On Polymorphic Recursion, Type Systems, and Abstract Interpretation

Marco Comini[1], Ferruccio Damiani[2], Samuel Vrech[1]

[1] Dipartimento di Matematica e Informatica, Università di Udine
Via delle Scienze, 206; I-33100 Udine, Italy
[2] Dipartimento di Informatica, Università di Torino
Corso Svizzera, 185; I-10149 Torino, Italy

Abstract. The problem of typing *polymorphic recursion* (i.e., recursive function definitions $\mathsf{rec}\,\{x = e\}$ where different occurrences of x in e are used with different types) has been investigated both by people working on *type systems* and by people working on *abstract interpretation*.
Recently, Gori and Levi have developed an abstract interpreter that is able to type all the ML typable recursive definitions and interesting examples of polymorphic recursion. The problem of finding a type system corresponding to their abstract interpreter was open.
In this paper we present a type system corresponding to the Gori-Levi abstract interpreter. Interestingly enough, the type system is derived from the system of simple types (which is the let-free fragment of the ML type system) by adapting a general technique for extending a decidable type system enjoying *principal typings* by adding a decidable rule for typing recursive definitions. The key role played in our investigation by the notion of principal typing suggests that this notion might be useful in other investigations about the relations between type systems and type inference algorithms synthesized by abstract interpretation.
Keywords. Principal Typing, Type Inference Algorithm

1 Introduction

The *Hindley-Milner system* (a.k.a. the *ML type system*) [2], which is the core of the type systems of functional programming languages like SML, OCaml, and Haskell, is only able to infer types for *monomorphic recursion* (i.e., recursive function definitions $\mathsf{rec}\,\{x = e\}$ where all the occurrences of x in e are used with exactly the same type inferred for e). The problem of inferring types for *polymorphic recursion* (i.e., recursive function definitions $\mathsf{rec}\,\{x = e\}$ where different occurrences of x in e are used with different types that specialize the type inferred for e) [11,14] has been studied both by people working on type systems [7,10] and by people working on abstract interpretation [12,13].

Building on results by Cousot [1], Gori and Levi [5,6] have developed a type abstract interpreter that is able to type all the ML typable recursive definitions and interesting examples of polymorphic recursion. In particular, as explained in [5,6], the type abstract interpreter is able to assign the expected type to all the

examples used in [14,1,12,13] to motivate polymorphic recursion. As pointed out in [5,6], the problem of finding a type system corresponding to the type abstract interpreter was open. Such a type system would lye between the *Curry-Hindley system* (a.k.a. the *system of simple types*) [8], which is the let-free fragment of the ML type system, and the let-free fragment of the *Milner-Mycroft system* [14].

In this paper we present a type system corresponding to the Gori-Levi abstract interpreter. Interestingly enough, the type system is derived from the Curry-Hindley system by adapting a general technique (developed in [4]) for extending a type system enjoying decidable typabilty and *principal typings* [9,17] by adding a decidable rule for typing rec-expressions. The key role played in our investigation by the notion of principal typing suggests that this notion might be useful in other investigations about the relations between *type systems* [16] and *type inference algorithms synthesized by abstract interpretation* [1] and, more in general, between program analyses specified via type systems and program analyses specified via abstract interpretation.

Organization of the Paper. Section 2 introduces a core functional programming language (which can be considered the kernel of languages like SML, OCaml, and Haskell) together with other basic definitions that will be used in the rest of the paper. Section 3 applies the technique of [4] to the Curry-Hindley type system obtaining a system equivalent to the let-free fragment of the Milner-Mycroft type system. Section 4 briefly illustrates the Gori-Levi type abstract interpreter [5,6] and Section 5 presents a new type system that corresponds to this abstract interpreter.

2 Preliminary Definitions

2.1 A Small ML-like Language

Expressions (ranged over by e) are defined by the pseudo-grammar:

$$e ::= x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{rec}\,\{x = e\},$$

where x ranges over program variables (in $\mathbf{P_V}$) and c ranges over constants.

Free and *bound* occurrences of a variable in an expression are defined as usual. The (finite) set of the free variables of an expression e is denoted by $\mathrm{FV}(e)$.

The set of constants includes the booleans (ranged over by $\flat$), the integer numbers (ranged over by ι), the constructors for pairs (pair) and lists (nil and cons), some logical and arithmetic operators, and the functions for decomposing pairs (fst and snd) and lists (null, hd and tl).

We have omitted conditionals from the syntax of expressions since, for typing purposes, the expression "if e_0 then e_1 else e_2" can be considered as syntactic sugar for the application "ifc e_0 e_1 e_2", where ifc is a constant of suitable type.

2.2 Types, Environments, Typings, and Principal Typings

The set of *simple types* ($\mathbf{T}_0$), ranged over by u, is defined by the pseudo-grammar:

$$u ::= \alpha \mid u_1 \to u_2 \mid \mathsf{bool} \mid \mathsf{int} \mid u_1 \times u_2 \mid u\,\mathsf{list}$$

We have *type variables* (ranged over by α), arrow types, and a selection of *ground types* and *parametric data-types*. The ground types are bool (the type of booleans) and int (the type of integers). The other types are pair types and list types.

The constructor $\rightarrow$ is right associative, e.g., $u_1 \rightarrow u_2 \rightarrow u_3$ means $u_1 \rightarrow (u_2 \rightarrow u_3)$, and the constructors $\times$ and list bind more tightly than $\rightarrow$, e.g., $u_1 \rightarrow u_2 \times u_3$ means $u_1 \rightarrow (u_2 \times u_3)$.

We assume a countable set $\mathbf{T_V}$ of type variables. A *substitution* $\mathbf{s}$ is a function from type variables to simple types which is the identity on all but a finite number of type variables. Substitutions will be denoted by $[\alpha_1 \leftarrow u_1, \ldots, \alpha_n \leftarrow u_n]$ $(n \geq 0)$; the empty substitution will be denoted by $[\,]$. The application of a substitution $\mathbf{s}$ to a simple type u, denoted by $\mathbf{s}(u)$, is defined as usual. The *composition* of two substitutions $\mathbf{s}_1$ and $\mathbf{s}_2$ is the substitution, denoted by $\mathbf{s}_1 \cdot \mathbf{s}_2$, such that $\mathbf{s}_1 \cdot \mathbf{s}_2(\alpha) \stackrel{\text{def}}{=} \mathbf{s}_1(\mathbf{s}_2(\alpha))$, for all type variables α. We say that $\mathbf{s}$ is *more general* than $\mathbf{s}'$, written $\mathbf{s} \leq \mathbf{s}'$, if there is a substitution $\mathbf{s}''$ such that $\mathbf{s}' = \mathbf{s}'' \cdot \mathbf{s}$.

An *environment* E is a set $\{x_1 : \rho_1, \ldots, x_n : \rho_n\}$ of assumptions for program variables such that every variable x_i $(1 \leq i \leq n)$ can occur at most once in E. The expression $Dom(E)$ denotes the *domain* of E, which is the set $\{x_1, \ldots, x_n\}$. Given a set of program variables X, the expression $E|_X$ denotes the *restriction of* E *to* X, which is the environment $\{x : \rho \in E \mid x \in X\}$. Given two environments E_1 and E_2, we write $E_1 \oplus E_2$ to denote the environment $E_1 \cup E_2$ under the assumption that $x : \rho_1 \in E_1$ and $x : \rho_2 \in E_2$ imply $\rho_1 = \rho_2$. We write $E, x : \rho$ as short for $E \cup \{x : \rho\}$ under the assumption that $x \notin Dom(E)$. The application of a substitution $\mathbf{s}$ to an environment E, denoted by $\mathbf{s}(E)$, is defined as usual.

