

# On Polymorphic Recursion, Type Systems, and Abstract Interpretation

Marco Comini<sup>1</sup>, Ferruccio Damiani<sup>2</sup>, Samuel Vrech<sup>1</sup>

<sup>1</sup> Dipartimento di Matematica e Informatica, Università di Udine  
Via delle Scienze, 206; I-33100 Udine, Italy

<sup>2</sup> 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  $\text{rec } \{x = e\}$  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 a family of abstract interpreters that are able to type all the ML typable recursive definitions and interesting examples of polymorphic recursion. The problem of finding type systems corresponding to their abstract interpreters was open (such systems would lie between the let-free fragments of the ML and of the Milner-Mycroft systems).

In this paper we exploit the notion of principal typing to: (i) provide a complete stratification of (let-free) Milner-Mycroft typability, and (ii) solve the problem of finding type systems corresponding to the type abstract interpreters proposed by Gori and Levi.

**Keywords.** Principal Typing, Type Inference Algorithm

## 1 Introduction

The *Hindley-Milner system* (a.k.a. the *ML type system*) [3], 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*<sup>3</sup>. The problem of inferring types for *polymorphic recursion*<sup>4</sup> [13,16] has been studied both by people working on type systems [8,12] (see also [1,10,11,18,5]) and by people working on abstract interpretation [14,15,6,7].

Building on results by Cousot [2], Gori and Levi [6,7] have developed a family of type abstract interpreters that are able to type all the ML typable recursive definitions and interesting examples of polymorphic recursion.

As pointed out in [6,7], the problem of finding a type system corresponding to the type abstract interpreter was open. Such type systems would lie between the *Curry-Hindley system* (a.k.a. the *system of simple types*) [9], which is the let-free

---

<sup>3</sup> I.e., recursive function definitions  $\text{rec } \{x = e\}$  where all the occurrences of  $x$  in  $e$  are used with exactly the same type inferred for  $e$ .

<sup>4</sup> I.e., recursive function definitions  $\text{rec } \{x = e\}$  where different occurrences of  $x$  in  $e$  are used with different types that specialize the type inferred for  $e$ .

fragment of the ML type system, and the let-free fragment of the *Milner-Mycroft system* [16]. Previous work of the second author [5] proposes a technique for extending a type system enjoying decidable typability and *principal typings* [10,19] by adding a decidable rule for typing *rec*-expressions.

In this paper we exploit the technique of [5] to: (1) Develop a family of decidable type systems that lie between the 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; and (2) Solve the problem of finding type systems corresponding to the type abstract interpreters proposed by Gori and Levi [6,7], thus providing a precise characterization of the expressive power of these type abstract interpreters.

**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 develops a family of decidable type systems that provide a complete stratification of (let-free) Milner-Mycroft typability. Section 4 briefly illustrates the Gori-Levi family of type abstract interpreters [6,7] and Section 5 presents a family of type systems that corresponds to these abstract interpreters.

## 2 Preliminary Definitions

Program *Expressions*, ranged over by  $e$ , are defined as  $e ::= x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \text{rec } \{x = e\}$ , where  $x$  ranges over program variables (in  $\mathbf{P}_V$ ) and  $c$  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  $\text{FV}(e)$ .

The set of constants includes the booleans (ranged over by  $b$ ), the integer numbers (ranged over by  $\iota$ ), 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.

The set of *simple types* ( $\mathbf{T}_0$ ), ranged over by  $u$ , is defined as  $u ::= \text{bool} \mid \text{int} \mid \alpha \mid u_1 \rightarrow u_2 \mid u_1 \times u_2 \mid u \text{ list}$ . We have *type variables* (ranged over by  $\alpha$ ), 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 and the constructors  $\times$  and *list* bind more tightly than  $\rightarrow$ .

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. The *domain* of a substitution  $\mathbf{s}$  is the set of variables  $\mathbf{Dom}(\mathbf{s}) \stackrel{\text{def}}{=} \{\alpha \mid \mathbf{s}(\alpha) \neq \alpha\}$ . Substitutions will be denoted by  $[\alpha_1 \leftarrow u_1, \dots, \alpha_n \leftarrow u_n]$  ( $n \geq 0$ ); 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  $\alpha$ . 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  $\mathbf{s}' = \mathbf{s}'' \cdot \mathbf{s}$ .

A *type environment*  $E$  is a set  $\{x_1 : \rho_1, \dots, x_n : \rho_n\}$  of type assumptions for program variables such that every variable  $x_i$  ( $1 \leq i \leq n$ ) can occur at most once in  $E$ . The expression  $\text{Dom}(E)$  denotes the *domain* of  $E$ , which is the set  $\{x_1, \dots, x_n\}$ . 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  $x \notin \text{Dom}(E)$ . 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  $\{x_1 : u_1, \dots, x_n : u_n\}$  of simple type assumptions for variables.

According to Wells [19], 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  $\leq_{\text{spsc}}$  is reflexive and transitive).

