Properties of a type abstract interpreter

Roberta Gori and Giorgio Levi

Dipartimento di Informatica, Università di Pisa, Pisa, Italy
{levi,gori}@di.unipi.it

Abstract In a previous paper [7], we have developed a type abstract interpreter which was shown to be more precise then the classical ML type inference algorithm in inferring monomorphic types, represented as Herbrand terms with variables à la Hindley. In order to deal with recursive functions, we introduce a new abstract fixpoint operator which generalizes the one used in the Hindley and ML inference algorithms by performing k fixpoint computation steps (as done in [11] in the case of polymorphic types). Our abstract interpreter has many interesting properties. It is possible to reconstruct the ML result by just one fixpoint computation step ($k = 1$) and to show that for every $k \geq 1$, either we reach the least fixpoint (which is in general more precise than the ML result), or we get exactly the same result as ML. One important result is that our type interpreter turns out to correspond to a type system, which lies between monomorphism and polymorphic recursion.

1 Introduction

In a recent paper [7], we have developed a type abstract interpreter, following the methodology in [2], that is more precise than the classical ML type inference algorithm in inferring monomorphic types, represented as Herbrand terms with variables à la Hindley. The difference w.r.t. the Hindley (and ML) algorithm is related to recursive functions. We use an operator similar to the one used in the above algorithms after k fixpoint computation steps (as done in [11] in the case of polymorphic types). In this paper, we formally study the properties of our type interpreter and we show some interesting results. Namely,

- If we perform just one fixpoint computation step ($k = 1$), we get exactly the ML result.
- For every $k \geq 1$, either we reach the least fixpoint (which is in general more precise than the ML result), or we get exactly the same result as ML. Hence, unlike what is claimed in [10,11] we get an improvement in precision only if we reach the least fixpoint in k steps.
- Our type interpreter turns out to correspond to a type system, which lies between monomorphism and polymorphic recursion. The type inference algorithm is much simpler than the one for polymorphic recursion (we do not need quantification in type terms and do not perform type generalization). Yet it succeeds in typing all the programs which are used to show the power of polymorphism.

In Sections 2.1 and 2.2 we summarize known results on monomorphic type systems (and type inference algorithms) and on abstract interpretation. In Section 3 we formalize the abstract interpreter in [7] and we prove its properties for the family of widenings. In Section 4 we compare its behavior with the ML algorithm on some examples. Finally, in Section 5, we prove that our abstract interpreter realizes a form of recursion which lies between monomorphic and polymorphic recursion.

For the sake of conciseness we have omitted the proofs. However all the lemmas and intermediate results are stated in order to help the reader to follow our reasoning.

2 Preliminaries

2.1 Monomorphic Types

The language we consider is the following version of untyped eager λ-calculus.

$$x, f, \ldots \in \mathbb{X}: \text{ program variables}$$
$$e, e_1, \ldots \in \mathbb{E}: \text{ program expressions}$$
$$e ::= id\ x \mid \lambda x.e \mid e_1 + e_2 \mid if\ e_1\ then\ e_2\ else\ e_3 \mid int\ n \mid e_1(e_2) \mid \mu f.\lambda x.e$$

$\lambda x.e$ denotes λ-abstraction and $e_1(e_2)$ denotes function application. In $\mu f.\lambda x.e$ the function f with formal parameter x is defined recursively. The guard in the conditional $if\ e_1\ then\ e_2\ else\ e_3$ is the test for zero.

In the Church/Curry monotype semantics the type of an expression e is a set of typings $< H, m >$ stating that the standard evaluation of e in an environment, where the global variables x have type $H(x)$ given by the type environment H, returns a value of type m. In the Church/Curry monotype system the type value m is a monotype, i.e.,

$$m, m_1, \ldots \in \mathbb{M}^C \qquad m ::= int \mid m_1 \to m_2$$

The rule-based form of the Church/Curry monotype semantics allows us to *check* that a given type m is indeed the right type for the expression e, by considering $H \in \mathbb{H}^C$, where $\mathbb{H}^C$ is the domain of functions: $\mathbb{X} \to \mathbb{M}^C$.

$$H \overset{C}{\vdash} int\ n \Rightarrow int \qquad\qquad H \overset{C}{\vdash} id\ x \Rightarrow H(x)$$

$$\frac{H \overset{C}{\vdash} e_1 \Rightarrow int \quad H \overset{C}{\vdash} e_2 \Rightarrow int}{H \overset{C}{\vdash} e_1 + e_2 \Rightarrow int} \qquad \frac{H \overset{C}{\vdash} e_1 \Rightarrow int \quad H \overset{C}{\vdash} e_2 \Rightarrow m \quad H \overset{C}{\vdash} e_3 \Rightarrow m}{H \overset{C}{\vdash} if\ e_1\ then\ e_2\ else\ e_3 \Rightarrow m}$$

$$\frac{H \overset{C}{\vdash} e_1 \Rightarrow m_1 \to m_2 \quad H \overset{C}{\vdash} m_1}{H \overset{C}{\vdash} e_1(e_2) \Rightarrow m_2} \qquad \frac{H[x \leftarrow m_1] \overset{C}{\vdash} e \Rightarrow m_2}{H \overset{C}{\vdash} \lambda x.e \Rightarrow m_1 \to m_2}$$

$$\frac{H[f \leftarrow m] \overset{C}{\vdash} \lambda x.e \Rightarrow m}{H \overset{C}{\vdash} \mu f.\lambda x.e \Rightarrow m}$$

