Electronic Notes in Theoretical Computer Science 40 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume40.html 18 pages

Assertion based Inductive Verification Methods for Logic Programs

Marco Comini [a], Roberta Gori [b] and Giorgio Levi [b]

[a] *Dipartimento di Matematica e Informatica, Università di Udine, Via delle Scienze 206, 33100 Udine, Italy*

[b] *Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy*

Abstract

This paper is an overview of our results on the application of abstract interpretation concepts to the derivation of a verification method for logic programs. These include the systematic design of semantics modeling various proof methods and the characterization of assertions as abstract domains. We first apply the verification framework defined in [5] to derive inductive sufficient conditions for partial correctness. Then the domain of assertions is formalized as an abstract domain. We can therefore derive an assertion based verification method. We finally show two methods based on different assertion languages: a decidable assertion language and Horn clause logic used as assertion language.

Key words: Inductive verification, Abstract interpretation, Assertion language, Transformation of logic programs.

1 Introduction

The aim of verification is to define conditions which allow us to formally prove that a program behaves as expected, i.e., that the program is correct w.r.t. a given specification, a description of the program's expected behavior.

There are essentially two ways to represent the *expected* behavior of a program. We can represent the behavior of a program *extensionally*, i.e., by listing all the results, or *intensionally*, i.e., by means of a property which must be satisfied by the computation results.

In order to express properties of programs, assertions — formulas in a suitable assertion language — are commonly used. A formula in an assertion language represents all the results which satisfy the property expressed by the formula. This allows us to express sets of results by means of a single formula.

By applying the verification framework defined in [5], we derive inductive conditions on abstract domains which are sufficient for proving partial correct-

ness. The key idea is that formulas of an assertion language can be viewed as abstract domains. Then a new verification method based on assertions can be derived. Of course the sufficient conditions which we obtain are parametric w.r.t. the specific assertion language. Therefore, depending on the choice of the assertion language, we define different verification methods able to prove different properties.

It is worth noting that, if the entailment relation in the assertion language is decidable, then the partial correctness conditions that we derive are effectively provable.

In this paper we first present a verification method based on a simple assertion language, which is able to express properties of terms, including types and other properties relevant to static analysis. The language is decidable. However, the properties which can be specified are given once for all.

As a further step, we propose a verification method based on an assertion language where properties can be defined by the user through a logic program (user programs in the following). This yields a very powerful and expressive assertion language.

However, in general there exists no effective method to decide whether the resulting conditions are verified. We will show that such conditions can often be proved by using well-known program transformation techniques. Program transformation rules (such as fold and unfold) allow one to syntactically transform formulas while preserving their semantics. In our case, we prove the sufficient correctness conditions by means of transformations on the user program.

It is worth noting that logic programs have already been used as specifications in the literature [13,10,3,21,20]. However our approach is different. In [3,21,20], in fact, assertions — associated to program points — are evaluated at run time by using the user programs and the run time values. Hence the logic implementation of the specification language is used to check by evaluation that each result of the actual program verifies the specification. In our approach, the same user program is used to syntactically prove sufficient conditions for partial correctness.

The reader is assumed to be familiar with the terminology and the basic results in the semantics of logic programs [1,14] and with the theory of abstract interpretation as presented in [7,8].

2 Inductive Abstract Verification

In order to prove that a program behaves as expected we can use a semantic approach based on abstract interpretation techniques. This approach allows us to derive in a uniform way sufficient conditions for proving partial correctness w.r.t. different properties. The ideas behind this approach are the following:

- The concrete semantics $[\![P]\!]$ of a program P is defined as the least fixpoint

of a semantic evaluation function $\mathcal{T}_P$ on the concrete domain $(\mathbb{C}, \sqsubseteq)$.

- As in standard abstract interpretation based program analysis, the class of properties we want to verify is formalized as an abstract domain $(\mathbb{A}, \leq)$, related to $(\mathbb{C}, \sqsubseteq)$ by the usual Galois connection $\alpha : \mathbb{C} \to \mathbb{A}$ and $\gamma : \mathbb{A} \to \mathbb{C}$ (abstraction and concretization functions). The corresponding *abstract semantic evaluation function* $\mathcal{T}_P^\alpha$ is systematically derived from $\mathcal{T}_P$, α and γ. The resulting abstract semantics is a correct approximation of the concrete semantics by construction and no additional "correctness" theorems need to be proved.
- An element $\mathcal{S}_\alpha$ of the domain $(\mathbb{A}, \leq)$ is the specification, i.e., the abstraction of the intended concrete semantics.
- The partial correctness of a program P w.r.t. a specification $\mathcal{S}_\alpha$ can be expressed as $\alpha([\![P]\!]) \leq \mathcal{S}_\alpha$.
- Since $[\![P]\!]$ is defined as the least fixpoint of the operator $\mathcal{T}_P$, a sufficient condition [1] for the partial correctness is

$$\mathcal{T}_P^\alpha(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha. \tag{1}$$

Following the above approach, verification techniques inherit the nice features of abstract interpretation. Namely, we can define a verification framework, parametric with respect to the (abstract) property we want to model. Given a specific property, the corresponding verification conditions are systematically derived from the framework and guaranteed to be indeed sufficient partial correctness conditions.

The inductive verification method based on the sufficient condition (1) does not require to compute fixpoints. In order to make it effectively applicable, we need

- a concrete fixpoint (denotational) semantics, which allows us to observe the property we want to verify.
- a finite representation of the intended abstract behavior (specification).

