a *warning* on *calls* since the predicate `lex/2` *may* or *may not* conform to the `t_cmp` predicate property. Intuitively, this is because `lex/2` has a *more general* call pattern than that of `t_cmp`, and thus, it will not raise a run-time check error when called with an element that is not of type `t/1`, introducing unexpected behavior.

However, consider defining a *wrapper* for `lex/2` with `t_cmp`:

```
20 :- pred lex_t(X, Y) : (t(X), t(Y)).
21 lex_t(X, Y) :- lex(X, Y).
```

Intuitively, for a query (`qsort(Xs, P, Ys)`, {`P` = `lex_t`}), *Hiord*$^\sharp$ is able to *prove* that such call would behave correctly w.r.t. `qsort/3`'s higher-order assertion, since `lex_t/2` *trivially* conforms to `t_cmp`.

As a follow-up, consider a predicate property which represents a parameterizable sorter of lists of elements of type `t/1`, defined "in terms of" the `t_cmp` predicate property:

```
22 t_sort := { :- pred _(Xs, P, Ys)
23              : (list(t, Xs), t_cmp(P)) => list(t, Ys). }.
```

For determining that `qsort/3` *abstractly conforms* to `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.

HTTP Server. Consider the following schematic interface for a generic HTTP server where we use regular types for representing requests and responses, and a predicate

property for representing handler predicates:

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

and the following partial specification of a higher-order *server* predicate `server/3`:

```
8  :- pred server(H, Req, Res) : (handler(H), req(Req)) => res(Res).
```

Such server is parameterized by a predicate that must be able to handle the four main REST-API operations that may be received by the server.

Consider the following (schematic) implementation of a handler predicate `h/2`:

```
9  h('DELETE', Res) :- ..., Res = 'OK'.
10 h('POST',   Res) :- ..., Res = 'CREATED'.
11 h('PUT',    Res) :- ..., Res = 'BAD_REQ'. % Typo, should be `BAD_REQUEST`.
12 h('GET',    Res) :- ..., Res = 'NOT_FOUND'.
```

Since `h/2` does not have a user-provided assertion, CiaoPP would infer the following assertion and regular type:

```
1  % Inferred by CiaoPP
2  :- pred h(Req, Res) : req(Req) => rt1(Res).
3
4  :- regtype rt1/1.
5  rt1 := 'OK'
6       | 'CREATED'
7       | 'BAD_REQ' % This case is different from that of `res/1`.
8       | 'NOT_FOUND'.
```

Given the definition of `rt1/1`, *Hiord*$^\sharp$ reports a warning stating that the `h/2` predicate *may* or *may not* conform to `handler`, since we introduced a typo in the definition of `h/2` (line 11). Note that in this situation, *Hiord*$^\sharp$ cannot conclude that `h/2` *does not definitely conform* to `handler` since `res/1` and `rt1/1` are not completely disjoint, and thus *conformance* would depend on the concrete semantics of the `h/2` predicate.

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).

A naive solution can be implemented by splitting the input list into three separate lists, one for each color, based on comparisons with a designated *neutral* element (often white). These lists are then concatenated in the desired order, yielding a sorted result. However, we want to generalize the solution by allowing the user to provide a comparator that given two elements, yields the result of the comparison.

We first define regular types to represent the colored elements and the result of their comparison:

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

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

Next, we define the dutch_cmp predicate property, that represents comparator predicates between two elements of type rwb/1:

```
dutch_cmp := { :- pred _(X, R, Y) : (rwb(X), rwb(Y)) => lge(R). }.
```

and also implement and partially specify a (naive) dutch_flag/3 higher-order predicate by providing a higher-order assertion:

```
:- pred dutch_flag(Cmp, L1, L2)
   : (dutch_cmp(Cmp), list(rwb, L1)) => list(rwb, L2).
dutch_flag(Cmp, L1, L2) :-
    dutch_flag_(Cmp, L1, Red, White, Blue),
    append(~append(Red, White), Blue, L2).

dutch_flag_(_,   [],     [],      [],    []).
dutch_flag_(Cmp, [E|L1], [E|Red], White, Blue) :-
    Cmp(E, <, w),
    dutch_flag_(Cmp, L1, Red, White, Blue).
dutch_flag_(Cmp, [E|L1], Red, [E|White], Blue) :-
    Cmp(E, =, w),
    dutch_flag_(Cmp, L1, Red, White, Blue).
dutch_flag_(Cmp, [E|L1], Red, White, [E|Blue]) :-
    Cmp(E, >, w),
    dutch_flag_(Cmp, L1, Red, White, Blue).
```

