How to transform an analyzer into a verifier

Marco Comini
Dipartimento di Matematica e Informatica – Università di Udine
Udine, Italy

Roberta Gori, Giorgio Levi
Dipartimento di Informatica – Università di Pisa
Pisa, Italy

Abstract

In this paper we push forward the idea of applying the abstract interpretation concepts to the problem of verification of programs. We consider the theory of abstract verification as proposed in [6, 5] and we show how it is possible to transform static analyzers with some suitable properties to obtain automatic verification tools based on sufficient verification conditions. We prove that the approach is general and flexible by showing three different verification tools based on different domains of types for functional, logic programming and *CLP* programming. The verifier for functional programs is obtained from a static analyzer which implements one of the polymorphic type domains introduced by Cousot [9]. The one for logic programs is obtained from a static analyzer on a type domain designed by Codish and Lagoon [3], while the verifier for *CLP* programs is obtained starting from the type analyzer described in [17].

1 Abstract Interpretation

Abstract interpretation [10, 11] is a general theory for approximating the semantics of discrete dynamic systems, originally developed by Patrick and Radhia Cousot, in the late 70's, as a unifying framework for specifying and validating static program analyses. The *abstract semantics* is an approximation of the concrete one, where exact (concrete) properties are replaced by approximated properties, modeled by an abstract domain. The framework of abstract interpretation can be useful to study hierarchies of semantics and to reconstruct data-flow analysis methods. It can be used to prove the safety of an analysis algorithm. However, it can also be used to systematically derive "optimal" abstract semantics from the abstract domain.

From the very beginning, abstract interpretation was shown to be useful for the automatic generation of program invariants. Even more recently [12, 14, 8], it was shown to be very useful to understand, organize and synthesize proof methods for *program verification*. In particular, we are interested in one specific approach to the generation of abstract interpretation-based partial correctness conditions [21, 24], which is used also in abstract debugging [1, 7, 2].

2 Verification and Abstract Interpretation

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.

In order to formally 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.

Assume we have a semantic evaluation function $\mathcal{T}_P$ on a concrete domain $(\mathbb{C}, \sqsubseteq)$, whose least fixpoint $\mathrm{lfp}_{\mathbb{C}}(\mathcal{T}_P)$ is the semantics of the program P. The ideas behind this approach are the following.

- 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 $\mathrm{lfp}_{\mathbb{A}}(\mathcal{T}_P^{\alpha})$ is a correct approximation of the concrete semantics by construction, i.e., $\alpha(\mathrm{lfp}_{\mathbb{C}}(\mathcal{T}_P)) \leq \mathrm{lfp}_{\mathbb{A}}(\mathcal{T}_P^{\alpha})$, 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

  $$\alpha(\mathrm{lfp}_{\mathbb{C}}(\mathcal{T}_P)) \leq \mathcal{S}_\alpha. \qquad (2.1)$$

- Since condition (2.1) requires the computation of the concrete fixpoint semantics, this condition is not always effectively computable. Then, we can prove instead the condition

  $$\mathrm{lfp}_{\mathbb{A}}(\mathcal{T}_P^{\alpha}) \leq \mathcal{S}_\alpha \qquad (2.2)$$

  which implies partial correctness [1]. Note that the new verification condition does not require the computation of the concrete fixpoint semantics. However an abstract fixpoint computation is still needed.
- A simpler condition, which is *sufficient* [2] for (2.2) and, therefore, for partial correctness to hold, is

  $$\mathcal{T}_P^{\alpha}(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha. \qquad (2.3)$$

  Note that this is Park's *fixpoint induction* condition [22].

[1]Since, by correctness, $\alpha(\mathrm{lfp}_{\mathbb{C}}(\mathcal{T}_P)) \leq \mathrm{lfp}_{\mathbb{A}}(\mathcal{T}_P^{\alpha})$, (2.2) implies (2.1).

[2]In fact $\mathcal{T}_P^{\alpha}(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha$ implies $\mathrm{lfp}_{\mathbb{A}}(\mathcal{T}_P^{\alpha}) \leq \mathcal{S}_\alpha$ since the specification $\mathcal{S}_\alpha$ is a *pre-fixpoint* of the abstract semantic evaluation function $\mathcal{T}_P^{\alpha}$.

Following the above approach, 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.

An important result is that, following our abstract interpretation approach, the issue of completeness of a verification method can be addressed in terms of properties of the chosen abstract interpretation. In general, in fact, given an inductive proof method, if a program is correct with respect to a specification $\mathcal{S}$ (i.e., if (2.1) is satisfied) the sufficient condition (2.3) might not hold for $\mathcal{S}$. However, if the method is *complete*, then when the program is correct with respect to $\mathcal{S}$, there exists a property $\mathcal{X}$, stronger than $\mathcal{S}$, which verifies the sufficient condition. [21, 24] proved that the method is complete if and only if the abstraction is *precise* with respect to $\mathcal{T}_P$, that is if $\alpha(\mathrm{lfp}_{\mathbb{C}}(\mathcal{T}_P)) = \mathrm{lfp}_{\mathbb{A}}(\mathcal{T}_P^{\alpha})$. This approach allows us to use some standard methods (see for example [20]), which allow us to systematically enrich a domain of properties so as to obtain an abstraction which is *fully precise* ($\alpha \cdot F = F^{\alpha} \cdot \alpha$) w.r.t. a given function F. Since full precision w.r.t. the semantic function $\mathcal{T}_P$ implies precision with respect to $\mathcal{T}_P$, these methods can be viewed as the basis for the systematic development of complete proof methods.