3 Verification methods

The partial correctness condition (1) was initially used (in the case of logic programs) in *abstract diagnosis* [6], a technique which extends declarative debugging [22,11] to a debugging framework parametric w.r.t. abstractions. A similar approach is taken in [3], where different approximations (modeled by abstract interpretation) can be used in the semantics and in the specification.

More general specifications (including pre and post conditions) are considered in [12], which defines a verification framework, where well known verifica-

[1] In fact $\mathcal{T}_P^\alpha(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha$ implies $[\![P]\!]^\alpha \leq \mathcal{S}_\alpha$ and, since $\alpha([\![P]\!]) \leq [\![P]\!]^\alpha$, the condition $\alpha([\![P]\!]) \leq \mathcal{S}_\alpha$ can be derived. Note that (1) means that the specification $\mathcal{S}_\alpha$ is a *pre-fixpoint* of the abstract semantic evaluation function $\mathcal{T}_P^\alpha$.

tion methods can be reconstructed, by simply choosing different abstractions.

The approach can be explained in terms of two steps of abstraction. The first step is concerned with the derivation of the most adequate semantics. Every proof method can be reconstructed as the condition (1) for a suitable choice of computational properties modelled by the abstraction. The second step performs the abstraction needed to model specific classes of properties which can lead to finitely representable specifications.

Therefore we can deal with different notions of partial correctness and their associated proof methods.

Success-correctness. In this case we consider post-conditions only. The most adequate semantics models *computed answers*.

I/O correctness. In this case specifications are pairs of pre- and post-conditions. With this method one can prove that the post-condition holds whenever the pre-condition is satisfied. The most adequate semantics models the functional dependencies between the initial and the resulting bindings for the variables of the goal.

I/O and call correctness. Specifications are still pairs of pre-post conditions. With this method one can prove also that the pre-conditions are satisfied by all the procedure calls. The most adequate semantics models the functional dependencies between the initial and the resulting bindings for the variables of the goal and information on *call patterns*.

As already mentioned, the second abstraction step is concerned with the choice of an abstract domain to approximate the properties. Of course we can make available to program verification all the abstract domains designed for static analysis, such as modes, types, groundness dependencies, etc. As in the case of static analysis, in general we lose precision. However we succeed in getting finite specifications.

4 Assertions and specification languages

As shown in [5], an alternative choice for the second abstraction step consists in defining an abstract domain whose elements are formulas (assertions) in a formal specification language. In this case we can specify properties of programs as assertions in a suitable specification language. Assertions, in fact, do define an abstract domain (as shown by the Cousots in the early papers on abstract interpretation).

Let us consider a first order language $\mathcal{L}$. We assume the signature of $\mathcal{L}$ to include functions, constants and variables of the programs we want to verify. Let $\mathbb{F}$ be a set of formulas (*assertions*) of $\mathcal{L}$, expressing properties of the arguments of predicates. We choose an interpretation $\mathcal{I}$ to define the semantics of the formulas of $\mathbb{F}$. The validity of a formula Φ in $\mathcal{I}$ under the *valuation* σ, written $\mathcal{I} \models_\sigma \Phi$, is defined as usual. Notice that substitutions can naturally be viewed as valuations.

A natural pre-order is induced on $\mathbb{F}$ by implication under the interpretation $\mathcal{I}$, i.e., $\Psi \preceq \Phi$ if and only if $\mathcal{I} \models \Psi \Rightarrow \Phi$. Our idea is to use formulas of $\mathbb{F}$ as abstract values to describe sets of substitutions. Basically we consider the following concretization from assertions to substitutions:

$$\gamma_{\mathbb{F}}(\Phi) := \{\sigma \in Subst \mid \mathcal{I} \models_\sigma \Phi\}.$$

If $\mathbb{F}$ is a complete lattice, closed under arbitrary conjunctions, the function $\gamma_{\mathbb{F}}$ is meet-additive. Then, by standard abstract interpretation results, it induces a Galois connection between $(\mathbb{F}, \preceq)$ and the power-set of sets of substitutions ordered by set inclusion. We can exploit this relation to provide assertional versions of the verification conditions for the previous proof methods which were given in terms of sets of substitutions. In the following we will restrict our attention to monotonic assertions.

Definition 4.1 *The assertion* Φ *is* monotonic *if for each* σ *such that* $\mathcal{I} \models_\sigma \Phi$, *whenever* $\eta \geq \sigma$ *then* $\mathcal{I} \models_\eta \Phi$.

By using monotonic assertions, we can derive the verification conditions of the methods of [4,2,9]. In order to prove I/O (and call) correctness, we deal with pre-post specifications $\mathcal{S}_I, \mathcal{S}_O$, functions which associate to each pure atom $p(\mathbf{x})$ an assertion Φ, with free variables in $\{\mathbf{x}\}$.

I/O correctness The sufficient verification conditions obtained from (1) in the case of I/O correctness are the following.

For each clause $c := p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n) \in P$,

$$\mathcal{I} \models \mathcal{S}_I(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \wedge \Phi_1 \wedge \cdots \wedge \Phi_n \Rightarrow \mathcal{S}_O(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}], \qquad (c)$$

where

$$\Phi_j := \begin{cases} \mathcal{S}_O(p_j(\mathbf{x}_j))[\mathbf{x}_j/\mathbf{t}_j] & \text{if } \mathcal{I} \models \mathcal{S}_I(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \Rightarrow \mathcal{S}_I(p_j(\mathbf{x}_j))[\mathbf{x}_j/\mathbf{t}_j] \\ \mathit{TRUE} & \text{otherwise} \end{cases}$$

I/O and call correctness The sufficient verification conditions obtained from (1) in the case of I/O and call correctness are the following.