Definition 1 (Simple type environments). A *simple type environment* U is an environment $\{x_1 : u_1, \ldots, x_n : u_n\}$ of simple type assumptions for variables.

According to Wells [17], in a given type system $\vdash$, "a *typing* t for a typable term e is the collection of all the information other than e which appears in the final judgement of a proof derivation showing that e is typable". In this paper we are interested in typings of the shape $\langle U;\, u\rangle$, where U is a simple type environment and u is a simple type. The following definitions are fairly standard (note that the relation $\leq_{\mathbf{spc}}$ is reflexive and transitive).

Definition 2 (Typing specialization relation $\leq_{\mathbf{spc}}$). A typing $\langle U;\, u\rangle$ *can be specialized to* $\langle U';\, u'\rangle$ (notation $\langle U;\, u\rangle \leq_{\mathbf{spc}} \langle U';\, u'\rangle$) if $\mathbf{s}(U) = U'$ and $\mathbf{s}(u) = u'$, for some substitution $\mathbf{s}$. We will write $\langle U;\, u\rangle =_{\mathbf{spc}} \langle U';\, u'\rangle$ to mean that both $\langle U;\, u\rangle \leq_{\mathbf{spc}} \langle U';\, u'\rangle$ and $\langle U';\, u'\rangle \leq_{\mathbf{spc}} \langle U;\, u\rangle$ hold.

Definition 3 (Principal typings). Let $\vdash$ be a type system with judgements of the shape $\vdash e : t$. A typing t is *principal for a term* e if $\vdash e : t$, and if $\vdash e : t'$ implies $t \leq_{\mathbf{spc}} t'$. We say that system $\vdash$ has the *principal typing property* to mean that every typable term has a principal typing.

$$\text{(Spc)}\ \frac{\vdash e : t}{\vdash e : t'}\ \text{where } t \leq_{\mathbf{spc}} t' \qquad \text{(Con)}\ \vdash c : \langle \emptyset;\, u \rangle\ \text{where } u = \mathbf{type}(c) \qquad \text{(Var)}\ \vdash x : \langle \{x : u\};\, u \rangle\ \text{where } u \in \mathbf{T}_0$$

$$\text{(App)}\ \frac{\vdash e_1 : \langle U_1;\, u_0 \to u \rangle \qquad \vdash e_2 : \langle U_2;\, u_0 \rangle}{\vdash e_1\, e_2 : \langle U_1 \oplus U_2;\, u \rangle}$$

$$\text{(Abs)}\ \frac{\vdash e : \langle U, x : u_0;\, u \rangle}{\vdash \lambda x.e : \langle U;\, u_0 \to u \rangle} \qquad \text{(AbsVac)}\ \frac{\vdash e : \langle U;\, u \rangle}{\vdash \lambda x.e : \langle U;\, u_0 \to u \rangle}\ \text{where } x \notin \mathrm{FV}(e) \text{ and } u_0 \in \mathbf{T}_0$$

Figure 1. Typing rules for the rec-free fragment of the language (system $\vdash_0$)

c	$\mathbf{type}(c)$	c	$\mathbf{type}(c)$	c	$\mathbf{type}(c)$
$\flat$	bool	not	bool $\to$ bool	fst	$\alpha_1 \times \alpha_2 \to \alpha_1$
ι	int	and, or	bool $\times$ bool $\to$ bool	snd	$\alpha_1 \times \alpha_2 \to \alpha_2$
pair	$\alpha_1 \to \alpha_2 \to (\alpha_1 \times \alpha_2)$	$+, -, *$	int $\times$ int $\to$ int	null	α list $\to$ bool
nil	α list	$=, <$	int $\times$ int $\to$ bool	hd	α list $\to \alpha$
cons	$\alpha \to \alpha$ list $\to \alpha$ list	ifc	bool $\to \alpha \to \alpha \to \alpha$	tl	α list $\to \alpha$ list

Figure 2. Types for constants

3 Derivation of a Type System for Recursive Definitions

In this section we derive a type system for recursive definitions by applying the technique of [4] to the Curry-Hindley system [8].

3.1 System $\vdash_0$: Typing the rec-free Fragment of the Language

The first step of the technique prescribes to take a type system that satisfies the following requirements (parameterized over the actual shape of the typing and of the typing specialization relation).

- It has typing judgements of the shape $\vdash e : t$, where the typing t contains assumptions for exactly the variables in $\mathrm{FV}(e)$.
- It has the principal typing property (that is the property described by Definition 3 where $\leq_{\mathbf{spc}}$ denotes the suitable typing specialization relation).
- It is decidable to establish whether a pair t_1 can be specialized to a pair t_2 (i.e., whether $t_1 \leq_{\mathbf{spc}} t_2$ holds).
- There is an algorithm that for every term e decides whether e is typable and, if so, returns a principal typing for e.

System $\vdash_0$ in Figure 1, which is just a is a reformulation of the system of simple types [8], satisfies the above requirements.

Rule (Spc), which is the only non-structural rule, allows to specialize (in the sense of Definition 2) the typing inferred for an expression. The rule for typing constants, (Con), uses the function **type** (tabulated in Figure 2) which specifies a type for each constant. Note that, by rule (Spc), it is possible to assign to a constant c all the specializations of the typing $\langle \emptyset;\, \mathbf{type}(c) \rangle$.

Since $\vdash_0 e : \langle U; u\rangle$ implies $Dom(U) = \mathrm{FV}(e)$, we have two rules for typing an abstraction $\lambda x.e$, (Abs) and (AbsVac), corresponding to the two cases $x \in \mathrm{FV}(e)$ and $x \notin \mathrm{FV}(e)$.

3.2 System $\vdash_0^{\mathrm{P}}$: Typing Polymorphic Recursive Definitions

The second step of the technique prescribes to extend system $\vdash_0$ with a typing rule that allows to assign to $\mathsf{rec}\,\{x = e\}$ any typing t that can be assigned to e by assuming the typing t itself for x. This requires to introduce the notion of *typing environment*.

Definition 4 (Typing environments). A *typing environment* D is an environment $\{x_1 : t_1, \ldots, x_n : t_n\}$ of typing assumptions for variables such that $Dom(D) \cap VR(D) = \emptyset$, where $VR(D) \stackrel{\text{def}}{=} \cup_{x:\langle U;\, u\rangle \in D} Dom(U)$ is the *set of variables occurring in the range of* D. Every typing t occurring in D is implicitly universally quantified over all type variables occurring in t.[3]

The typing rules of system $\vdash_0^{\mathrm{P}}$ (where "P" stands for "polymorphic") are given in Figure 3. The judgement $D \vdash_0^{\mathrm{P}} e : \langle U; u\rangle$ means "e is $\vdash_0^{\mathrm{P}}$-typable in D with typing $\langle U; u\rangle$", where

- D is a typing environment specifying typing assumptions for variables that may or may not occur free in e, and
- $\langle U; u\rangle$ is the typing inferred for e, where U is simple type environment containing the type assumptions for the free variables of e which are not in $Dom(D)$, and u is a simple type.

