Running $HH(\mathcal{C})$ to deal with inductive inference

Javier Leach, Susana Nieva *
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
leach, nieva@sip.ucm.es

Abstract

The language $HH(\mathcal{C})$ is a combination of two orthogonal extensions of traditional Logic Programming. On one hand, the object logic is hereditary Harrop formulas (HH), instead of Horn clauses. On the other, it includes constraints similarly as it occurs in the Constraint Logic Programming paradigm. A suitable application of $HH(\mathcal{C})$ is meta-programming, including specification and validation of properties of different objects. In this context, recursive data-types and programs are very important. Some properties concerning these objects require mathematical induction to be proved. Hence $HH(\mathcal{C})$ should support inductive reasoning in some comfortable fashion. In this paper we show the advantages of this combination in the field of theorem proving using induction rules. The expressivity that the presence of universal quantification and implication in HH goals provides to the language is very useful to specify induction schemes. The incorporation of constraints has repercussions on the efficiency of inductive inference.

1 Introduction

Inductive inference is required for reasoning about objects containing recursion or iteration such as data structures, programs, or electronic circuits. In fact many properties of those objects can not be proved without the use of mathematical induction. For this reason, inductive inference has a vital importance in the definition of formal methods for synthesizing, verifying and transforming software and hardware.

It is possible to distinguish two different paradigms on automated induction: explicit and implicit induction. Implicit induction is also termed *inductionless induction* or proof by consistency, and it is based on Knuth-Bendix completion procedures [4]. Explicit induction does make use of explicit induction rules. Automated induction theorem proving, based on this paradigm, requires to solve special search problems such as: the choice of the induction rule, the choice of the induction variable, computing generalizations, or discovering intermediate lemmata [2, 16]. Important examples of explicit inductive theorem provers are the *Boyer/Moore Theorem Prover* [1], that incorporates the first inductive proof techniques such as recursion analysis, destructor elimination or generalization of subterms, and *Oyster/CL^{A}M* [3] whose contributions are, for instance, proof planning and rippling.

This paper concerns explicit induction using $HH(\mathcal{C})$. In [11] the scheme $HH(\mathcal{X})$ is introduced as a combination of the logic of Hereditary Harrop formulas (HH) and Constraint Logic Programming (CLP) [9]. $HH(\mathcal{X})$ may be particularized with any constraint system $\mathcal{C}$, providing an instance $HH(\mathcal{C})$. In [10] a higher-order version is defined, and in [8] several declarative semantics are presented. One suitable facet of $HH(\mathcal{C})$ is its use as meta-language. This is also the case of λ-Prolog, that is based on higher-order HH [13]. Meta-programming includes specification and val-

*The authors are partially supported by the Spanish project TIC2002-01167 'MELODIAS'.

idation of properties of recursive data-types and programs. Then, since mathematical induction plays a crucial role in the proof of theorems stating such properties, $HH(\mathcal{C})$ should support inductive inference.

HH extends Horn logic allowing disjunction, intuitionist implication and universal quantifier in goals. These constructions are essential for capturing module structure, hypothetical queries and data abstraction. Here we investigate their usefulness on inductive reasoning. Notice that induction rules infer universal statements, and the axioms specifying the induction step are universally quantified implications between the induction hypothesis and the induction conclusion. So both, universal quantifiers and implications, are appropriate tools to specify induction rules as clauses of $HH(\mathcal{C})$ programs.

On the other hand, it is well known that the main benefit of the CLP approach is the efficiency, because the satisfiability of constraints may be checked by efficient constraint solvers. Working with constraints on particular domains, different from Herbrand terms, certain parts of a proof by induction will consist of checking the satisfiability of particular constraints, so it will be done apart from the logic by the constraint solver. Then the search tree is pruned, and the proof is more concise and efficient. This work opens a new application field of CLP as an efficient tool to mechanize inductive inference.

The organization of the paper is the following. Section 2 overviews the principal aspects of the syntax of $HH(\mathcal{C})$. It contains the rules of a goal solver procedure, that can be understood as an operational semantics of the language. Some examples of the use of $HH(\mathcal{C})$ as a constraint logic programming language are also included. In Section 3 we focus on the use of this language to handle inductive inferences. Several common problems of automated theorem proving by induction are treated. The intended advantages of $HH(\mathcal{C})$ are explained using examples. The paper finishes with Section 4, where we expose the conclusions and our perspectives in this area.

