Rank-2 Intersection and Polymorphic Recursion

Ferruccio Damiani[*]

Dipartimento di Informatica, Università di Torino
Corso Svizzera 185, I-10149 Torino, Italy
damiani@di.unito.it

Abstract. Let $\vdash$ be a rank-2 intersection type system. We say that a term is $\vdash$-*simple* (or just *simple* when the system $\vdash$ is clear from the context) if system $\vdash$ can prove that it has a simple type. In this paper we propose new typing rules and algorithms that are able to type recursive definitions that are not simple. At the best of our knowledge, previous algorithms for typing recursive definitions in the presence of rank-2 intersection types allow only simple recursive definitions to be typed. The proposed rules are also able to type interesting examples of *polymorphic recursion* (i.e., recursive definitions $\mathsf{rec}\,\{x = e\}$ where different occurrences of x in e are used with different types). Moreover, the underlying techniques do not depend on particulars of rank-2 intersection, so they can be applied to other type systems.

1 Introduction

The Hindley-Milner type system (a.k.a. the ML type system) [5], which is the core of the type systems of modern functional programming languages (like SML, OCaml, and Haskell), has several limitations that prevent safe programs from being typed. In particular, it does not allow different types to be assigned to different occurrences of a formal parameter in the body of a function. To overcome these limitations, various extensions of the ML system based on *universal types* [7,19], *intersection types* [3,1], *recursive types*, and combinations of them, have been proposed in the literature.

The system of *rank-2 intersection types* [16,20,23,13,6] is particularly interesting since it is able to type all ML programs, has the principal pair property (a.k.a. principal typing property [13,22]), decidable type inference, and the complexity of type inference is of the same order as in ML.

Intersection types are obtained from *simple types* [11] by adding the *intersection type constructor* $\wedge$. A term has type $u_1 \wedge u_2$ (u_1 intersection u_2) if it has both type u_1 and type u_2. For example, the identity function $\lambda x.x$ has both type $\mathsf{int} \to \mathsf{int}$ and $\mathsf{bool} \to \mathsf{bool}$, so it has type $(\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{bool} \to \mathsf{bool})$. Rank-2 intersection types may contain intersections only to the left of a single arrow. Therefore, for instance, $((\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{bool} \to \mathsf{bool})) \to \mathsf{int} \to \mathsf{int}$ is a rank-2

intersection type (as usual, the arrow type constructor is right associative), while $(((\mathsf{int} \to \mathsf{int}) \land (\mathsf{bool} \to \mathsf{bool})) \to \mathsf{int}) \to \mathsf{int}$ is not a rank-2 intersection type.

The problem of typing standard programming language constructs like local definitions, conditional expressions, and recursive definitions without losing the extra information provided by rank-2 intersection is more difficult than one might expect (see, e.g., [12,13,21,6]).

Definition 1 ($\vdash$-simple terms). Let $\vdash$ be a rank-2 intersection type system. We say that a term is $\vdash$-*simple* (or just *simple* when the system $\vdash$ is clear from the context) if system $\vdash$ can prove that it has a simple type.

An example of *non-simple* term is $\lambda x.xx$ (which has principal type $((\alpha_1 \to \alpha_2) \land \alpha_1) \to \alpha_2$, where α_1, α_2 are type variables).

In previous work [6] we introduced rules and algorithms for typing *non-simple local definitions* (i.e., terms of the shape $\mathsf{let}\, x = e_0 \,\mathsf{in}\, e_1$ where e_0 is non-simple) and *non-simple conditional expressions* (i.e., non-simple terms of the shape $\mathsf{if}\, e_0 \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$). In this paper we propose new rules and algorithms for typing *non-simple recursive definitions* (i.e., non-simple terms of the shape $\mathsf{rec}\, \{x = e\}$). At the best of our knowledge, previous algorithms for typing recursive definitions in the presence of rank-2 intersection types [12,13,21,6] allow only simple recursive definitions to be typed.

Note that, the ability of typing non-simple recursive definitions is not, a priori, a fair measure for the expressive power of a system. It fact, it might be possible that a given system could type a term that is non-simple *in the given system* only because the given system somehow prevents the term from having a simple type. In another system that allows only simple terms to be typed, the same term might have a simple type and therefore be simple. When designing the new rules we will be careful to avoid such a pathological situation.

Inferring types for *polymorphic recursion* (i.e., recursive definitions $\mathsf{rec}\, \{x = e\}$ where different occurrences of x in e are used with different types) [17,18,10,15] is a recurring topic on the mailing list of popular typed programming languages (see, e.g., [9] for a discussion of several examples). The rules proposed in this paper are able to type interesting examples of polymorphic recursion. So, besides providing a solution to the problem of finding decidable rules for typing non-simple recursive definitions, the techniques proposed in this paper address also the related topic of polymorphic recursion. Moreover (as we will point out in Section 8), these techniques do not depend on particulars of rank-2 intersection. So they can be applied to other type systems.

Organization of the Paper. Section 2 introduces a small functional programming language, which can be considered the kernel of languages like SML, OCaml, and Haskell. Section 3 introduces some basic definitions. Section 4 presents the rank-2 intersection type system ($\vdash_2$) for the rec-free subset of the language. Section 5 briefly reviews the rules for typing recursive definitions in the presence of rank-2 intersection types that have been proposed in the literature. Section 6 extends $\vdash_2$ with new rules for typing non-simple recursive definitions and Section 7 outlines how to adapt the rules to deal with mutual recursion. We conclude by discussing some further work.

The proofs and an on-line demonstration of a prototype implementation of the typing algorithm are available at the url http://lambda.di.unito.it/pr.

2 A Small ML-like Language

We consider a quite rich set of constants (that will be useful to formulate the examples considered through the paper) including the booleans, the integer numbers, the constructors for pairs and lists, some logical and arithmetic operators, and the functions for decomposing pairs (fst and snd) and lists (null, hd and tl). The syntax of constants (ranged over by c) is as follows:

$$c ::= \flat \mid \iota \mid \mathsf{pair} \mid \mathsf{nil} \mid \mathsf{cons} \mid \mathsf{not} \mid \mathsf{and} \mid \mathsf{or} \mid + \mid - \mid * \mid = \mid < \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{null} \mid \mathsf{hd} \mid \mathsf{tl}$$

where $\flat$ ranges over booleans (true and false) and ι ranges over integer numbers.

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

$$e ::= x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{rec}\,\{x = e\} \mid \mathsf{let}\,x = e_0\,\mathsf{in}\,e_1 \mid \mathsf{if}\,e_0\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2,$$

where x ranges over variables. The finite set of the free variables of an expression e is denoted by $\mathrm{FV}(e)$.

3 Basic Definitions

In this section we introduce the syntax of our rank-2 intersection types, together with other basic definitions that will be used in the rest of the paper.