Let D be a typing environment, "$x \notin D$" is short for "$x \notin Dom(D) \cup VR(D)$" and $\mathrm{FV}_D(e) \stackrel{\text{def}}{=} (\mathrm{FV}(e) - Dom(D)) \cup VR(D|_{\mathrm{FV}(e)})$ is the *set of the free variables of the expression* e *in* D. In any valid judgement $D \vdash_0^{\mathrm{P}} e : \langle U; u\rangle$ it holds that $Dom(D) \cap Dom(U) = \emptyset$ and $Dom(U) = \mathrm{FV}_D(e)$.

Rules (Spc), (Con), (Var), (Abs), (AbsVac), and (App) are just the rules of system $\vdash_0$ (in Figure 1) modified by adding the typing environment D on the left of the typing judgements and, when necessary, side conditions (like "$x \notin Dom(D)$" in rule (Var)) to ensure that $Dom(D) \cap Dom(U) = \emptyset$.

Rule (Rec-P) allows to assign to a recursive definition $\mathsf{rec}\,\{x = e\}$ any typing t that can be assigned to e by assuming the typing t for x. Note that the combined use of rules (Var-P) and (Spc) allows to assign different specializations $t_i = \langle U_i; u_i\rangle$ $(1 \le i \le n)$ of $t = \langle U; u\rangle$ to different occurrences of x in e, provided that $\oplus_{1\le i\le n} U_i$ is defined.

The following theorem (taken from [4]) shows that system $\vdash_0^{\mathrm{P}}$ has the same expressive power of Milner-Mycroft system [14]. This implies that typability in system $\vdash_0^{\mathrm{P}}$ is undecidable (as is in the Milner-Mycroft system [7,10]).

Theorem 5 ([4]). *Let e be a closed expression. Then $\emptyset \vdash_0^{\mathrm{P}} e : \langle \emptyset; u\rangle$ if and only if $\emptyset \vdash e : u$ is Milner-Mycroft derivable.*

[3] To emphasize this fact we might have used assumption of the shape $\forall\vec{\alpha}.t$ where $\vec{\alpha}$ is the sequence of all type variables occurring in t.

$$\text{(Spc)}\ \frac{D \vdash e : t}{D \vdash e : t'}\ \text{where } t \leq_{\mathbf{spc}} t' \qquad \text{(Con)}\ D \vdash c : \langle \emptyset;\, u \rangle\ \text{where } u = \mathbf{type}(c) \qquad \text{(Var)}\ D \vdash x : \langle \{x : u\};\, u \rangle\ \text{where } u \in \mathbf{T}_0 \text{ and } x \notin Dom(D)$$

$$\text{(Abs)}\ \frac{D \vdash e : \langle U, x : u_0;\, u \rangle}{D \vdash \lambda x.e : \langle U;\, u_0 \to u \rangle}\ \text{where } x \notin D \qquad \text{(AbsVac)}\ \frac{D \vdash e : \langle U;\, u \rangle}{D \vdash \lambda x.e : \langle U;\, u_0 \to u \rangle}\ \text{where } x \notin \mathrm{FV}(e),\ u_0 \in \mathbf{T}_0, \text{ and } x \notin D$$

$$\text{(App)}\ \frac{D \vdash e_1 : \langle U_1;\, u_0 \to u \rangle \qquad D \vdash e_2 : \langle U_2;\, u_0 \rangle}{D \vdash e_1\, e_2 : \langle U_1 \oplus U_2;\, u \rangle}$$

$$\text{(Rec-P)}\ \frac{D, x : \langle U;\, u \rangle \vdash e : \langle U;\, u \rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle U;\, u \rangle}\ \text{where } Dom(U) = \mathrm{FV}_D(\mathsf{rec}\,\{x = e\}) \text{ and } x \notin D \qquad \text{(Var-P)}\ D,\, x : t \vdash x : t$$

Figure 3. Typing rules of system $\vdash_0^{\mathrm{P}}$

3.3 Systems $\vdash_{\mathbf{0}}^{\boldsymbol{k}}$ ($k \geq 1$): a Family of Decidable Restrictions of $\vdash_{\mathbf{0}}^{\mathbf{P}}$

The third step of the technique prescribes to introduce a family of decidable restrictions of rule (Rec-P). This requires to introduce the notion of *principal-in-D typing*, which adapts the notion of principal typing (see Definition 3) to deal with the typing environment D.

Definition 6 (Principal-in-D typings). Let $\vdash$ be a system with judgements of the shape $D \vdash e : t$. A typing t is *principal-in-D for a term e* if $D \vdash e : t$, and if $D \vdash e : t'$ implies $t \leq_{\mathbf{spc}} t'$. We say that system $\vdash$ has the *principal-in-D typing property* to mean that every typable term has a principal-in-D typing.

For every finite set of variables $X = \{x_1, \ldots, x_n\}$ ($n \geq 0$) let

- $\mathsf{T}_X \stackrel{\text{def}}{=} \{\langle U;\, u \rangle \mid \langle U;\, u \rangle \text{ is typing such that } Dom(U) = X\}$, and
- $\mathsf{B}_X \stackrel{\text{def}}{=} \{\langle \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\};\, \alpha \rangle \mid \text{the type variables } \alpha_1, \ldots, \alpha_n, \alpha \text{ are all distinct}\} \subseteq \mathsf{T}_X$.

The typing specialization relation $\leq_{\mathbf{spc}}$ (see Definition 2) is a preorder over T_X and, for all typings $b \in \mathsf{B}_X$ and $t \in \mathsf{T}_X$, $b \leq_{\mathbf{spc}} t$. For every subset S_X of T_X,

$$\mathbf{Min}_{\leq_{\mathbf{spc}}}(\mathsf{S}_X) \stackrel{\text{def}}{=} \{t \in \mathsf{S}_X \mid t \leq_{\mathbf{spc}} t' \text{ for all } t' \in \mathsf{S}_X\}$$

is the (possibly empty) *set of the $\leq_{\mathbf{spc}}$-minimum elements of S_X*.

The following proposition holds (the proof is straightforward, by Definition 6).

Proposition 7. *Let $\vdash$ be a system with judgements of the shape $D \vdash e : t$. A typing t is a principal-in-D typing for a term e if and only if $t \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t' \mid D \vdash e : t'\})$.*

For every $k \geq 1$, let $\vdash_0^k$ be the system obtained from $\vdash_0^{\mathrm{P}}$ by replacing rule (Rec-P) with the rule:

$$\text{(Rec-}k\text{)}\ \frac{D, x : t_0 \vdash e : t_1 \quad \cdots \quad D, x : t_{k-1} \vdash e : t_k}{D \vdash \mathsf{rec}\,\{x = e\} : t_k}$$

where $x \notin D$,

$$t_0 \in \mathsf{B}_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e\})},$$
$$(\forall i \in \{1,\ldots,k\})\; t_i \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D, x : t_{i-1} \vdash e : t\}), \tag{3.1}$$
$$t_{k-1} = t_k$$

(note that $D \vdash_0^k e : t$ implies $D \vdash_0^{k+1} e : t$).