3 Sufficient Verification Conditions

As we have already pointed out, trying to prove condition (2.1) leads to a non effective verification method. This is due to the fact that (2.1) requires the computation of the concrete fixpoint semantics, which, in general, is not effective. A verification method based on condition (2.2) is effective only if the abstract domain is Noetherian or otherwise if we use widening operators to ensure the termination of the computation of the abstract fixpoint semantics. This is the approach adopted, for example, in the verifier of the Ciao Prolog Development System [3].

Even if the methods based on condition (2.2) may seem stronger than the methods based on condition (2.3), this is not always the case. When the domain is non-Noetherian the use of widening operators leads to an unavoidable loss of precision, which, in some case, makes condition (2.2) weaker than condition (2.3). We will show an example of this in the case of the polymorphic type domain for functional languages considered in Section 4.2. In particular we will show that, using a verification method based on condition (2.2) with the ML widening, it is not possible to prove that the function in Figure 1 on page 4 has type $(`a \to `a) \to (`a \to `b) \to \mathit{int} \to `a \to `b$ while it is possible using condition (2.3). Moreover, even when the abstract domain is Noetherian, the computation of the abstract fixpoint semantics may be very expensive.

On the contrary, the inductive verification method based on (sufficient) condition (2.3) does not require to compute fixpoints. Therefore proving $\mathcal{T}_P^{\alpha}(\mathcal{S}_{\alpha}) \leq \mathcal{S}_{\alpha}$ is, in general, not expensive even for complex abstract domains. Moreover, when the function $\mathcal{T}_P^{\alpha}$ can be viewed as the union of functions $\mathcal{T}_c^{\alpha}$ defined on the primitive components of a program, using condition (2.3) has another advantage. Proving

[3]Available at URL: http://www.clip.dia.fi.upm.es/Software/Ciao/

```
let rec f f1 g n x =
  if n=0 then g(x)
  else f(f1)(function x -> (function h -> g(h(x)))) (n-1)
```

Figure 1: The recursive function f.

condition $\mathcal{T}_P^\alpha(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha$ boils down to verify $\mathcal{T}_{P\,c}^\alpha(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha$ for every $c \in P$. In this case, this allows us to prove condition (2.3) *compositionally.* For example, in logic programming the condition $\mathcal{T}_P^\alpha(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha$ can be verified by proving for each clause $c \in P$, $\mathcal{T}_{P\,c}^\alpha(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha$. This approach is also useful for abstract debugging [7]. If, in the compositional proof of $\mathcal{T}_P^\alpha(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha$ we fail in the program component c, there is a possible bug in c.

For all the above reasons, we consider verification methods based on condition (2.3). Therefore, in order to derive effective verification methods we need to choose an abstract domain $(\mathbb{A}, \leq)$ where

- the intended abstract behavior (specification) $\mathcal{S}_\alpha \in \mathbb{A}$ has a finite representation;
- $\leq$ is a decidable relation.

This allows us to use, in addition to all the Noetherian abstract domains used in static analysis, non-Noetherian domains (such as polymorphic type domains for functional languages), which lead to finite abstract semantics, and finite representations of properties.

Hence, every time we have a static analysis computed by a fixpoint abstract semantics operator, we can systematically construct a verifier based on conditions (2.3). We only need to realize the $\leq$ operation on the abstract domain $\mathbb{A}$. The verifier is a tool which applies once the abstract fixpoint semantic operator to the user specification $\mathcal{S}_\alpha \in \mathbb{A}$ and verifies that the result is indeed $\leq \mathcal{S}_\alpha$.

In this paper we show how easy this process can be by showing three different examples. All verification tools we will present here are obtained by starting from static analyzers defined on type domains. We prove that our approach is very general and flexible by defining verifications tools for three different paradigms: Functional Programming, Logic and Constraint Logic Programming. In particular, the verification tool for functional programming, presented in Section 4, is obtained from a static analyzer which implements one of the polymorphic type domains introduced by Cousot in [9].

The verification tool for logic programming, presented in Section 5, is obtained from a static analyzer on a type domain designed by Codish and Lagoon [3].

Finally, the verifier for *CLP* programs, presented in Section 6, is obtained starting from the type analyzer described in [17].

4 Type inference in higher order functional languages via abstract interpretation

As we will show in the following, the "higher-order types" abstract domain is non-Noetherian. This is therefore a typical case of application of our approach based on the effective sufficient condition (2.3), which does not require fixpoint computations.