**Definition 2 (Typing specialization relation  $\leq_{\text{spsc}}$ ).** A typing  $\langle U; u \rangle$  can be specialized to  $\langle U'; u' \rangle$  (notation  $\langle U; u \rangle \leq_{\text{spsc}} \langle U'; u' \rangle$ ) if  $\mathbf{s}(U) = U'$  and  $\mathbf{s}(u) = u'$ , for some substitution  $\mathbf{s}$ . We will write  $\langle U; u \rangle =_{\text{spsc}} \langle U'; u' \rangle$  to mean that both  $\langle U; u \rangle \leq_{\text{spsc}} \langle U'; u' \rangle$  and  $\langle U'; u' \rangle \leq_{\text{spsc}} \langle U; u \rangle$  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  $t \leq_{\text{spsc}} t'$ . We say that system  $\vdash$  has the *principal typing property* to mean that every typable term has a principal typing.

### 3 A stratification of (let-free) Milner-Mycroft typability

In this section we obtain a complete stratification of (let-free) Milner-Mycroft typability by applying the technique of [5] to the Curry-Hindley system [9].

#### 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  $\text{FV}(e)$ .

$$\begin{array}{c}
(\text{SPC}) \frac{\vdash e : t}{\vdash e : t'} \quad \text{where } t \leq_{\text{spc}} t' \qquad (\text{CON}) \vdash c : \langle \emptyset; u \rangle \quad \text{where } u = \mathbf{type}(c) \qquad (\text{VAR}) \vdash x : \langle \{x : u\}; u \rangle \quad \text{where } u \in \mathbf{T}_0 \\
(\text{APP}) \frac{\vdash e_1 : \langle U_1; u_0 \rightarrow u \rangle \quad \vdash e_2 : \langle U_2; u_0 \rangle}{\vdash e_1 e_2 : \langle U_1 \oplus U_2; u \rangle} \\
(\text{ABS}) \frac{\vdash e : \langle U, x : u_0; u \rangle}{\vdash \lambda x. e : \langle U; u_0 \rightarrow u \rangle} \quad \text{where } x \in \text{FV}(e) \qquad (\text{ABSVAC}) \frac{\vdash e : \langle U; u \rangle}{\vdash \lambda x. e : \langle U; u_0 \rightarrow u \rangle} \quad \text{where } x \notin \text{FV}(e) \text{ and } u_0 \in \mathbf{T}_0
\end{array}$$

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

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

**Figure 2.** Types for constants

- It has the principal typing property (see Definition 3).
- It is decidable to establish whether a pair  $t_1$  can be specialized to a pair  $t_2$  (i.e., whether  $t_1 \leq_{\text{spc}} t_2$  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 reformulation of the system of simple types [9], 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  $\mathbf{type}$  (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  $\text{Dom}(U) = \text{FV}(e)$ , we have two rules for typing an abstraction  $\lambda x. e$ , (ABS) and (ABSVAC), corresponding to the two cases  $x \in \text{FV}(e)$  and  $x \notin \text{FV}(e)$ .

### 3.2 System $\vdash_0^P$ : 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  $\text{rec}\{x = e\}$  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  $\{x_1 : t_1, \dots, x_n : t_n\}$  of typing assumptions for variables such that  $\text{Dom}(D) \cap \text{VR}(D) = \emptyset$ , where  $\text{VR}(D) \stackrel{\text{def}}{=} \cup_{x:\langle U; u \rangle \in D} \text{Dom}(U)$  is the set of variables occurring in the range of  $D$ .

$$\begin{array}{c}
\text{(SPC)} \frac{D \vdash e : t}{D \vdash e : t'} \quad \text{where } t \leq_{\text{SPC}} t' \quad \text{(CON)} D \vdash c : \langle \emptyset; u \rangle \quad \text{where } u = \mathbf{type}(c) \quad \text{(VAR)} D \vdash x : \langle \{x : u\}; u \rangle \quad \text{where } u \in \mathbf{T}_0 \text{ and } x \notin \text{Dom}(D) \\
\text{(ABS)} \frac{D \vdash e : \langle U, x : u_0; u \rangle}{D \vdash \lambda x. e : \langle U; u_0 \rightarrow u \rangle} \quad \text{where } x \notin D \quad \text{(ABSVAC)} \frac{D \vdash e : \langle U; u \rangle}{D \vdash \lambda x. e : \langle U; u_0 \rightarrow u \rangle} \quad \text{where } x \notin \text{FV}(e), u_0 \in \mathbf{T}_0, \text{ and } x \notin D \\
\text{(APP)} \frac{D \vdash e_1 : \langle U_1; u_0 \rightarrow u \rangle \quad D \vdash e_2 : \langle U_2; u_0 \rangle}{D \vdash e_1 e_2 : \langle U_1 \oplus U_2; u \rangle} \\
\text{(REC-P)} \frac{D, x : \langle U; u \rangle \vdash e : \langle U; u \rangle}{D \vdash \text{rec } \{x = e\} : \langle U; u \rangle} \quad \text{where } \text{Dom}(U) = \text{FV}_D(\text{rec } \{x = e\}) \text{ and } x \notin D \quad \text{(VAR-P)} D, x : t \vdash x : t
\end{array}$$

**Figure 3.** Typing rules of system  $\vdash_0^P$

Every typing  $t$  occurring in  $D$  is implicitly universally quantified over all type variables occurring in  $t$ .<sup>5</sup>

The typing rules of system  $\vdash_0^P$  (where “P” stands for “polymorphic”) are given in Figure 3. The judgement  $D \vdash_0^P e : \langle U; u \rangle$  means “ $e$  is  $\vdash_0^P$ -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 a simple type environment containing the type assumptions for the free variables of  $e$  which are not in  $\text{Dom}(D)$ , and  $u$  is a simple type.