According to Proposition 7, in rule (Rec-k), the requirement (3.1) is equivalent to

(for all $i \in \{1,\ldots,k\}$) t_i is a principal-in-$(D, x : t_{i-1})$ typing for e.

Therefore, as pointed out in [4], the checking of a purported $\vdash_0^k$ derivation requires the ability to decide whether a typing is principal-in-D. Note that requirement (3.1) is crucial. In fact, removing it would make rule (Rec-k) equivalent to rule (Rec-P) for all $k \geq 2$.

For all $k \geq 1$, system $\vdash_0^k$ has the principal-in-D typing property and $\vdash_0^k$-typability is decidable — see the explanations before Theorem 10.

An Inference Algorithm for $\vdash_0^k$ ($k \geq 1$) An *unification problem* is a set of equalities between simple types. A solution to an unification problem P (*unifier*) is a substitution $\mathbf{s}$ such that $\mathbf{s}(u_1) = \mathbf{s}(u_2)$ for all $(u_1 = u_2) \in P$. A *Most General Unifier* is a solution minimal w.r.t. $\leq$ (and thus all Most General Unifiers are equivalent up to renaming of type variables). We will write $\mathbf{MGU}(P)$ for the set of all the most general unifiers for P and $\mathbf{mgu}(P)$ for any element of $\mathbf{MGU}(P)$.

The inference algorithm makes use of an algorithm for checking whether the $\leq_{\mathbf{spc}}$ relation (see Definition 2) holds and of the standard algorithm for finding a most general solution to an unification problem. Note that the first algorithm is a particular case of the latter.

The inference algorithm is presented by defining (for all $k \geq 1$) a function $\mathbf{PT}_0^k$ which, for every expression e and environment D, returns a set of typings $\mathbf{PT}_0^k(D, e)$ such that

$$\mathbf{PT}_0^k(D, e) = \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D \vdash_0^k e : t\}).$$

Definition 8 (Inductive characterization of the set of principal-in-D typings for e w.r.t. $\vdash_0^k$). For every expression e and environment D, the set $\mathbf{PT}_0^k(D, e)$ is defined by structural induction on e.

- If $e = x$, then
  - If $x : \langle U; u\rangle \in D$ and the substitution $\mathbf{s}$ is a fresh renaming of $\overrightarrow{\alpha} = \mathrm{FTV}(U) \cup \mathrm{FTV}(u)$, then $\langle \mathbf{s}(U); \mathbf{s}(u)\rangle \in \mathbf{PT}_0^k(D, x)$.
  - If $x \notin \mathit{Dom}(D)$ and α is a type variable, then $\langle \{x : \alpha\}; \alpha\rangle \in \mathbf{PT}_0^k(D, x)$.
- If $e = c$ and $\mathbf{type}(c) = u$, then $\langle \emptyset; u\rangle \in \mathbf{PT}_0^k(D, c)$.
- If $e = \lambda\, x.e_0$ and $\langle U; u_0\rangle \in \mathbf{PT}_0^k(D, e_0)$, then
  - If $x \notin \mathrm{FV}(e_0)$ and α is a fresh type variable, then $\langle U; \alpha \to u_0\rangle \in \mathbf{PT}_0^k(D, \lambda\, x.e_0)$.
  - If $x \in \mathrm{FV}(e_0)$ and $U = U'$, $x : u$, then $\langle U'; u \to u_0\rangle \in \mathbf{PT}_0^k(D, \lambda\, x.e_0)$.

– If $e = e_0\, e_1$ and $\langle U_0;\, u_0\rangle \in \mathbf{PT}_0^k(D, e_0)$, then
  - If $u_0 = \alpha$ (a type variable), α_1 and α_2 are fresh type variables, $\langle U_1;\, u_1\rangle \in \mathbf{PT}_0^k(D, e_1)$ is fresh, and $\mathbf{s} \in \mathbf{MGU}(\{u_1 = \alpha_1,\, \alpha = \alpha_1 \to \alpha_2\} \cup \{u' = u'' \mid x : u' \in U_0 \text{ and } x : u'' \in U_1\}$, then $\langle \mathbf{s}(U_0) \oplus \mathbf{s}(U_1);\, \mathbf{s}(\alpha_2)\rangle \in \mathbf{PT}_0^k(D, e_0\, e_1)$.
  - If $u_0 = u_2 \to u$, the pair $\langle U_1;\, u_1\rangle \in \mathbf{PT}_0^k(D, e_1)$ is fresh, and $\mathbf{s} \in \mathbf{MGU}(\{u_1 = u_2\} \cup \{u' = u'' \mid x : u' \in U_0 \text{ and } x : u'' \in U_1\})$, then $\langle \mathbf{s}(U_0) \oplus \mathbf{s}(U_1);\, \mathbf{s}(u)\rangle \in \mathbf{PT}_0^k(D, e_0\, e_1)$.
– If $e = \mathsf{rec}\,\{x = e_0\}$, then
  - If $h \in \{1, \ldots, k\}$, $t_0 \in \mathsf{B}_{\mathrm{FV}_D(e)}$, $t_1 \in \mathbf{PT}_0^k((D, x : t_0), e_0)$, …, $t_h \in \mathbf{PT}_0^k((D, x : t_{h-1}), e_0)$, $t_1 \not\leq_{\mathbf{spc}} t_0$, … $t_{h-2} \not\leq_{\mathbf{spc}} t_{h-1}$, and $t_h \leq_{\mathbf{spc}} t_{h-1}$, then $t_{h-1} \in \mathbf{PT}_0^k(D, e)$.

For every $k \geq 1$, expression e, and typing environment D, the set $\mathbf{PT}_0^k(D, e)$ is an equivalence class of typings modulo renaming of the type variables in a typing. The following proposition holds (the proof is straightforward, by induction on Definition 8)

Proposition 9. *For every $k \geq 1$, expression e and environment D, if $\langle U;\, u\rangle \in \mathbf{PT}_0^k(D, e)$, then*

1. *$Dom(U) = \mathrm{FV}_D(e)$, and*
2. *$\langle U';\, u'\rangle \in \mathbf{PT}_0^k(D, e)$ if and only if there is a bijection $\mathbf{s} : \mathbf{T_V} \to \mathbf{T_V}$ such that $\mathbf{s}(U) = U'$ and $\mathbf{s}(u) = u'$.*

Indeed Definition 8 specifies a sound, complete, and terminating inference algorithm: to perform type inference on an expression e w.r.t. the environment D simply try to build a typing by following the definition of $\mathbf{PT}_0^k(D, e)$, choosing fresh type variables and using the unification and $\leq_{\mathbf{spc}}$-checking algorithms as necessary.

Theorem 10 (Soundness and completeness of $\mathbf{PT}_0^k$ for $\vdash_0^k$). *For every $k \geq 1$, expression e, and environment D:*

– *If $t \in \mathbf{PT}_0^k(D, e)$, then $D \vdash_0^k e : t$.*
– *If $D \vdash_0^k e : t'$, then $t \leq_{\mathbf{spc}} t'$ for some $t \in \mathbf{PT}_0^k(D, e)$.*

Corollary 11. *For every $k \geq 1$, expression e, and environment D:* $\mathbf{PT}_0^k(D, e) = \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D \vdash_0^k e : t\})$.