Our language is a small variation of untyped λ-calculus as considered in [9]. [9] shows that several known type systems and corresponding type inference algorithms can systematically be derived from (the collecting version of) a concrete denotational semantics. The main advantage of the abstract interpretation approach to type inference is that the type abstract interpreters are correct, by construction, w.r.t. the concrete semantics. This means that "the type of the concrete value computed by the concrete semantics is in general more precise than the type computed by the type abstract interpreter". As it is often the case with abstract interpretation, approximation is introduced by abstract computations. By looking at the relation between the concrete semantics and the abstract interpreters, we can reason about the relative precision of different type inference algorithms.

The traditional point of view of type systems is quite different, since type inference is viewed as an extension of syntactic analysis. Namely, the (concrete) semantics is only defined for those programs which can be typed. Traditional type inference algorithms do also introduce approximation. However this cannot be directly related to a concrete semantics, because the latter is based on the result of type inference. The result is that there exist programs which cannot be typed, yet would have a well-defined concrete semantics, i.e., there exist non typeable programs which would never go wrong, if provided with a concrete semantics with " dynamic" type checking. Let us look at a couple of examples, where we use the ML syntax.

The ML expression

```
let rec f f1 g n x =
  if n=0 then g(x)
  else f(f1)(function x -> (function h -> g(h(x)))) (n-1) x f1
    in f (function x -> x+1) (function x -> x+1) 10 5;;

This expression has type ('a -> 'a) -> 'b
but is here used with type 'b.
```

(taken from [9]) which is an application of the function f of Figure 1 on page 4, cannot be typed by the ML type inference algorithm. By using a concrete semantics with "dynamic" type checking, we would obtain a correct concrete result (`-: int=16`).

The expression is indeed a type correct application of the function $f\ f_1\ g\ n\ x = g(f_1^n(x))$ which has the type $(`a \rightarrow `a) \rightarrow (`a \rightarrow `b) \rightarrow int \rightarrow `a \rightarrow `b$. As we will see in the following, the function cannot be typed by the ML type inference algorithm, because of an approximation related to recursive functions. The same approximation does not allow the ML algorithm to type the expression

```
# let rec f x = x and g x = f (1+x) in f f 2;;
```

```
This expression has type int -> int but is here used with type int
```

Because of the approximation related to the "syntactic" mutual recursion, the type assigned to f is $int \rightarrow int$ rather than $`a \rightarrow `a$. Again a concrete semantics with dynamic type checking, would compute a correct concrete result (`-: int=2`). The abstract interpreter considered in the next section, succeeds in correctly typing the above expressions and is, therefore, more precise than the ML type inference algorithm.

4.1 A type abstract interpreter

Following the approach in [9], we have developed (and implemented in OCAML [23]) several abstract interpreters for inferring principal types, represented as Herbrand terms with variables, for various notions of types (monomorphic types à la Hindley, let polymorphism and polymorphic recursion).

For the current presentation we will represent programs using the syntax of ML. Since we will sometimes compare our results with those obtained by the ML type inference algorithm, we will just consider the let-polymorphic abstract interpreter [4], which corresponds to the ML type system.

The language has one basic type only: *int*. Types are Herbrand terms, built with the basic type *int*, variables and the (functional) type constructor $\rightarrow$.

The actual domain of the abstract interpreter is more complex, and contains explicitly quantified terms and constraints. For the sake of our discussion, abstract values will simply be (equivalence classes under variable renaming of) terms. The partial order relation is the usual instantiation relation, i.e., $t_1 \leq t_2$ if t_2 is an instance of t_1. Note that the resulting abstract domain is non-Noetherian since there exist infinite ascending chains.

Our let-polymorphic type interpreter turns out to be essentially equivalent to the ML type inference algorithm, with one important difference, related to the abstract semantics of recursive functions. Such a semantics should in principle require a least fixpoint computation. However since the domain is non-Noetherian, the fixpoint cannot, in general, be reached in finitely many iterations. The problem is solved in the ML type inference algorithm, by using a widening operator (based on unification) after the first iteration. Widening operators [13] give an upper approximation of the least fixpoint and guarantee termination by introducing further approximation. We apply the same widening operator after k iterations. This allows us to get often the least fixpoint and, in any case, to achieve a better precision.

The "let" and "let rec" constructs are handled as "declarations"; the abstract semantic evaluation function for declarations has the following type

$$semd : declaration \rightarrow env \rightarrow int \rightarrow env,$$

[4]The abstract syntax of the language, together with the concrete semantics, the implementation of the abstract domain and the resulting abstract interpreter can be found at `http://www.di.unipi.it/~levi/typesav/pagina.html`.

where *env* is the domain of abstract environments, which associates types to identifiers and the integer parameter is used to control the approximation of the widening operator.