2 The language $HH(\mathcal{C})$

$HH(\mathcal{C})$ can be regarded as a constraint logic programming language, not founded in Horn logic, as usual, but in the extended logic of hereditary Harrop formulas. In this section we summarize the principal aspects of this language, regarding the syntax and the goal computation.[1]

2.1 The syntax

As most CLP languages, $HH(\mathcal{C})$ is in fact a parameterized scheme that can be instantiated by particular constraint systems. Some requirements are imposed to such generic constraint system $\mathcal{C}$. Specifically the set of constraints includes $\top$ (true), $\bot$ (false), and the equations $t = t'$ for any Σ-terms t and t' (built from a signature Σ), and it is closed under $\wedge, \Rightarrow, \exists, \forall$ and the application of substitutions of terms for variables. In addition a binary entailment relation, $\vdash_{\mathcal{C}}$, between sets of constraints, Γ, and constraints, C, is assumed. In fact $\vdash_{\mathcal{C}}$ is a deduction system for the constraint system $\mathcal{C}$, such that the inference rules associated to $\wedge, \Rightarrow, \exists, \forall$ and the equality relation valid in the intuitionist fragment of first-order logic are also valid in it. Given a constraint system $\mathcal{C}$, Γ is said to be *$\mathcal{C}$-satisfiable* if $\emptyset \vdash_{\mathcal{C}} \overline{\exists}(\bigwedge \Gamma)$, where $\overline{\exists}$ denotes the existential closure, and $\bigwedge \Gamma$ the conjunction of the constraints in Γ.

Example 1 Let $Ax_{\mathcal{CFT}}$ be the Smolka and Treinen's axiomatization of the domain of feature trees [14]. An example of constraint system, called $\mathcal{CFT}$, can be defined considering the whole set of first-order formulas as constraints, and $\Gamma \vdash_{\mathcal{CFT}} C$ if and only if $\Gamma \cup Ax_{\mathcal{CFT}} \vdash C$, where $\vdash$ is the entailment relation of classical first-order logic with equality.

Another example is the constraint system $\mathcal{R}$ that can be defined analogously, but using $Ax_{\mathcal{R}}$, the Tarski's axiomatization of the closed field of real numbers [15]. It is also possible to restrict the constraint domain to natural numbers. $\mathcal{N}$ can be defined using the Peano axiomatization of the ordered set of natural

[1] See [11] for more detailed explanations.

numbers and the same entailment relation as the examples above.

Hybrid constraint systems can be useful. See [7], where the system $\mathcal{RH}$, that combines the field of real numbers with finite trees, is defined. □

Many of the constraint systems employed in practice —$\mathcal{CFT}$, $\mathcal{R}$ or $\mathcal{N}$ e.g.— include negation $\neg$. If $\mathcal{C}$ owns negation, the semantics of $\neg C$ is determined by the entailment relation $\vdash_{\mathcal{C}}$, as it is the case of any other constraint. Usually, the notation, $\neg t = t'$ is simplified by $t \neq t'$. Naturally it must not be interpreted as an inequality in the Herbrand universe, but as the negation of an equation in the domain of $\mathcal{C}$.

Let us see how constraints can be found embedded in clauses and goals. *Clauses* D and *goals* G are defined by the following mutually recursive rules.

$$G ::= A \mid C \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \Rightarrow G \mid C \Rightarrow G \mid \exists x G \mid \forall x G,$$
$$D ::= A \mid G \Rightarrow D \mid D_1 \wedge D_2 \mid \forall x D,$$

where A is an atom and C is a constraint.

A *program*, denoted by Δ, is a finite set of clauses. In order to simplify the presentation of this paper we will consider, without loss of generality, that every clause of a program is of the form $\forall x_1 \ldots \forall x_n (G \Rightarrow A)$.

The following examples illustrate the use of the language and its expressive power. A Prolog-like notation, enriched with constraints and the logic connectives $\forall$, $\exists$ and $\Rightarrow$, will be used to write programs. As usual, capital letters represent implicitly universally quantified variables. Small letters are used for explicitly quantified variables.

Example 2 This is a simple $HH(\mathcal{R})$ program to specify the divisibility relation:.

```
divisor(Y, X):- modulo(X,Y,0).
modulo(X,Y,Z):- ∃ w (X = w * Y + Z),
                0 <= Z, Z < Y.
```

Realize that the variable w, that represents the quotient, has been explicitly existentially quantified. □

Example 3 Consider the program Δ below of the instance $HH(\mathcal{R})$:

```
triangle(A,B,C):- A > 0, B > 0, C > 0,
      A < C + B, B < A + C, C < A + B.
isosceles(A,B,C):- triangle(A,B,C),
                   ((A = B, A ≠ C) ;
                    (A = C, A ≠ B) ;
                    (B = C, A ≠ B)).
```

The predicate `triangle(A,B,C)` becomes true when it is possible to build a triangle with sides of lengths `A`, `B` and `C`.

Suppose that, from Δ, we want to know which conditions on a variable y guarantee that, for any $x > 1$, it is possible to build an isosceles triangle with sides $\langle x, x, y \rangle$. A goal which captures that query may be:

$G \equiv (\forall x (x > 1 \Rightarrow \mathit{isosceles}(x, x, y)))$.