The above type rules can be used to check whether a given type is indeed the type of an expression e. However it is not possible to use directly the above rules to *infer* the right type of an expression. Hindley's algorithm [8] allows us to find the *principal* typing for the Church/Curry monotype system. A principal typing is a typing that subsumes all the others [6,9]. More generally this means that there is an exact representation for all possible typings. Hindley's algorithm uses a richer type domain, the domain of Herbrand terms, i.e., monotypes with variables. Hindley types are as follows.

$$\begin{aligned} a, \ldots \in \mathbb{V} &: \text{type variables} \\ \tau, \tau_1, \ldots \in \mathbb{M}_v^H &: \text{monotype with variables} \\ \tau ::= & \, int \mid a \mid \tau_1 \to \tau_2 \end{aligned}$$

In this case the type environment domain $\mathbb{H}^H$ is the set of functions: $\mathbb{X} \to \mathbb{M}_v^H$.

In [2] Hindley's inference algorithm is reconstructed as an *exact* abstraction of the Church/Curry monotype semantics. This gives rise to an abstract semantics, which "implements" Hindley's inference algorithm as an abstract function $\mathcal{T}^H[\![.]\!]$, which, given a type environment $H \in \mathbb{H}^H$ returns a monotype with variables τ.

It is worth noting that the ML type inference algorithm is based on the Damas-Milner inference algorithm [6] which coincides with Hindley's inference algorithm on the language fragment (without let-polymorphism) we are considering.

The principal type computed by Hindley's inference algorithm is related to the Church/Curry monotype system by the following properties.

Theorem 1. *[2] Assume that* $< H, \tau > \in \mathcal{T}^H[\![e]\!]$. *Then, for all grounding substitutions* $\vartheta : \mathbb{V} \to \mathbb{M}^C$, $\vartheta(H) \overset{C}{\vdash} e \Rightarrow \vartheta(\tau)$.

Theorem 2. *[2] For all* m *such that* $H \overset{C}{\vdash} e \Rightarrow m$, *there exists a suitable* $H' \in \mathbb{H}^H$ *and* τ *such that* $< H', \tau > \in \mathcal{T}^H[\![e]\!]$ *and* τ *is the least common generalization of such* m.

2.2 Abstract Interpretation

Abstract interpretation [3,4] is a general theory for approximating the semantics of discrete dynamic systems, originally developed by Patrick and Radhia Cousot, in the late 70's. The *abstract semantics* is an approximation of the concrete one, where exact (concrete) values are replaced by (abstract) properties, modeled by an abstract domain.

In abstract interpretation based static program analysis we compute an abstract fixpoint semantics. 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 (concrete) semantics of the program P. The class of properties we want to consider is formalized as an abstract domain $(\mathbb{A}, \leq)$, related to $(\mathbb{C}, \sqsubseteq)$ by a Galois connection

$\alpha : \mathbb{C} \rightarrow \mathbb{A}$ and $\gamma : \mathbb{A} \rightarrow \mathbb{C}$ (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. The abstract semantics $\mathrm{lfp}_{\mathbb{A}}(\mathcal{T}_P^\alpha)$ models a safe approximation of the property of interest: if the property is verified in $\mathrm{lfp}_{\mathbb{A}}(\mathcal{T}_P^\alpha)$ it will also be verified in $\mathrm{lfp}_{\mathbb{C}}(\mathcal{T}_P)$. An analysis method based on the computation of the abstract semantics $\mathrm{lfp}_{\mathbb{A}}(\mathcal{T}_P^\alpha)$ is effective only if the least fixpoint is reached in finitely many iterations, i.e., if the abstract domain is Noetherian. If this is not the case, *widening operators* can be used to ensure the termination. Widening operators [5] give an upper approximation of the least fixpoint and guarantee termination by introducing further approximation.

3 Our type inference algorithm and some of its properties

We have followed the general approach of [2] in using abstract interpretation theory to derive abstract type interpreters on suitable type domains. In our experiment (described in [7]) the type domain was still $\mathbb{M}_v^H$. Following the presentation of the ML type inference algorithm in [1], we have implemented the Herbrand type domain, by using (equality) constraints on type variables. Our abstract interpreter was obtained as an abstraction of the concrete denotational semantics for the untyped λ-calculus we have presented, extended with mutual recursion. We will use mutual recursion in the examples of Section 4 only, while the operational abstract semantics of this section (and the related properties) do not handle mutual recursion, for the sake of simplicity in the presentation.