We now show the analysis of a recursive function *pi* which computes $\prod_{n=a}^{n=b} f(n)$. The result is an environment where the function identifier *pi* is bound to its inferred type.

```
# semd (let rec pi f a b =
  if (a - (b +1) = 0) then 1 else (f a) * (pi f (a +1) b))[] 0;;

- : env = [ pi <- (int -> int) -> int -> int -> int ]
```

Consider now the recursive function f in Figure 1 on page 4. The evaluation of the abstract semantic evaluation function *semd* with the control parameter set to 0, gives us the result of the ML inference algorithm, i.e., f cannot be typed.

```
# semd(let rec f f1 g n x =
  if n=0 then g(x)
  else f(f1)(function x -> (function h ->g(h(x)))) (n-1) x f1) [] 0;;

- : env = [ f <- Notype ]
```

However, the application of *semd* with the control parameter set to 3 computes the (following) right type for function f, which is indeed the least fixpoint.

```
# semd(let rec f f1 g n x =
  if n=0 then g(x)
  else f(f1)(function x -> (function h -> g(h(x)))) (n-1) x f1)[] 3;;

- : env = [ f <- ('a -> 'a) -> ('a -> 'b) -> int -> 'a -> 'b ]
```

Setting the control parameter to -1, will allow us to compute the least fixpoint without even using the widening operator.

4.2 From the type interpreter to the type verifier

Once we have the abstract semantic evaluation function *semd*, we can easily use it for program verification, by taking an abstract environment as specification (abstraction of the intended semantics). Assume we want to verify a declaration d w.r.t. a specification S, by "implementing" the sufficient condition (2.3) ($\mathcal{T}_P^{\alpha}(\mathcal{S}_\alpha) \leq \mathcal{S}_\alpha$) where, $\leq$ is the lifting to abstract environments of the partial order relation on terms. The application of the abstract semantic evaluation to the specification can be implemented as follows:

1. if $d = (let\ f = e)$ is the declaration of a non recursive function f, $semd\ d\ s\ k$ returns a new environment S', where f is bound to the type computed by assuming that all the global names have the types given in the specification.

2. if $d = (let\ rec\ f = e)$ is a declaration of a recursive function f, $semd\ (let\ f = e)\ s\ k$ returns a new environment S', where f is bound to the type computed, by assuming that all the global names *and* f have the types given in the specification. In other words, we take the type for f given in the specification and we evaluate once the body of the recursive function, to get a new type for f. Note that we abstractly execute the recursive function body just once and we do not compute the fixpoint. Note also that the control parameter is possibly used only for approximating fixpoints corresponding to recursive functions occurring within e.

We are then left with the problem of establishing whether $S' \leq S$. Since S' can only be different from S in the denotation of f, we can just show that $S(f)$ is an instance of $S'(f)$. The verification method can then be implemented by a function *checkd* : *declaration* $\rightarrow$ *specification* $\rightarrow$ *int* $\rightarrow$ *bool* consisting of three lines of ML code.

We have also implemented the verification condition (2.2) ($\text{lfp}_{\mathbb{A}}(\mathcal{T}_P^{\alpha}) \leq \mathcal{S}_{\alpha}$) by an even simpler ML function *checkdf*.

It is worth noting that both *checkd* and *checkdf* allow us to verify a program consisting of a set of function declarations in a modular way. Each declaration is verified in a separate step, by using the specification for determining the types of global identifiers.

In the following section we show and discuss some examples.

4.3 Examples

We show now two examples of verification (through *checkd*) of *pi*: in the second example, the verification fails since the type given in the specification is too general.

```
# checkd (let rec pi f a b =
  if (a - (b +1) = 0) then 1 else (f a) * (pi f (a +1) b))
[ pi <- (int -> int) -> int -> int -> int ] 0;;

- : bool = true

# checkd (let rec pi f a b =
  if (a - (b +1) = 0) then 1 else (f a) * (pi f (a +1) b))
[ pi <- ('a -> 'a) -> 'a -> 'a -> 'a ] 0;;

- : bool = false
```

We can consider also the verification of the traditional identity function *id*. Note that *id* is also correct w.r.t. a specification which is an instance of the correct principal type.

```
# checkd (let id x = x)[ id <- 'a -> 'a ] 0;;

- : bool = true
```

```
# checkd (let id x = x)[ id <- int -> int ] 0;;

- : bool = true
```

Now we verify a function, which is intended to compute the factorial and which is defined by a suitable composition of *pi* and *id*. In the verification, we use the specification rather than the semantics for *pi* and *id*. Note that if we take a wrong specification for *id*, we cannot prove the type correctness of *fact*.

```
# checkd (let fact = pi id 1)  [ pi <- (int->int)->int->int->int;
id <- 'a -> 'a; fact <- int -> int ] 0;;

- : bool = true

# checkd (let fact = pi id 1)  [ pi <- (int->int)->int->int->int;
id <- 'a -> 'b -> 'a; fact <- int -> int ] 0;;

- : bool = false
```

Now we show an example involving polymorphism, where two occurrences of the polymorphic function *id* take different instances of the type in the specification.

```
# checkd (let g = id id)  [ id <- 'a -> 'a; g <- 'b -> 'b ] 0;;

- : bool = true

# checkd (let g = id id)
[ id <- 'a -> 'a; g <- ('b -> 'b) -> ('b -> 'b) ] 0;;

- : bool = true
```

In Figure 2 on page 10 we consider again the recursive function f of Figure 1 on page 4. We now show that if we use the verification condition defined by *checkdf* (with widening control parameter set to 0), we fail in establishing the correctness w.r.t. the principal type (since, as we have already shown, the ML type inference algorithm fails). On the contrary, the verification condition defined by *checkd* succeeds.