The intended meaning of the program Δ should secure that $0 < y \wedge y \leq 2$ is a correct answer constraint for G from Δ. □

Example 4 The following is a reversible naive program to compute Fibonacci numbers.

```
fib(0,1).
fib(1,1).
fib(N,X1+X2):- N >= 2, fib(N-1,X1),
                       fib(N-2,X2).
```

In [11] appears the following more efficient version of the computation of Fibonacci numbers, that is also reversible, but for which none Fibonacci number must be recalculated, and goals of the form *fibonacci*(n, x), n given, run in linear time.

```
fibonacci(N,X):- memfib(0,1)
       ⇒(memfib(1,1)⇒getfib(N,X,1)).
getfib(N,X,M):- 0 <= N, N <= M,
                       memfib(N,X).
getfib(N,X,M):- N > M,
       memfib(M-1, F1), memfib(M, F2),
  (memfib(M+1,F1+F2)⇒getfib(N,X,M+1)).
```

In accordance with the semantics presented in [8], both programs are equivalent. In Section 3, we refer to the naive (and exponential) version for a better understanding of the induction examples. □

2.2 Goal resolution

When $HH(\mathcal{C})$ was defined [11], its meaning was explained by means of a proof system, $\vdash_{\mathcal{UC}}$, that merges inference rules from an intuitionist sequent calculus with the entailment relation $\vdash_{\mathcal{C}}$. That proof system guarantees uniform (goal oriented) proofs, therefore $HH(\mathcal{C})$ can be considered as an *abstract logic programming language* in the sense of [12]. A goal solving procedure —sound and complete w.r.t. $\mathcal{UC}$— was also introduced. Such procedure can be seen as an operational semantics of $HH(\mathcal{C})$.

Now we briefly present that goal solving procedure, in order to explain the behavior of the constraint logic programming language $HH(\mathcal{C})$.

A computation of such procedure consists on a transformation of states. Each transformation step corresponds to the application of a rule which is guided by the structure of the goal to be solved. When no goals remain to be solved, the procedure ends in a final state that contains an answer constraint.

A *state* $\mathcal{S}$, has the form $\Pi[S \Box \mathcal{G}]$, where $\mathcal{G}$ is a multiset of triples $\langle \Delta, C, G \rangle$ (Δ *local program*, C *local constraint* and G *local goal*). Π is a quantifier prefix $Q_1 x_1 \ldots Q_k x_k$ where Q_i, $1 \leq i \leq k$, is $\forall$ or $\exists$. S is a *global constraint* formula.

The *rules for transformation of states* are defined as follows:

i) Conjunction.

$\Pi[S \Box \mathcal{G}, \langle \Delta, C, G_1 \wedge G_2 \rangle] \Vdash$
$\Pi[S \Box \mathcal{G}, \langle \Delta, C, G_1 \rangle, \langle \Delta, C, G_2 \rangle]$.

ii) Disjunction.

$\Pi[S \Box \mathcal{G}, \langle \Delta, C, G_1 \vee G_2 \rangle] \Vdash$
$\Pi[S \Box \mathcal{G}, \langle \Delta, C, G_i \rangle]$, for $i = 1$ or 2
(don't know choice).

iii) Implication with local clause.

$\Pi[S \Box \mathcal{G}, \langle \Delta, C, D \Rightarrow G \rangle] \Vdash$
$\Pi[S \Box \mathcal{G}, \langle \Delta \cup \{D\}, C, G \rangle]$.

iv) Implication with local constraint.

$\Pi[S \Box \mathcal{G}, \langle \Delta, C, C' \Rightarrow G \rangle] \Vdash$
$\Pi[S \Box \mathcal{G}, \langle \Delta, C \wedge C', G \rangle]$.

v) Existential quantification.

$\Pi[S \Box \mathcal{G}, \langle \Delta, C, \exists x G \rangle] \Vdash$
$\Pi \exists w[S \Box \mathcal{G}, \langle \Delta, C, G[w/x] \rangle]$,
(w is a new variable).

vi) Universal quantification.

$\Pi[S \Box \mathcal{G}, \langle \Delta, C, \forall x G \rangle] \Vdash$
$\Pi \forall w[S \Box \mathcal{G}, \langle \Delta, C, G[w/x] \rangle]$,
(w is a new variable).

vii) Constraint.

$\Pi[S \Box \mathcal{G}, \langle \Delta, C, C' \rangle] \Vdash$
$\Pi[S \wedge (C \Rightarrow C') \Box \mathcal{G}]$.
If $\Pi(S \wedge (C \Rightarrow C'))$ is $\mathcal{C}$-satisfiable.

viii) Clause of the program.

$\Pi[S \Box \mathcal{G}, \langle \Delta, C, A \rangle] \Vdash$
$\Pi[S \Box \mathcal{G}, \langle \Delta, C, \exists x_1 \ldots \exists x_n ((\bigwedge_{1 \leq i \leq m} t_i = t'_i) \wedge G) \rangle]$.
Provided that $\forall x_1 \ldots \forall x_n (G \Rightarrow A')$ is a variant of some clause in Δ (don't know choice), $x_1, \ldots, x_n$ new variables, and $A \equiv P(t_1, \ldots, t_m)$, $A' \equiv P(t'_1, \ldots, t'_m)$.