For each clause $c := p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n) \in P$ and each $1 \leq k \leq n$,

$$\begin{aligned} \mathcal{I} \models \mathcal{S}_I(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \wedge \mathcal{S}_O(p_1(\mathbf{x}_1))[\mathbf{x}_1/\mathbf{t}_1] \wedge \cdots \wedge \\ \mathcal{S}_O(p_{k-1}(\mathbf{x}_{k-1}))[\mathbf{x}_{k-1}/\mathbf{t}_{k-1}] \Rightarrow \mathcal{S}_I(p_k(\mathbf{x}_k))[\mathbf{x}_k/\mathbf{t}_k], \end{aligned} \qquad (c_I^k)$$

and

$$\begin{aligned} \mathcal{I} \models \mathcal{S}_I(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}] \wedge \mathcal{S}_O(p_1(\mathbf{x}_1))[\mathbf{x}_1/\mathbf{t}_1] \wedge \cdots \wedge \\ \mathcal{S}_O(p_n(\mathbf{x}_n))[\mathbf{x}_n/\mathbf{t}_n] \Rightarrow \mathcal{S}_O(p(\mathbf{x}))[\mathbf{x}/\mathbf{t}]. \end{aligned} \qquad (c_O)$$

It is worth noting that whenever the relation $\models$ is decidable, we have an effective way to check the conditions. In the next section, as an example, we

take the language of properties in [23], which allows us to express the first order theory of types, groundness, freeness and sharing properties of terms. The language extends the language of [15,16], by providing also an effective procedure to decide the validity of formulas.

5 A simple assertion language

Consider the first order language obtained by closing with the usual first order connectives the predicates $ground(X)$ and $list(X)$, specifying ground terms and lists, respectively. Informally they are defined as

$$\mathcal{I} \models_\sigma ground(X) \text{ if and only if } \sigma(X) \text{ contains no variables.}$$

and

$$\mathcal{I} \models_\sigma list(X) \text{ if and only if } \sigma(X) \text{ is a list}$$

In [23] these and other properties have been considered and defined in detail. For example, the class of typing properties (like $list(X)$) has been formally defined using regular term grammars. We refer to that paper for the decision procedure.

Example 5.1 *Let us consider the following* naive sort. *The procedures* `leq` *and* `perm` *(not shown) are assumed to have the following properties:* `leq(X,Y)` *is successful if X and Y are numbers and $X \leq Y$, and* `perm(Xs,Ys)` *returns in Ys a permutation of the list Xs.*

```
c1:  sort(Xs,Ys) :- perm(Xs,Ys), ord(Ys).
c2:  ord([]).
c3:  ord([X,Y|Zs]) :- leq(X, Y), ord([Y|Zs]).
```

We can apply the I/O correctness proof method to show that the program is correct w.r.t. the following specification.

$$\mathcal{S}_I := \begin{cases} sort(X,Y) & \mapsto list(X) \wedge ground(X) \\ perm(X,Y) & \mapsto list(X) \wedge ground(X) \\ ord(X) & \mapsto list(X) \wedge ground(X) \\ leq(X,Y) & \mapsto ground(X) \wedge ground(Y) \end{cases}$$

$$\mathcal{S}_O := \begin{cases} sort(X,Y) & \mapsto list(Y) \wedge ground(Y) \\ perm(X,Y) & \mapsto list(Y) \wedge ground(Y) \\ ord(X) & \mapsto TRUE \\ leq(X,Y) & \mapsto TRUE \end{cases}$$

In our case we can show that, for example, the clause `c1` *is correct by*

showing the validity of the following formulas (which is straightforward).

$$list(Xs) \wedge ground(Xs) \Rightarrow list(Xs) \wedge ground(Xs)$$
$$list(Xs) \wedge ground(Xs) \wedge list(Ys) \wedge ground(Ys) \Rightarrow list(Ys) \wedge ground(Ys)$$
$$list(Xs) \wedge ground(Xs) \wedge list(Ys) \wedge ground(Ys) \wedge TRUE \Rightarrow list(Ys) \wedge ground(Ys)$$

It is worth noting that this approach allows us to perform a kind of program diagnosis. In fact, let us consider a small change in the program, obtained by inverting the order of the predicates in the body of the clause `c1`*, thus obtaining*

```
c1':  sort(Xs,Ys) :- ord(Ys), perm(Xs,Ys).
```

In this case the predicate `ord` *may be called with a non ground argument, even if the predicate* `sort` *is called correctly w.r.t. its pre-condition. This possibly wrong situation is detected by observing that the verification condition* $list(Xs) \wedge ground(Xs) \Rightarrow list(Ys) \wedge ground(Ys)$ *associated to the first clause is false. In other terms, a failure in proving one of the verification conditions allows us to detect possibly wrong clauses.*

6 Logic Programs as specifications

The specification language considered in Section 5 is decidable. However, the properties which can be used in a specification are given once for all. A more interesting case would be to let the user to define its own properties, by means of a logic program. As already mentioned, logic programs have already been proposed as specifications in the literature [13,10,3,21,20]. In particular, in [21] assertions associated to program points are verified at run time by evaluating the logic programs on the actual run time values. [10] proposes a new language to let the user communicate with the debugger. In this language specifications are logic programs and the user assertions are used to interactively diagnose errors.

In all these approaches the role of the user defined logic programs is to allow to *extensionally* derive information on the intended behavior, i.e., the specification. They are in fact used to evaluate the assertion on run time values and therefore to check that each program answer does indeed satisfy the assertion. In this section we propose a different approach, where the user defined logic programs are used to *intensionally* derive information on the intended behavior. This is obtained by syntactic program transformation techniques, which often allow us to prove the verification conditions.

