

Rank-2 Intersection and Polymorphic Recursion

(extended version with proofs)

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 $\text{rec}\{x = e\}$ 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 $\text{int} \rightarrow \text{int}$ and $\text{bool} \rightarrow \text{bool}$, so it has type $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$. Rank-2 intersection types may contain intersections only to the left of a single arrow. Therefore, for instance, $((\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})) \rightarrow \text{int} \rightarrow \text{int}$ is a rank-2

* Partially supported by IST-2001-33477 DART and MIUR cofin'04 EOS projects. The funding bodies are not responsible for any use that might be made of the results presented here.

intersection type (as usual, the arrow type constructor is right associative), while $((\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})) \rightarrow \text{int} \rightarrow \text{int}$ 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 $((\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1) \rightarrow \alpha_2$, 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 $\text{let } x = e_0 \text{ in } e_1$ where e_0 is non-simple) and *non-simple conditional expressions* (i.e., non-simple terms of the shape $\text{if } e_0 \text{ then } e_1 \text{ else } e_2$). In this paper we propose new rules and algorithms for typing *non-simple recursive definitions* (i.e., non-simple terms of the shape $\text{rec } \{x = e\}$). 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. In 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 $\text{rec } \{x = e\}$ 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.

An on-line demonstration of a prototype implementation of the typing algorithm is 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:

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

where b ranges over booleans (**true** and **false**) and ι ranges over integer numbers.

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

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

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

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:

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

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 \rightarrow is right associative, e.g., $u_1 \rightarrow u_2 \rightarrow u_3$ means $u_1 \rightarrow (u_2 \rightarrow u_3)$, and the constructors \times and **list** bind more tightly than \rightarrow , e.g., $u_1 \rightarrow u_2 \times u_3$ means $u_1 \rightarrow (u_2 \times u_3)$. We 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 \rightarrow , e.g., $u_1 \wedge u_2 \rightarrow u_3$ means $(u_1 \wedge u_2) \rightarrow u_3$, 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 $\{x_1 : t_1, \dots, x_n : t_n\}$ of assumptions for variables such that every variable x_i ($1 \leq i \leq n$) can occur at most once in T . The expression $\text{Dom}(T)$ denotes the *domain* of T , which is the set $\{x_1, \dots, x_n\}$. We write $T, x : t$ for the environment $T \cup \{x : t\}$ where it is assumed that $x \notin \text{Dom}(T)$. 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 $\{x_1 : w_1, \dots, x_n : w_n\}$ 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:

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

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_{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.

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

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

- $\text{Dom}(A) = \text{Dom}(A')$,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 $\langle A; v \rangle \leq_{\text{spc}} \langle A'; v' \rangle$) if $A' \leq_1 \mathbf{s}(A)$ and $\mathbf{s}(v) \leq_2 v'$, for some substitution \mathbf{s} .

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

Note that the relation \leq_{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 $\text{Dom}(A) = \text{Dom}(A')$ in the definition of $A \leq_1 A'$ is not present in most other papers. The “usual” definition drops this requirement, thus allowing $\text{Dom}(A) \supseteq \text{Dom}(A')$. 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	$\mathbf{type}(c)$
b	\mathbf{bool}	\mathbf{not}	$\mathbf{bool} \rightarrow \mathbf{bool}$	\mathbf{fst}	$\alpha_1 \times \alpha_2 \rightarrow \alpha_1$
ι	\mathbf{int}	$\mathbf{and, or}$	$\mathbf{bool} \times \mathbf{bool} \rightarrow \mathbf{bool}$	\mathbf{snd}	$\alpha_1 \times \alpha_2 \rightarrow \alpha_2$
\mathbf{pair}	$\alpha_1 \rightarrow \alpha_2 \rightarrow (\alpha_1 \times \alpha_2)$	$+, -, *$	$\mathbf{int} \times \mathbf{int} \rightarrow \mathbf{int}$	\mathbf{null}	$\alpha \mathbf{list} \rightarrow \mathbf{bool}$
\mathbf{nil}	$\alpha \mathbf{list}$	$=, <$	$\mathbf{int} \times \mathbf{int} \rightarrow \mathbf{bool}$	\mathbf{hd}	$\alpha \mathbf{list} \rightarrow \alpha$
\mathbf{cons}	$\alpha \rightarrow \alpha \mathbf{list} \rightarrow \alpha \mathbf{list}$			\mathbf{tl}	$\alpha \mathbf{list} \rightarrow \alpha \mathbf{list}$

Fig. 1. Types for constants

$\text{(SPC)} \frac{\frac{\vdash e : p}{\vdash e : p'}}{\text{where } p \leq_{\text{spc}} p'}$	$\text{(CON)} \vdash c : \langle \emptyset; v \rangle$ <p style="text-align: center; margin: 0;">where $v = \mathbf{type}(c)$</p>	$\text{(VAR)} \vdash x : \langle \{x : u\}; u \rangle$ <p style="text-align: center; margin: 0;">where $u \in \mathbf{T}_0$</p>
$\text{(APP)} \frac{\frac{\vdash e : \langle A; u_1 \wedge \dots \wedge u_n \rightarrow v \rangle}{\vdash e e_0 : \langle A \wedge A_1 \wedge \dots \wedge A_n; v \rangle} \quad \vdash e_0 : \langle A_1; u_1 \rangle \quad \dots \quad \vdash e_0 : \langle A_n; u_n \rangle}{\vdash e e_0 : \langle A \wedge A_1 \wedge \dots \wedge A_n; v \rangle}$		
$\text{(ABS)} \frac{\frac{\vdash e : \langle A, x : w; v \rangle}{\vdash \lambda x. e : \langle A; w \rightarrow v \rangle}}$	$\text{(ABSVAC)} \frac{\frac{\vdash e : \langle A; v \rangle}{\vdash \lambda x. e : \langle A; u \rightarrow v \rangle}}{\text{where } x \notin \text{FV}(e) \text{ and } u \in \mathbf{T}_0}$	

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 $\text{Dom}(A) = \text{FV}(e)$.