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

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 “ $x \notin \text{Dom}(D)$ ” in rule (VAR)) to ensure that  $\text{Dom}(D) \cap \text{Dom}(U) = \emptyset$ .

Rule (REC-P) allows to assign to a recursive definition  $\text{rec } \{x = e\}$  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  $t_i = \langle U_i; u_i \rangle$  ( $1 \leq i \leq n$ ) of  $t = \langle U; u \rangle$  to different occurrences of  $x$  in  $e$ , provided that  $\bigoplus_{1 \leq i \leq n} U_i$  is defined.

The following theorem (taken from [5]) shows that system  $\vdash_0^P$  has the same expressive power as the Milner-Mycroft system [16]. This implies that typability in system  $\vdash_0^P$  is undecidable (as is in the Milner-Mycroft system [8,12]).

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

<sup>5</sup> To emphasize this fact we might have used assumptions of the shape  $\forall \vec{\alpha}. t$  where  $\vec{\alpha}$  is the sequence of all the type variables occurring in  $t$ .

### 3.3 Systems $\vdash_0^k$ ( $k \geq 1$ ): a Family of Decidable Restrictions of $\vdash_0^P$

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  $t \leq_{\text{spc}} t'$ . 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  $X = \{x_1, \dots, x_n\}$  ( $n \geq 0$ ) let  $\mathbb{T}_X \stackrel{\text{def}}{=} \{\langle U; u \mid \langle U; u \rangle \text{ is typing such that } \text{Dom}(U) = X\}, \text{ and } \mathbb{B}_X \stackrel{\text{def}}{=} \{\langle \{x_1 : \alpha_1, \dots, x_n : \alpha_n\}; \alpha \rangle \mid \text{the type variables } \alpha_1, \dots, \alpha_n, \alpha \text{ are all distinct}\} \subseteq \mathbb{T}_X$ . The typing specialization relation  $\leq_{\text{spc}}$  (see Definition 2) is a preorder over  $\mathbb{T}_X$  and, for all typings  $b \in \mathbb{B}_X$  and  $t \in \mathbb{T}_X$ ,  $b \leq_{\text{spc}} t$ . Moreover for every subset  $S_X$  of  $\mathbb{T}_X$ ,

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

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

The following proposition holds.

**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  $t \in \mathbf{Min}_{\leq_{\text{spc}}}(\{t' \mid D \vdash e : t'\})$ .*

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

$$\text{(REC-}k\text{)} \frac{D, x : t_0 \vdash e : t_1 \cdots D, x : t_{k-1} \vdash e : t_k}{D \vdash \text{rec } \{x = e\} : t_k} \text{ where} \quad (3.1)$$

$$x \notin D, t_0 \in \mathbf{B}_{\text{FV}_D(\text{rec } \{x=e\})},$$

$$(\forall i \in \{1, \dots, k\}) t_i \in \mathbf{Min}_{\leq_{\text{spc}}}(\{t \mid D, x : t_{i-1} \vdash e : t\}), t_{k-1} = t_k$$

(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  $i \in \{1, \dots, k\}$ )  $t_i$  is a principal-in- $(D, x : t_{i-1})$  typing for  $e$ . Therefore, as pointed out in [5], 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  $\vdash_0^k$  ( $k \geq 1$ )** A *unification problem*  $P$  is a set of equalities between simple types. A *solution* to  $P$  (*unifier*) is an idempotent 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 a 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  $e$ .

$e = \mathbf{x}$  If  $x : \langle U; u \rangle \in D$  and the substitution  $\mathbf{s}$  is a renaming of  $\vec{\alpha} = \mathbf{FTV}(U) \cup \mathbf{FTV}(u)$  with fresh type variables, then  $\langle \mathbf{s}(U); \mathbf{s}(u) \rangle \in \mathbf{PT}_0^k(D, x)$ .

If  $x \notin \text{Dom}(D)$  and  $\alpha$  is a type variable, then  $\langle \{x : \alpha\}; \alpha \rangle \in \mathbf{PT}_0^k(D, x)$ .

$e = \mathbf{c}$  If  $\mathbf{type}(c) = u$ , then  $\langle \emptyset; u \rangle \in \mathbf{PT}_0^k(D, c)$ .

$e = \lambda x. e_0$  If  $\langle U; u_0 \rangle \in \mathbf{PT}_0^k(D, e_0)$ , then

– If  $x \notin \mathbf{FV}(e_0)$  and  $\alpha$  is a fresh type variable, then  $\langle U; \alpha \rightarrow u_0 \rangle \in \mathbf{PT}_0^k(D, \lambda x. e_0)$ .

– If  $x \in \mathbf{FV}(e_0)$  and  $U = U'$ ,  $x : u$ , then  $\langle U'; u \rightarrow u_0 \rangle \in \mathbf{PT}_0^k(D, \lambda x. e_0)$ .

$e = e_0 e_1$  If  $\langle U_0; u_0 \rangle \in \mathbf{PT}_0^k(D, e_0)$ , then

– If  $u_0 = \alpha$  (a type variable),  $\alpha_1$  and  $\alpha_2$  are fresh type variables,  $\langle U_1; u_1 \rangle \in \mathbf{PT}_0^k(D, e_1)$  is renamed apart, and  $\mathbf{s} \in \mathbf{MGU}(\{u_1 = \alpha_1, \alpha = \alpha_1 \rightarrow \alpha_2\} \cup \{u' = u'' \mid x : u' \in U_0 \text{ and } x : u'' \in U_1\})$ , then  $\langle \mathbf{s}(U_0) \oplus \mathbf{s}(U_1); \mathbf{s}(u_0) \rangle \in \mathbf{PT}_0^k(D, e_0 e_1)$ .