In our specification language, assertions are formulas built on user defined predicates. The meaning of such predicates is specified by some user defined logic program. Once the verification conditions are derived, they can be proved by using the program and transformation techniques similar to the

ones described in [18].

Depending on the property we want to verify, different versions of these techniques can be used. For example, if we want to prove the partial correctness of a program w.r.t. computed answers we should be careful to use transformations preserving the computed answers semantics.

This section essentially presents some examples which show how our verification method works. As we will show in the following examples, our verification conditions will often be proved simply by using unfolding steps. In more complex examples, we need to prove some intermediate lemmata by using the goal replacement rule [18], which allows us to replace a goal with an equivalent (w.r.t. the chosen semantics) one. It is worth noting, however, that also the generation of the intermediate lemmata can often be obtained by using an unfold/fold proof method, as shown in [19]. This suggests that the process of proving the verification conditions can be automatized or at least semi-automatized.

6.1 Verification of properties of a reactive system

We consider the logic program of Figure 1 intended to model the behavior of a simple coffee machine which accepts 10 cents of Euro coins and gives back water for 10 cents and coffee for 20. The water is given immediately when requested, while the coffee can take a while to be served since the machine has to warm up. Streams (possibly infinite lists) of pairs (input,output) are used to model sequences of machine actions. The possible inputs are 'no actions', 'a 10 cents coin', 'the water request button' and 'the coffee request button'. The outputs are 'no actions', 'an error beep', 'a water cup' and 'a coffee cup'.

The concrete semantics needs to model partial answers in order to cope with the infinite behavior. However, condition (1) on the assertion domain boils down to the same sufficient conditions presented on Page 5.

The property we want to prove is that if we insert 20 cents and press the coffee request button, the coffee cup eventually comes. The specification is then

$$\mathcal{S}_I := \begin{cases} e00(X) & \mapsto \texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X) \\ e10(X) & \mapsto \texttt{sublistX}([(\texttt{10},\_),(\texttt{coffee},\_)],X) \\ e20(X) & \mapsto \texttt{sublistX}([(\texttt{coffee},\_)],X) \\ warm(X) & \mapsto TRUE \\ warm1(X) & \mapsto TRUE \end{cases}$$

```
c1:  e00( [ (null, null) | X] ) :- e00( X ).
c2:  e00( [ (10, null) | X] ) :- e10( X ).
c3:  e00( [ (water, beep) | X] ) :- e00( X ).
c4:  e00( [ (coffee, beep) | X] ) :- e00( X ).

c5:  e10( [ (null, null) | X] ) :- e10( X ).
c6:  e10( [ (10, null) | X] ) :- e20( X ).
c7:  e10( [ (water, water) | X] ) :- e00( X ).
c8:  e10( [ (coffee, beep) | X] ) :- e10( X ).

c9:  e20( [ (null, null) | X] ) :- e20( X ).
cA:  e20( [ (water, water) | X] ) :- e10( X ).
cB:  e20( [ (coffee, coffee) | X] ) :- e00( X ).
cC:  e20( [ (coffee, null) | X] ) :- warm( X ).

cD:  warm( [ (null, null) | X] ) :- warm1( X ).
cE:  warm( [ (null, coffee) | X] ) :- e00( X ).

cF:  warm1( [ (null, coffee) | X] ) :- e00( X ).
```

Figure 1. The vending machine program

$$\mathcal{S}_O := \begin{cases} e00(X) & \mapsto \mathtt{match}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],(\_,\mathtt{coffee}),X) \\ e10(X) & \mapsto \mathtt{matchX}([(\mathtt{10},\_),(\mathtt{coffee},\_)],(\_,\mathtt{coffee}),X) \\ e20(X) & \mapsto \mathtt{matchX}([(\mathtt{coffee},\_)],(\_,\mathtt{coffee}),X) \\ warm(X) & \mapsto \mathtt{matchX}([\,],(\_,\mathtt{coffee}),X) \\ warm1(X) & \mapsto \mathtt{matchX}([\,],(\_,\mathtt{coffee}),X) \end{cases}$$

where the user defined predicates are given in Figure 2. Since the property expressed by the precondition does not need to be *definitely* verified by all the traces of the system, we are not concerned with call correctness. Therefore we use the I/O correctness schema which leads to the following conditions.

clause `c1` By using some unfolding steps in the premise we can prove that $\mathcal{I} \models \mathcal{S}_I(e00(Y))[Y/[(\mathtt{null},\mathtt{null})|X]] \Rightarrow \mathcal{S}_I(e00(Z))[Z/X]$, i.e.,

$$\begin{array}{l} \mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],[(\mathtt{null},\mathtt{null})|X]) \Longrightarrow \\ \mathtt{sublist}([(\mathtt{10},\_),(\mathtt{10},\_),(\mathtt{coffee},\_)],X) \end{array}$$

```
sublist(Xs, Ys) :- sublistX(Xs,Ys).
sublist(Xs, [Y|Ys]) :- sublist(Xs,Ys).

sublistX([], Xs).
sublistX([Y|Xs],[Y|Ys]) :- sublistX(Xs,Ys).

match(Xs,X,Ys) :- matchX(Xs,X,Ys).
match(Xs,X,[Y|Ys]) :- match(Xs,X,Ys).

matchX([],X,[X|_]).
matchX([],X,[Y|Ys]) :- matchX([],X,Ys).
matchX([Y|Xs],X,[Y|Ys]) :- matchX(Xs,X,Ys).
```