Assume that we need to program a color comparator predicate cmp/3. Consider a first attempt at implementing cmp/3 where we write the full name of the color instead of the first letter:

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

As in a previous example, since cmp/3 has no user-provided assertion, CiaoPP would infer the following assertion and regular type:

```
% Inferred by CiaoPP
:- pred cmp(X, R, Y) : (rt2(X), rt2(Y)) => lge(R).

:- regtype rt2/1.
rt2 := red
     | white
     | blue.
```

When determining its conformance to `dutch_cmp`, *Hiord*$^\sharp$ finds that it is *definitely not conformant*, since the regular type `rt2/1` inferred by CiaoPP is *completely disjoint* from that of the `dutch_cmp` predicate property (namely `rwb/1`). That is, in the eterms abstract domain: `rt2` $\sqcap$ `rwb` $= \bot$.

We proceed by correcting it, but in the process we accidentally mistype some of the `r` atoms for `o` atoms in lines 22 and 23:

```
22 cmp(o, =, r).  cmp(w, =, w).  cmp(b, =, b).
23 cmp(o, <, w).  cmp(w, >, r).  cmp(b, >, r).
24 cmp(r, <, b).  cmp(w, <, b).  cmp(b, >, w).
```

Again, CiaoPP would infer the following assertion and regular type:

```
1  % Inferred by CiaoPP
2  :- pred cmp(X, R, Y) : (rt3(X), rwb(Y)) => lge(R).
3
4  :- regtype rt3/1.
5  rt3 := r
6       | w
7       | b
8       | o.
```

However, *Hiord*$^\sharp$ is again able to detect that this version of `cmp/3` *does not definitely conform* to `dutch_cmp`, since it would not raise a run-time check error when called with an `o` on its first argument. That is, it has *more* admissible call patterns than `dutch_cmp` since `rwb` $\sqsubset$ `rt3` in the eterms abstract domain.

In an attempt at improving the precision of the ordering, we try to refine `cmp/3` to yield more informative results. In particular, we introduce new comparison outcomes `<<` and `>>` to indicate that `X` is "much lower" than `Y` and vice-versa. We also explicitly define an assertion for `cmp/3`, and define a new regular type with the new comparison outcomes.

```
22 :- pred cmp(X, R, Y) : (rwb(X), rwb(Y)) => lgLGe(R).
23 cmp(r,  =, r).  cmp(w, =, w).  cmp(b,  =, b).
24 cmp(r,  <, w).  cmp(w, >, r).  cmp(b, >>, r).
25 cmp(r, <<, b).  cmp(w, <, b).  cmp(b,  >, w).
26
27 :- regtype lgLGe/1.
28 lgLGe :=  < |  >
29        | << | >>
30        |  = .
```

However, when determining conformance of this refined version of `cmp/3` to `dutch_cmp`, *Hiord*$^\sharp$ also detects that `cmp/3` *is not definitely conformant*, since `cmp/3` could yield comparison results that are not reflected in `dutch_cmp`. That is, `lge` $\sqsubset$ `lgeLG` in the eterms abstract domain.

Finally, we develop the following implementation and assertion of `cmp/3`:

```
22 :- pred cmp(X, R, Y) : (rwb(X), rwb(Y)) => lge(R).
23 cmp(r, =, r).  cmp(w, =, w).  cmp(b, =, b).
24 cmp(r, <, w).  cmp(w, >, r).  cmp(b, >, r).
25 cmp(r, <, b).  cmp(w, <, b).  cmp(b, >, w).
```

For which *Hiord*$^\sharp$ determines that it *definitely conforms* to `dutch_cmp`, since it behaves exactly as expected, and we end up with a solution to the higher-order *dutch national flag* problem where the assertions are formally verified.

Chapter 6

Conclusions & Future Work

Looking back to the main objective of this thesis presented in §1.1, we have presented a novel approach for the *compile-time* verification of higher-order (C)LP programs with assertions that describe higher-order arguments.

We started by refining both the syntax and semantics of *predicate properties*, a special kind of property that allow using the full power of the (Ciao) assertion language for describing higher-order arguments.

Then, we introduced the notions of *conformance* in the concrete domain setting—in general not computable, and *abstract conformance* in the abstract domain setting which can be computed at compile-time by leveraging the *abstract interpretation* mathematical theory for approximating the concrete semantics of a program. We proposed (compile-time) conditions for determining abstract conformance, and then formally proved the relation between the notions of conformance and abstract conformance.