Comparison with System $\vdash_{\mathbf{0}}^{\mathbf{P}}$ The relation between system $\vdash_0^k$ and system $\vdash_0^{\mathrm{P}}$ is stated by the following theorems. Roughly speaking, the former says that when rule (Rec-k) works at all, it works as well as rule (Rec-P) does, and the latter says that the family of systems $\vdash_0^k$ ($k \geq 1$) provides a complete stratification of $\vdash_0^{\mathrm{P}}$-typability.

Theorem 12. *For every $k \geq 1$:*

1. *If $D \vdash_0^k e : t$, then $D \vdash_0^{\mathrm{P}} e : t$.*

2. If e is $\vdash_0^k$-typable in D and $D \vdash_0^{\mathrm{P}} e : t$, then $D \vdash_0^k e : t$.

Theorem 13. *If $D \vdash_0^{\mathrm{P}} e : t$, then there exists $k \geq 1$ such that $D \vdash_0^k e : t$.*

Note that Theorem 10 (which implies that, for all $k \geq 1$, system $\vdash_0^k$ has the principal-in-D typing property), Theorem 13, and Theorem 12.2, imply that system $\vdash_0^{\mathrm{P}}$ has the principal-in-D typing property.

Comparison with the ML Type System For all $k \geq 1$, system $\vdash_0^k$ is able to type recursive definitions that are not ML-typable. The examples for $k = 1$ are not particularly interesting: the prototypical term is the always divergent function $\mathsf{rec}\,\{x = x\,x\}$, that has principal-in-$\emptyset$ typing $\langle \emptyset; \alpha \rangle$). Instead, with $k = 2$ it is already possible to type many interesting examples of polymorphic recursion. Consider for instance the OCaml program (taken from [15])

```
type 'a seq = EMPTY | SEQ of 'a * ('a * 'a) seq ;;

let rec size s = match s with
                   EMPTY     -> 0
                 | SEQ(x,ps) -> 1 + 2 * (size ps) ;;
```

where `'a seq` is a polymorphic sequence type (a sequence is either empty or made of an element paired with a sequence of pairs of elements) and `size` is a function that returns the number of elements contained in a sequence. Although OCaml allows the definition of the `'a seq` recursive data-type, the ML type system (and, therefore, OCaml) is not able to type the function `size`. Instead, for all $k \geq 2$, rule (Rec-k) is able to assign the principal-in-$\emptyset$ pair `<{},'a seq -> int>` to the function `size`.

The following example shows that, for all $k \geq 1$, system $\vdash_0^k$ is not able to type all the ML-typable recursive definitions.

Example 14. The ML-typable term $\mathsf{rec}\,\{f = \lambda\, g\, y.\mathsf{if\ false\ then}\ y\ \mathsf{else}\ g\,(f\,g\,y)\}$ is not $\vdash_0^2$-typable and $\vdash_0^3$-typable with principal-in-$\emptyset$ typing $\langle \emptyset; (\alpha \to \alpha) \to \alpha \to \alpha \rangle$. The ML-typable term $\mathsf{rec}\,\{f = \lambda\, g\, h_1\, y.\mathsf{if\ false\ then}\ y\ \mathsf{else}\ g\,(f\,g\,h_1\,(f\,h_1\,g\,y))\}$ is not $\vdash_0^3$-typable and $\vdash_0^4$-typable with principal-in-$\emptyset$ typing $\langle \emptyset; (\alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha \to \alpha \rangle$. In general, for all $m \geq 0$, the ML-typable term

$$\mathsf{rec}\,\{f = \lambda\, g\, h_1\, h_2 \cdots h_m\, y.\ \mathsf{if\ false\ then}\ y\ \mathsf{else}\ g\,(f\,g\,h_1\,h_2 \cdots h_m\,(f\,h_1\,g\,h_2 \cdots h_m\,(\cdots(f\,h_1 \cdots h_{m-1}\,g\,y)\cdots))\}$$

is not $\vdash_0^{m+2}$-typable and $\vdash_0^{m+3}$-typable with principal-in-$\emptyset$ typing

$$\langle \emptyset; \underbrace{(\alpha \to \alpha) \to (\alpha \to \alpha) \to \cdots \to (\alpha \to \alpha)}_{m+1} \to \alpha \to \alpha \rangle.$$

3.4 Systems $\vdash_0^{k,\mathbf{ML}}$ ($k \geq 1$): Recovering ML-typability

The fourth step of the technique allows to extend system $\vdash_0^k$ (for any given $k \geq 1$) to type not $\vdash_0^k$-typable expressions that can be typed by a given decidable restriction of $\vdash_0^{\mathrm{P}}$, while preserving decidable typability and principal-in-D typings.

We say that *a typing rule for recursive definitions* (Rec-?) *is* $\vdash_0^{\mathrm{P}}$*-suitable* to mean that the system $\vdash_0^?$ obtained from $\vdash_0^{\mathrm{P}}$ by replacing rule (Rec-P) with rule (Rec-?):

1. is a restriction of system $\vdash_0^{\mathrm{P}}$ (i.e., $D \vdash_0^? e : \langle U; u\rangle$ implies $D \vdash_0^{\mathrm{P}} e : \langle U; u\rangle$),
2. has the principal-in-D typing property, and
3. there is an algorithm that, given a typing environment D and a term e, returns a principal-in-D typing for e.

Theorems 10 and 12 guarantee that, for all $k \geq 1$, adding to system $\vdash_0^k$ a $\vdash_0^{\mathrm{P}}$-suitable rule (Rec-?) with an additional side condition ensuring that the rule can be applied only if rule (Rec-k) is not applicable, results in a system, denoted by $\vdash_0^{k,?}$, with both decidable typability and principal-in-D typing property. So, to extend system $\vdash_0^k$ to type all the ML typable recursive definitions, we have just to add to system $\vdash_0^k$ a $\vdash_0^{\mathrm{P}}$-suitable rule which (without the additional side condition) is at least as expressive as the ML rule for recursive definitions. The simplest way of doing this would be to add (a version, modified to fit into system $\vdash_0^k$, of) the ML rule itself:

$$\text{(Rec-ML)}\ \frac{D \vdash e : \langle U, x : u;\, u\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle U;\, u\rangle}\ \text{where } x \notin D$$

and to restrict it with the additional side condition:

> and there are no $t_0, t_1, \cdots, t_k$ such that:
> $t_0 \in \mathsf{B}_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e\})}$,
> (for all $i \in \{1, \ldots, k\}$) $t_i \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D, x : t_{i-1} \vdash e : t\})$, and
> $t_{k-1} = t_k$

which ensures that the rule can be applied only when rule (Rec-k) is not applicable. Let $\vdash_0^{k,\mathrm{ML}}$ denote the resulting system.

An Inference Algorithm for $\vdash_0^{k,\mathbf{ML}}$ ($k \geq 1$) A sound and complete inference algorithm for system $\vdash_0^{k,\mathrm{ML}}$ ($\mathbf{PT}_0^{k,\mathrm{ML}}$) can be obtained from the inference algorithm $\mathbf{PT}_0^k$, given in Definition 8, by adding the following sub-clause to the clause for rec-expression.