Figure 2. The user defined predicates for the program of Figure 1

In fact, by unfolding the atom in the premise, this condition is rewritten in

$$\begin{array}{l}\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X)\vee\\ \texttt{sublistX}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{null},\texttt{null})|X])\Longrightarrow\\ \texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X)\end{array}$$

By unfolding `sublistX` we obtain

$$\begin{array}{l}\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X)\vee \mathit{FALSE}\Longrightarrow\\ \texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],X)\end{array}$$

Then (by some unfolding steps and logical implication properties) we can prove the verification condition $\texttt{c1}_O$

$$\begin{array}{l}\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{null},\texttt{null})|X]))\wedge\\ \texttt{match}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X)\Longrightarrow\\ \texttt{match}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{null},\texttt{null})|X])\end{array}$$

Indeed the condition can be rewritten as

$$\begin{array}{l}\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{null},\texttt{null})|X]))\wedge\\ \texttt{match}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X)\Longrightarrow\\ \texttt{match}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X)\vee\\ \texttt{matchX}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{null},\texttt{null})|X])\end{array}$$

which is a propositional tautology (since it has the form $A\wedge B\Longrightarrow B\vee C$).

clause `c2` By using some unfolding steps in the premise we can prove that

$$\texttt{sublist}([(\texttt{10},\_),(\texttt{10},\_),(\texttt{coffee},\_)],[(\texttt{10},\texttt{null})|X])\Longrightarrow$$

$$\texttt{sublistX}([(10,\_),(\texttt{coffee},\_)],X)$$

Then (by some unfolding steps and logical implication properties) we can prove the verification condition

$$\begin{array}{l}\texttt{sublist}([(10,\_),(10,\_),(\texttt{coffee},\_)],[(10,\texttt{null})|X])\wedge\\ \texttt{matchX}([(10,\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X)\Longrightarrow\\ \texttt{match}([(10,\_),(10,\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(10,\texttt{null})|X])\end{array}$$

clause `c3` Analogous to `c1`

clause `c4` Analogous to `c1`

clause `c5` By using an unfolding step in the premise we can prove that

$$\begin{array}{l}\texttt{sublistX}([(10,\_),(\texttt{coffee},\_)],[(\texttt{null},\texttt{null})|X])\Longrightarrow\\ \texttt{sublistX}([(10,\_),(\texttt{coffee},\_)],X)\end{array}$$

since the premise is false. Then we can prove the verification condition

$$\begin{array}{l}\texttt{sublistX}([(10,\_),(\texttt{coffee},\_)],[(\texttt{null},\texttt{null})|X])\wedge\\ \texttt{matchX}([(10,\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X)\Longrightarrow\\ \texttt{matchX}([(10,\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{null},\texttt{null})|X])\end{array}$$

clause `c6` Analogous to `c2`

clause `c7` By using an unfolding step in the premise we can prove that

$$\begin{array}{l}\texttt{sublistX}([(10,\_),(\texttt{coffee},\_)],[(\texttt{water},\texttt{water})|X])\Longrightarrow\\ \texttt{sublist}([(10,\_),(10,\_),(\texttt{coffee},\_)],X)\end{array}$$

since the premise is false. Then we can prove the verification condition

$$\begin{array}{l}\texttt{sublistX}([(10,\_),(\texttt{coffee},\_)],[(\texttt{water},\texttt{water})|X])\wedge\\ \texttt{match}([(10,\_),(10,\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),X)\Longrightarrow\\ \texttt{matchX}([(10,\_),(\texttt{coffee},\_)],(\_,\texttt{coffee}),[(\texttt{water},\texttt{water})|X])\end{array}$$

clause `c8` Analogous to `c5`

clause `c9` Analogous to `c5`

clause `cA` Analogous to `c7`

clause `cB` By using an unfolding step in the premise we can prove that

$$\begin{array}{l}\texttt{sublistX}([(\texttt{coffee},\_)],[(\texttt{coffee},\texttt{coffee})|X])\not\Longrightarrow\\ \texttt{sublist}([(10,\_),(10,\_),(\texttt{coffee},\_)],X)\end{array}$$

since the premise is true and the conclusion is not. Then we can prove the verification condition

$$\texttt{sublistX}([(\texttt{coffee},\_)],[(\texttt{coffee},\texttt{coffee})|X])\wedge \mathit{TRUE}\Longrightarrow$$

$$\texttt{matchX}([(\texttt{coffee}, \_)], (\_, \texttt{coffee}), [(\texttt{coffee}, \texttt{coffee})|X])$$

clause `cC` By using an unfolding step in the premise we can prove that

$$\texttt{sublistX}([(\texttt{coffee}, \_)], [(\texttt{coffee}, \texttt{null})|X]) \Longrightarrow \mathit{TRUE}$$

Then we can prove the verification condition

$$\begin{array}{l}\texttt{sublistX}([(\texttt{coffee}, \_)], [(\texttt{coffee}, \texttt{null})|X]) \wedge \\ \texttt{matchX}([\,], (\_, \texttt{coffee}), X) \Longrightarrow \\ \texttt{matchX}([(\texttt{coffee}, \_)], (\_, \texttt{coffee}), [(\texttt{coffee}, \texttt{null})|X])\end{array}$$

clause `cD` Since $\mathit{TRUE} \Longrightarrow \mathit{TRUE}$ we can prove the verification condition

$$\begin{array}{l}\mathit{TRUE} \wedge \texttt{matchX}([\,], (\_, \texttt{coffee}), X) \Longrightarrow \\ \texttt{matchX}([\,], (\_, \texttt{coffee}), [(\texttt{null}, \texttt{null})|X])\end{array}$$

clause `cE` By using an unfolding step in the premise we can prove that

$$\mathit{TRUE} \not\Longrightarrow \texttt{sublist}([(10, \_), (10, \_), (\texttt{coffee}, \_)], X)$$

since the premise is true and the conclusion is not. Then we can prove the verification condition

$$\mathit{TRUE} \Longrightarrow \texttt{matchX}([\,], (\_, \texttt{coffee}), [(\texttt{null}, \texttt{coffee})|X])$$

clause `cF` Analogous to `cE`

We conclude that the program is partially correct w.r.t. the specification. Note that if we use a stronger notion of partial correctness (including call correctness), we do not succeed in proving it, because we have no guarantee that every procedure call verifies the preconditions.