We will be defining several classes of types. The set of *simple types* ($\mathbf{T}_0$), ranged over by u, the set of *rank-1 intersection types* ($\mathbf{T}_1$), ranged over by w, and the set of *rank-2 intersection types* ($\mathbf{T}_2$), ranged over by v, are defined by the pseudo-grammar:

$$
\begin{array}{llll}
u & ::= \alpha \mid u_1 \to u_2 \mid \mathsf{bool} \mid \mathsf{int} \mid u_1 \times u_2 \mid u\,\mathsf{list} & & \text{(simple types)} \\
w & ::= u_1 \wedge \cdots \wedge u_n & & \text{(rank-1 types)} \\
v & ::= u \mid w \to v & & \text{(rank-2 types)}
\end{array}
$$

where $n \geq 1$. We have *type variables* (ranged over by α), arrow types, and a selection of *ground types* and *parametric datatypes*. The ground types are bool (the type of booleans) and int (the type of integers). The other types are pair types and list types. Note that $\mathbf{T}_0 = \mathbf{T}_1 \cap \mathbf{T}_2$ (for technical convenience, following [13] and other papers, rank-1 types are not included into rank-2 types).

The constructor $\to$ is right associative, e.g., $u_1 \to u_2 \to u_3$ means $u_1 \to (u_2 \to u_3)$, and the constructors $\times$ and list bind more tightly than $\to$, e.g., $u_1 \to u_2 \times u_3$ means $u_1 \to (u_2 \times u_3)$. We consider $\wedge$ to be associative, commutative, and idempotent. Therefore any type in $\mathbf{T}_1$ can be considered (modulo elimination of duplicates) as a set of types in $\mathbf{T}_0$. The constructor $\wedge$ binds more tightly than $\to$, e.g., $u_1 \wedge u_2 \to u_3$ means $(u_1 \wedge u_2) \to u_3$, and less tightly than $\times$ and list (which can be applied only to simple types).

We assume a countable set 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 application of a substitution $\mathbf{s}$ to a type t, denoted by $\mathbf{s}(t)$, is defined as usual. Note that, since substitutions replace type variables by simple types, we have that $\mathbf{T}_0$, $\mathbf{T}_1$, and $\mathbf{T}_2$ are closed under substitution.

An *environment* T is a set $\{x_1 : t_1, \ldots, x_n : t_n\}$ of assumptions for variables such that every variable x_i $(1 \leq i \leq n)$ can occur at most once in T. The expression $\mathrm{Dom}(T)$ denotes the *domain* of T, which is the set $\{x_1, \ldots, x_n\}$. We write $T, x : t$ for the environment $T \cup \{x : t\}$ where it is assumed that $x \notin \mathrm{Dom}(T)$. The application of a substitution $\mathbf{s}$ to an environment T, denoted by $\mathbf{s}(T)$, is defined as usual.

Definition 2 (Rank-1 environments). A *rank-1 environment* A is an environment $\{x_1 : w_1, \ldots, x_n : w_n\}$ of rank-1 type assumptions for variables.

Given two rank-1 environments A_1 and A_2, we write $A_1 \wedge A_2$ to denote the rank-1 environment:

$$\{x : w_1 \wedge w_2 \mid x : w_1 \in A_1 \text{ and } x : w_2 \in A_2\}$$
$$\cup \{x : w_1 \in A_1 \mid x \notin \mathrm{Dom}(A_2)\} \cup \{x : w_2 \in A_2 \mid x \notin \mathrm{Dom}(A_1)\}.$$

A *pair* is a formula $\langle A; v \rangle$ where A is a rank-1 environment and v is a rank-2 type. Let p range over pairs.

The following relation is fairly standard.

Definition 3 (Pair specialization relation $\leq_{\mathbf{spc}}$). The subtyping relations $\leq_1$ ($\subseteq \mathbf{T}_1 \times \mathbf{T}_1$) and $\leq_2$ ($\subseteq \mathbf{T}_2 \times \mathbf{T}_2$) are defined by the following rules.

$$(\wedge) \ \frac{\cup_{1 \leq i \leq n}\{u_i\} \supseteq \cup_{1 \leq j \leq m}\{u'_j\}}{u_1 \wedge \cdots \wedge u_n \leq_1 u'_1 \wedge \cdots \wedge u'_m} \qquad (\mathrm{REF}) \ \frac{u \in \mathbf{T}_0}{u \leq_2 u} \qquad (\rightarrow) \ \frac{w' \leq_1 w \quad v \leq_2 v'}{w \rightarrow v \leq_2 w' \rightarrow v'}$$

Given two rank-1 environments A and A' we write $A \leq_1 A'$ to mean that

- $\mathrm{Dom}(A) = \mathrm{Dom}(A')$,[1] and
- for every assumption $x : w' \in A'$ there is an assumption $x : w \in A$ such that $w \leq_1 w'$.

A pair $\langle A; v \rangle$ *can be specialized to* $\langle A'; v' \rangle$ (notation $\langle A; v \rangle \leq_{\mathbf{spc}} \langle A'; v' \rangle$) if $A' \leq_1 \mathbf{s}(A)$ and $\mathbf{s}(v) \leq_2 v'$, for some substitution $\mathbf{s}$.

Example 4. We have $\langle \{y : \beta\}; \alpha \rightarrow \beta \rangle \leq_{\mathbf{spc}} \langle \{y : \gamma\}; ((\gamma \rightarrow \gamma) \wedge \gamma) \rightarrow \gamma \rangle$.

Note that the relation $\leq_{\mathbf{spc}}$ is reflexive and transitive.

4 Typing the "rec-free" Fragment of the Language

In this section we introduce the type system $\vdash_2$ for the "rec-free" fragment of the language (i.e., the fragment without recursive definitions). System $\vdash_2$ is just a reformulation of Jim's system $\mathbf{P}_2$ [13].

---

[1] The requirement $\mathrm{Dom}(A) = \mathrm{Dom}(A')$ in the definition of $A \leq_1 A'$ is not present in most other papers. The "usual" definition drops this requirement, thus allowing $\mathrm{Dom}(A) \supseteq \mathrm{Dom}(A')$. We have added such a requirement since it will simplify the presentation of the new typing rules for recursive definitions (in Section 6).

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			tl	α list $\to \alpha$ list

Fig. 1. Types for constants

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

$$(\textsc{App})\ \frac{\vdash e : \langle A; u_1 \wedge \cdots \wedge u_n \to v \rangle \qquad \vdash e_0 : \langle A_1; u_1 \rangle \quad \cdots \quad \vdash e_0 : \langle A_n; u_n \rangle}{\vdash e\, e_0 : \langle A \wedge A_1 \wedge \cdots \wedge A_n; v \rangle}$$

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

Fig. 2. Typing rules for the rec-free fragment of the language (system $\vdash_2$)

4.1 System $\vdash_2$

Following Wells [22], in the type inference system $\vdash_2$ we use typing judgements of the shape $\vdash_2 e : \langle A; v \rangle$, instead of the more usual notation $A \vdash_2 e : v$. This slight change of notation will simplify the presentation of the new typing rules for recursive definitions (in Section 6). The judgement $\vdash_2 e : \langle A; v \rangle$ means "e is $\vdash_2$-typable with pair $\langle A; v \rangle$", where

- A is a rank-1 environment containing the type assumptions for the free variables of e, and
- v is a rank-2 type.