In Figure 3 on page 11, we show some aspects related to the incompleteness of the verification method defined by *checkd*, still in the case of the function f of Figure 2 on page 10. In fact, *checkd* fails to establishing the correctness of f w.r.t. a specification in which all the variables in the principal type are instantiated to *int*. If we use the stronger verification method, based on the computation of the fixpoint (condition (2.2), without widening), we succeed. The last example shows that if we verify a specific application of f, we succeed again even with *checkd*, because the recursive definition, being inside a declaration, leads anyway to a fixpoint computation.

Let us finally consider the issue of termination. The recursive function in the first example of Figure 4 on page 12 is not typed by ML. If we try to verify it w.r.t.

```
# checkdf(let rec f f1 g n x =
  if n=0 then g(x)
  else f(f1)(function x -> (function h -> g(h(x)))) (n-1) x f1)
[ f <- ('a -> 'a) -> ('a -> 'b) -> int -> 'a -> 'b ] 0;;

- : bool = false

# checkd(let rec f f1 g n x =
  if n=0 then g(x)
  else f(f1)(function x -> (function h -> g(h(x)))) (n-1) x f1)
[ f <- ('a -> 'a) -> ('a -> 'b) -> int -> 'a -> 'b ] 0;;

- : bool = true
```

Figure 2: Verification of the recursive function f.

a specification assigning to it the type $`a \to `a$, we correctly fail by using both the verification method based on condition (2.3) and the verification method based on condition (2.2) with widening. If we apply the condition (2.2) without widening, the verification process does not terminate.

5 Type verification in logic languages via abstract interpretation

In this section we will show an example of transformation of an analyzer into a verifier for logic programming. As in the case of functional programming we will consider an abstract domain of types. This abstract domain of types for logic programming was introduced in [3]. In order to formally introduce this domain, we have first to define the abstraction from concrete terms to type terms $\tau : \mathit{Terms} \to \mathit{TypeTerms}$. Type terms in this domain are associative, commutative and idempotent. They are built using a binary set constructor $+$ and a collection of *monomorphic* and *polymorphic* description symbols. The monomorphic symbols are constants (e.g. $\mathit{num}/0$, $\mathit{nil}/0$) and the polymorphic symbols are unary (e.g. $\mathit{list}/1$, $\mathit{tree}/1$). Intuitively, the description symbols represent sets of function symbols in the corresponding concrete alphabet. For example, the description symbol *list* might be defined to represent the $\mathit{cons}/2$ symbol in the concrete alphabet and the description of the constant *num* might represent the symbols 0, 1, etc.

```
# checkd(let rec f f1 g n x =
  if n=0 then g(x)
  else f(f1)(function x -> (function h -> g(h(x)))) (n-1) x f1)
[ f <- (int -> int) -> (int -> int) -> int -> int -> int ] 0;;

- : bool = false

# checkdf(let rec f f1 g n x =
  if n=0 then g(x)
  else f(f1)(function x -> (function h -> g(h(x)))) (n-1) x f1)
[ f <- (int -> int) -> (int -> int) -> int -> int -> int ] (-1);;

- : bool = true

# checkd(let g = (let rec f f1 g n x =
  if n=0 then g(x)
  else f(f1)(function x -> (function h -> g(h(x)))) (n-1) x f1
    in f)(function x -> x + 1))
[ g <- (int -> int) -> int -> int -> int ] (-1);;

- : bool = true
```

Figure 3: Verification of the recursive function f.

The abstraction function is defined by induction on terms:

$$\tau(t) := \begin{cases} X & \text{if } t \text{ is the variable } X \\ \mathit{num} & \text{if } t \text{ is a number} \\ \mathit{nil} & \text{if } t = [\,] \\ \mathit{list}(\tau(t_1)) + \tau(t_2) & \text{if } t = [t_1|t_2] \\ \mathit{void} & \text{if } t = \mathit{void} \\ \mathit{tree}(\tau(t_1)) + \tau(t_2) + \tau(t_3) & \text{if } t = \mathit{tree}(t_1, t_2, t_3) \\ \mathit{other} & \text{otherwise} \end{cases}$$

Thus, the abstractions of terms $[-3, 0, 7]$, $[X, Y]$, $[X|Y]$ and $\mathit{tree}(2, \mathit{void}, \mathit{void})$ are $\mathit{list}(\mathit{num}) + \mathit{nil}$ [5], $\mathit{list}(X) + \mathit{list}(Y) + \mathit{nil}$, $\mathit{list}(X) + Y$ and $\mathit{tree}(\mathit{num}) + \mathit{void}$ respectively.

Abstract atoms are simply built with abstract terms, and $\tau(p(t_1, \ldots, t_n)) := p(\tau(t_1), \ldots, \tau(t_n))$. Our abstract domain will be the types domain $\mathcal{D}_\tau$, which is the power-set of abstract atoms ordered by set inclusion.