A *resolution of a goal G from a program Δ* is a finite sequence of states $\mathcal{S}_0, \ldots, \mathcal{S}_n$, such that: $\mathcal{S}_0 \equiv [\top \Box \langle \Delta, \top, G \rangle]$; $\mathcal{S}_{i-1} \Vdash \mathcal{S}_i$, $1 \leq i \leq n$, by means of any of the transformation rules; $\mathcal{S}_n \equiv \Pi_n [S_n \Box \emptyset]$. The constraint $\Pi_n S_n$ is called the *answer constraint* of this resolution. We say that G *fails* from Δ if there is not any resolution of G from Δ. By the completeness theorem, this implies that there is not a proof of G from Δ in $\vdash_{\mathcal{UC}}$.

Example 5 Let Δ, G and C given at Example 3. The following is a rough description of the computation of a resolution of G from Δ with answer constraint C:
• Use rule *vi)* to eliminate the quantifier $\forall x$.
• Then apply *iv)* which introduces $x > 1$ as a local constraint.
• Now *viii)* with the clause for the predicate `isosceles`.
• Next apply the rule *v)* three times to eliminate the existential quantifiers introduced by the previous rule.
• Then *i)* to eliminate the conjunction of the local goal. That allows to choose *triangle*(x, x, y) as the next subgoal.

• Apply *viii)* with the clause for `triangle` and *v)* three times.
• The partially calculated answer is now equivalent to the constraint $\forall x \exists w(x > 1 \Rightarrow (y = w \wedge x > 0 \wedge w > 0 \wedge x < w + x \wedge w < x + x))$, applying *vii)* its satisfiability is checked.
• The rule *ii)* can choose $x = x \wedge x \neq w$ as the goal to be solved.
• Hence *vii)* allows to finish the proof by proving the satisfiability of $\forall x \exists w(x > 1 \Rightarrow (y = w \wedge x > 0 \wedge w > 0 \wedge x < w + x \wedge w < x + x \wedge x = x \wedge x \neq w))$. This answer constraint is in fact equivalent, in the system $\mathcal{R}$, to the intended one: $0 < y \wedge y \leq 2$. □

3 Proving theorems by induction in $HH(\mathcal{C})$

The specification, verification and proof of properties of programs is one of the applications of $HH(\mathcal{C})$. As in the case of λ-Prolog, it can even be used as the base language of an interactive theorem prover. Then, by the importance of the induction in that frame, it must be considered how $HH(\mathcal{C})$ can deal with inductive inference.

Besides the characteristic expressivity of HH, that is also the logic basis of λ-Prolog, $HH(\mathcal{C})$ advantages λ-Prolog by the presence of a constraint system. It can be specially beneficial in the context of inductive inference, because, as it will be seen, it permits to reduce the difficulty of proofs, transferring to the constraint solver the responsibility of some parts of them. This improves the clarity of the inference and the efficiency of the theorem prover.

3.1 Implementing induction proofs

We will use examples to show the benefits of the presence of quantifiers and implications in the goals, and of the incorporation of constraints, for theorem proving using induction rules.

Example 6 Let us consider the instance $HH(\mathcal{N})$, where the domain of the constraints is the ordered set of natural numbers with the arithmetic operations + and −.

In this context, the following program Δ specifies a *necessary and sufficient* condition for a natural number to be even. Here programs are operationally interpreted, then the disyunction in the second clause is needed[2].

```
even(0).
even(X):- even(X + 2);
          (X >= 2, even(X - 2)).
```

From this program we want to prove that $\forall x P(x)$ holds, where

$P(x) \equiv \forall y(even(x) \wedge even(y) \Rightarrow even(x + y))$.

Due to the form of the recursive definition of `even`, the better option in order to prove this theorem is to use the following induction rule.

$$\frac{P(0), P(1), \forall x(P(x) \Rightarrow P(x + 2))}{\forall x P(x)}$$

The great expressive power of our language allows to translate the induction axioms of that rule into the form of a goal. Then, the inductive inference will consist on a resolution, from Δ, of the conjunction of the three following subgoals.

$G_0 \equiv \forall y(even(0) \wedge even(y) \Rightarrow even(0 + y))$.
$G_1 \equiv \forall y(even(1) \wedge even(y) \Rightarrow even(1 + y))$.
$G_2 \equiv \forall x(\forall y$
$(even(x) \wedge even(y) \Rightarrow even(x + y)) \Rightarrow$
$\forall y(even(x+2) \wedge even(y) \Rightarrow even((x+2)+y)))$.