In any valid judgement $\vdash_2 e : \langle A; v \rangle$ it holds that $\mathrm{Dom}(A) = \mathrm{FV}(e)$.

The typing rules of system $\vdash_2$ are given in Fig. 2. Rule (Spc), which is the only non-structural rule, allows to specialize (in the sense of Definition 3) the pair inferred for an expression.

The rule for typing constants, (Con), uses the function **type** (tabulated in Figure 1) 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 pair $\langle \emptyset; \mathbf{type}(c) \rangle$.

Since $\vdash_2 e : \langle A; v \rangle$ implies $\mathrm{Dom}(A) = \mathrm{FV}(e)$, 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 $x \notin \mathrm{FV}(e)$. Note that, by rule (Var), it is possible to assume a different simple type for each occurrence of the λ-bound variable.

The rule for typing function application, (APP), allows to use a different pair for each expected type of the argument (the operator $\wedge$ on rank-1 environments has been defined immediately after Definition 2).

To save space, we have omitted the typing rule for local definitions, which handles expressions "let $x = e_0$ in e_1" like syntactic sugar for "$(\lambda x.e_1)\,e_0$", and the typing rule for for conditional expressions, which handles expressions "if e_0 then e_1 else e_2" like syntactic sugar for the application "ifc $e_0\,e_1\,e_2$", where ifc is a special constant of type bool $\rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$. Note that, according to these rules, only simple local definitions and conditional expressions can be typed.

4.2 Principal Pairs and Decidability for $\vdash_2$

Definition 5 (Principal pairs). Let $\vdash$ be a type system with judgements of the shape $\vdash e : p$. A pair p is *principal for a term e* if $\vdash e : p$, and if $\vdash e : p'$ implies $p \leq_{\mathbf{spc}} p'$. We say that system $\vdash$ has the *principal pair property* to mean that every typable term has a principal pair.

System $\vdash_2$ has the principal pair property and is decidable (see, e.g., [13]).

5 Typing Simple Recursive Definition

In this section we briefly recall the typing rules for recursive definitions proposed by Jim [12,13] (see also [6]) which, at the best of our knowledge, are the more powerful rules for typing recursive definitions in presence of rank-2 intersection types that can be found in the literature.

In order to be able to formulate these typing rules we need some auxiliary definitions. The set of *rank-2 intersection type schemes* ($\mathbf{T}_{\forall 2}$), ranged over by vs, is defined by the following pseudo-grammar:

$$vs ::= \forall \overrightarrow{\alpha}.v \qquad \text{(rank-2 schemes)},$$

where $\overrightarrow{\alpha}$ denotes a finite (possibly empty) unordered sequence of distinct type variables $\alpha_1 \cdots \alpha_m$ and $v \in \mathbf{T}_2$. Let ϵ denote the empty sequence, $\forall \epsilon.v$ is a legal expression, syntactically different from v, so that $\mathbf{T}_2 \cap \mathbf{T}_{\forall 2} = \emptyset$. *Free* and *bound* type variables are defined as usual. For every type or scheme $t \in \mathbf{T}_1 \cup \mathbf{T}_2 \cup \mathbf{T}_{\forall 2}$ let $\mathrm{FTV}(t)$ denote the set of free type variables of t. For every scheme $\forall \overrightarrow{\alpha}.v$ it is assumed that $\{\overrightarrow{\alpha}\} \subseteq \mathrm{FTV}(v)$. Moreover, schemes are considered equal modulo renaming of bound type variables. Given a type $v \in \mathbf{T}_2$ and a rank-1 environment A, we write $\mathrm{Gen}(A, v)$ for the $\forall$-*closure of v in A*, that is the rank-2 scheme $\forall \overrightarrow{\alpha}.v$, where $\overrightarrow{\alpha}$ is the sequence of the type variables in $\mathrm{FTV}(v) - \cup_{x:w \in A}\mathrm{FTV}(w)$.

Definition 6 (Scheme instantiation relation). For every rank-2 scheme $\forall \overrightarrow{\alpha}.v$ and for every rank-1 type $u_1 \wedge \cdots \wedge u_n$, let $\forall \overrightarrow{\alpha}.v \leq_{\forall 2,1} u_1 \wedge \cdots \wedge u_n$ mean that, for every $i \in \{1, \ldots, n\}$, $u_i = \mathbf{s}_i(v)$, for some substitution $\mathbf{s}_i$ such that $\mathbf{s}_i(\beta) = \beta$ for all $\beta \notin \overrightarrow{\alpha}$.

Example 7. We have (remember that $\wedge$ is idempotent): $\forall \alpha\beta\gamma.((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma)) \rightarrow \alpha \rightarrow \gamma \leq_{\forall 2,1} ((\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}) \wedge ((\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool} \rightarrow \text{bool})$.

We can now present the typing rules. All these rules type recursive definitions $\mathsf{rec}\,\{x = e\}$ by assigning simple types to the occurrences of x in e (each occurrence may be assigned a different simple type). Therefore, they allow only simple (in the sense of Definition 1) *non-vacuous* recursive definitions to be typed (we say that a recursive definition $\mathsf{rec}\,\{x = e\}$ is *vacuous* to mean that $x \notin \mathrm{FV}(e)$). Jim [12] proposed the following rules for typing recursive definitions:[2]

$$(\textsc{Rec})\ \dfrac{\vdash e : \langle A, x : w;\, v\rangle}{\vdash \mathsf{rec}\,\{x = e\} : \langle A;\, v\rangle} \qquad\qquad (\textsc{RecVac})\ \dfrac{\vdash e : p}{\vdash \mathsf{rec}\,\{x = e\} : p}$$
$$\text{where } \mathrm{Gen}((A, x : w),\, v) \leq_{\forall 2,1} w \qquad\qquad \text{where } x \notin \mathrm{FV}(e)$$

These two rules corresponds to the two cases $x \in \mathrm{FV}(e)$ and $x \notin \mathrm{FV}(e)$. Note that the rule for non-vacuous recursion, (\textsc{Rec}), requires that the rank-2 type v assigned to $\mathsf{rec}\,\{x = e\}$ must be such that $\mathrm{Gen}((A, x : w),\, v) \leq_{\forall 2,1} w$. I.e., $\mathbf{s}(v) = u_i$ for some substitution $\mathbf{s}$ and simple type u_i such that $\mathbf{s}(A) = A$ and $w = u_1 \wedge \cdots \wedge u_i \wedge \cdots \wedge u_n$ $(1 \leq i \leq n)$. This implies $\vdash \mathsf{rec}\,\{x = e\} : \langle A;\, u_i\rangle$. Therefore, rule (\textsc{Rec}) allows only simple recursive definitions to be typed. System $\vdash_2 + (\textsc{Rec}) + (\textsc{RecVac})$ has the principal pair property and is decidable (see, e.g., [12,6]).