– If  $u_0 = u_2 \rightarrow u$ , the pair  $\langle U_1; u_1 \rangle \in \mathbf{PT}_0^k(D, e_1)$  is renamed apart, 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  $\langle \mathbf{s}(U_0) \oplus \mathbf{s}(U_1); \mathbf{s}(u) \rangle \in \mathbf{PT}_0^k(D, e_0 e_1)$ .

$e = \mathbf{rec} \{x = e_0\}$  If  $h \in \{1, \dots, k\}$ ,  $t_0 \in \mathbf{B}_{\mathbf{FV}_D(e)}$ ,  $t_1 \in \mathbf{PT}_0^k((D, x : t_0), e_0)$ ,

$\dots$ ,  $t_h \in \mathbf{PT}_0^k((D, x : t_{h-1}), e_0)$ ,  $t_1 \not\leq_{\mathbf{spc}} t_0$ ,  $\dots$ ,  $t_{h-1} \not\leq_{\mathbf{spc}} t_{h-2}$ , 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.

**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.  $\text{Dom}(U) = \mathbf{FV}_D(e)$ , and
2.  $\langle U'; u' \rangle \in \mathbf{PT}_0^k(D, e)$  if and only if there is a bijection  $\mathbf{s} : \mathbf{T}_{\mathbf{V}} \rightarrow \mathbf{T}_{\mathbf{V}}$  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  $e$  w.r.t.  $D$  simply follow the definition of  $\mathbf{PT}_0^k(D, e)$ , choosing fresh type variables and using the unification and  $\leq_{\text{spc}}$ -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  $t \leq_{\text{spc}} t'$  for some  $t \in \mathbf{PT}_0^k(D, e)$ .*