Let us simulate how the goal solver procedure will work with G_2. We represent $G_2 \equiv \forall x G$. Where $G \equiv (D \Rightarrow \forall y G')$, $D \equiv \forall y(even(x) \wedge even(y) \Rightarrow even(x+y))$, and $G' \equiv (even(x + 2) \wedge even(y)) \Rightarrow even((x + 2) + y)$. This resolution can be computed with the prototipe presented in [6]. The inference is shown in Figure 1 where the following simplifications are used:
$\Delta' \equiv \Delta \cup \{D, even(x + 2), even(y)\}$.
$\Pi' \equiv \forall x \forall y \exists x_1 \exists x_2$.
$S^i \equiv x + x_1 = (x + 2) + y$.
$S^{ii} \equiv S^i \wedge x_2 = x$.
$S^{iii} \equiv S^{ii} \wedge x_2 + 2 = x + 2$.
$S^{iv} \equiv S^{iii} \wedge x_3 = x_1$.
$S^v \equiv S^{iv} \wedge x_3 \geq 2$.
$S^{vi} \equiv S^v \wedge x_3 - 2 = y$.

[2] See rows nine and twelve of Figure 1.

ΠS	**G**	***rule***
$\forall x \top$	$\langle \Delta, \top, G \rangle$	*vi)*
$\forall x \top$	$\langle \Delta \cup \{D\}, \top, \forall y G' \rangle$	*iii)*
$\forall x \forall y \top$	$\langle \Delta \cup \{D\}, \top, G' \rangle$	*vi)*
$\forall x \forall y \top$	$\langle \Delta', \top, even((x+2)+y) \rangle$	*iii)*
$\forall x \forall y \exists x_1 \top$	$\langle \Delta', \top, x + x_1 = (x+2) + y \wedge even(x) \wedge even(x_1) \rangle$	*viii), v)*
$\forall x \forall y \exists x_1 S^i$	$\langle \Delta', \top, even(x) \wedge even(x_1) \rangle$	*i), vii)*
$\forall x \forall y \exists x_1 S^i$	$\langle \Delta', \top, even(x) \rangle, \mathcal{G}' \equiv \langle \Delta', \top, even(x_1) \rangle$	*i)*
$\Pi' S^{ii}$	$\langle \Delta', \top, even(x_2+2) \vee (x_2 \geq 2 \wedge even(x_2-2)) \rangle, \mathcal{G}'$	*viii), v), i), vii)*
$\Pi' S^{ii}$	$\langle \Delta', \top, even(x_2+2) \rangle, \mathcal{G}'$	*ii)*
$\Pi' S^{iii}$	$\mathcal{G}' \equiv \langle \Delta', \top, even(x_1) \rangle$	*viii), vii)*
$\Pi' \exists x_3 S^{iv}$	$\langle \Delta', \top, even(x_3+2) \vee (x_3 \geq 2 \wedge even(x_3-2)) \rangle$	*viii), v), i), vii)*
$\Pi' \exists x_3 S^{v}$	$\langle \Delta', \top, even(x_3-2) \rangle$	*ii), i), vii)*
$\Pi' \exists x_3 S^{vi}$	$\emptyset$	*viii), vii)*

Figure 1: Computation of G_2.

Notice that D corresponds to the induction hypothesis. The rule *iii)*, in the second row, adds it to the program as a local clause. Then it will be used during the reasoning, particularly in the proof of the subgoal $even((x+2)+y)$ (fifth row). The proof is concluded with success, because the computed answer constraint, $\Pi' \exists x_3 S^{vi}$, is equivalent to $\top$.

It is also remarkable the fact that the constraint solver is who manipulates the operation $+$, hence there is not any clause specifying it, and no additional steps using these hypothetical clauses will be needed in the goal resolution. In such way, the search tree results significantly pruned. □

One of the main problems in automatizing inductive proofs is the choice of the variable on which induction is done. An additional benefit of transferring some tasks to the constraint system is that this problem can be avoided, in the cases specified next.

Example 7 Consider Δ and G of Example 6. If the induction rule is applied to the second variable y instead of x. The new goals will be:

$G'_0 \equiv \forall x(even(x) \wedge even(0) \Rightarrow even(x+0))$.
$G'_1 \equiv \forall x(even(x) \wedge even(1) \Rightarrow even(x+1))$.
$G'_2 \equiv \forall y(\forall x$
$(even(x) \wedge even(y) \Rightarrow even(x+y)) \Rightarrow$
$\forall y(even(x) \wedge even(y+2) \Rightarrow even(x+(y+2))))$.

Many of the inductive proof techniques (e.g. rewriting) make use, in some explicit way during the proof, of the definition of the recursive function involved in the property to be proved. If, for instance, a rewriting technique is considered, the equations $0 + y = y$ and $s(x) + y = s(x + y)$, which recursively specify $+$, will be expressed as rewrite rules to be used on inductive inferences. Applying those rules, the goals G_0, G_1, G_2 of the previoud example will succeed, but G'_0, G'_1, G'_2 will not.