As pointed out by Jim [13], rule (\textsc{Rec}) might be generalized along the lines of Mycroft's rule (\textsc{Fix}$'$) [18]: just replace the condition $\mathrm{Gen}((A, x : w),\, v) \leq_{\forall 2,1} w$ by the condition $\mathrm{Gen}(A,\, v) \leq_{\forall 2,1} w$. Let us call (\textsc{Rec}$'$) the generalized rule. Rule (\textsc{Rec}$'$) is strictly more powerful than rule (\textsc{Rec}) (see Section 3 of [13] for an example) but, again, it allows only simple recursive definitions to be typed.

Jim [12] proposed also the following rule:[2]

$$(\textsc{RecInt})\ \dfrac{\vdash e : \langle A, x : u_1 \wedge \cdots \wedge u_m;\, u_1\rangle \quad \cdots \quad \vdash e : \langle A, x : u_1 \wedge \cdots \wedge u_m;\, u_m\rangle}{\vdash \mathsf{rec}\,\{x = e\} : \langle A;\, u_{i_0}\rangle}$$
$$\text{where } i_0 \in \{1, \ldots, m\}$$

The decidability of $\vdash_2 + (\textsc{RecInt})$ is an open question (there is no obvious way to find an upper bound on the value of m used in the rule) [12]. Note that, since $u_1, \cdots, u_m \in \mathbf{T}_0$ (and, in particular, $u_{i_0} \in \mathbf{T}_0$), also this rule allows only simple recursive definitions to be typed.

6 Typing Non-Simple Recursive Definitions

In this section we extend system $\vdash_2$ to type non-simple recursive definitions.

6.1 System $\vdash_2^{\mathbf{P}}$

In order to be able to type non-simple recursive definitions, we propose to adopt the following strategy: allow one to assign to $\mathsf{rec}\,\{x = e\}$ any pair p that can be assigned to e by assuming the pair p itself for x.

To implement the above strategy, we introduce the notion of *pair environment* (taken from [6]).

---

[2] We still give these rules the original name used in [12], but we adapt them to fit in the type assignment system $\vdash_2$.

(SPC) $\dfrac{D \vdash e : p}{D \vdash e : p'}$ (CON) $D \vdash c : \langle \emptyset; v \rangle$ (VAR) $D \vdash x : \langle \{x : u\}; u \rangle$

    where $p \leq_{\mathbf{spc}} p'$ where $v = \mathbf{type}(c)$ where $u \in \mathbf{T}_0$ and $x \notin \mathrm{Dom}(D)$

(APP) $\dfrac{D \vdash e : \langle A; u_1 \wedge \cdots \wedge u_n \rightarrow v \rangle \qquad D \vdash e_0 : \langle A_1; u_1 \rangle \quad \cdots \quad D \vdash e_0 : \langle A_n; u_n \rangle}{D \vdash e\, e_0 : \langle A \wedge A_1 \wedge \cdots \wedge A_n; v \rangle}$

(ABS) $\dfrac{D \vdash e : \langle A, x : w; v \rangle}{D \vdash \lambda\, x.e : \langle A; w \rightarrow v \rangle}$ (ABSVAC) $\dfrac{D \vdash e : \langle A; v \rangle}{D \vdash \lambda\, x.e : \langle A; u \rightarrow v \rangle}$

    where $x \notin D$ where $x \notin \mathrm{FV}(e)$, $u \in \mathbf{T}_0$, and $x \notin D$

(VARP) $D, x : p \vdash x : p$ (RECP) $\dfrac{D, x : \langle A; v \rangle \vdash e : \langle A; v \rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle A; v \rangle}$

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

Fig. 3. Typing rules of system $\vdash_2^{\mathrm{P}}$

Definition 8 (Pair environments). A *pair environment D* is an environment $\{x_1 : p_1, \ldots, x_n : p_n\}$ of pair assumptions for variables such that $\mathrm{Dom}(D) \cap \mathrm{VR}(D) = \emptyset$, where $\mathrm{VR}(D) = \cup_{x:\langle A; v \rangle \in D}\mathrm{Dom}(A)$ is the *set of variables occurring in the range of D*. Every pair p occurring in D is implicitly universally quantified over all type variables occurring in p.[3]

The typing rules of system $\vdash_2^{\mathrm{P}}$ (where "P" stands for "polymorphic") are given in Fig. 3. The judgement $D \vdash_2^{\mathrm{P}} e : \langle A; v \rangle$ means "e is $\vdash_2^{\mathrm{P}}$-typable in D with pair $\langle A; v \rangle$", where

- D is a pair environment specifying pair assumptions for the variables introduced by a rec-binder in the program context surrounding e,
- $\langle A; v \rangle$ is the pair inferred for e, where A is a rank-1 environment containing the type assumptions for the free variables of e which are not in $\mathrm{Dom}(D)$, and v is a rank-2 type.

Let D be a pair environment, "$x \notin D$" is short for "$x \notin \mathrm{Dom}(D) \cup \mathrm{VR}(D)$" and $\mathrm{FV}_D(e) = (\mathrm{FV}(e) - \mathrm{Dom}(D)) \cup \mathrm{VR}(\{x : p \in D \mid x \in \mathrm{FV}(e)\})$ is the *set of the free variables of the expression e in D*. In any valid judgement $D \vdash_2^{\mathrm{P}} e : \langle A; v \rangle$ it holds that $\mathrm{Dom}(D) \cap \mathrm{Dom}(A) = \emptyset$ and $\mathrm{Dom}(A) = \mathrm{FV}_D(e)$.

Rules (SPC), (CON), (VAR), (ABS), (ABSVAC), and (APP) are just the rules of system $\vdash_2$ (in Fig. 2) modified by adding the pair environment D on the left of the typing judgements and, when necessary, side conditions (like "$x \notin \mathrm{Dom}(D)$" in rule (VAR)) to ensure that $\mathrm{Dom}(D) \cap \mathrm{Dom}(A) = \emptyset$.

Rule (RECP) allows to assign to a recursive definition $\mathsf{rec}\,\{x = e\}$ any pair p that can be assigned to e by assuming the pair p for x. Note that the combined use of rules (VARP) and (SPC) allows to assign a different specialization of p to each occurrence of x in e.

---

[3] To emphasize this fact the paper [6] uses *pair schemes*. I.e., formulae of the shape $\forall \overrightarrow{\alpha}.p$, where $\overrightarrow{\alpha}$ is the sequence of *all* the type variables occurring in the pair p.

6.2 On the Expressive Power of System $\vdash^P_2$

System $\vdash^P_2$ is able to type non-simple recursive definitions, as illustrated by the following example (for more interesting examples see http://lambda.di.unito.it/pr).