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

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

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

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

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

**Theorem 13.** *If  $D \vdash_0^{\text{P}} e : t$ , 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  $\vdash_0^{\text{P}}$  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  $\text{rec } \{x = x x\}$ , 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 [17])

```
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  $\langle \{\}, 'a \text{ seq} \rightarrow \text{int} \rangle$  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  $\text{rec}\{f = \lambda g y. \text{if false then } y \text{ else } g(f g y)\}$  is not  $\vdash_0^2$ -typable but  $\vdash_0^3$ -typable with principal-in- $\emptyset$  typing  $\langle \emptyset; (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \rangle$ . The ML-typable term  $\text{rec}\{f = \lambda g h_1 y. \text{if false then } y \text{ else } g(f g h_1 (f h_1 g y))\}$  is not  $\vdash_0^3$ -typable but  $\vdash_0^4$ -typable with principal-in- $\emptyset$  typing  $\langle \emptyset; (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \rangle$ . In general, for all  $m \geq 0$ , the ML-typable term

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

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

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

### 3.4 Systems $\vdash_0^{k, \text{ML}}$ ( $k \geq 1$ ): 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  $\vdash_0^P$ , while preserving decidable typability and principal-in- $D$  typings.

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

1. is a restriction of system  $\vdash_0^P$  (i.e.,  $D \vdash_0^? e : \langle U; u \rangle$  implies  $D \vdash_0^P e : \langle U; u \rangle$ ),
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  $\vdash_0^P$ -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  $\vdash_0^P$ -suitable rule which (without the additional side condition) is at least as expressive as the rule for monomorphic recursions. 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:

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

and to restrict it with the additional side condition:

and there are no  $t_0, t_1, \dots, t_k$  such that:

$$t_0 \in \mathbf{B}_{\text{FV}_D(\text{rec}\{x=e\})},$$

$$\text{(for all } i \in \{1, \dots, k\}) \quad t_i \in \mathbf{Min}_{\leq \text{spc}}(\{t \mid D, x : t_{i-1} \vdash e : t\}), \text{ and } t_{k-1} = t_k$$

$$\begin{array}{l}
(2) \ H \triangleright x \Rightarrow H(x) \\
\text{where } x \in \mathbf{P}_V \\
(5) \ \frac{H \triangleright e_1 \Rightarrow (u_1, \mathbf{s}_1) \quad H \triangleright e_2 \Rightarrow (u_2, \mathbf{s}_2)}{\mathbf{s} = \mathbf{mgu}(\{u_1 = f_1 \rightarrow f_2, u_2 = f_1\} \cup \mathbf{eqs}(\mathbf{s}_1) \cup \mathbf{eqs}(\mathbf{s}_2))} \\
H \triangleright e_1 e_2 \Rightarrow (\mathbf{s}(f_2), \mathbf{s}) \\
(6) \ \frac{H[x \leftarrow (f_1, \epsilon)] \triangleright e \Rightarrow (u, \mathbf{s}) \quad u_1 = \mathbf{s}(f_1)}{H \triangleright (\lambda x. e) \Rightarrow ((u_1 \rightarrow u), \mathbf{s})} \\
(7) \ \frac{H \triangleright \mathbf{rec}\{x = e\} \Rightarrow_{T_P}^n (u_n, \mathbf{s}_n) \\
H \triangleright \mathbf{rec}\{x = e\} \Rightarrow_{T_P}^{n+1} (u_{n+1}, \mathbf{s}_{n+1}) \\
(u_n, \mathbf{s}_n) = (u_{n+1}, \mathbf{s}_{n+1})}{H \triangleright \mathbf{rec}\{x = e\} \Rightarrow (u_n, \mathbf{s}_n)} \\
(8) \ H \triangleright \mathbf{rec}\{x = e\} \Rightarrow_{T_P}^0 (\alpha, \epsilon) \\
\text{with } \alpha \in \mathbf{T}_V \text{ fresh} \\
(9) \ \frac{H \triangleright \mathbf{rec}\{x = e\} \Rightarrow_{T_P}^{n-1} (u_1, \mathbf{s}_1) \\
H \triangleright \mathbf{rec}\{x = e\}(u_1, \mathbf{s}_1) \Rightarrow_{T_P} (u_2, \mathbf{s}_2)}{H \triangleright \mathbf{rec}\{x = e\} \Rightarrow_{T_P}^n (u_2, \mathbf{s}_2)} \\
(10) \ \frac{H[x \leftarrow (u, \mathbf{s})] \triangleright e \Rightarrow (u_1, \mathbf{s}_1)}{H \triangleright \mathbf{rec}\{x = e\}(u, \mathbf{s}) \Rightarrow_{T_P} (u_1, \mathbf{s}_1)} \\
(1) \ H \triangleright c \Rightarrow (\text{type}(c), \epsilon)
\end{array}$$

**Figure 4.** Rules of the Gori-Levi type abstract interpreter  $\triangleright_{\text{GL}}$

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

A sound and complete inference algorithm for system  $\vdash_0^{k, \text{ML}}$  ( $\mathbf{PT}_0^{k, \text{ML}}$ ) 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  $\mathbf{rec}$ -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  $\mathbf{PT}_0^{k, \text{ML}}$  w.r.t.  $\vdash_0^{k, \text{ML}}$ ).**

For every  $k \geq 1$ , expression  $e$ , and environment  $D$ :

(*Soundness*) If  $t \in \mathbf{PT}_0^{k, \text{ML}}(D, e)$ , then  $D \vdash_0^{k, \text{ML}} e : t$ .

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

**Corollary 16.** For every  $k \geq 1$ , expression  $e$ , and environment  $D$ :

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