- Otherwise, if $\langle U, x : u;\, u'\rangle \in \mathbf{PT}_0^k(D, e_0)$, and $\mathbf{s} \in \mathbf{MGU}(\{u = u'\})$, then $\mathbf{s}(\langle U;\, u\rangle) \in \mathbf{PT}_0^k(D, e)$.

Theorem 15 (Soundness and completeness of $\mathbf{PT}_0^{k,\mathbf{ML}}$ w.r.t. $\vdash_0^{k,\mathbf{ML}}$). *For every $k \geq 1$, expression e, and environment D:*
(Soundness). *If $t \in \mathbf{PT}_0^{k,\mathrm{ML}}(D, e)$, then $D \vdash_0^{k,\mathrm{ML}} e : t$.*
(Completeness). *If $D \vdash_0^{k,\mathrm{ML}} e : t'$, then $t \leq_{\mathbf{spc}} t'$ for some $t \in \mathbf{PT}_0^{k,\mathrm{ML}}(D, e)$.*

(2) $H \rhd x \Rightarrow H(x)$ where $x \in \mathbf{P_V}$

(5) $$\frac{H \rhd e_1 \Rightarrow (u_1, \mathbf{s}_1) \qquad H \rhd e_2 \Rightarrow (u_2, \mathbf{s}_2) \qquad \mathbf{s} = \mathbf{mgu}\left(\{u_1 = f_1 \to f_2, u_2 = f_1\} \cup \mathbf{eqs}(\mathbf{s}_1) \cup \mathbf{eqs}(\mathbf{s}_2)\right)}{H \rhd e_1 e_2 \Rightarrow (\mathbf{s}(u_2), \mathbf{s})}$$

(6) $$\frac{H[x \leftarrow (f_1, \epsilon)] \rhd e \Rightarrow (u, \mathbf{s}) \qquad u_1 = \mathbf{s}(f_1)}{H \rhd (\lambda x.e) \Rightarrow ((u_1 \to u), \mathbf{s})}$$

(7) $$\frac{H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{n}_{T_P} (u_n, \mathbf{s}_n) \qquad H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{n+1}_{T_P} (u_{n+1}, \mathbf{s}_{n+1}) \qquad (u_n, \mathbf{s}_n) = (u_{n+1}, \mathbf{s}_{n+1})}{H \rhd \mathsf{rec}\{x = e\} \Rightarrow (u_n, \mathbf{s}_n)}$$

(8) $H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{0}_{T_P} (\alpha, \epsilon)$ with $\alpha \in \mathbf{T_V}$ fresh

(9) $$\frac{H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{n-1}_{T_P} (u_1, \mathbf{s}_1) \qquad H \rhd \mathsf{rec}\{x = e\}(u_1, \mathbf{s}_1) \Rightarrow_{T_P} (u_2, \mathbf{s}_2)}{H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{n}_{T_P} (u_2, \mathbf{s}_2)}$$

(10) $$\frac{H[x \leftarrow (u, \mathbf{s})] \rhd e \Rightarrow (u_1, \mathbf{s}_1)}{H \rhd \mathsf{rec}\{x = e\}(u, \mathbf{s}) \Rightarrow_{T_P} (u_1, \mathbf{s}_1)}$$

(1) $H \rhd c \Rightarrow (type(c), \epsilon)$

Figure 4. Rules of the Gori-Levi type abstract interpreter $\rhd_{\mathrm{GL}}$

Corollary 16. *For every $k \geq 1$, expression e, and environment D:* $\mathbf{PT}^k_0(D, e) = \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D \vdash^{k,\mathrm{ML}}_0 e : t\})$.

4 The Gori-Levi Type Abstract Interpreter Revisited

In this section we briefly recall the type abstract interpreter of Gori and Levi [5,6]. The syntax of the small ML-like language used in the present paper is slightly different from the one in [5,6]. Namely, our language uses the rec-notation instead of the μ-notation, uses booleans instead of integers in the test of conditionals, and has more constants. In order to simplify the presentation of the correspondence result (see Section 5.2) we reformulate in the following the type abstract interpreter by using the language syntax and the notations that we have used so far.

The *abstract semantics* (or *abstract typing*) of an expression e is the type $u \in \mathbf{T}_0$ of the expression e together with a substitution $\mathbf{s} \in \mathbf{T_V} \to \mathbf{T}_0$ representing a constraint on the type variables. It is computed w.r.t. a given *abstract environment* H, which is a partial function mapping program variables $\mathbf{P_V}$ to abstract typings $(\mathbf{T}_0 \times (\mathbf{T_V} \to \mathbf{T}_0))$. The intuition is that the substitution $\mathbf{s} \in \mathbf{T_V} \to \mathbf{T}_0$ collects all the constraints generated on the type variables of H while inferring the type u for the expression e.

Gori and Levi obtained systematically the rules of a non-effective type abstract interpreter $\rhd_{\mathrm{GL}}$ by abstracting the concrete semantics on the type domain $\mathbf{P_V} \to (\mathbf{T}_0 \times (\mathbf{T_V} \to \mathbf{T}_0))$. These rules are (modulo the change of notation) those in Figure 4. Rule numbers correspond exactly to those in [5,6]. Note that, rule (3) (for integer additions) and rule (4) (for conditionals) are missing because they are encompassed by the use of the function **type** for typing constants in rule (1) — see the discussion in Section 3.1. The operation **eqs** (used in rule (5))

$$(11)\ \frac{H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{k}_{\mathrm{wid}} (u, \mathbf{s})}{H \rhd \mathsf{rec}\{x = e\} \Rightarrow (u, \mathbf{s})} \qquad (12)\ \frac{\begin{array}{c} H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{k-1}_{T_P} (u_1, \mathbf{s}_1) \\ H \rhd \mathsf{rec}\{x = e\}(u_1, \mathbf{s}_1) \Rightarrow_{T_P} (u_2, \mathbf{s}_2) \\ (u_1, \mathbf{s}_1) = (u_2, \mathbf{s}_2) \end{array}}{H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{k}_{\mathrm{wid}} (u_1, \mathbf{s}_1)}$$

$$(13)\ \frac{\begin{array}{cc} H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{k-1}_{T_P} (u_1, \mathbf{s}_1) & H \rhd \mathsf{rec}\{x = e\}(u_1, \mathbf{s}_1) \Rightarrow_{T_P} (u_2, \mathbf{s}_2) \\ (u_1, \mathbf{s}_1) \neq (u_2, \mathbf{s}_2) & \mathbf{s} = \mathbf{mgu}(\{\mathbf{s}_2(u_1) = u_2\} \cup \mathbf{eqs}(\mathbf{s}_2)) \end{array}}{H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{k}_{\mathrm{wid}} (\mathbf{s}(u_1), \theta)}$$

Figure 5. Rules of the Gori-Levi decidable type abstract interpreter $\rhd^{k,\mathrm{ML}}_{\mathrm{GL}}$

is defined as $\mathbf{eqs}([\alpha_1 \leftarrow u_1, \ldots, \alpha_n \leftarrow u_n]) \stackrel{\mathrm{def}}{=} \{\alpha_1 = u_1, \ldots, \alpha_n = u_n\}$ and $H[x \leftarrow \rho]$ (used in rules (6) and (10)) is a destructive update.