Example 9. The recursive definition $\mathsf{rec}\,\{f = e\}$, where

$$e \;=\; \lambda\, g\, l.\, \mathsf{if}\ (\mathsf{null}\ l)\ \mathsf{then}\ \mathsf{nil}\ \mathsf{else}\ \mathsf{cons}\ (\mathsf{pair}\ (g\,(\mathsf{hd}\ l)\,5)\ (g\,y\,\mathsf{true}))\ (f\,g\,(\mathsf{tl}\ l)),$$

is non-simple since it defines a function that uses its parameter g with two different non unifiable types. The following $\vdash^P_2$-typing judgement holds.

$$\emptyset \vdash \mathsf{rec}\,\{f = e\} : \langle \{y : \alpha_2\};\ ((\alpha_1 \to \mathsf{int} \to \beta_1) \wedge (\alpha_2 \to \mathsf{bool} \to \beta_2)) \to \alpha_1\ \mathsf{list} \to (\beta_1 \times \beta_2)\ \mathsf{list}\rangle.$$

System $\vdash^P_2$ has more expressive power than the system $\vdash_2 + (\mathrm{REC}) + (\mathrm{RECVAC})$ of Section 5 (and therefore of system $\mathbf{P}^R_2$ [13,12]) and of the Milner-Mycroft system [18] (see also [17]), in the sense that the set of typable terms increases, and types express better the behaviour of terms. In particular, the following theorems hold.

Theorem 10. *If $\vdash_2 + (\mathrm{REC}) + (\mathrm{RECVAC})\ e : \langle A;\, v\rangle$, then $\emptyset \vdash^P_2 e : \langle A;\, v\rangle$.*

Theorem 11. *If $\emptyset \vdash e : u$ is Milner-Mycroft derivable, then $\emptyset \vdash^P_2 e : \langle \emptyset;\, u\rangle$.*

6.3 Principal-in-D Pairs and (Un)decidability of $\vdash^P_2$

The following notion of *principal-in-D pair* (taken from [6]) adapts the notion of principal pair (see Definition 5) to deal with the pair environment D.

Definition 12 (Principal-in-D pairs). Let $\vdash$ be a system with judgements of the shape $D \vdash e : p$. A pair p is *principal-in-D for a term e* if $D \vdash e : p$, and if $D \vdash e : p'$ implies $p \leq_{\mathbf{spc}} p'$. We say that system $\vdash$ has the *principal-in-D pair property* to mean that every typable term has a principal-in-D pair.

We don't know whether system $\vdash^P_2$ has the principal-in-D pair property. The following theorem implies that the restriction of $\vdash^P_2$ which uses only simple types, that we will call $\vdash^P_0$, is undecidable (as is the Milner-Mycroft system [10,15]). We conjecture that also the whole $\vdash^P_2$ is undecidable.

Theorem 13. *Let e be a let-free expression. Then $\emptyset \vdash^P_0 e : \langle \emptyset;\, u\rangle$ iff $\emptyset \vdash e : u$ is Milner-Mycroft derivable.*

6.4 Systems $\vdash^{P_k + -}_2 (k \geq 1)$: a Family of Decidable Restrictions of $\vdash^P_2$

By taking inspiration from the idea of iterating type inference (see, e.g., [18,17]) and by relying on the notion of principal-in-D pair (see Definition 12) we will now design a family of decidable restrictions of rule (RECP).

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

- $\mathsf{P}_X = \{\langle A;\, v\rangle \mid \langle A;\, v\rangle$ is pair such that $\mathrm{Dom}(A) = X\}$, and

- $\mathsf{bot}_X = \langle \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}; \alpha \rangle$, where the type variables $\alpha_1, \ldots, \alpha_n, \alpha$ are all distinct.

The relation $\leq_{\mathbf{spc}}$ is a preorder over P_X and, for all pairs $p \in \mathsf{P}_X$, $\mathsf{bot}_X \leq_{\mathbf{spc}} p$.

For every $k \geq 1$, let $\vdash_2^{\mathrm{P}_k}$ be the system obtained from $\vdash_2^{\mathrm{P}}$ by replacing rule (RECP) with the following rule:

$$(\mathrm{RECP}_k) \quad \frac{D, x : p_0 \vdash e : p_1 \quad \cdots \quad D, x : p_{k-1} \vdash e : p_k}{D \vdash \mathsf{rec}\,\{x = e\} : p_k}$$
$$\text{where} \quad x \notin D, \ p_0 = \mathsf{bot}_{\mathrm{FV}(\mathsf{rec}\,\{x=e\})}, \ p_{k-1} = p_k, \text{ and}$$
$$\text{for all } i \in \{1, \ldots, k\} \ p_i \text{ is a principal-in-}(D, x : p_{i-1}) \text{ pair for } e$$

(note that $D \vdash_2^{\mathrm{P}_k} e : p$ implies $D \vdash_2^{\mathrm{P}_{k+1}} e : p$). For all $k \geq 1$, system $\vdash_2^{\mathrm{P}_k}$ has the principal-in-D pair property and is decidable — the result follows from Theorem 20 (soundness and completeness of the inference algorithm for $\vdash_2^{\mathrm{P}_k+\mathrm{J}}$, which is an extension of $\vdash_2^{\mathrm{P}_k}$) of Section 6.5. The relation between system $\vdash_2^{\mathrm{P}_k}$ and system $\vdash_2^{\mathrm{P}}$ is stated by the following theorem which, roughly speaking, says that when rule (RECP_k) works at all, it works as well as rule (RECP) does.

Theorem 14. *For every $k \geq 1$:*
1. *If $D \vdash_2^{\mathrm{P}_k} e : p$, then $D \vdash_2^{\mathrm{P}} e : p$.*
2. *If e is $\vdash_2^{\mathrm{P}_k}$-typable in D and $D \vdash_2^{\mathrm{P}} e : p$, then $D \vdash_2^{\mathrm{P}_k} e : p$.*

Unfortunately, for every $k \geq 1$, system $\vdash_2^{\mathrm{P}_k}$ is not able to type all the ML-typable recursive definitions (see Example 15 below).

Example 15 (Rule (RECP_k) and the ML rule are incomparable). The expression $\mathsf{rec}\,\{f = e\}$ of Example 9 in Section 6.2 is non-simple, therefore it is not ML-typable. Since

$$p = \langle \{y : \alpha_2\}; ((\alpha_1 \to \mathsf{int} \to \beta_1) \wedge (\alpha_2 \to \mathsf{bool} \to \beta_2)) \to \alpha_1\,\mathsf{list} \to (\beta_1 \times \beta_2)\,\mathsf{list} \rangle,$$

is both a principal-in-$\{f : \langle \{y : \alpha_1\}; \alpha \rangle\}$ and a principal-in-$\{f : p\}$ pair for e in system $\vdash_2^{\mathrm{P}_2}$, we have that p is a principal-in-$\emptyset$ pair for $\mathsf{rec}\,\{f = e\}$ in system $\vdash_2^{\mathrm{P}_k}$ (for all $k \geq 2$).

The expression $\mathsf{rec}\,\{f = e'\}$, where $e' = \lambda g\,y.\,\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ y\ \mathsf{else}\ g(f\,g\,y)$, is ML-typable with principal pair $\langle \emptyset; (\alpha \to \alpha) \to \alpha \to \alpha \rangle$ and is not $\vdash_2^{\mathrm{P}_k}$-typable (for all $k \geq 1$) — in fact, for every $k \geq 1$, the expression e' has principal-in-$\{f : p_{k-1}\}$ pair

$$p_k = \langle \emptyset; ((\alpha_0 \to \alpha_1) \wedge (\alpha_1 \to \alpha_2) \wedge \cdots \wedge (\alpha_{k-1} \to \alpha_k)) \to (\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_k) \to \alpha_k \rangle$$

in system $\vdash_2^{\mathrm{P}}$.