The abstract semantics of an expression e in a given type environment H produces the type τ of the expression e together with some constraint γ on the type variables in H. The intuition is that the constraint γ is how H should be instantiated in order for e to have the type τ. We define $\mathbb{M}_v^N = \mathbb{M}_v^H \times (V \rightarrow \mathbb{M}_v^N)$.

$$
\begin{aligned}
a, \ldots \in & \mathbb{V} : \text{type variables} \\
(\tau, \gamma), (\tau_1, \gamma_1), \ldots \in & \mathbb{M}_v^N \\
\tau, \tau_1, \ldots \in & \mathbb{M}_v^H : \text{monotype with variables} \\
\tau ::= & \mathit{int} \mid a \mid \tau_1 \rightarrow \tau_2 \\
\gamma, \gamma_1 \in & \mathbb{V} \rightarrow \mathbb{M}_v^H : \text{idempotent substitutions such that} \\
& \gamma(\tau) = \tau.
\end{aligned}
$$

In this case the type environment domain $\mathbb{H}^N$ is the set of functions: $\mathbb{X} \rightarrow \mathbb{M}_v^N$.

An abstract environment $H \in \mathbb{H}^N$ maps identifiers to abstract values. Substitutions are always restricted to the set of type variables occurring in the environment. The abstract partial order relation is defined in terms of the relations on terms and substitutions. Namely, $(\tau_1, \gamma_1) \leq (\tau_2, \gamma_2)$ if $\tau_1 \leq \tau_2$ and $\gamma_1 \leq \gamma_2$, where $\tau_1 \leq \tau_2$ if $\exists \vartheta$ such that $\vartheta(\tau_1) = \tau_2$. As usual, elements of the abstract domain are equivalence classes w.r.t. the corresponding equivalence relation. The *lub* operation can easily be defined by means of the unification algorithm. The bottom element $\perp$ is a pair consisting of a type variable and the empty substitution. The equivalence class corresponding to the top element contains all the pairs, whose first component is the distinguished term *Notype*. Note that the abstract domain is non-Noetherian since there exist infinite ascending chains.

In the following we present the rules of an abstract type interpreter on the domain $\mathbb{M}_v^N$. The inference rules are obtained systematically by abstracting the concrete semantics on the $\mathbb{M}_v^N$ type domain. We use the operators *apply* and *unify*, where $apply(\gamma, \tau) = \gamma(\tau)$ and $unify(a = b, \gamma_1, \gamma_2)$ computes the solved form of the set of equations $\{a = b, x_i^1 = t_i^1, x_i^2 = t_i^2\}$ with $x_i^1/t_i^1 \in \gamma_1$ and $x_i^2/t_i^2 \in \gamma_2$.

$$H \vdash int\ n \Rightarrow (int, \epsilon) \tag{1}$$

$$H \vdash id\ x \Rightarrow H(x) \tag{2}$$

$$\frac{\begin{array}{c} H \vdash e_1 \Rightarrow (\tau_1, \gamma_1) \quad H \vdash e_2 \Rightarrow (\tau_2, \gamma_2) \\ \gamma = unify(\{\tau_1 = int, \tau_2 = int, \gamma_1, \gamma_2\}) \end{array}}{H \vdash e_1 + e_2 \Rightarrow (apply(\gamma, \tau_1), \gamma)} \tag{3}$$

$$\frac{\begin{array}{c} H \vdash e_1 \Rightarrow (\tau_1, \gamma_1) \quad H \vdash e_2 \Rightarrow (\tau_2, \gamma_2) \quad H \vdash e_3 \Rightarrow (\tau_3, \gamma_3) \\ \gamma = unify(\{\tau_1 = int, \tau_2 = \tau_3, \gamma_1, \gamma_2, \gamma_3\}) \end{array}}{H \vdash if\ e_1\ then\ e_2\ else\ e_3 \Rightarrow (apply(\gamma, \tau_2), \gamma)} \tag{4}$$

$$\frac{\begin{array}{c} H \vdash e_1 \Rightarrow (\tau_1, \gamma_1) \quad H \vdash e_2 \Rightarrow (\tau_2, \gamma_2) \\ \gamma = unify(\{\tau_1 = f_1 \rightarrow f_2, \tau_2 = f_1, \gamma_1, \gamma_2\}) \end{array}}{H \vdash e_1 e_2 \Rightarrow (apply(\gamma, f_2), \gamma)} \tag{5}$$

$$\frac{H[x \leftarrow (f_1, \epsilon)] \vdash e \Rightarrow (\tau, \gamma) \quad \tau_1 = apply(\gamma, f_1)}{H \vdash \lambda x.e \Rightarrow ((\tau_1 \rightarrow \tau), \gamma)} \tag{6}$$