[5] $\tau([-3, 0, 7]) = \mathit{list}(\tau(-3)) + \tau([0, 7]) = \mathit{list}(\mathit{num}) + \mathit{list}(\tau(0)) + \tau([7]) = \mathit{list}(\mathit{num}) + \mathit{list}(\mathit{num}) + \mathit{list}(\tau(7)) + \tau([]) = \mathit{list}(\mathit{num}) + \mathit{list}(\mathit{num}) + \mathit{list}(\mathit{num}) + \mathit{nil} = \mathit{list}(\mathit{num}) + \mathit{nil}$

```
# let rec f x = f;;

This expression has type 'a -> 'b but is here used with type 'b

# checkd (let rec f x = f) [ f <- 'a -> 'a ] 0;;

- : bool = false

# checkdf (let rec f x = f)[ f <- 'a -> 'a ]10;;

- : bool = false

# checkdf (let rec f x = f)[ f <- 'a -> 'a ](-1);;

Interrupted.
```

Figure 4: Verification of the non terminating function f.

5.1 From the type analyzer to the type verifier

We have already discussed how, once we have the abstract semantic evaluation function $\mathcal{T}_P^\alpha$, we can easily define a verification method based on condition (2.3). Actually, depending on the chosen evaluation function $\mathcal{T}_P^\alpha$, we can prove partial correctness of a program w.r.t. different properties. For example, we can prove that a program P is correct w.r.t. the intended types of the successful atoms or w.r.t. the intended types of the computed answers and so on. More concrete the semantic evaluation functions lead to stronger verification methods. Here, in order to show some interesting examples we consider a very strong method: the *I/O and call correctness* method *over the type domain*. It is obtained by instantiating the $\mathcal{T}_P^\alpha$ semantic evaluation function of condition (2.3) with an abstract semantic operator which is able to model the functional type dependencies between the initial and the resulting bindings for the variables of the goal *plus* information on call patterns.

Specifications are therefore pairs of pre and post conditions, which describe the intended input-output type dependencies. They are formalized as partial functions from *GAtoms* (the set of all generic atoms) to the domain $\mathcal{D}_\tau$ (denoted by $\mathbb{A}_\tau := [GAtoms \rightharpoonup \mathcal{D}_\tau]$) and are ordered by $\sqsubseteq$, the pointwise extension of $\subseteq$ on $\mathbb{A}_\tau$.

Proving condition (2.3) guarantees that for every procedure the post condition holds whenever the pre conditions are satisfied and that the pre conditions are satisfied by all the procedure calls. It is worth noting that the verification conditions, obtained in this case from condition (2.3) are a slight generalization of the ones defined by the Drabent-Maluszynski method [16].

We have developed a prototype verifier [6] which is able to test our verification conditions on the types domain. The verifier is obtained by using the existing abstract

[6]Available at URL: http://www.dimi.uniud.it/~comini/Projects/PolyTypesVerifier/.

operations defined in the type analyzer implemented by Lagoon [7]. The verification method can then be implemented by a function *verifyIOcall* : *clause* $\rightarrow$ *InputSpec* $\rightarrow$ *OutputSpec* $\rightarrow$ *bool* consisting in several Prolog predicates which implement condition (2.3) in case of a $\mathcal{T}_P^\alpha$ function which models the type dependencies of *I/O and call pattern* [6]. The code turns out to essentially compute ACI-unification between the abstractions of the atoms of a clause and all the matching items of the specification. Since the $\sqsubseteq$ operation on $\mathbb{A}_\tau$ is the pointwise extension of subset inclusion, the resulting set (which is necessarily finite) is then checked to be a subset of the specification.

In the following section we show and discuss some examples.

5.2 Examples

The `queens` program of Figure 5 on page 14 is proved to be correct w.r.t. the following intended specification w.r.t. the type domain $\mathbb{A}_\tau$.

$$\mathcal{S}_\tau^I := \begin{cases} queens(X,Y) & \mapsto \{queens(nil + list(num), T), queens(nil, T)\} \\ perm(X,Y) & \mapsto \{perm(nil + list(num), T), perm(nil, T)\} \\ delete(X,Y) & \mapsto \{delete(T, nil + list(num), U), delete(T, nil, U)\} \\ safe(X,Y) & \mapsto \{safe(nil + list(num)), safe(nil)\} \\ noattack(X,Y,Z) & \mapsto \left\{\begin{array}{l} noattack(num, nil, num), \\ noattack(num, nil + list(num), num) \end{array}\right\} \end{cases}$$

$$\mathcal{S}_\tau^O := \begin{cases} queens(X,Y) & \mapsto \left\{\begin{array}{l} queens(nil, nil), \\ queens(nil + list(num), nil + list(num)) \end{array}\right\} \\ perm(X,Y) & \mapsto \left\{\begin{array}{l} perm(nil, nil), \\ perm(nil + list(num), nil + list(num)) \end{array}\right\} \\ delete(X,Y) & \mapsto \left\{\begin{array}{l} delete(num, nil + list(num), nil), \\ delete(num, nil + list(num), nil + list(num)) \end{array}\right\} \\ safe(X,Y) & \mapsto \{safe(nil + list(num)), safe(nil)\} \\ noattack(X,Y,Z) & \mapsto \left\{\begin{array}{l} noattack(num, nil, num), \\ noattack(num, nil + list(num), num) \end{array}\right\} \end{cases}$$

As we have already pointed out, the verification method based on condition (2.3) is *compositional*. Therefore, in order to perform the I/O and call correctness verification, we apply the predicate *verifyIOcall* to the clause to be verified and to the pre-post program specifications (both given as lists of type atoms). In this way, if the predicate *verifyIOcall* returns *false* we can have a hint on the clause that may be wrong.