Note that $\mathsf{rec}\,\{f = e'\}$ is $\vdash_2^{\mathrm{P}}$-typable with pair $\langle \emptyset; ((\alpha \to \gamma) \wedge (\gamma \to \alpha) \wedge (\alpha \to \beta)) \to (\alpha \wedge \beta \wedge \gamma) \to \beta \rangle$.

We will now show that, for all $k \geq 1$, it is quite easy to modify system $\vdash_2^{\mathrm{P}_k}$ in order to make it to extend the ML system while preserving decidability and principal-in-D pair property. To this aim, we will say that *a typing rule for recursive definitions* (REC_-) *is $\vdash_2^{\mathrm{P}}$-suitable* to mean that: the system $\vdash_{\overline{2}}$ obtained from $\vdash_2^{\mathrm{P}}$ by replacing rule (RECP) with rule (REC_-)

- is a restriction of system $\vdash^P_2$ (i.e., $D \vdash_{\bar 2} e : \langle A; v \rangle$ implies $D \vdash^P_2 e : \langle A; v \rangle$),
- is decidable, and
- has the principal-in-D pair property.

For instance, for every $k \geq 1$, rule $(\textsc{RecP}_k)$ is $\vdash^P_2$-suitable. Theorem 14 guarantees that, for all $k \geq 1$, adding to system $\vdash^{P_k}_2$ a $\vdash^P_2$-suitable rule $(\textsc{Rec}_\text{-})$ results in a system, denoted by $\vdash^{P_k+\text{-}}_2$, with both decidability and principal-in-D pair property. So, to extend system $\vdash^{P_k}_2$ to type all the ML typable recursive definitions, we have just to add to system $\vdash^{P_k}_2$ a $\vdash^P_2$-suitable rule which 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 $\vdash^{P_k}_2$, of) the ML rule itself:

$$(\textsc{RecML})\ \frac{D \vdash e : \langle A, x : u; u \rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle A; u \rangle}\ \text{where } x \notin D$$

(producing system $\vdash^{P_k+\text{ML}}_2$). Another possibility, is to add the following rule

$$(\textsc{RecJ})\ \frac{D \vdash e : \langle A, x : w; v \rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle A; v \rangle}\ \text{where } \text{Gen}((A, x : w), v) \leq_{\forall 2,1} w \text{ and } x \notin D$$

which corresponds to rule $(\textsc{Rec})$ of Section 5. In this way we obtain a system, $\vdash^{P_k+\text{J}}_2$, which is more expressive that system $\vdash_2 +(\textsc{Rec})+(\textsc{RecVac})$ of Section 5 (observe that $\vdash_2 +(\textsc{RecVac})$-typabily implies $\vdash^{P_k}_2$-typability).

Example 16. The expression $\mathsf{rec}\,\{f = e'\}$ of Example 15, which is not $\vdash^{P_k}_2$-typable (for all $k \geq 1$), has principal-in-$\emptyset$ pairs $\langle \emptyset; (\alpha \to \alpha) \to \alpha \to \alpha \rangle$ and $\langle \emptyset; ((\alpha \to \alpha) \wedge (\alpha \to \beta)) \to (\alpha \wedge \beta) \to \beta \rangle$ in systems $\vdash^{P_1+\text{ML}}_2$ and $\vdash^{P_1+\text{J}}_2$, respectively.

6.5 An Inference Algorithm for $\vdash^{P_k+\text{J}}_2$ $(k \geq 1)$

The inference algorithm makes use of an algorithm for checking whether the $\leq_{\textbf{spc}}$ relation (see Definition 3) holds and of an algorithm for finding a most general solution to a $\leq_{\forall 2,1}$-*satisfaction problem* ($\leq_{\forall 2,1}$ is the relation of Definition 6).

Existence of an algorithm for checking whether the $\leq_{\textbf{spc}}$ relation holds is stated by the following theorem.

Theorem 17 (Decidability of $\leq_{\textbf{spc}}$). *There is an algorithm that, for every p and p', decides whether $p \leq_{\textbf{spc}} p'$ holds.*

A $\leq_{\forall 2,1}$-*satisfaction problem* [13,12] (see also [6]) is a formula $\exists \overrightarrow{\alpha}.P$, where $\overrightarrow{\alpha}$ is a (possibly empty) sequence of type variables occurring free in P, and P is a set in which every element is either: 1) an equality between $\textbf{T}_0$ types; or 2) an inequality between a $\textbf{T}_{\forall 2} \cup \textbf{T}_2$ type and a $\textbf{T}_1$ type. A substitution $\textbf{s}$ is a *solution* to $\exists \overrightarrow{\alpha}.P$ if there exists a substitution $\textbf{s}'$ such that: $\textbf{s}(\alpha) = \textbf{s}'(\alpha)$ for all $\alpha \notin \overrightarrow{\alpha}$, $\textbf{s}'(u_1) = \textbf{s}'(u_2)$ for every equality $(u_1 = u_2) \in P$, and $\textbf{s}'(vs) \leq_{\forall 2,1} \textbf{s}'(w)$ (resp. $\textbf{s}'(\forall \epsilon.v) \leq_{\forall 2,1} \textbf{s}'(w)$) for every inequality $(vs \leq w) \in P$ (resp. $(v \leq w) \in P$). We will write $\textbf{MGS}(\exists \overrightarrow{\alpha}.P)$ for the set of most general solutions to the $\leq_{\forall 2,1}$-satisfaction problem $\exists \overrightarrow{\alpha}.P$ (a $\leq_{\forall 2,1}$-satisfaction problem $\exists \overrightarrow{\alpha}.P$ generalizes unification and, as with unification, most general solutions are not unique). Existence

of an algorithm for finding a most general solution to a $\leq_{\forall 2,1}$-satisfaction problem is stated by the following theorem [13,12] (see also [6]).

Theorem 18. *There is an algorithm that decides, for any $\leq_{\forall 2,1}$-satisfaction problem, whether it is solvable, and, if so, returns a most general solution.*

The inference algorithm is presented in a style favored by the intersection type community. For all $k \geq 1$, we define a function $\mathbf{PP}_k$ which, for every expression e and environment D, returns a set of pairs $\mathbf{PP}_k(D, e)$ such that

- if $\mathbf{PP}_k(D, e) = \emptyset$, then e can not be typed by $\vdash_2^{\mathrm{P}_k+\mathrm{J}}$ w.r.t. D, and
- every element of $\mathbf{PP}_k(D, e)$ is a principal pair for e w.r.t. D.[4]

Definition 19 (The function $\mathbf{PP}_k$). For every expression e and environment D, the set $\mathbf{PP}_k(D, e)$ is defined by structural induction on e.