## 4 The Gori-Levi Type Abstract Interpreter Revisited

In this section we briefly recall the type abstract interpreter of Gori and Levi [6,7]. The syntax of the small ML-like language used in the present paper is slightly different from the one in [6,7]. Namely, our language uses the  $\mathbf{rec}$ -notation instead of the  $\mu$ -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 we 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  $\mathbf{s} \in \mathbf{T}_V \rightarrow \mathbf{T}_0$  representing a constraint on the type variables. It is computed w.r.t. a given *abstract*

$$\begin{array}{c}
(11) \frac{H \triangleright \text{rec}\{x = e\} \Rightarrow_{\text{wid}}^k (u, \mathbf{s})}{H \triangleright \text{rec}\{x = e\} \Rightarrow (u, \mathbf{s})} \qquad (12) \frac{H \triangleright \text{rec}\{x = e\} \Rightarrow_{T_P}^{k-1} (u_1, \mathbf{s}_1) \quad H \triangleright \text{rec}\{x = e\}(u_1, \mathbf{s}_1) \Rightarrow_{T_P} (u_2, \mathbf{s}_2) \quad (u_1, \mathbf{s}_1) = (u_2, \mathbf{s}_2)}{H \triangleright \text{rec}\{x = e\} \Rightarrow_{\text{wid}}^k (u_1, \mathbf{s}_1)} \\
(13) \frac{H \triangleright \text{rec}\{x = e\} \Rightarrow_{T_P}^{k-1} (u_1, \mathbf{s}_1) \quad H \triangleright \text{rec}\{x = e\}(u_1, \mathbf{s}_1) \Rightarrow_{T_P} (u_2, \mathbf{s}_2) \quad (u_1, \mathbf{s}_1) \neq (u_2, \mathbf{s}_2) \quad \mathbf{s} = \mathbf{mgu}(\{\mathbf{s}_2(u_1) = u_2\} \cup \mathbf{eqs}(\mathbf{s}_2))}{H \triangleright \text{rec}\{x = e\} \Rightarrow_{\text{wid}}^k (\mathbf{s}(u_1), \mathbf{s})}
\end{array}$$

**Figure 5.** Rules of the Gori-Levi decidable type abstract interpreter  $\triangleright_{\text{GL}}^{k, \text{ML}}$

*environment*  $H$ , which is a partial function mapping program variables  $\mathbf{P}_{\mathbf{V}}$  to abstract typings  $(\mathbf{T}_0 \times (\mathbf{T}_{\mathbf{V}} \rightarrow \mathbf{T}_0))$ . The intuition is that the substitution  $\mathbf{s} \in \mathbf{T}_{\mathbf{V}} \rightarrow \mathbf{T}_0$  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  $\triangleright_{\text{GL}}$  by abstracting the concrete semantics on the type domain  $\mathbf{P}_{\mathbf{V}} \rightarrow (\mathbf{T}_0 \times (\mathbf{T}_{\mathbf{V}} \rightarrow \mathbf{T}_0))$ . These rules are (modulo the change of notation) those in Figure 4. Rule numbers correspond exactly to those in [6,7]. 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)) is defined as  $\mathbf{eqs}([\alpha_1 \leftarrow u_1, \dots, \alpha_n \leftarrow u_n]) \stackrel{\text{def}}{=} \{\alpha_1 = u_1, \dots, \alpha_n = u_n\}$ , the operation **mgu** returns a most general unifier, 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 [6,7] 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  $\triangleright_{\text{GL}}^{k, \text{ML}}$ .

## 5 The Type Systems Corresponding to $\triangleright_{\text{GL}}$ and $\triangleright_{\text{GL}}^{k, \text{ML}}$

In this section we solve the open problem [6,7] of finding type systems corresponding to the  $\triangleright_{\text{GL}}$  and  $\triangleright_{\text{GL}}^{k, \text{ML}}$  type abstract interpreters.

As stated in [6,7], the type systems corresponding to the abstract interpreters  $\triangleright_{\text{GL}}$  and  $\triangleright_{\text{GL}}^{k, \text{ML}}$  would lie between the Curry-Hindley and Milner-Mycroft type systems. In Section 5.1 we will define (less powerful) variants of systems  $\vdash_0^{k, \text{ML}} / \vdash_0^{\text{P}}$ , that we will call  $\vdash_{\text{GL}}^{k, \text{ML}} / \vdash_{\text{GL}}$ , by requiring that all the occurrences of  $x$  in  $e$  are used with *the same* type. In Section 5.2 we will prove that systems  $\vdash_{\text{GL}}^{k, \text{ML}} / \vdash_{\text{GL}}$  are equivalent to the  $\triangleright_{\text{GL}}^{k, \text{ML}} / \triangleright_{\text{GL}}$  abstract interpreters. I.e., we will prove that the type abstract interpreter  $\triangleright_{\text{GL}}$  infers a type to a recursive function definition  $\text{rec}\{x = e\}$  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  $e$ . This result disproves the misconception that in the type system corresponding to the abstract

interpreters  $\triangleright_{\text{GL}}^{k,\text{ML}} / \triangleright_{\text{GL}}$  “the different function applications of a recursive function can lead to different (but *compatible*) instantiation of the recursive function type” ([7], page 142). Namely, our result shows that two different instantiations are *compatible* if and only if they are *the same*. Finally, in Section 5.3, we provide a simpler type system that has the same expressive power of the non-effective abstract interpreter  $\triangleright_{\text{GL}}$ .

*Example 17.* The non ML-typable term (taken from [2] — see also [7], page 140)

$$\text{rec } \{ f = \lambda f_1 g n x. \text{if } n = 0 \text{ then } g x \\ \text{else } f f_1 (\lambda y. (\lambda h. g (h y))) (n - 1) x f_1 \}$$

(which defines the function  $F$  such that  $F f_1 g n x = g (\overbrace{f_1 \cdots f_1}^{n \text{ times}} x) \cdots$ ) is typed by  $\triangleright_{\text{GL}}$  (and, for  $k \geq 3$ , also by  $\triangleright_{\text{GL}}^{k,\text{ML}}$ ) with type  $u = (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \beta) \rightarrow \text{int} \rightarrow \alpha \rightarrow \beta$ . The function  $f$  is recursively called with the specialized type  $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \beta) \rightarrow \text{int} \rightarrow \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \beta$  (obtained from  $u$  by substituting  $\beta$  with  $(\alpha \rightarrow \alpha) \rightarrow \beta$ ). The typing  $\langle \emptyset; u \rangle$  is principal-in- $\emptyset$  w.r.t.  $\vdash_0^{\text{P}}$  (and, for  $k \geq 3$ , also w.r.t.  $\vdash_0^{k,\text{ML}}$ ).