6.2 A simple property of append

We consider now the append program

```
c1:  append([], Ys, Ys).
c2:  append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

We want to prove that the expected relation among the list lengths holds. Thus the specification is

$$\begin{array}{rl}\mathcal{S}_I := \mathit{append}(X, Y, Z) \mapsto & \texttt{list}(X) \wedge \texttt{length}(X, Lx) \wedge \texttt{list}(Y) \wedge \\ & \texttt{length}(Y, Ly) \\ \mathcal{S}_O := \mathit{append}(X, Y, Z) \mapsto & \texttt{list}(Z) \wedge \texttt{length}(Z, Lz) \wedge Lz = Lx + Ly\end{array}$$

where the user defined predicates are as follows

```
list([]).
list([X|Xs]) :- list(Xs).

length([],0).
length([X|Xs],Lx) :- length(Xs, Lxs), Lx = Lxs + 1.
```

The property expressed by the precondition has now to be *definitely* verified by all the inputs. Therefore we use the I/O and call correctness schema which leads to the following conditions.

clause $\texttt{c1}_O$ The condition is

$$\texttt{list}([\,]) \wedge \texttt{length}([\,], Lx) \wedge \texttt{list}(Y) \wedge \texttt{length}(Y, Ly) \Longrightarrow \\ \texttt{list}(Y) \wedge \texttt{length}(Y, Lz) \wedge Lz = Lx + Ly$$

It can be proved by first proving the functionality of $\texttt{length}(Y, Ly)$ (i.e., $\texttt{length}(Xs, X) \wedge \texttt{length}(Xs, Y) \Longleftrightarrow \texttt{length}(Xs, X) \wedge X = Y$) by using the fold/unfold proof techniques of [19] and then using an unfolding step in the premise. In fact by unfolding $\texttt{length}([\,], Lx)$ and $\texttt{list}([\,])$ we obtain

$$\texttt{list}(Y) \wedge \texttt{length}(Y, Ly) \Longrightarrow \texttt{list}(Y) \wedge \texttt{length}(Y, Lz) \wedge Lz = 0 + Ly$$

By functionality we obtain

$$\texttt{list}(Y) \wedge \texttt{length}(Y, Ly) \Longrightarrow \texttt{list}(Y) \wedge \texttt{length}(Y, Ly) \wedge Ly = 0 + Ly$$

clause $\texttt{c2}_I$ The condition is

$$\texttt{list}([X|Xs]) \wedge \texttt{length}([X|Xs], Lxxs) \wedge \texttt{list}(Ys) \wedge \texttt{length}(Ys, Lys) \Longrightarrow \\ \texttt{list}(Xs) \wedge \texttt{length}(Xs, Lxs) \wedge \texttt{list}(Ys) \wedge \texttt{length}(Ys, Lys)$$

which can be proved by unfolding.

clause $\texttt{c2}_O$ The condition is

$$\texttt{list}([X|Xs]) \wedge \texttt{length}([X|Xs], Lxxs) \wedge \texttt{list}(Ys) \wedge \texttt{length}(Ys, Lys) \wedge \\ \texttt{list}(Zs) \wedge \texttt{length}(Zs, Lzs) \wedge Lzs = Lxs + Lys \Longrightarrow \\ \texttt{list}([X|Zs]) \wedge \texttt{length}([X|Zs], Lxzs) \wedge Lxzs = Lxxs + Lys$$

which (by unfolding and functionality of `length`) becomes

$$\texttt{list}(Xs) \wedge \texttt{length}(Xs, Lxs) \wedge Lxxs = Lxs + 1 \wedge \texttt{list}(Ys) \wedge \\ \texttt{length}(Ys, Lys) \wedge \texttt{list}(Zs) \wedge \texttt{length}(Zs, Lzs) \wedge Lzs = Lxs + Lys \Longrightarrow \\ \texttt{list}(Zs) \wedge \texttt{length}(Zs, Lxzs) \wedge Lxzs = Lzs + 1 \wedge Lxzs = Lxxs + Lys$$

which is true by arithmetic properties.

We conclude that the program is partially correct w.r.t. the specification.