- If $e = x$, then
  - If $x : \langle A; v \rangle \in D$ and the substitution $\mathbf{s}$ is a fresh renaming of $\overrightarrow{\alpha} = \mathrm{FTV}(A) \cup \mathrm{FTV}(v)$, then $\langle \mathbf{s}(A); \mathbf{s}(v) \rangle \in \mathbf{PP}_k(D, x)$.
  - If $x \notin \mathrm{Dom}(D)$ and α is a type variable, then $\langle \{x : \alpha\}; \alpha \rangle \in \mathbf{PP}_k(D, x)$.
- If $e = c$ and $\mathbf{type}(c) = v$, then $\langle \emptyset; v \rangle \in \mathbf{PP}_k(D, c)$.
- If $e = \lambda x.e_0$ and $\langle A; v \rangle \in \mathbf{PP}_k(D, e_0)$, then
  - If $x \notin \mathrm{FV}(e_0)$ and α is a fresh type variable, then $\langle A; \alpha \rightarrow v \rangle \in \mathbf{PP}_k(D, \lambda x.e_0)$.
  - If $x \in \mathrm{FV}(e_0)$ and $A = A', x : w$, then $\langle A'; w \rightarrow v \rangle \in \mathbf{PP}_k(D, \lambda x.e_0)$.
- If $e = e_0 e_1$ and $\langle A_0; v_0 \rangle \in \mathbf{PP}_k(D, e_0)$, then
  - If $v_0 = \alpha$ (a type variable), α_1 and α_2 are fresh type variables, $\langle A_1; v_1 \rangle \in \mathbf{PP}_k(D, e_1)$ is fresh, and $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon.\{v_1 \leq \alpha_1, \alpha = \alpha_1 \rightarrow \alpha_2\})$, then $\langle \mathbf{s}(A_0 \wedge A_1); \mathbf{s}(\alpha_2) \rangle \in \mathbf{PP}_k(D, e_0 e_1)$.
  - If $v_0 = u_1 \wedge \cdots \wedge u_n \rightarrow v$, for all $i \in \{1, \ldots, n\}$ the pairs $\langle A_i; v_i \rangle \in \mathbf{PP}_k(D, e_1)$ are fresh, and $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon.\{v_i \leq u_i \mid i \in \{1, \ldots, n\}\})$, then $\langle \mathbf{s}(A_0 \wedge A_1 \wedge \cdots \wedge A_n); \mathbf{s}(v) \rangle \in \mathbf{PP}_k(D, e_0 e_1)$.
- If $e = \mathsf{rec}\,\{x = e_0\}$ and $\langle A; v \rangle \in \mathbf{PP}_k(D, e_0)$, then
  - If $x \notin \mathrm{FV}(e_0)$, then $\langle A; v \rangle \in \mathbf{PP}_k(D, e)$.
  - If $x \in \mathrm{FV}(e_0)$, then
    * If h is the minimum number in $\{1, \ldots, k\}$ such that $p_0 = \mathsf{bot}_{\mathrm{FV}(e)}$, $p_1 \in \mathbf{PP}_k((D, x : p_0), e_0)$, ..., $p_h \in \mathbf{PP}_k((D, x : p_{h-1}), e_0)$, and $p_h \leq_{\mathbf{spc}} p_{h-1}$, then $p_{h-1} \in \mathbf{PP}_k(D, e)$.
    * Otherwise (if such an h does not exist), if $A = A'$, $x : w$ and $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon.\{\mathrm{Gen}(A', v) \leq w\})$, then $\langle \mathbf{s}(A'); \mathbf{s}(v) \rangle \in \mathbf{PP}_k(D, e)$.

For every $k \geq 1$, expression e, and environment D, the set $\mathbf{PP}_k(D, e)$ is an equivalence class of pairs modulo renaming of the type variables in a pair. Indeed Definition 19 specifies an inference algorithm: to perform type inference on an expression e w.r.t. the environment D simply follow the definition of $\mathbf{PP}_k(D, e)$, choosing fresh type variables and using the $\leq_2$-satisfaction and $\leq_{\mathbf{spc}}$-checking algorithms as necessary.

---

[4] The set $\mathbf{PP}_k(D, e)$ does not contain all the principal pairs for e w.r.t. D. For instance, for all $k \geq 1$, $\langle \emptyset; (\alpha_1 \wedge \alpha_2) \rightarrow \alpha_1 \rangle$ is a principal-in-$\emptyset$ pair for $\lambda x.x$ in $\vdash_2^{\mathrm{P}_k+\mathrm{J}}$, but $\langle \emptyset; (\alpha_1 \wedge \alpha_2) \rightarrow \alpha_1 \rangle \notin \mathbf{PP}_k(\emptyset, \lambda x.x)$.

Theorem 20. *(Soundness and completeness of* $\mathbf{PP}_k$ *for* $\vdash_2^{\mathrm{P}_k+\mathrm{J}}$*). For every* $k \geq 1$*, expression* e*, and environment* D*:*
(Soundness). *If* $p \in \mathbf{PP}_k(D, e)$*, then* $D \vdash_2^{\mathrm{P}_k+\mathrm{J}} e : p$*.*
(Completeness). *If* $D \vdash_2^{\mathrm{P}_k+\mathrm{J}} e : p'$*, then* $p \leq_{\mathbf{spc}} p'$ *for some* $p \in \mathbf{PP}_k(D, e)$*.*

7 Typing Non-Simple Mutually Recursive Definitions

The results presented in Section 6 can be straightforwardly adapted to mutually recursive definitions. Let $\mathsf{rec}_i\{x_1 = e_1, \ldots, x_n = e_n\}$ (where $1 \leq i \leq n$) denote the i-th expression defined by the mutually recursive definition $\{x_1 = e_1, \ldots, x_n = e_n\}$.

Let $\vdash_2^{\mathrm{P}2}$ be the extension of system $\vdash_2^{\mathrm{P}}$ to mutual recursion, obtained by replacing rule (RECP) by the following rule:

$$
\text{(RecP2)} \quad \frac{\begin{array}{c} D, x_1 : \langle A_1;\, v_1\rangle, \ldots, x_n : \langle A_n;\, v_n\rangle \vdash e_1 : \langle A_1;\, v_1\rangle \\ \cdots \\ D, x_1 : \langle A_1;\, v_1\rangle, \ldots, x_n : \langle A_n;\, v_n\rangle \vdash e_n : \langle A_n;\, v_n\rangle \end{array}}{D \vdash \mathsf{rec}_{i_0}\{x_1 = e_1, \ldots, x_n = e_n\} : \langle A_1 \wedge \cdots \wedge A_n;\, v_{i_0}\rangle}
$$

where $i_0 \in \{1, \ldots, n\}$, $x_1, \ldots, x_n \notin D$, and (for all $i \in \{1, \ldots, n\}$)
$$\mathrm{Dom}(A_i) = \mathrm{FV}_D(e_i \underbrace{[x_1 := e_1, \ldots, x_n := e_n] \cdots [x_1 := e_1, \ldots, x_n := e_n]}_{n-1 \text{ times}}) - \{x_1, \ldots, x_n\}$$

The design of decidable systems $\vdash_2^{\mathrm{P}2_k}$, $\vdash_2^{\mathrm{P}2_k+\mathrm{ML}}$, and $\vdash_2^{\mathrm{P}2_k+\mathrm{J}}$, corresponding to the decidable systems $\vdash_2^{\mathrm{P}_k}$, $\vdash_2^{\mathrm{P}_k+\mathrm{ML}}$, and $\vdash_2^{\mathrm{P}_k+\mathrm{J}}$, is straightforward.