As mentioned in Example 6, $+$ is not a predicate defined by program clauses, but an operation of $\mathcal{N}$, so the goals G'_0, G'_1, G'_2 can be proved in a similar way as G_0, G_1, G_2, without using an explicit recursive definition of the operation $+$. For instance, in the execution of G_2 in Example 6, the $\mathcal{N}$-satisfiability of $\forall x \forall y \exists x_1 (x + x_1 = (x+2) + y)$ has been proved, while, solving G'_2, the corresponding constraint to be checked will be $\forall y \forall x \exists x_1 (x_1 + y = x + (y+2))$, which is also true. Hence no decision about the induction variable have to be done. □

The reasoning of the previous example is likewise valid if a property related to any operation or predicate of the constraint system, different of $+$, has to be proved. Therefore, the benefit of $HH(\mathcal{C})$ to avoid the chioce of the induction variable can be generalized to the cases in which the candidate induction vari-

ables appear as argument of any operation or predicate of $\mathcal{C}$.

The recursive datatype list has a wide application in declarative programming. The next example illustrates how to deal with the recursion/induction duality for this datatype.

Example 8 We will use here a hybrid constraint system that combines the type of the lists with that of the real numbers, similarly as it was presented in [7]. For that mixed system we will use type annotations of the form $t : \tau$ for terms[3]. We suppose that there is available in the constraint system an operation *in*, such that $x : real\ in\ l : list$ determines if a real number x appears in a list l. We want to prove the following property: $\forall l : list\ P(l)$, where

$P(l) \equiv \forall x : real(\neg(x\ in\ l) \Rightarrow remove(x, l, l))$.

$\neg(x\ in\ l)$ is a constraint, and the predicate `remove(X,L,L')`, defined by the clauses below, affirms that the list `L'` coincides with the list `L`, except for the element `X`, that is removed from `L'`.

```
remove(X,[],[]).
remove(X,[X | L],L'):- remove(X,L,L').
remove(X,[Y | L],[Y | L']):- X ≠ Y,
                             remove(X,L,L').
```

For example, the goal $remove(4, [4, 2+2], [\,])$, is successful, but $\exists x(remove(x, [4, 2+2], [\,]) \wedge x < 4)$ is not.

To prove the property $\forall l : list\ P(l)$ above, structural induction for lists is applied:

$$\frac{P([\,]), \forall y : real\ \forall t : list(P(t) \Rightarrow P([y|t]))}{\forall l : list\ P(l)}$$

In this example, the conjecture to be proved corresponds to the conjunction of the goals:

$G_1 \equiv \forall x : real(\neg(x\ in\ [\,]) \Rightarrow remove(x, [\,], [\,]))$,
$G_2 \equiv \forall y : real\ \forall t : list((\forall x : real(\neg(x\ in\ t) \Rightarrow remove(x, t, t))) \Rightarrow (\forall x : real(\neg(x\ in\ [y|t]) \Rightarrow remove(x, [y|t], [y|t]))))$.

The induction hypothesis (inside G_1):
$\forall x : real(\neg(x\ in\ t) \Rightarrow remove(x, t, t))$,
would be locally incorporated to the program, as in the previous example. So, during the process of proving the induction conclusion, the local clause corresponding to the induction hypothesis would be utilized along with the remainder of the clauses that define the predicate `remove`. The constraint solver will decide the satisfiability of the constraints related to the predicate *in* as, for example, $\forall y \forall t \forall x(\neg(x\ in\ [y|t]) \Rightarrow x \neq y)$. □

Another drawback in automatizing induction are the proofs of existential theorems. Those theorems appear frequently when proving properties of programs and in synthesis problems. In the latter, the form of inductions may determine the form of the recursive program to be synthesized. But most of the techniques to automatize inductive inference are designed to deal only with universal variables and must be reconsidered to manage existential quantifiers.

In our language explicit existential quantifiers can appear in goals (and constraints), but not in clauses. Some times formulas containing an existential quantified clause can be rewritten to an equivalent one, avoiding this quantifier, by means of intuitionist logical rules. For instance, $\exists x D \Rightarrow G$ can be transformed to the legal goal $\forall x(D \Rightarrow G)$.

An alternate solution to represent existential clauses is the use of skolem functions as we show in the next example.

Example 9 Consider the naive version of the Fibonacci program of Example 4. Any proof of the goal $G \equiv \forall x \exists y fib(x, y)$ from the set of clauses defining the predicate `fib` must fail if the considered constraint system is $\mathcal{R}$. Still considering the constraint system $\mathcal{N}$, instead of $\mathcal{R}$, no proof of G from Δ can be found. But the property $\forall x P(x)$ specified by G can be proved using induction. For this case, the following rule will be needed:

$$\frac{P(0), P(1), \forall x \geq 2(\forall k < x(P(k) \Rightarrow P(x)))}{\forall x P(x)}$$

The base cases: $\exists y fib(0, y)$, $\exists y fib(1, y)$, are easy to prove. But the inductive step yields to a formula that is not a goal, since the inductive hypothesis, $\forall k(k \geq 0 \wedge k < x \Rightarrow \exists y fib(k, y))$,

[3] In the example we will suppress these annotations when the type can be deduced from the context.

is not a clause because of the existential quantifier.