We have presented the rules for all the syntactic constructs, except recursion. If we abstract systematically the concrete semantics, the abstract semantics of recursion should be an abstract fixpoint computation. In our case we obtain the following rules.

$$\frac{\begin{array}{c} H \vdash \mu f.\lambda x.e \Rightarrow_{T_P}^{n-1} (\tau_1, \gamma_1) \quad H \vdash \mu f.\lambda x.e \Rightarrow_{T_P}^{n} (\tau_2, \gamma_2) \\ (\tau_1, \gamma_1) = (\tau_2, \gamma_2) \end{array}}{H \vdash \mu f.\lambda x.e \Rightarrow (\tau_2, \gamma_2)} \tag{7}$$

$$H \vdash (\mu f.\lambda x.e) \Rightarrow^{0}_{T_P} (x, \epsilon) \tag{8}$$

$$\frac{H \vdash (\mu f.\lambda x.e) \Rightarrow^{n-1}_{T_P} (\tau_1, \gamma_1) \quad H \vdash (\mu f.\lambda x.e(\tau_1, \gamma_1)) \Rightarrow_{T_P} (\tau_2, \gamma_2)}{H \vdash (\mu f.\lambda x.e) \Rightarrow^{n}_{T_P} (\tau_2, \gamma_2)} \tag{9}$$

$$\frac{H[f \leftarrow (\tau, \gamma)] \vdash e \Rightarrow (\tau_1, \gamma_1)}{H \vdash (\mu f.\lambda x.e, (\tau, \gamma)) \Rightarrow_{T_P} (\tau_1, \gamma_1)} \tag{10}$$

However, our abstract domain is not Noetherian. This implies that we can have infinite ascending chains. As a consequence, we are not guaranteed to be able to find a solution to Rule 7 in a finite number of steps.

Therefore we introduce a family of widening operators. These are obtained by generalizing the operator introduced by Hindley's inference algorithm [8]. We obtain a family of widening operators simply by replacing Rule 7 (which requires the computation of the fixpoint for $\Rightarrow_{T_P}$), by the following rules.

Definition 1. *Given a k.*

$$\frac{H \vdash \mu f.\lambda x.e(\tau, \gamma) \Rightarrow^{k}_{wid} (\tau_1, \gamma_1)}{H \vdash \mu f.\lambda x.e \Rightarrow (\tau_1, \gamma_1)} \tag{11}$$

$$\frac{\begin{array}{c} H \vdash \mu f.\lambda x.e \Rightarrow^{k-1}_{T_P} (\tau_1, \gamma_1) \quad H \vdash (\mu f.\lambda x.e, (\tau_1, \gamma_1)) \Rightarrow_{T_P} (\tau_2, \gamma_2) \\ (\tau_1, \gamma_1) = (\tau_2, \gamma_2) \end{array}}{H \vdash \mu f.\lambda x.e \Rightarrow^{k}_{wid} (\tau_1, \gamma_1)} \tag{12}$$

$$\frac{\begin{array}{c} H \vdash \mu f.\lambda x.e \Rightarrow^{k-1}_{T_P} (\tau_1, \gamma_1) \quad H \vdash (\mu f.\lambda x.e, (\tau_1, \gamma_1)) \Rightarrow_{T_P} (\tau_2, \gamma_2) \\ (\tau_1, \gamma_1) \neq (\tau_2, \gamma_2) \quad \vartheta = unify(\tau_1 \gamma_2 = \tau_2, \gamma_2) \end{array}}{H \vdash \mu f.\lambda x.e \Rightarrow^{k}_{wid} (apply(\vartheta, \tau_1), \vartheta)} \tag{13}$$

This gives us a family of widenings obtained simply by choosing a different value for k.

Consider now the Hindley type inference algorithm. We can prove that the Hindley inference algorithm can be obtained simply by choosing the widening corresponding to $k = 1$.

First we need some definitions which allow us to relate our type environments $H \in \mathbb{H}^N$ to the environment in $H \in \mathbb{H}^H$.

Definition 2. *Let $H \in \mathbb{H}^N$. Then $\tilde{H} \in \mathbb{H}^H$ is defined as follows:*

$$\tilde{H}(x) = \begin{cases} \tau & \textit{if } H(x) = (\tau, \gamma) \\ \textit{Undefined} & \textit{otherwise} \end{cases}$$

We now define the application of a constraint γ to an environment.

Definition 3. *Let $H \in \mathbb{H}^H$. Then $\gamma(H) \in \mathbb{H}^H$ is defined as follows:*

$$\gamma(H)(x) = \begin{cases} \gamma(\tau) & \text{if } H(x) = \tau \\ \text{Undefined otherwise} \end{cases}$$

We first prove that, if we infer (τ, γ) for the expression e, by using the widening operator with $k = 1$, then we obtain the type τ for the expression e in the environment $\gamma(H)$.