8 Conclusion

In this paper we have taken the system of rank-2 intersection type for the λ-calculus and have extended it with a decidable rule for typing non-simple recursive definitions. The new rules can be integrated without problems with the rules for typing non-simple local definitions and conditional expressions that we have proposed in previous work [6].

The technique developed in this paper does not depend on particulars of rank-2 intersection and could therefore be applied to other type systems. To clarify this point, we consider the following definition (taken from [22]):

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 (for instance, in system $\vdash_2$ a typing is a pair $\langle A; v\rangle$).

We can now describe our technique as follows:

1. Take a type system for the λ-calculus, with judgements of the shape $\vdash e : t$ where t mentions exactly the variables in $\mathrm{FV}(e)$ (this requirement is not restrictive, since it is always possible to adjust the judgment and the rules of a type system in order to satisfy it), such that: system $\vdash$ has the principal typing property (which is the property described by Definition 5 by replacing

$\leq^{\vdash}_{\mathbf{spc}}$); it is decidable to establish whether a typing t_1 can be specialized to a typing t_2 (i.e, whether $t_1 \leq^{\vdash}_{\mathbf{spc}} t_2$ holds); there is a known algorithm that for every term e decides whether e is $\vdash$-typable and, if so, returns a principal typing for e. We also require that system $\vdash$ contains the analogous of rule (SPC) (this additional requirement is not restrictive since, whenever the previous requirements are satisfied, such a rule is admissible).

2. Modify system $\vdash$ by introducing a *typing environment D* (containing typing assumptions for the rec-bound identifiers) and by adding typing rules analogous to (VARP) and (RECP). Let $\vdash^{P}$ be the resulting system (which has judgements of the shape $D \vdash^{P} e : t$).

3. Prove that, for every set of variables $X = \{x_1, \ldots, x_n\}$ ($n \geq 0$) the relation $\leq^{\vdash}_{\mathbf{spc}}$ is a preorder over $\mathsf{P}_X = \{t \mid \text{the typing } t \text{ mentions exactly the variables in } X\}$ and there is a known typing $\mathsf{bot}_X \in \mathsf{P}_X$ such that, for all typings $t \in \mathsf{P}_X$, $\mathsf{bot}_X \leq^{\vdash}_{\mathbf{spc}} t$. For every $k \geq 1$, define the analogous of rule (RECP$_k$) and prove the analogous of Theorem 14. Let $\vdash^{P_k}$ denote the system obtained from $\vdash^{P}$ by replacing the analogous of rule (RECP) with the analogous of rule (RECP$_k$). System $\vdash^{P_k}$ is guaranteed to have the principal-in-D typing property (which is the property described by Definition 12 by replacing *pair* with *typing* and $\leq_{\mathbf{spc}}$ with $\leq^{\vdash}_{\mathbf{spc}}$).

4. If necessary, add to $\vdash^{P_k}$ a $\vdash^{P}$-*suitable* (the analogous for $\vdash^{P}$ of the notion of $\vdash^{P}_2$-*suitable* given in Section 6.4) rule (REC_). Let $\vdash^{P_k+\text{-}}$ denote the resulting system.

The above steps describe a procedure that transforms a type system $\vdash$ (without rules for rec-expressions) enjoying the principal typing property into a system $\vdash^{P_k+\text{-}}$ (with rules for rec-expressions) enjoying the *principal-in-D* (and, in general, not the *principal*) typing property.

It worth examining what happens when the procedure is applied to the system of simple types [11]: the system at step 1 turns out to be $\vdash_0$ (the restriction of $\vdash_2$ which uses only simple types), the one obtained at step 2 is $\vdash^{P}_0$, which is essentially a reformulation of the Milner-Mycroft system (see Theorem 13), and the systems obtained at step 4 by adding the ML typing rule for recursion, $\vdash^{P_k+\text{ML}}_0$, are of intermediate power between the let-free fragments of the ML system and of the Milner-Mycroft system — we believe that the systems $\vdash^{P_k+\text{ML}}_0$ correspond to the family of abstract interpreters described in [8].

Further work includes, on the one side, designing more expressive decidable extensions of systems $\vdash^{P_k+\text{J}}_2$ (we are investigating the possibility of integrating techniques from the theory of abstract interpretation [4]). On the other, verifying the applicability of the technique to other type systems, like the system with rank-2 universal and intersection types proposed in [21], System **P** [14], and System E [2].

referees and the anonymous referees of earlier versions of this paper for many insightful and constructive comments.

References

1. H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. of Symbolic Logic*, 48:931–940, 1983.
2. S. Carlier, J. Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *ESOP'04*, volume 2986 of *LNCS*, pages 294–309. Springer, 2004.
3. M. Coppo and M. Dezani-Ciancaglini. An extension of basic functional theory for lambda-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.
4. P. Cousot. Types as Abstract Interpretations. In *POPL'97*, pages 316–331. ACM, 1997.
5. L. M. M. Damas and R. Milner. Principal type schemas for functional programs. In *POPL'82*, pages 207–212. ACM, 1982.
6. F. Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Trans. Prog. Lang. Syst.*, 25(4):401–451, 2003.
7. J. Y. Girard. *Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur*. PhD thesis, Université Paris VII, 1972.
8. R. Gori and G. Levi. Properties of a type abstract interpreter. In *VMCAI'03*, volume 2575 of *LNCS*, pages 132–145. Springer, 2003.
9. J. J. Hallett and A. J. Kfoury. Programming examples needing polymorphic recursion. In *ITRS'04 (workshop affiliated to LICS'04)*, ENTCS. Elsevier, to appear.
10. F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Prog. Lang. Syst.*, 15(2):253–289, 1993.
11. R. Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, London, 1997.
12. T. Jim. Rank 2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, LCS, Massachusetts Institute of Technology, 1995.
13. T. Jim. What are principal typings and what are they good for? In *POPL'96*, pages 42–53. ACM, 1996.
14. T. Jim. A polar type system. In *ICALP Workshops*, volume 8 of *Proceedings in Informatics*, pages 323–338. Carleton-Scientific, 2000.
15. 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.
16. D. Leivant. Polymorphic Type Inference. In *POPL'83*, pages 88–98. ACM, 1983.
17. L. Meertens. Incremental polymorphic type checking in B. In *POPL'83*, pages 265–275. ACM, 1983.
18. A. Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228. Springer, 1984.
19. J. C. Reynolds. Towards a Theory of Type Structure. In *Colloque sur la Programmation*, volume 19 of *LNCS*. Springer, 1974.
20. S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Katholieke Universiteit Nijmegen, 1993.
21. S. van Bakel. Rank 2 types for term graph rewriting. In *TIP'02*, volume 75 of *ENTCS*. Elsevier, 2003.
22. J.B. Wells. The essence of principal typings. In *ICALP'02*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.
23. H. Yokouchi. Embedding a Second-Order Type System into an Intersection Type System. *Information and Computation*, 117:206–220, 1995.