```
c1:  isort([], []).
c2:  isort([X|Xs], Ys) :- isort(Xs, Zs), insert(X, Zs, Ys).

c3:  insert(X, [], [X]).
c4:  insert(X, [Y|Ys], [Y|Zs]) :- X > Y, insert(X, Ys, Zs).
c5:  insert(X, [Y|Ys], [X, Y|Ys]) :- X =< Y.
```

Figure 3. The insertion sort program

```
intlist([]).
intlist([X|Xs]) :- integer(X), intlist(Xs).

sort(Xs, Ys) :- perm(Xs, Ys), ord(Ys).

ord([]).
ord([X]).
ord([X,Y|Xs]) :- X =< Y, ord([Y|Xs]).

perm(Xs, [Z|Zs]) :- select(Z, Xs, Ys), perm(Ys, Zs).
perm([], []).

select(X, [X|Xs], Xs).
select(X, [Y|Xs], [Y|Zs]) :- select(X, Xs, Zs).
```

Figure 4. The user defined predicates for the program of Figure 3

6.3 Specifications and algorithms

In this example we want to prove that a clever implementation of sort (the insertion sort of Figure 3) is correct w.r.t. a specification given by the declarative (inefficient) specification of sort. Thus the specification is

$$\mathcal{S}_I := \begin{cases} isort(X,Y) & \mapsto \texttt{intlist}(X) \\ insert(X,Y,Z) & \mapsto int(X) \wedge \texttt{intlist}(Y) \wedge \texttt{ord}(Y) \end{cases}$$

$$\mathcal{S}_O := \begin{cases} isort(X,Y) & \mapsto \texttt{intlist}(Y) \wedge \texttt{sort}(X,Y) \\ insert(X,Y,Z) & \mapsto \texttt{intlist}(Z) \wedge \texttt{sort}([X|Y],Z) \end{cases}$$

where the user defined predicates are given in Figure 4. We assume the following specification for the built-ins.

$$\mathcal{S}_I := \begin{cases} X\texttt{=<}Y & \mapsto int(X) \wedge int(Y) \\ X\texttt{>}Y & \mapsto int(X) \wedge int(Y) \\ \texttt{integer}(X) & \mapsto \mathit{TRUE} \end{cases}$$

$$\mathcal{S}_O := \begin{cases} X\texttt{=<}Y & \mapsto X \leq Y \\ X\texttt{>}Y & \mapsto X > Y \\ \texttt{integer}(X) & \mapsto int(X) \end{cases}$$

Since the property in the precondition has to be *definitely* verified by all the inputs, we use the I/O and call correctness schema, which leads to the following partial correctness conditions.

clause $\texttt{c1}_O$ The condition is $\texttt{intlist}([\,]) \Longrightarrow \texttt{intlist}([\,]) \wedge \texttt{sort}([\,],[\,])$ which can be proved by a few unfolding steps.

clause $\texttt{c2}_I$ The conditions are $\texttt{intlist}([X|Xs]) \Longrightarrow \texttt{intlist}(Xs)$ and

$$\texttt{intlist}([X|Xs]) \wedge \texttt{intlist}(Zs) \wedge \texttt{sort}(Xs, Zs) \Longrightarrow \\ int(X) \wedge \texttt{intlist}(Zs) \wedge \texttt{ord}(Zs)$$

Both can be proved by a few unfolding steps in the premises.

clause $\texttt{c2}_O$ The condition is

$$\texttt{intlist}([X|Xs]) \wedge \texttt{intlist}(Zs) \wedge \texttt{sort}(Xs, Zs) \wedge \texttt{intlist}(Ys) \wedge \\ \texttt{sort}([X|Zs], Ys) \Longrightarrow \texttt{intlist}(Ys) \wedge \texttt{sort}([X|Xs], Ys)$$

It can be proved by first proving a property of `perm`, i.e., $\texttt{perm}(Xs, Zs) \wedge \texttt{perm}([X|Zs], Ys) \Longleftrightarrow \texttt{perm}([X|Xs], Ys)$.

clause $\texttt{c3}_O$ The condition is

$$int(X) \wedge \texttt{intlist}([\,]) \wedge \texttt{ord}([\,]) \Longrightarrow \texttt{intlist}([X]) \wedge \texttt{sort}([X],[X])$$

which can be proved by a few unfolding steps.

clause $\texttt{c4}_I$ The conditions are

$$int(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \Longrightarrow int(X) \wedge int(Y)$$

$$int(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \wedge X > Y \Longrightarrow \\ int(X) \wedge int(Y) \wedge \texttt{ord}(Ys)$$

Both can be proved by a few unfolding steps in the premises.

clause $\texttt{c4}_O$ The condition is

$$int(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \wedge X > Y \wedge \texttt{intlist}(Zs) \wedge \\ \texttt{sort}([X|Ys], Zs) \Longrightarrow \texttt{intlist}([Y|Zs]) \wedge \texttt{sort}([X, Y|Ys], [Y|Zs])$$

It can be proved by first proving a property of `sort`, i.e., $\texttt{sort}([X|Ys], Zs) \wedge \texttt{ord}([Y|Ys]) \wedge X > Y \implies \texttt{sort}([X, Y|Ys], [Y|Zs])$, and then by a few unfolding steps in the premises.

clause $\texttt{c5}_I$ The condition is

$$int(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \implies int(X) \wedge int(Y)$$

which can be proved by an unfolding step.

clause $\texttt{c5}_O$ The condition is

$$int(X) \wedge \texttt{intlist}([Y|Ys]) \wedge \texttt{ord}([Y|Ys]) \wedge X \leq Y \implies \texttt{intlist}([X, Y|Ys]) \wedge \texttt{sort}([X, Y|Ys], [X, Y|Ys])$$

It can be proved by first proving a property of `perm`, i.e., $\texttt{intlist}(Xs) \implies \texttt{perm}(Xs, Xs)$, and then by a few unfolding steps in the premises.

We conclude that the program is partially correct w.r.t. the specification.

7 Conclusions

In this paper we have first applied the verification framework defined in [5] to the derivation of sufficient partial correctness conditions on suitable abstract domains. The verification framework can be instantiated to specifications given in terms of assertions (which can be viewed as an intensional semantics). We have shown that assertions can indeed be handled as abstract domains and have shown two applications with different specification languages.

The first one is a simple decidable assertion language, which is able to express properties of terms, including types and other properties relevant to static analysis. An open interesting issue is the definition of more expressive (still decidable) specification languages.