Theorem 3. *Assume that $H \vdash e \Rightarrow (\tau, \gamma)$ using Rules 11-13 with $k = 1$. Then $< \gamma(\tilde{H}), \tau > \in \mathcal{T}^H[\![e]\!]$.*

We now prove that also the opposite relation holds. We first define the most general environment $\mathcal{H} \in \mathbb{H}^N$. $\mathcal{H}$ is such that $\forall x$, $\mathcal{H}(x) = (v, \epsilon)$ where v is a variable and $\forall x, y,\ x \neq y : \mathcal{H}(x) \neq \mathcal{H}(y)$.

Theorem 4. *Assume that $< H, \tau > \in \mathcal{T}^H[\![e]\!]$ then $\mathcal{H} \vdash e \Rightarrow (\tau, \gamma)$, using Rules 11-13 with $k = 1$ and $\gamma(\mathcal{H}) = H$.*

This allows us to conclude that the Hindley inference algorithm is equivalent to our abstract semantics with the widening operator corresponding to $k = 1$.

We establish now several interesting properties of our abstract semantics.

We first formally prove that Rules 11-13 are indeed a family of widening operators, that is a correct approximation of the abstract fixpoint computation.

First we establish some elementary properties of the abstract semantics we have defined. The first properties state that starting from an environment $H \in \mathbb{N}^H$ which maps variables to abstract elements, our semantics assign to each expression an element which still belongs to the abstract domain we have defined.

Lemma 1. *If H is such that for all x, $H(x) = (\tau, \gamma)$ with $apply(\gamma, \tau) = \tau$. Then for all (τ, γ) such that $H \vdash e \Rightarrow (\tau, \gamma)$ or $H \vdash e \Rightarrow^n_{T_P} (\tau, \gamma)$ then $apply(\gamma, \tau) = \tau$.*

The next result is useful to prove Corollaries 1 and 2: the first one states that $\Rightarrow_{T_P}$ is monotone, while the latter guarantees that the computation of $\Rightarrow^n_{T_P}$, for all n leads to an ascending chain of abstract values.

Lemma 2. *For all e, if $H[f \leftarrow (\tau_0, \gamma_0)] \vdash e \Rightarrow (\tilde{\tau}_1, \tilde{\gamma}_1)$ and $H[f \leftarrow (\tau_1, \gamma_1)] \vdash e \Rightarrow (\tilde{\tau}_2, \tilde{\gamma}_2)$ with $(\tau_0, \gamma_0) \leq (\tau_1, \gamma_1)$, then $(\tilde{\tau}_1, \tilde{\gamma}_1) \leq (\tilde{\tau}_2, \tilde{\gamma}_2)$.*

Corollary 1. *Assume $(\tau_0, \gamma_0) \leq (\tau_1, \gamma_1)$. Let $(\mu f.\lambda x.e, (\tau_0, \gamma_0)) \vdash e \Rightarrow_{T_P} (\tau_2, \gamma_2)$ and $(\mu f.\lambda x.e, (\tau_1, \gamma_1)) \vdash e \Rightarrow_{T_P} (\tau_3, \gamma_3)$, then $(\tau_2, \gamma_2) \leq (\tau_3, \gamma_3)$.*

Corollary 2. *If $(\mu f.\lambda x.e, (\tau, \gamma)) \Rightarrow^{n-1}_{T_P} (\tau_1, \gamma_1)$ and $(\mu f.\lambda x.e, (\tau, \gamma)) \Rightarrow^n_{T_P} (\tau_2, \gamma_2)$, then $\tau_1 \leq \tau_2$ and $\gamma_1 \leq \gamma_2$.*

By Tarski's theorem we know that the solution of Rule 7 consists in computing the least fixpoint of $\Rightarrow_{T_P}$, i.e., the least n such that $H \vdash \mu f.\lambda x.e \Rightarrow^{k-1}_{T_P} (\tau_1, \gamma_1)$ $H \vdash \mu f.\lambda x.e \Rightarrow^k_{T_P} (\tau_2, \gamma_2)$ and $(\tau_1, \gamma_1) = (\tau_2, \gamma_2)$.

The next step is to formally prove that for each k, $\Rightarrow^k_{wid}$, computes a correct approximation of the least fixpoint of $\Rightarrow_{T_P}$.

We first need the following intermediate result.

Lemma 3. *Assume e contains f, then $H[f \leftarrow (\tau,\gamma)] \vdash e \Rightarrow (\tau_1,\gamma_1)$, then for all ϑ, $H[f \leftarrow (\tau\gamma_1\vartheta, unify(\gamma,\gamma_1\vartheta))] \vdash e \Rightarrow (\tau_2,\gamma_2)$, with $\tau_2 = \tau_1\vartheta$, $\gamma_2 = \gamma_1\vartheta$.*

We now prove that any abstract element (τ_k^w,γ_k^w) computed by $\Rightarrow_{wid}^k$ is indeed a fixpoint of $\Rightarrow_{T_P}$.