Afterwards, we motivated the use of *wrappers*, a technique for restricting the set of admissible calls of a predicate to match that of a predicate property, reducing the abstract conformance on calls condition to a trivial check in a program analysis friendly manner.

Next, we presented a *reduction* from predicate properties to regular types, which are first-order properties that are natively supported by the CiaoPP framework; and we provided a high-level algorithm for the verification of a higher-order program with higher-order assertions based in all the concepts presented before.

Finally, we reported on *Hiord*$^\sharp$, the prototype implementation of the technique that is part of the CiaoPP framework; and presented a representative set of experiments that were not possible to verify until this point.

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

6.1 Future Work

We end this thesis with some lines of future work.

Integration with Modular and Incremental Analysis. This thesis presented the theoretical foundations of an approach to tackle the problem at hand, and provided an initial working prototype integrated in the CiaoPP framework. However, it would be interesting to study how the approach can be adapted to the *modular* and *incremental* analysis techniques already present in CiaoPP.

More General forms of Higher-Order. Another interesting direction would be to explore how the approach can be extended to more general forms of higher-order constructs such as *predicate abstractions* or *partial applications*.

...As long as the Sun, the Moon, and the Earth exist, everything will be all right.

The End of Evangelion

Bibliography

[1] P. Körner, M. Leuschel, J. Barbosa, V. Santos-Costa, V. Dahl, M. V. Hermenegildo, J. F. Morales, J. Wielemaker, D. Diaz, S. Abreu, and G. Ciatto, "Fifty Years of Prolog and Beyond," *Theory and Practice of Logic Programming, 20th Anniversary Special Issue*, vol. 22, no. 6, pp. 776–858, May 2022. [Online]. Available: https://arxiv.org/abs/2201.10816

[2] S. Marlow, Ed., *Haskell 2010 Language Report*, 2010.

[3] M. Hermenegildo, G. Puebla, and F. Bueno, "Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging," in *The Logic Programming Paradigm: a 25–Year Perspective*, K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, Eds. Springer-Verlag, July 1999, pp. 161–192.

[4] G. Puebla, F. Bueno, and M. Hermenegildo, "Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs," in *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, ser. LNCS, no. 1817. Springer-Verlag, March 2000, pp. 273–292.

[5] L. Cardelli, "Typeful programming," in *Formal Description of Programming Concepts, based on a seminar organized by IFIP Working Group 2.2 and held near Rio de Janeiro in April 1989*, ser. IFIP State-of-the-Art Reports, E. J. Neuhold and M. Paul, Eds. Springer, 1989, p. 431. [Online]. Available: http://lucacardelli.name/Papers/TypefulProg.pdf

[6] J. G. Siek and W. Taha, "Gradual Typing for Functional Languages," in *Scheme and Functional Programming Workshop*, 2006, pp. 81–92.

[7] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *ACM Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, 1977, pp. 238–252.

[8] N. Stulova, J. F. Morales, and M. Hermenegildo, "Assertion-based Debugging of Higher-Order (C)LP Programs," in *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*. ACM Press, September 2014, pp. 225–235.

[9] D. Miller, "A logic programming language with λ-abstraction, function variables, and simple unification," *Journal of Logic and Computation*, vol. 1(4), pp. 497–536, 1991.

[10] Z. Somogyi, F. Henderson, and T. Conway, "The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language," *Journal of Logic Programming*, vol. 29, no. 1–3, pp. 17–64, October 1996.

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

[12] R. Kowalski and D. Kuehner, "Linear resolution with selection function," *Artificial Intelligence*, vol. 2, pp. 227–260, 1971.

[13] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel, "Un systeme de communication homme-machine en francais," *Rapport preliminaire, Groupe de Res. en Intell. Artif*, 1973.

[14] J. Jaffar and J.-L. Lassez, "Constraint Logic Programming," in *ACM Symposium on Principles of Programming Languages*. ACM, 1987, pp. 111–119.

[15] P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks," in *Sixth ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, 1979, pp. 269–282.

[16] P. Cousot, *Principles of abstract interpretation*. MIT Press, 2021.

[17] K. Muthukumar and M. Hermenegildo, "Compile-time Derivation of Variable Dependency Using Abstract Interpretation," *Journal of Logic Programming*, vol. 13, no. 2/3, pp. 315–347, July 1992.