In the following, for the sake of readability, we have chosen to skip the specification arguments in the calls to the tool (except for the first). We can now prove that the `queens` program is correct w.r.t. the I/O and call correctness conditions.

```
| ?- verifyIOcall(  (queens(X,Y) :- perm(X,Y), safe(Y)),
    [queens(nil+list(num),U), queens(nil,U),
```

[7]Available at URL: http://www.cs.bgu.ac.il/~mcodish/Software/aci-types-poly.tgz.

```
c1: queens(X,Y) :- perm(X,Y), safe(Y).

c2: perm([],[]).
c3: perm([X|Y],[V|Res]) :- delete(V,[X|Y],Rest), perm(Rest,Res).

c4: delete(X,[X|Y],Y).
c5: delete(X,[F|T],[F|R]) :- delete(X,T,R).

c6: safe([]).
c7: safe([X|Y]) :- noattack(X,Y,1), safe(Y).

c8: noattack(X,[],N).
c9: noattack(X,[F|T],N) :- X =\= F, X =\= F + N, F =\= X + N,
N1 is N + 1, noattack(X,T,N1).
```

Figure 5: The queens program

```
    perm(nil+list(num),U), perm(nil,U),
    delete(T,nil+list(num),U), delete(T,nil,U),
    safe(nil+list(num)), safe(nil),
    noattack(num,nil,num), noattack(num,nil+list(num),num)],
      [queens(nil+list(num),nil+list(num)), queens(nil,nil),
      perm(nil+list(num),nil+list(num)), perm(nil,nil),
      delete(num,nil+list(num),nil+list(num)),
      delete(num,nil+list(num),nil),
      safe(nil+list(num)), safe(nil),
      noattack(num,nil,num), noattack(num,nil+list(num),num)]).
Clause is OK.

|?- verifyIOcall((perm([],[])), [...], [...]).
Clause is OK.
| ?- verifyIOcall((delete(X,[X|Y],Y)), [...], [...]).
Clause is OK.
| ?- verifyIOcall((delete(X,[F|T],[F|R]) :- delete(X,T,R)),
[...], [...]).
Clause is OK.
| ?- verifyIOcall((safe([X|Y]) :- noattack(X,Y,1), safe(Y)),
[...], [...]).
Clause is OK.
| ?- verifyIOcall(  (noattack(X,[F|T],N) :- X =\= F, X =\= F+N,
  F =\= X+N, N1 is N+1, noattack(X,T,N1)), [...], [...]).
Clause is OK.
```

Note that if we change the order of the atoms in the body of clause c1 we obtain the clause

```
c1':  queens(X,Y) :- safe(Y), perm(X,Y)
```

which can no longer be proved correct w.r.t. the considered specification. Indeed, now Y in the call $safe(Y)$ is not assured to be a list of numbers. The tool detects that there is something potentially wrong

```
| ?- verifyIOcall((queens(X,Y):-safe(Y),perm(X,Y)), [...],[...]).
Clause may be wrong because call safe(U) (atom number 1 of body)
is not in the call-specification.
```

6 Type verification in constraint logic languages

Another example of the transformation methodology of an analyzer is [4], where the resulting tool is employed to diagnose CHIP programs w.r.t. type information. Types are over-approximations of program semantics. This is the *descriptive* approach to types. The abstract domain is a space of types, described by a restricted class of *CLP* programs called *regular unary constraint logic* (RULC) programs [17]. This class is a generalization to constraint logic programming of regular unary programs (used by [19, 18]). Thus, the specification S is a RULC program and the abstract immediate consequence operator is the one operating on RULC programs of [17]. If the type generated for the head of a clause c by $\mathcal{T}^{\alpha}_{\{c\}}(S)$ is not a subtype of the corresponding predicate in S, then the clause is responsible of generating wrong type information. S is given "by need" by querying the user about the correctness of the actual semantics (which is computed by the analyzer) and (in case is needed) about the intended types.

A prototype [8] of the type diagnoser has been implemented as an extension of the type analyzer of CHIP programs of [17]. The actual implementation of both the diagnoser and the analyzer provides a more user-friendly syntax for types. Types are acquired and presented using the *regular term grammar* formalism (see e.g. [15]).

Consider the *wrong* CHIP version of the `queens` program of Figure 6 on page 16. The call `safe(X,T,K1)` in the recursive definition of `safe` has been replaced by the wrong call `safe(T,Y,K1)`. The `:-entry` declaration says that the predicate `nqueens/2` should be called with an *integer* term as first argument and *any* term as second argument.