The non ML-typable term  $\text{rec } \{ g = \lambda x y. (g (\lambda z. x) (\lambda z. y)) + (g (\lambda z. y) (\lambda z. x)) \}$  is  $\vdash_0^{\text{P}}$ -typable and, for  $k \geq 2$ , also  $\vdash_0^{k,\text{ML}}$ -typable with principal-in- $\emptyset$  typing  $\langle \emptyset; \alpha \rightarrow \beta \rightarrow \text{int} \rangle$  (the function  $g$  is recursively called with the typings  $\langle \emptyset; (\alpha' \rightarrow \alpha) \rightarrow (\beta' \rightarrow \beta) \rightarrow \text{int} \rangle$  and  $\langle \emptyset; (\beta' \rightarrow \beta) \rightarrow (\alpha' \rightarrow \alpha) \rightarrow \text{int} \rangle$ ). However, the  $\triangleright_{\text{GL}}$  and, for  $k \geq 2$ , the  $\triangleright_{\text{GL}}^{k,\text{ML}}$  type abstract interpreters assign to the term the type  $\alpha \rightarrow \alpha \rightarrow \text{int}$ .

### 5.1 Systems $\vdash_{\text{GL}}$ , $\vdash_{\text{GL}}^k$ , and $\vdash_{\text{GL}}^{k,\text{ML}}$

Given a typing environment  $D$  and an expression  $e$ , the set  $\text{FV}_D^{\text{GL}}(e) \stackrel{\text{def}}{=} \text{FV}(e) \cup \text{VR}(D|_{\text{FV}(e)})$  is the *set of the free variables of  $e$  and of the free variables of  $e$  in  $D$*  (note that  $\text{FV}_D^{\text{GL}}(e) = \text{FV}(e) \cup \text{FV}_D(e)$ ). The rules of system  $\vdash_{\text{GL}}$  are obtained by those of system  $\vdash_0^{\text{P}}$  (in Figure 3) by replacing the rules (VAR-P) and (REC-P) with the following three rules:

$$\text{(VAR-GL)} \quad D, x : \langle U; u \rangle \vdash x : \langle U, x : u; u \rangle$$

$$\text{(REC-GL)} \quad \frac{D, x : \langle U; u \rangle \vdash e : \langle U, x : u'; u \rangle}{D \vdash \text{rec } \{ x = e \} : \langle U; u \rangle} \quad \text{(RECVAC-GL)} \quad \frac{D, x : \langle U; u \rangle \vdash e : \langle U; u \rangle}{D \vdash \text{rec } \{ x = e \} : \langle U; u \rangle}$$

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

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  $\text{rec } \{ x = e \}$  is typed by assigning the same simple type  $u'$  to all the occurrences of  $x$ . Rule (RECVAC-GL) is needed to type *vacuous recursive definitions* (i.e., expressions  $\text{rec } \{ x = e \}$  where  $x \notin \text{FV}(e)$ ).

System  $\vdash_{\text{GL}}$  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  $\vdash_{\text{GL}}$ , obtaining two new type systems that we will call  $\vdash_{\text{GL}}^k$  and  $\vdash_{\text{GL}}^{k,\text{ML}}$ , respectively.

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

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

where  $x \notin D$ ,

$$\begin{aligned} &\langle U_0; u_0 \rangle \in \mathbf{B}_{\text{FV}_D^{\text{GL}}(\text{rec}\{x=e\})}, \\ &(\forall i \in \{1, \dots, k\}) \langle U_i, x : u'_i; u_i \rangle \in \mathbf{Min}_{\leq \text{spc}}(\{t \mid D, x : \langle U_{i-1}; u_{i-1} \rangle \vdash e : t\}), \\ &\langle U_{k-1}; u_{k-1} \rangle = \langle U_k; u_k \rangle \end{aligned}$$

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

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

where  $x \notin D$  and there are no  $\langle U_i; u_i \rangle$  ( $1 \leq i \leq k$ ) such that:

$$\begin{aligned} &\langle U_0; u_0 \rangle \in \mathbf{B}_{\text{FV}_D^{\text{GL}}(\text{rec}\{x=e\})}, \\ &(\forall i \in \{1, \dots, k\}) \langle U_i, x : u'_i; u_i \rangle \in \mathbf{Min}_{\leq \text{spc}}(\{t \mid D, x : \langle U_{i-1}; u_{i-1} \rangle \vdash e : t\}), \\ &\langle U_{k-1}; u_{k-1} \rangle = \langle U_k; u_k \rangle \end{aligned}$$

## 5.2 Soundness and completeness of $\triangleright_{\text{GL}}^{k,\text{ML}} / \triangleright_{\text{GL}}$ w.r.t. $\vdash_{\text{GL}}^{k,\text{ML}} / \vdash_{\text{GL}}$

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

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

We can now define the equivalence relations  $\approx_H$  and state the correspondence results between  $\triangleright_{\text{GL}}^{k,\text{ML}} / \triangleright_{\text{GL}}$  and  $\vdash_{\text{GL}}^{k,\text{ML}} / \vdash_{\text{GL}}$ .

**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:  $u = u'$ ,  $\text{Dom}(\mathbf{s}) \subseteq \text{Range}(H|_{\text{Dom}(U)})$ , and (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:  $\text{Dom}(D) \subseteq \text{Dom}(H)$  and (for all  $x \in \text{Dom}(D)$ )  $x : t \in D$  if and only if  $t \approx_H H(x)$ .

**Theorem 20 (Soundness and completeness of  $\triangleright_{\text{GL}}^{k, \text{ML}}$  w.r.t.  $\vdash_{\text{GL}}^{k, \text{ML}}$ ).** 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  $H \triangleright_{\text{GL}}^k e \Rightarrow (u, \mathbf{s})$ , then  $D \vdash_{\text{GL}}^{k, \text{ML}} e : t$ , with  $t \approx_H (u, \mathbf{s})$  and  $t \in \text{Min}_{\leq \text{spc}}(\{t' \mid D \vdash_{\text{GL}}^{k, \text{ML}} e : t'\})$ .

**(Completeness)** If  $D \vdash_{\text{GL}}^{k, \text{ML}} e : t$ , then  $H \triangleright_{\text{GL}}^k e \Rightarrow (u, \mathbf{s})$ , with  $t' \approx_H (u, \mathbf{s})$  for some  $t' \in \text{Min}_{\leq \text{spc}}(\{t'' \mid D \vdash_{\text{GL}}^{k, \text{ML}} e : t''\})$ .