If a skolem function symbol f is used, the inductive step can be reformulated as the goal $\forall x(x \geq 2 \Rightarrow (D \Rightarrow \exists y fib(n, y)))$, where $D \equiv \forall k(k \geq 0 \wedge k < x \Rightarrow fib(k, f(k))$ is already a clause. □

When skolem symbols are emploied, the signature and the axiomatization of $\mathcal{C}$ must be conveniently extended.

3.2 Implementing induction rules

As we have previously mentioned, the implementation of theorem provers is one of the many applications of λ-Prolog [5]. Since the higher-order version of $HH(\mathcal{C})$, that we have developed in [10], includes *ho-HH*, in which λ-Prolog is based, it is also possible to implement the rules of a logical calculus using clauses of $HH(\mathcal{C})$ (even in many cases it is not necessary to resort to higher-order).

In previous examples we have used the induction rule for the proof of some properties on well founded sets. In each case we have just used appropriate formulations of the induction rule, and we have shown how the induction axioms, particularized to the properties to be proven, can be specified as goals of diverse instances of $HH(\mathcal{C})$ that are executed by the goal solving procedure.

We concentrate now in the implementation of those induction rules. Certain mechanism, responsible for producing automatically the goals corresponding to the axioms of a suitable induction rule, will be designed. With such aim, the demonstration by induction of a property P is identified with a predicate `induction(P)`, defined by a clause whose body is formed by the goals `inducbase(P)`, `inducstep(P)`, corresponding to the induction axioms.

In order to properly define this clause, the following consideration has to be taken into account: the different induction rules are in fact scheme-rules that are particularized when they are instantiated with specific properties. Therefore, second-order predicates are needed to specify the different induction schemes. Hence the higher-order version of $HH(\mathcal{C})$ [10], that combines *ho-HH* formulas and first-order constraints, will be used.

Peano simple induction scheme can be specified in $hoHH(\mathcal{N})$ by:

```
pinduc(P):- pinducbase(P),pinducstep(P).
pinducbase(P):- P(0).
pinducstep(P):- ∀ x (P(x) ⇒ P(x + 1)).
```

The scheme used in Example 6 can be translated to:

```
induc2(P):- inducbase2(P),inducstep2(P).
inducbase2(P):- P(0), P(1).
inducstep2(P):- ∀ x (P(x) ⇒ P(x + 2)).
```

The induction proof of the property $\forall x \forall y(even(x) \wedge even(y) \Rightarrow even(x + y))$ corresponds to the resolution of the goal:

$induc2(\lambda x.(\forall y\ even(x) \wedge even(y) \Rightarrow even(x+y))$.

For the Example 9, the adequate induction scheme can be implemented by:

```
stinduct(P,L):- stinducbase(P,L),
                stinducstep(P,L).
stinducbase(P,[]).
stinducbase(P,[X | L]):- P(X),
                stinducbase(P,L).
stinducstep(P,L):- greatest(L,N),
                  ∀ n (n >= N ⇒
 (∀ k(k >= 0,k < n ⇒ P(k)) ⇒ P(n))).
```

L represents the list of the base cases, and `greatest(L,N)` calculates in the second argument the greatest element of the list in the first argument.

The inductive inference of the property $\forall x \exists y fib(x, y)$ is equivalent to the resolution of the goal:

$stinduc(\lambda x.(\exists y fib(x, y)), [0, 1])$.

The scheme for structural induction on lists, used in Example 8 corresponds to the clauses:

```
linduc(P):- linducbase(P),linducstep(P).
linducbase(P):- P([]).
linducstep(P):- ∀ x ∀ l
                (P(l) ⇒ P([x | l])).
```

To prove by induction the property of Example 8, it is necessary to compute the goal:

$$linduc(\lambda l : list.\ \forall x : real \\ (\neg(x\ in\ l) \Rightarrow remove(x, l, l))).$$

If the constraint system $\mathcal{C}$ is a well-founded order with respect to some order $\prec$, the Noetherian induction schema can be specified by:

```
ntinduction(P):- ∀ x ( ∀ y
          (y ≺ x ⇒ P(y)) ⇒ P(x)).
```

A constraint solver would be available for $\mathcal{C}$, being responsible for checking $\mathcal{C}$-satisfiability of constraints that may contain the order symbol $\prec$ with the intended meaning.

Each of the previous induction schemes are valid clauses only for the corresponding instance of $HH(\mathcal{X})$. And, indeed, it is necessary to involve a human user to determine which induction rule to use, but this is a common process in interactive theorem provers.

Now that several induction rules have been implemented, we remark another benefit of our system, referred constructor versus destructor style.