Corollary 3. *If $H \vdash \mu f.\lambda x.e \Rightarrow_{wid}^k (\tau_1,\gamma_1)$ then $H \vdash (\mu f.\lambda x.e,(\tau_1,\gamma_1)) \Rightarrow_{T_P} (\tau_1,\gamma_1)$.*

Since the solution of Rule 7 consists in computing the least fixpoint of $\Rightarrow_{T_P}$, we can state the following.

Corollary 4. *Assume that $H \vdash \mu f.\lambda x.e \Rightarrow (\tau_1,\gamma_1)$ and $H \vdash \mu f.\lambda x.e \Rightarrow_{wid}^k (\tau_2,\gamma_2)$. Then $\tau_1 \leq \tau_2$ and $\gamma_1 \leq \gamma_2$.*

The previous result shows that Rule13 is indeed a widening operator.

Now if we consider two widenings, different in the number of iterations (with different values of k), both terminating using Rule 13, we might wonder which is the relation among the computed abstract values. In principle, the most precise widening (the one with more iterations) might lead to a more precise result, unless we get to the fixpoint. This was claimed in [10,11] in the case of a polymorphic type system. On the contrary, the next theorem states that the result does not depend on the number of iterations.

Theorem 5. *For all k, v, if $H \vdash \mu f.\lambda x.e \Rightarrow_{wid}^k (\tau_1^w,\gamma_1^w)$ then $H \vdash \mu f.e \Rightarrow_{wid}^v (\tau_1^w,\gamma_1^w)$.*

Theorem 5 together with Theorems 3 and 4, allows us to conclude that Hindley's inference algorithm is equivalent to our abstract semantics with the widening operator for any possible k, unless we succeed in reaching the least fixpoint.

4 Examples

In this Section we show some examples, most of which were presented in [7] as well. The theoretical results shown in the last Section allow one to better understand the experimental results. All the examples use the ML syntax for programs and type expressions.

Example 1. Consider the function (taken from [2])

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

The expression cannot be typed by the ML algorithm (the underlined expression is the one to which the type error message applies, in the OCAML implementation). The expression defines the function $f\ f_1\ g\ n\ x = g(f_1^n(x))$ which has the type $(`a \rightarrow `a) \rightarrow (`a \rightarrow `b) \rightarrow int \rightarrow `a \rightarrow `b$. This type is correctly computed (by a fixpoint computation) in [2], by using a more complex type system (a polytype system à la Church-Curry). We infer the correct type, simply by using the widening corresponding to $k = 3$, because we reach the least fixpoint.

We show the sequence of abstract values (approximations for f) computed in the example.

1. step 0:
   $\tau_0 =$ `'a1`
   $\gamma_0 = \epsilon$
2. step 1:
   $\tau_1 =$ `'a5->('a4->'a2)->int->'a4->'a2`,
   $\gamma_1 =$ `'a1` $\leftarrow$ `'a5->('a3->(('a3->'a4)->'a2))->int->'a4->('a5->'a2)`$\}$
3. step 2:
   $\tau_2 =$ `('a7->'a7)->('a7->'a6)->int->'a7->'a6`,
   $\gamma_2 = \{$`'a2`$\leftarrow$ `('a7->'a7)->'a6`, `'a4`$\leftarrow$`'a7`$\}$
4. step 3:
   $\tau_3 =$ `('a->'a) ->('a->'b)->int->'a->'b`,
   $\gamma_3 = \{$`'a6`$\leftarrow$ `('a->'a)->'b`, `'a7`$\leftarrow$`'a`$\}$

Note that at step 3 we reach the fixpoint since $(\tau_2, \gamma_2) = (\tau_3, \gamma_3)$ (both γ_2 and γ_3 are empty when restricted to the global environment). The sequence is obviously increasing. Looking at the sequence, we can also check the result we would obtain using a less precise widening, in particular, the ML widening ($k = 1$). In this case, we would use the Rule 13 after step 1. The computation of the $lub((apply(\gamma_1, \tau), \epsilon), (\tau_1, \gamma_1))$ would lead to the computation of the solved form of the equation

```
'a5->('a4->'a2)->int->'a4->'a2 =
'a5->('a3->(('a3->'a4)->'a2))->int->'a4->('a5->'a2).
```

The unification algorithm would generate the unsolvable equation

```
'a2=('a3->'a3)->'a2,
```

which explains the ML type error message. The result of the *lub* operation is therefore the top abstract value, i.e., the expression cannot be typed. When the widening terminates by using Rule 13 we always compute an upper approximation of the least fixpoint, i.e., a less precise type.

The next example, taken from [10], shows that ML does sometimes type a recursive function with a type which is not precise enough.