As this interpreter is possibly non-effective, because the type domain is not Nötherian, Gori and Levi [5,6] replace rule (7) with the rules (11), (12), (13) of Figure 5 (that arises from a family of widening operators as k varies) yielding the effective type abstract interpreter $\rhd^{k,\mathrm{ML}}_{\mathrm{GL}}$.

5 The Type Systems Corresponding to $\rhd_{\mathbf{GL}}$ and $\rhd^{\boldsymbol{k},\mathbf{ML}}_{\mathbf{GL}}$

In this section we solve the open problem [5,6] of finding a type system corresponding to the $\rhd_{\mathrm{GL}}$ and $\rhd^{k,\mathrm{ML}}_{\mathrm{GL}}$ type abstract interpreters.

As stated in [5,6], the type systems corresponding to the abstract interpreters $\rhd_{\mathrm{GL}}$ and $\rhd^{k,\mathrm{ML}}_{\mathrm{GL}}$ would lye between the Curry-Hindley and Milner-Mycroft type systems. Example 17 (see below) suggests that the type abstract interpreter $\rhd_{\mathrm{GL}}$ infers a type to a recursive function definition $\mathsf{rec}\,\{x = e\}$ by requiring that all the occurrences of x in e are used with a same type (as in the Curry-Hindley type system) that specializes (as in the Milner-Mycroft type system) the type inferred for e. Following this intuition, in Section 5.1 we will define a (less powerful) variant of system $\vdash^{\mathrm{P}}_{0}$, that we will call $\vdash_{\mathrm{GL}}$, by requiring that all the occurrences of x in e are used with *the same* type. In Section 5.2 we will prove that system $\vdash_{\mathrm{GL}}$ is equivalent to the $\rhd_{\mathrm{GL}}$ type abstract interpreter. This result disproves the misconception that in the type system corresponding to the abstract interpreter $\rhd_{\mathrm{GL}}$ "the different function applications of a recursive function can lead to different (but *compatible*) instantiation of the recursive function type" ([6], page 142). Namely, our result shows that two different instantiations are *compatible* if and only if they are *the same*.

Example 17. The term $\mathsf{rec}\,\{f = \lambda x.((\lambda y.\mathsf{true})\,(\mathsf{not}\,(f\,\mathsf{false}))\}$, where the function f is recursively called with type $\mathsf{bool} \to \mathsf{bool}$, is Curry-Hindley typable with principal typing $\langle \emptyset; \mathsf{bool} \to \mathsf{bool} \rangle$ and is typed by $\rhd_{\mathrm{GL}}$ with the more general typing $\langle \emptyset; \alpha \to \mathsf{bool} \rangle$, which is also the principal-in-$\emptyset$ typing w.r.t. system $\vdash^{\mathrm{P}}_{0}$.

The term $\mathsf{rec}\,\{g = \lambda x.((g\,2) + (g\,\mathsf{false}))\}$, where the function g is recursively called with the two non-unifiable types $\mathsf{int} \to \mathsf{int}$ and $\mathsf{bool} \to \mathsf{int}$, cannot be typed by the $\rhd_{\mathrm{GL}}$ type abstract interpreter and can be typed by system $\vdash^{\mathrm{P}}_{0}$ with principal-in-$\emptyset$ typing $\langle \emptyset; \alpha \to \mathsf{int} \rangle$.

5.1 Systems $\vdash_{\mathbf{GL}}$, $\vdash^{\boldsymbol{k}}_{\mathbf{GL}}$, and $\vdash^{\boldsymbol{k,ML}}_{\mathbf{GL}}$

Given a typing environment D and an expression e, the set $\mathrm{FV}^{\mathrm{GL}}_D(e) \stackrel{\text{def}}{=} \mathrm{FV}(e) \cup VR(D|_{\mathrm{FV}(e)})$ is the *set of the free variables of e and of the free variables of e in D* (note that $\mathrm{FV}^{\mathrm{GL}}_D(e) = \mathrm{FV}(e) \cup \mathrm{FV}_D(e)$). The rules of system $\vdash_{\mathrm{GL}}$ are obtained by those of system $\vdash^{\mathrm{P}}_0$ (in Figure 3) by replacing the rules (Rec-P) and (Var-P) with the following two rules:

$$\text{(Rec-GL)}\ \frac{D, x : \langle U; u\rangle \vdash e : \langle U, x : u'; u\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle U; u\rangle} \qquad \text{(Var-GL)}\ D,\ x : \langle U; u\rangle \vdash x : \langle U, x : u; u\rangle$$

where $Dom(U) = \mathrm{FV}^{\mathrm{GL}}_D(\mathsf{rec}\,\{x = e\})$ and $x \notin D$

Note that the combined use of rules (Var-GL) and (App) ensures that, in the premise of rule (Rec-GL), the body e of the recursive definition $\mathsf{rec}\,\{x = e\}$ is typed by assigning the same simple type u' to all the occurrences of x.

System $\vdash_{\mathrm{GL}}$ can be seen as derived from system $\vdash_0$ (in Section 3.1) by adapting the second step (see Section 3.2) of the technique illustrated in Section 3. We now adapt the third (see Section 3.3) and the fourth (see Section 3.4) steps to system $\vdash_{\mathrm{GL}}$, obtaining two new type systems that we will call $\vdash^{k}_{\mathrm{GL}}$ and $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}$, respectively.

Adapting the Third Step For every $k \geq 1$, let $\vdash^{k}_{\mathrm{GL}}$ be the system obtained from $\vdash_{\mathrm{GL}}$ by replacing rule (Rec-GL) with the rule:

$$\text{(Rec-GL-}k\text{)}\ \frac{D, x : \langle U_0; u_0\rangle \vdash e : \langle U_1, x : u'_1; u_1\rangle \ \cdots\ D, x : \langle U_{k-1}; u_{k-1}\rangle \vdash e : \langle U_k, x : u'_k; u_k\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle U_k; u_k\rangle}$$

where $x \notin D$,
$\langle U_0; u_0\rangle \in \mathsf{B}_{\mathrm{FV}^{\mathrm{GL}}_D(\mathsf{rec}\{x=e\})}$,
$(\forall i \in \{1, \ldots, k\})\ \langle U_i, x : u'_i; u_i\rangle \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D, x : \langle U_{i-1}; u_{i-1}\rangle \vdash e : t\})$,
$\langle U_{k-1}; u_{k-1}\rangle = \langle U_k; u_k\rangle$

Adapting the Fourth Step For every $k \geq 1$, let $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}$ the system obtained from $\vdash^{k}_{\mathrm{GL}}$ by adding the rule:

$$\text{(Rec-GL-ML)}\ \frac{D \vdash e : \langle U, x : u; u\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle U; u\rangle}$$

where $x \notin D$ and there are no $\langle U_i; u_i\rangle$ $(1 \leq i \leq k)$ such that:
$\langle U_0; u_0\rangle \in \mathsf{B}_{\mathrm{FV}^{\mathrm{GL}}_D(\mathsf{rec}\{x=e\})}$,
$(\forall i \in \{1, \ldots, k\})\ \ \langle U_i, x : u'_i; u_i\rangle \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D, x : \langle U_{i-1}; u_{i-1}\rangle \vdash e : t\})$,
$\langle U_{k-1}; u_{k-1}\rangle = \langle U_k; u_k\rangle$

5.2 Soundness and completeness of $\rhd^{\boldsymbol{k,ML}}_{\mathbf{GL}}/\rhd_{\mathbf{GL}}$ w.r.t. $\vdash^{\boldsymbol{k,ML}}_{\mathbf{GL}}/\vdash_{\mathbf{GL}}$

It is not possible to give a pointwise straightforward correspondence between the structure of the proof tree in system $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}/\vdash_{\mathrm{GL}}$ and the steps of abstract interpreter $\rhd^{k,\mathrm{ML}}_{\mathrm{GL}}/\rhd_{\mathrm{GL}}$ because of the way environments are built. However when $\rhd^{k,\mathrm{ML}}_{\mathrm{GL}}/\rhd_{\mathrm{GL}}$ terminates we reach an abstract typing which is isomorphic w.r.t. the corresponding typing in $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}/\vdash_{\mathrm{GL}}$.