Inductive theorem proving systems usually work in constructor style induction (e.g. $\forall x(P(x) \Rightarrow P(x+1))$ in the natural numbers). When the conjecture contains terms with destructor functions (e.g. $x-1$ in the natural numbers), a destructor elimination technique takes place before applying a constructor induction rule. This process can be obviated in our system if the constructor and destructor operators are constraint operations.

let $\forall x(x > 0 \Rightarrow P(x-1))$ be a conjecture, if $induction(\lambda x.(x > 0 \Rightarrow P(x-1)))$ is executed, the induction step corresponds to the proof of $\forall x(P(x-1) \Rightarrow P((x-1)+1))$[4].

The induction hypothesis $P(x-1)$ is added as local clause, and the treatment of the induction conclusion $P((x-1)+1))$ will be similar to that of $P(x)$, since both arguments are equal in the constraint system. Techniques as rewrite or fertilization make use of matching. For instance, fertilization can be applied when the induction conclusion contains an instance of the induction hypothesis. Then, in this case, $P(x-1)$ is not a useful induction hypothesis. This is the reason to employ destructor elimination. Because in $HH(\mathcal{C})$ not matching but $\mathcal{C}$-satisfiability is the mechanism that makes a clause able to be used, $P(x-1)$ can be directly used as induction hypothesis, avoiding the bond.

[4]For simplicity the conditions x>0 have been abbreviated.

4 Conclusions

Our work shows that $HH(\mathcal{C})$ is a suitable tool to implement inductive theorem proving system. We have taken advantage of the flexibility of the $HH(\mathcal{C})$ scheme both, from the point of view of the richness of the base logic HH, and from the possibility to use different constraint systems, choosing the domain of application according to the type of problem to be solved.

Due to the embedding of implications and universal quantifiers inside goals (and so inside programs), using the $HH(\mathcal{C})$ framework makes it possible to specify the induction axioms directly as goals to be solved. Then an inductive inference is identified with the resolution of a goal by the goal solving procedure. In addition, the induction schemes can be elegantly implemented as clauses using second-order predicates. On the other hand, relegating certain tasks to the constraint solver, the search space is reduced improving the efficiency and clarity of the proof. We have shown the particular benefits brought by the incorporation of a constraint system, to the processes that habitually arise in the execution of induction inferences.

Therefore, this paper explores a new efficient tool to automatize inductive inference. Making use of CLP, it is possible to implement systems that improve the usual inductive proof mechanisms in different aspects, as the proof of existential conjectures, the choice of the induction variable or the technique of destructive elimination. When the base logic

is *HH*, the improvements are stretched to the specification of the induction schemes.

We have propose an implementation of the mathematical induction as a deduction mechanism for interactive theorem provers based on $HH(\mathcal{C})$. Our intention is to undertake the study of heuristics that permit a broader automation of the search control problems related to: the choice of the induction rule, the introduction of lemmas, and the generalization of the conjecture that is usually required on inductive inferences. We foresee to be able to deal with such search control problems, also making use of the abstraction and the expressive power of the logic, as well as of the possibility of liberating the goal solving procedure from some concrete tasks that can be executed by the constrain solver.

References

[1] R. S. Boyer and J. S. Moore. A computational logic handbook. In *Perspectives in Computing*. Academic Press, 1988.

[2] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (1)*, pages 845–911. Elsevier Science, 2001.

[3] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The oyster-clam system. In M. Stickel, editor, *CADE-10*, LNAI 449, pages 647–648. Springer, 1990.

[4] H. Comon. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (1)*, pages 913–962. Elsevier Science, 2001.

[5] A. Felty. Tutorial on λ-prolog and its application to theorem proving. Technical report, Bell Labs. Lucent Technologies, sep 1997.

[6] M. García-Díaz. Algunas extensiones de la Programación Lógica. una implementación para un caso concreto. Technical report, Dpto. Sistemas Informáticos y Programación (UCM), 2001.

[7] M. García-Díaz and S. Nieva. Solving constraints for an instance of an extended CLP language over a domain based on real numbers and Herbrand terms. *Journal of Functional and Logic Programming*, 2003(2), September 2003.

[8] M. García-Díaz and S. Nieva. Providing declarative semantics for HH extended constraint logic programs. In *PPDP'04*, pages 55–66. ACM Press, 2004.

[9] J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[10] J. Leach and S. Nieva. A higher-order logic programming language with constraints. In H. Kuchen and K. Ueda, editors, *FLOPS'01*, LNCS 2024, pages 108–122. Springer, 2001.

[11] J. Leach, S. Nieva, and M. Rodríguez-Artalejo. Constraint logic programming with hereditary Harrop formulas. *Theory and Practice of Logic Programming*, 1(4):409–445, 2001.

[12] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[13] G. Nadathur. The metalanguage λ-prolog and its implementation. In H. Kuchen and K. Ueda, editors, *FLOPS'01*, LNCS 2024, pages 1–20. Springer, 2001.

[14] G. Smolka and R. Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, 1994.

[15] A. Tarski. *A decision method for elementary algebra and geometry*. University of California Press, 1951.

[16] C. Walther. Mathematical induction. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming (2)*, pages 127–228. Clarendon Press, 1994.