The second one allows the user to specify properties to be used in the assertions by means of logic programs considering an assertion language able to derive user defined properties. We have shown, through some examples, how the resulting sufficient verification conditions can be derived and proved by using program transformations techniques. Most of the verification conditions can very easily be proven by using some unfolding steps, while other transformation techniques, such as goal replacement, are needed to prove more complex properties. The generation of the intermediate lemmata needed for goal replacement can often be obtained by using an unfold/fold proof method, as shown in [19]. These considerations suggest that the process of proving verification conditions can easily be semi-automatized by using, for example, some of the recently implemented systems for the transformation of logic programs [17].

As a final remark, we want to point out that our approach can be generalized to other paradigms. We just need to define a denotational semantics on

the concrete domain. By using the approach of Section 6, logic programs are used for specification only, thus exploiting their declarative nature.

References

[1] Apt, K. R., *Introduction to Logic Programming*, in: J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Elsevier and The MIT Press, 1990 pp. 495–574.

[2] Bossi, A. and N. Cocco, *Verifying correctness of logic programs*, in: J. Diaz and F. Orejas, editors, *Proceedings of TAPSOFT'89*, 1989, pp. 96–110.

[3] Bueno, F., 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: M. Kamkar, editor, *Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging)* (1997), pp. 155–169.

[4] Clark, K. L., *Predicate Logic as a Computational Formalism*, Res. Report DOC 79/59, Imperial College, Dept. of Computing, London (1979).

[5] Comini, M., R. Gori, G. Levi and P. Volpe, *Abstract Interpretation based Verification of Logic Programs*, in: S. Etalle and J.-G. Smaus, editors, *Proceedings of the Workshop on Verification of Logic Programs*, Electronic Notes in Theoretical Computer Science **30** (2000).

[6] Comini, M., G. Levi, M. C. Meo and G. Vitiello, *Abstract Diagnosis*, Journal of Logic Programming **39** (1999), pp. 43–93, special Issue on Synthesis, Transformation and Analysis of Logic Programs.

[7] Cousot, P. and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, in: *Proceedings of Fourth ACM Symp. Principles of Programming Languages*, 1977, pp. 238–252.

[8] Cousot, P. and R. Cousot, *Systematic Design of Program Analysis Frameworks*, in: *Proceedings of Sixth ACM Symp. Principles of Programming Languages*, 1979, pp. 269–282.

[9] Deransart, P., *Proof Methods of Declarative Properties of Definite Programs*, Theoretical Computer Science **118** (1993), pp. 99–166.

[10] Drabent, W., S. Nadjm-Tehrani and J. Maluszynski, *Algorithmic Debugging with Assertions*, in: H. Abramson and M. H. Rogers, editors, *Meta-programming in Logic Programming* (1989), pp. 383–398.

[11] Ferrand, G., *Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method*, Journal of Logic Programming **4** (1987), pp. 177–198.

[12] Levi, G. and P. Volpe, *Derivation of Proof Methods by Abstract Interpretation*, in: C. Palamidessi, H. Glaser and K. Meinke, editors, *Principles of Declarative Programming. 10th International Symposium, PLILP'98*, Lecture Notes in Computer Science **1490** (1998), pp. 102–117.

[13] Lichtenstein, Y. and E. Y. Shapiro, *Abstract Algorithmic Debugging*, in: R. A. Kowalski and K. A. Bowen, editors, *Proceedings of Fifth Int'l Conf. and Symp. on Logic Programming*, Seattle, 1988, pp. 512–531.

[14] Lloyd, J. W., "Foundations of Logic Programming," Springer-Verlag, 1987, second edition.

[15] Marchiori, E., *A Logic for Variable Aliasing in Logic Programs*, in: G. Levi and M. Rodriguez-Artalejo, editors, *Proceedings of the 4th International Conference on Algebraic and Logic Programming (ALP'94)*, Lecture Notes in Computer Science **850** (1994), pp. 287–304.

[16] Marchiori, E., *Design of Abstract Domains using First-order Logic*, in: M. Hanus and M. Rodriguez-Artalejo, editors, *Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96)*, Lecture Notes in Computer Science **1139** (1996), pp. 209–223.

[17] Pettorossi, A. and M. Proietti, *MAP: A Tool for Program Transformation*, URL: `http://www.iasi.rm.cnr.it/~proietti/system.html`.

[18] Pettorossi, A. and M. Proietti, *Transformation of Logic Programs*, in: D. M. Gabbay, C. J. Hogger and J. A. Robinson, editors, *Handbook of Logic in Artificial Intellince and Logic Programming*, Oxford University Press, 1998 pp. 697–787.

[19] Pettorossi, A. and M. Proietti, *Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs*, Journal of Logic Programming **41** (1999), pp. 197–230.

[20] Puebla, G., F. Bueno and M. Hermenegildo, *A Generic Preprocessor for Program Validation and Debugging*, in: P. Deransart, M. Hermenegildo and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, Lecture Notes in Computer Science **1870**, Springer-Verlag, 2000 To appear.

[21] Puebla, G., F. Bueno and M. Hermenegildo, *An Assertion Language for Costraint Logic Programs*, in: P. Deransart, M. Hermenegildo and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, Lecture Notes in Computer Science **1870**, Springer-Verlag, 2000 To appear.

[22] Shapiro, E. Y., "Algorithmic Program Debugging," The MIT Press, 1983.

[23] Volpe, P., *A first-order language for expressing aliasing and type properties of logic programs*, Science of Computer Programming (2000), to appear.