In order to express explicitly the isomorphism between the abstract typing computed by $\rhd^{k,\mathrm{ML}}_{\mathrm{GL}}/\rhd_{\mathrm{GL}}$ and the typing in $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}/\vdash_{\mathrm{GL}}$, we need to introduce some preliminary notations. Let $Dom(H)$ denote the domain of the partial function H. Given H, we write H^1, H^2 as the projection functions (given H, if $x \in Dom(H)$ and $H(x) = (u, \mathbf{s})$, then $H^1(x) = u$ and $H^2(x) = \mathbf{s}$). Also we define $Range(H) = \{\mathrm{FTV}(u) \mid H^1(x) = u, x \in Dom(H)\}$ as the *range of* H. We can now define the equivalence relations $\approx_H$ and state the correspondence results between $\rhd^{k,\mathrm{ML}}_{\mathrm{GL}}/\rhd_{\mathrm{GL}}$ and $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}/\vdash_{\mathrm{GL}}$.

Definition 18. *Let H be an abstract environment, $\langle U; u\rangle$ a typing, and $(u, \mathbf{s})$ an abstract typing. We write $\langle U; u\rangle \approx_H (u', \mathbf{s})$ to mean that:*

$$u = u', \quad Dom(\mathbf{s}) \subseteq Range(H|_{Dom(U)}), \quad \text{and} \quad (\text{forall } x : u'' \in U)\ \mathbf{s}(H^1(x)) = u''.$$

Definition 19. *Let e be an expression, D a typing environment, and H an abstract environment. We write $D \cong H$ to mean that:*

$$Dom(D) \subseteq Dom(H) \text{ and (for all } x \in Dom(D))\ x : t \in D \text{ if and only if } t \approx_H H(x).$$

Theorem 20 (Soundness and completeness of $\rhd^{k,\mathbf{ML}}_{\mathbf{GL}}$ w.r.t. $\vdash^{k,\mathbf{ML}}_{\mathbf{GL}}$). *Let D be a typing environment and H be an abstract environment with $D \cong H$, then for every $k \geq 1$ and expression e:*
(Soundness). *If $H \rhd^{k}_{\mathrm{GL}} e \Rightarrow (u, \mathbf{s})$, then $D \vdash^{k,\mathrm{ML}}_{\mathrm{GL}} e : t$, with $t \approx_H (u, \mathbf{s})$ and $t \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t' \mid D \vdash^{k,\mathrm{ML}}_{\mathrm{GL}} e : t'\})$.*
(Completeness). *If $D \vdash^{k,\mathrm{ML}}_{\mathrm{GL}} e : t$, then $H \rhd^{k}_{\mathrm{GL}} e \Rightarrow (u, \mathbf{s})$, with $t' \approx_H (u, \mathbf{s})$ for some $t' \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t'' \mid D \vdash^{k,\mathrm{ML}}_{\mathrm{GL}} e : t''\})$.*

Theorem 21 (Soundness and completeness of $\rhd_{\mathbf{GL}}$ w.r.t. $\vdash_{\mathbf{GL}}$). *Let D be a typing environment, H be an abstract environment with $D \cong H$, then for every expression e:*
(Soundness). *If $H \rhd_{\mathrm{GL}} e \Rightarrow (u, \mathbf{s})$, then $D \vdash_{\mathrm{GL}} e : t$, with $t \approx_H (u, \mathbf{s})$ and $t \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t' \mid D \vdash_{\mathrm{GL}} e : t'\})$.*
(Completeness). *If $D \vdash_{\mathrm{GL}} e : t$, then $H \rhd_{\mathrm{GL}} e \Rightarrow (u, \mathbf{s})$ with $t' \approx_H (u, \mathbf{s})$ for some $t' \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t'' \mid D \vdash_{\mathrm{GL}} e : t''\})$.*

6 Conclusions and Future Work

In this paper we have exploited the general technique proposed in [4] to:

1. develop a family of decidable type systems that lye between of Curry-Hindley type system and (the let-free fragment of) the Milner-Mycroft type system and provide a complete stratification of (let-free) Milner-Mycroft typability;
2. solve the problem of finding type systems corresponding to the type abstract interpreters proposed by Gori and Levi [5,6], thus providing a precise characterization of the expressive power of these type abstract interpreters (see the discussion at the beginning of Section 5).

We plan to extend the type systems (and the results) illustrated in this paper to deal with let-expressions by exploiting a general technique (developed in [3]) for extending a type system (without rules for let-expressions) enjoying the principal typing property with a typing rule for let-expressions. Moreover, we would like to investigate whether the notion of principal typing can be exploited to develop a framework to systematize (to some extent) the task of relating program analyses specified via type systems and program analyses specified via abstract interpretation.

References

1. P. Cousot. Types as Abstract Interpretations. In *POPL'97*, pages 316–331. ACM, 1997.
2. L. M. M. Damas and R. Milner. Principal type schemas for functional programs. In *POPL'82*, pages 207–212. ACM, 1982.
3. F. Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Trans. Prog. Lang. Syst.*, 25(4):401–451, 2003.
4. F. Damiani. Rank 2 intersection for recursive definitions. *Fundamenta Informaticae*, 77(4):451–488, 2007.
5. R. Gori and G. Levi. An experiment in type inference and verification by abstract interpretation. In *VMCAI'02*, volume 2294 of *LNCS*, pages 225–239. Springer, 2002.
6. R. Gori and G. Levi. Properties of a type abstract interpreter. In *VMCAI'03*, volume 2575 of *LNCS*, pages 132–145. Springer, 2003.
7. F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Prog. Lang. Syst.*, 15(2):253–289, 1993.
8. R. Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, London, 1997.
9. T. Jim. What are principal typings and what are they good for? In *POPL'96*, pages 42–53. ACM, 1996.
10. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Prog. Lang. Syst.*, 15(2):290–311, 1993.
11. L. Meertens. Incremental polymorphic type checking in B. In *POPL'83*, pages 265–275. ACM, 1983.
12. B. Monsuez. Polymorphic typing by abstract interpretation. *Theoretical Computer Science*, 652:217–228, 1992.
13. B. Monsuez. Polymorphic types and widening operators. In *SAS'93*, volume 724 of *LNCS*, pages 224–281. Springer, 1993.
14. A. Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228. Springer, 1984.
15. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
16. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
17. J.B. Wells. The essence of principal typings. In *ICALP'02*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.