Example 2. `# let rec f x = (function x -> function y -> x) 0 (f 0);;`
`val f : int -> int = <fun>`

In the concrete semantics, f can be applied to a value of any type (not necessarily int), without causing any run time type error. If we choose $k = 2$, we reach the least fixpoint, and compute a more precise type

```
val f : 'a -> int = <fun>
```

The same type is inferred in [10], by using polymorphic recursion.

The next example, taken from [11], shows a problem similar to the one of Example 2, in an expression with mutual recursion.

Example 3.
```
# let rec p x = if q x = q 1 then p x else p x
    and q x = p 1;;
val p : int -> 'a = <fun>
val q : int -> 'a = <fun>
```

If we choose $k = 3$, we reach the least fixpoint, and compute more precise types

```
val p : 'a -> 'b = <fun>
val q : 'a -> 'b = <fun>
```

The same types are inferred in [11], by using polymorphic recursion.

5 The type system corresponding to our abstract interpreter

As we have already discussed the main difference between Hindley's monomorphic type inference algorithm and our abstract type interpreter lies in the way we deal with recursive definition. The examples presented in Section 4 show that our abstract type interpreter is more precise. Roughly speaking in a monomorphic type system each function application has to have the same type. Such a type is exactly the type of the recursive function. This is the reason why in the Hindley type inference algorithm each instantiation of the type of a recursive function produced by a function application has a direct effect in " guessing" the type of the recursive function: the instantiation produced by a function application has to produce the same instantiation on the type of the recursive function itself.

In a polymorphic type system each function application can have any type which is an instance of the recursive function type. This is the reason why in a polymorphic type inference algorithm the instantiation of the type of a recursive function in a function application does not produce the same instantiation on the type of the recursive function itself. In a polymorphic type system different function applications of a recursive function can lead to different (even *incompatible*) instantiations of the recursive function type.

Our type inference algorithm realizes a form of recursion which lies between monomorphism and polymorphic recursion. In the type system corresponding to our abstract type interpreter each function application can have any type which is an instance of the recursive function type as long as all these different instance are *compatible*. In this type system the different function applications of a recursive function can lead to different (but *compatible*) instantiations of the recursive function type. Intuitively this is due to the way we realize the computation of our abstract fixpoint. We collect all the constraints produced by the different instantiations due to different function applications in each computation step of the abstract fixpoint operator. This is the reason why such instantiations have to be *compatible* in order to produce a satisfiable constraint. On the other hand we never apply this constraint to the type of the recursive function we are trying to determine by fixpoint computation. As in a polymorphic type system, the instantiation of the type of a recursive function in a function application does not produce the same instantiation on the type of the recursive function itself. However, in our weak form of polymorphism such instantiations are checked to be *compatible*.

We now want to formally prove what we have just discussed. We first define a relation on environments in $\mathbb{H}^H$. $H, H' \in \mathbb{H}^H$ then $H \leq H'$ if $\forall x$, $H(x) \leq H'(x)$.

The next result guarantees us that for all the types checked by the Church/Curry monotype system there exists a principal type computed by our abstract type interpreter by using Rule 7. This result is a consequence of Theorem 2, 4 and Corollary 4.

Theorem 6. *Assume that* $H \overset{C}{\vdash} e \Rightarrow m$. *Then, there exists* (τ, γ) *such that and* $\mathcal{H} \vdash e \Rightarrow (\tau, \gamma)$ *with* $H \geq \gamma(\tilde{\mathcal{H}})$ *and* $m \geq \tau$.

On the other hand, as already shown in Section 4, there are types which can be inferred by our abstract type interpreter which can not be checked by the Church/Curry monotype system. Consider, for example, the function of Example 1.

```
# 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;;
```

This expression cannot be typed by the ML algorithm. However, our abstract type interpreter is able to infer the type $(`a \to `a) \to (`a \to `b) \to int \to `a \to `b$ for f.

The Damas-Milner-Mycroft polytype semantics associates a type to an expression e starting from a type environment which may contain universally quantified types. The quantified type $\forall a.\tau$ is used to allow one to consider any instance of τ obtained by renaming and instantiating the variable a whenever needed. This mechanism allows one to realize the recursive polymorphism since the type of a recursive function is inserted in the environment after having been generalized (i.e., all the free type variables are universally quantified). When a function application is reached, any instance obtained by a renaming of τ can be used. This allows the type τ of a recursive function to be independent from the type used in its applications.

In [2] the Damas-Milner-Mycroft polytype semantics and the corresponding type system are reconstructed on a suitable abstract domain. Let $ftv(t)$ be the set of free variables of t. Parametric/free variables (like a in $\forall b.b \to a$) are different from generic/bound type variables (like b in $\forall b.b \to a$):

$$a, \ldots \in \mathbb{V}_p : \text{parametric/free type variables}$$
$$b, \ldots \in \mathbb{V}_g : \text{generic/bound type variables}$$
$$a, b \in \mathbb{V} = \mathbb{V}_p \cup \mathbb{V}_g : \text{type variables } \mathbb{V}_p \cap \mathbb{V}_g = \emptyset$$
$$\pi \in \mathbb{P}_g^{DMM}, \quad \pi ::= \tau \mid \forall b_1 \ldots b_n.\tau \text{ where } \{b_1 \ldots b_n\} = ftv(\tau)$$

As already mentioned, our abstract domain is much simpler, i.e., it is the standard monomorphic types with variables domain without quantification. As a consequence, we do not have type generalization.