The interaction with the call-success diagnoser is as in the following.

```
Do you like Call-Type constraint_queens(list(anyfd))?  YES

Do you like Call-Type safe(list(anyfd),list(anyfd),int)?   NO
What should it be?  anyfd, list(anyfd), int.

Do you like Succ-Type safe(list(anyfd),[],int)?  NO
What should it be?  anyfd, list(anyfd), int.

Do you like Succ-type constraint_queens( ([]|[anyfd]) )? NO
```

[8]Available at URL: http://www.ida.liu.se/~pawpi/Diagnoser/diagnoser.html.

```
:-entry nqueens(int,any).

nqueens(N,List):- length(List,N), List::1..N,
                  constraint_queens(List),
                  labeling(List,0,most_constrained,indomain).

constraint_queens([X|Y]):- safe(X,Y,1), constraint_queens(Y).
constraint_queens([]).

safe(X,[Y|T],K):- noattack(X,Y,K), K1 is K+1, safe(T,Y,K1).
safe(_,[],_).

noattack(X,Y,K):- X #\= Y, Y #\= X+K, X #\= Y+K.
```

Figure 6: The CHIP queens program

```
what should it be?  list(anyfd).

Do you like Succ-Type noattack(anyfd,anyfd,int)?  YES

Diagnoser WARNING: Clause
safe(X, [Y|T], K) :- noattack(X, Y, K), K1 is K + 1, safe(T, Y, K1).
suspiciuos because of atom safe(T, Y, K1).

Do you like Call-Type noattack(list(anyfd),anyfd,int)?  NO
What should it be?  anyfd, anyfd, int.

Do you like Succ-Type nqueens(nnegint, ([]|[anyfd]) )?  NO
What should it be?  int, list(int).

end of diagnosis, no (more) warnings.
```

Thus we are warned about the (only) incorrect clause of the program.

7 Conclusions

Based on the theory of abstract verification, as proposed in [6, 5], we have shown how it is possible and "easy" to transform static analyzers (with suitable properties) into automatic verifiers. In this paper we have presented three different verification tools based on different type domains for functional, logic and *CLP* programming. However, our abstract verification approach is very general and flexible. Existing static analyzers can be transformed into verification tools dealing with different abstract domains for different programming paradigms, provided the analyzers are defined as construction of abstract fixpoint semantics.

References

[1] F. Bourdoncle. Abstract Debugging of Higher-Order Imperative Languages. In *Programming Languages Design and Implementation '93*, pages 46–55, 1993.

[2] 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 M. Kamkar, editor, *Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169. Linköping University, 1997.

[3] M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-unification. *Journal of Theoretical Computer Science*, 238:131–159, 2000.

[4] M. Comini, W. Drabent, and P. Pietrzak. Diagnosis of CHIP Programs Using Type Information. In M. C. Meo and M. Vilares Ferro, editors, *Appia-Gulp-Prode'99, Joint Conference on Declarative Programming*, pages 337–349, 1999.

[5] M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract Interpretation based Verification of Logic Programs. Submitted for publication.

[6] M. Comini, 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*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000. Available at URL: http://www.elsevier.nl/locate/entcs/volume30.html.

[7] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

[8] P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. In S. Brookes and M. Mislove, editors, *Proceedings of the 13th International Symposium on Mathematical Foundations of Programming Semantics MFPS'97*, volume 6 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1997. Available at URL: http://www.elsevier.nl/locate/entcs/volume6.html.

[9] P. Cousot. Types as abstract interpretations (Invited Paper). In *Conference Record of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 316–331. ACM Press, 1997.

[10] P. Cousot 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*, pages 238–252, 1977.

[11] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of Sixth ACM Symp. Principles of Programming Languages*, pages 269–282, 1979.

[12] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–549, 1992.

[13] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992.

[14] P. Cousot and R. Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *Proceedings of Nineteenth Annual ACM Symp. on Principles of Programming Languages*, pages 83–94. ACM Press, 1992.

[15] P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, Cambridge, Mass., 1992.

[16] W. Drabent and J. Maluszynski. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

[17] W. Drabent and P. Pietrzak. Type Analysis for CHIP. In *proceedings of Types for Constraint Logic Programming, post-conference workshop of JICSLP'98*, 1998.

[18] J. Gallagher and D. A. de Waal. Regular Approximations of Logic Programs and Their Uses. Technical Report CSTR-92-06, Department of Computer Science, University of Bristol, 1992.

[19] J. Gallagher and D. A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Cambridge, Mass., 1994. MIT Press.

[20] R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: A domain perspective. In M. Johnson, editor, *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[21] G. Levi 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*, volume 1490 of *Lecture Notes in Computer Science*, pages 102–117. Springer-Verlag, 1998.

[22] D. Park. Fixpoint Induction and Proofs of Program Properties. *Machine Intelligence*, 5:59–78, 1969.

[23] D. Rémy and J. Vouillon. Objective ML:An effective object-oriented extension to ML. *Theory and Practice of Object-Systems*, 4(1):27–50, 1998.

[24] P. Volpe. *Derivation of proof methods for logic programs by abstract interpretation.* PhD thesis, Dipartimento di Matematica, Università di Napoli Federico II, 1999.