[18] M. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, and G. Puebla, "An Overview of Ciao and its Design Philosophy," *TPLP*, vol. 12, no. 1–2, pp. 219–252, 2012.

[19] M. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia, "Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)," *Science of Computer Programming*, vol. 58, no. 1–2, pp. 115–140, October 2005.

[20] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, and G. Puebla, "The Ciao Approach to the Dynamic vs. Static Language Dilemma," in *Proceedings for the International Workshop on Scripts to Programs (STOP'11)*. New York, NY, USA: ACM, 2011.

[21] C. Flanagan, "Hybrid Type Checking," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. G. Morrisett and S. L. Peyton Jones, Eds. ACM, 2006, pp. 245–256. [Online]. Available: https://doi.org/10.1145/1111037.1111059

[22] A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris, "Safe & Efficient Gradual Typing for TypeScript," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, S. K. Rajamani and D. Walker, Eds. ACM, 2015, pp. 167–180. [Online]. Available: http://dl.acm.org/citation.cfm?id=2676726

[23] K. Muthukumar and M. Hermenegildo, "Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs," Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, Technical Report ACT-DC-153-90, April 1990. [Online]. Available: https://cliplab.org/papers/mcctr-fixpt.pdf

[24] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla, "On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs," in *Proc. of the 3rd Int'l. Workshop on Automated Debugging–AADEBUG'97*. Linköping, Sweden: U. of Linköping Press, May 1997, pp. 155–170. [Online]. Available: https://cliplab.org/papers/aadebug97-informal_bitmap.pdf

[25] E. Albert, G. Puebla, and M. Hermenegildo, "Abstraction-Carrying Code," in *Proc. of LPAR'04*, ser. LNAI, vol. 3452. Springer, 2005.

[26] M. Hermenegildo, G. Puebla, and F. Bueno, "Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging," in *The Logic Programming Paradigm: a 25–Year Perspective*. Springer-Verlag, 1999, pp. 161–192.

[27] G. Puebla, F. Bueno, and M. Hermenegildo, "An Assertion Language for Constraint Logic Programs," in *Analysis and Visualization Tools for Constraint Programming*, ser. LNCS. Springer-Verlag, 2000, no. 1870, pp. 23–61.

[28] N. Stulova, J. F. Morales, and M. Hermenegildo, "Some Trade-offs in Reducing the Overhead of Assertion Run-time Checks via Static Analysis," *Science of Computer Programming*, vol. 155, pp. 3–26, April 2018, selected and Extended papers from the 2016 International Symposium on Principles and Practice of Declarative Programming.

[29] M. Sanchez-Ordaz, I. Garcia-Contreras, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, and M. Hermenegildo, "VeriFly: On-the-fly Assertion Checking via Incrementality," *Theory and Practice of Logic Programming*, vol. 21, no. 6, pp. 768–784, September 2021. [Online]. Available: https://arxiv.org/abs/2106.07045

[30] D. Cabeza, M. Hermenegildo, and J. Lipton, "Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction," in *ASIAN'04*, ser. LNCS, no. 3321. Springer-Verlag, December 2004, pp. 93–108.

[31] B. C. Pierce, *Types and Programming Languages*. MIT Press, February 2002.

[32] S. Debray, P. Lopez-Garcia, and M. Hermenegildo, "Non-Failure Analysis for Logic Programs," in *1997 International Conference on Logic Programming*. Cambridge, MA: MIT Press, Cambridge, MA, June 1997, pp. 48–62.

[33] F. Bueno, P. Lopez-Garcia, and M. Hermenegildo, "Multivariant Non-Failure Analysis via Standard Abstract Interpretation," in *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, ser. LNCS, no. 2998. Heidelberg, Germany: Springer-Verlag, April 2004, pp. 100–116.

[34] L. Mauborgne and X. Rival, "Trace partitioning in abstract interpretation based static analyzers," in *European Symposium on Programming (ESOP'05)*, ser. Lecture Notes in Computer Science, M. Sagiv, Ed., vol. 3444. Springer-Verlag, 2005, pp. 5–20.

[35] X. Rival and L. Mauborgne, "The trace partitioning abstract domain," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 5, p. 26–es, Aug. 2007. [Online]. Available: https://doi.org/10.1145/1275497.1275501

[36] C. Vaucheret and F. Bueno, "More Precise yet Efficient Type Inference for Logic Programs," in *SAS'02*, ser. LNCS, no. 2477. Springer, 2002, pp. 102–116.