**Theorem 21 (Soundness and completeness of  $\triangleright_{\text{GL}}$  w.r.t.  $\vdash_{\text{GL}}$ ).** Let  $D$  be a typing environment,  $H$  be an abstract environment with  $D \cong H$ , then for every expression  $e$ :

**(Soundness)** If  $H \triangleright_{\text{GL}} e \Rightarrow (u, \mathbf{s})$ , then  $D \vdash_{\text{GL}} e : t$ , with  $t \approx_H (u, \mathbf{s})$  and  $t \in \text{Min}_{\leq \text{spc}}(\{t' \mid D \vdash_{\text{GL}} e : t'\})$ .

**(Completeness)** If  $D \vdash_{\text{GL}} e : t$ , then  $H \triangleright_{\text{GL}} e \Rightarrow (u, \mathbf{s})$  with  $t' \approx_H (u, \mathbf{s})$  for some  $t' \in \text{Min}_{\leq \text{spc}}(\{t'' \mid D \vdash_{\text{GL}} e : t''\})$ .

### 5.3 System $\vdash'_{\text{GL}}$ : a simpler characterization of $\triangleright_{\text{GL}}$

Let  $\vdash'_{\text{GL}}$  be the system obtained from system  $\vdash_0$  in Fig. 1 by removing rule (SPC) and adding the following rules:

$$\text{(SPC')} \frac{\vdash e : \langle U; u \rangle}{\vdash e : \langle U; \mathbf{s}(u) \rangle} \quad \text{where } \mathbf{Dom}(\mathbf{s}) = \text{FTV}(u) - \text{FTV}(U)$$

$$\begin{array}{l} \text{(REC-GL')} \frac{\vdash e : \langle U, x : \mathbf{s}(u); u \rangle}{\vdash \text{rec } \{x = e\} : \langle U; u \rangle} \\ \text{where } \mathbf{Dom}(\mathbf{s}) = \text{FTV}(u) - \text{FTV}(U) \end{array} \quad \begin{array}{l} \text{(RECVAC-GL')} \frac{\vdash e : \langle U; u \rangle}{\vdash \text{rec } \{x = e\} : \langle U; u \rangle} \\ \text{where } x \notin \text{FV}(e) \end{array}$$

Note that rule (SPC') is a restriction of rule (SPC) (rule (SPC) is admissible). The following theorem shows that system  $\vdash'_{\text{GL}}$  has the same expressive power as system  $\vdash_{\text{GL}}$  and, therefore, of the non-effective type abstract interpreter  $\triangleright_{\text{GL}}$ .

**Theorem 22.**  $\vdash'_{\text{GL}} e : \langle U; u \rangle$  if and only if  $\emptyset \vdash_{\text{GL}} e : \langle U; u \rangle$ .

## 6 Conclusions and Future Work

The stratification of (let-free) Milner-Mycroft typability, given in Section 3, is an adaptation of the results in [5] to the case without intersection types. Note that Theorem 13 and Example 14 are completely new (the analogue of Theorem 13 does not hold in presence of intersection types). We plan to extend the type systems (and the results) in Section 3 to deal with let-expressions by exploiting a general technique (developed in [4]) for extending a type system (without rules for let-expressions) enjoying the principal typing property with a typing rule for

let-expressions. The characterization of the expressive power of the type abstract interpreters of [6,7], given in Section 5, has been obtained by exploiting the notion of principal typing (which is at the basis of the technique of [5]). We believe that the notion of principal typing might be useful in other investigations on the relations between program analyses specified via type systems and program analyses specified via abstract interpretation. We don't know whether typability in the type system corresponding to the non-effective abstract interpreter  $\triangleright_{GL}$  (i.e., typability in systems  $\vdash_{GL}$  and  $\vdash'_{GL}$ ) is decidable.

## References

1. M. Coppo. An extended polymorphic type system. In *MFCS'80*, LNCS 88, pages 194–204. Springer, 1980.
2. P. Cousot. Types as abstract interpretations. In *POPL'97*, pages 316–331. ACM, 1997.
3. L. M. M. Damas and R. Milner. Principal type schemas for functional programs. In *POPL'82*, pages 207–212. ACM, 1982.
4. F. Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Trans. Prog. Lang. Syst.*, 25(4):401–451, 2003.
5. F. Damiani. Rank 2 intersection for recursive definitions. *Fundamenta Informaticae*, 77(4):451–488, 2007. (Special Issue on TLCA'95).
6. 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.
7. R. Gori and G. Levi. Properties of a type abstract interpreter. In *VMCAI'03*, volume 2575 of *LNCS*, pages 132–145. Springer, 2003.
8. F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Prog. Lang. Syst.*, 15(2):253–289, 1993.
9. R. Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, London, 1997.
10. T. Jim. What are principal typings and what are they good for? In *POPL'96*, pages 42–53. ACM, 1996.
11. A. J. Kfoury and S M. Pericas-Geertsen. Type inference for recursive definitions. In *LICS'99*, pages 119–128. IEEE, 1999.
12. 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.
13. L. Meertens. Incremental polymorphic type checking in B. In *POPL'83*, pages 265–275. ACM, 1983.
14. B. Monsuez. Polymorphic typing by abstract interpretation. *Theoretical Computer Science*, 652:217–228, 1992.
15. B. Monsuez. Polymorphic types and widening operators. In *SAS'93*, volume 724 of *LNCS*, pages 224–281. Springer, 1993.
16. A. Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228. Springer, 1984.
17. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
18. T. Terauchi and A. Aiken. On typability for polymorphic recursive rank-2 intersection types. In *LICS'06*, pages 111–122. IEEE, 2006.
19. J.B. Wells. The essence of principal typings. In *ICALP'02*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.