In this case the parametric type environment domain $\mathbb{H}_g^{DMM}$ is the set of functions: $\mathbb{X} \to \mathbb{P}_g^{DMM}$. The rule-based form of the Damas-Milner-Mycroft semantics allows us to *check* that a given type $\tau \in \mathbb{M}_v^H$ is indeed the right type for the expression e, by considering a given $H \in \mathbb{H}_g^{DMM}$. In this case we write $H \overset{DMM}{\vdash} e \Rightarrow \tau$.

First we need an operator which allows us to relate our type environments $H \in \mathbb{H}^N$ to the environment in $H \in \mathbb{H}_g^{DMM}$.

Definition 4. *Let $H \in \mathbb{H}^N$. Then $op(\gamma, H) \in \mathbb{H}_g^{DMM}$ is defined as follows:*

$$op(\gamma, H)(x) = \begin{cases} \gamma(\tau) & \text{if } H(x) = (\tau, \gamma') \text{ and } x \text{ is not a recursive program variable} \\ gen_H(\tau) & \text{if } H(x) = (\tau, \gamma') \text{ and } x \text{ a is recursive program variable} \\ Undefined & otherwise \end{cases}$$

where $gen_H(\tau)$ is a function which (universally) quantifies the free variable of τ w.r.t.the environment H.

The next result shows that any type that can be inferred with our abstract type interpreter can indeed be checked by the Damas-Milner-Mycroft polytype system.

Theorem 7. *Assume that $H \vdash e \Rightarrow (\tau, \gamma)$. Then $op(\gamma, \tilde{H}) \overset{DMM}{\vdash} e \Rightarrow \tau$.*

As already discussed at the beginning of this section polymorphic recursion is more powerful than the kind of polymorphic recursion we are able to capture with our abstract interpreter. Intuitively, the Damas-Milner-Mycroft polytype system could check an expression where the recursive function is called with two incomparable instances, for example, $int \to int$ and $bool \to int$. It is worth noting, however, that this case seems not to arise in meaningful programs. All the examples in [2,10,11,12], introduced as a motivation for the introduction of polymorphic recursion, can indeed be typed by our abstract interpreter.

6 Conclusion

We have developed an abstract interpreter on the Herbrand type domain. In this paper we show that the well known Hindley inference algorithm can be reconstructed by means of a particular widening operator on the fixpoint computation required by our abstract interpreter. We prove that all the widenings obtained as generalization of this particular operator give the same results unless the fixpoint is reached. Moreover, the type inference algorithm obtained by performing the fixpoint computation defines a new type system that we have shown to realize a form of recursion which lies between monomorphic and polymorphic recursion.

It is worth noting that our abstract domain is much simpler, i.e., it is the standard monomorphic types with variables domain without quantification. This is the reason why we do not need the type generalization. However, our abstract interpreter is able to assign the "right" type to all the examples introduced in [2,10,11,12] as a motivation for the introduction of polymorphic recursion.

One may wonder which is the type system corresponding to the inference algorithm defined by our abstract interpreter. We are currently working on its formal definition. It is worth noting that, as in the case of polymorphic recursion, not all the instances of the inferred type will be checked by the type system. This is true only for those instances which are comparable with all the function applications. In our case, since all the function applications are required to be comparable, the set of all the instances of the inferred type which can be actually checked will be finitely representable.

References

1. G. Cousineau and M. Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998.
2. P. Cousot. Types as abstract interpretations. In *Conference Record of the 24th ACM Symp. on Principles of Programming Languages* , pages 316–331. ACM Press, 1997.
3. 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.
4. 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.
5. 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.
6. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.
7. R. Gori and G. Levi. An experiment in type inference and verification by abstract interpretation. In A. Cortesi, editor, *Proc. of the VMCAI'02 workshop on Abstract Interpretaion and Model Checking*, volume 2294 of *LNCS*, pages 225–239, 2002.
8. J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions American Mathematical Society*, 146:29–60, 1969.
9. J. Mitchell. Type systems for programming languages. volume B of *Handbook of Theoretical Computer Science*, pages 365–458. Elsevier, 1990.
10. B. Monsuez. Polymorphic typing by abstract interpretation. In R. Shyamasundar, editor, *Proceedings of Foundation of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1992.

11. B. Monsuez. Polymorphic types and widening operators. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proceedings of Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 224–281. Springer-Verlag, 1993.
12. A. Mycroft. Polymorphic type schemes and recursive definitions. In G. Goos and J. Hartmanis, editors, *Proceedings of the International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.