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 $\text{Dom}(A) = \text{FV}(e)$, 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 $x \notin \text{FV}(e)$. 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 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 $\mathbf{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$. 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:

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

where $\vec{\alpha}$ 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 $\text{FTV}(t)$ denote the set of free type variables of t . For every scheme $\forall \vec{\alpha}. v$ it is assumed that $\{\vec{\alpha}\} \subseteq \text{FTV}(v)$. 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 $\text{Gen}(A, v)$ for the \forall -closure of v in A , that is the rank-2 scheme $\forall \vec{\alpha}. v$, where $\vec{\alpha}$ is the sequence of the type variables in $\text{FTV}(v) - \cup_{x:w \in A} \text{FTV}(w)$.

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

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

We can now present the typing rules. All these rules type recursive definitions $\text{rec } \{x = e\}$ 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 $\text{rec } \{x = e\}$ is *vacuous* to mean that $x \notin \text{FV}(e)$). Jim [12] proposed the following rules for typing recursive definitions:²

$$\begin{array}{c} \text{(REC)} \quad \frac{\vdash e : \langle A, x : w; v \rangle}{\vdash \text{rec } \{x = e\} : \langle A; v \rangle} \\ \text{where } \text{Gen}((A, x : w), v) \leq_{\forall 2,1} w \end{array} \qquad \begin{array}{c} \text{(RECVAC)} \quad \frac{\vdash e : p}{\vdash \text{rec } \{x = e\} : p} \\ \text{where } x \notin \text{FV}(e) \end{array}$$

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

As pointed out by Jim [13], rule (REC) might be generalized along the lines of Mycroft's rule (FIX') [18]: just replace the condition $\text{Gen}((A, x : w), v) \leq_{\forall 2,1} w$ by the condition $\text{Gen}(A, v) \leq_{\forall 2,1} w$. Let us call (REC') the generalized rule. Rule (REC') is strictly more powerful than rule (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

$$\text{(RECINT)} \quad \frac{\vdash e : \langle A, x : u_1 \wedge \dots \wedge u_m; u_1 \rangle \quad \dots \quad \vdash e : \langle A, x : u_1 \wedge \dots \wedge u_m; u_m \rangle}{\vdash \text{rec } \{x = e\} : \langle A; u_{i_0} \rangle} \\ \text{where } i_0 \in \{1, \dots, m\}$$

The decidability of $\vdash_2 + (\text{RECINT})$ 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 $u_1, \dots, u_m \in \mathbf{T}_0$ (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^P

In order to be able to type non-simple recursive definitions, we propose to adopt the following strategy: allow one to assign to $\text{rec } \{x = e\}$ 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 .

$$\begin{array}{c}
\text{(SPC)} \frac{D \vdash e : p}{D \vdash e : p'} \quad \text{where } p \leq_{\text{SPC}} p' \quad \text{(CON)} D \vdash c : \langle \emptyset; v \rangle \quad \text{where } v = \mathbf{type}(c) \quad \text{(VAR)} D \vdash x : \langle \{x : u\}; u \rangle \quad \text{where } u \in \mathbf{T}_0 \text{ and } x \notin \text{Dom}(D) \\
\text{(APP)} \frac{D \vdash e : \langle A; u_1 \wedge \dots \wedge u_n \rightarrow v \rangle \quad D \vdash e_0 : \langle A_1; u_1 \rangle \quad \dots \quad D \vdash e_0 : \langle A_n; u_n \rangle}{D \vdash e e_0 : \langle A \wedge A_1 \wedge \dots \wedge A_n; v \rangle} \\
\text{(ABS)} \frac{D \vdash e : \langle A, x : w; v \rangle}{D \vdash \lambda x. e : \langle A; w \rightarrow v \rangle} \quad \text{where } x \notin D \quad \text{(ABSVAC)} \frac{D \vdash e : \langle A; v \rangle}{D \vdash \lambda x. e : \langle A; u \rightarrow v \rangle} \quad \text{where } x \notin \text{FV}(e), u \in \mathbf{T}_0, \text{ and } x \notin D \\
\text{(VARP)} D, x : p \vdash x : p \quad \text{(RECP)} \frac{D, x : \langle A; v \rangle \vdash e : \langle A; v \rangle}{D \vdash \mathbf{rec} \{x = e\} : \langle A; v \rangle} \quad \text{where } \text{Dom}(A) = \text{FV}_D(\mathbf{rec} \{x = e\}) \text{ and } x \notin D
\end{array}$$

Fig. 3. Typing rules of system \vdash_2^P

Definition 8 (Pair environments). A *pair environment* D is an environment $\{x_1 : p_1, \dots, x_n : p_n\}$ of pair assumptions for variables such that $\text{Dom}(D) \cap \text{VR}(D) = \emptyset$, where $\text{VR}(D) = \cup_{x : \langle A; v \rangle \in D} \text{Dom}(A)$ 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^P (where “P” stands for “polymorphic”) are given in Fig. 3. The judgement $D \vdash_2^P e : \langle A; v \rangle$ means “ e is \vdash_2^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 \mathbf{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 $\text{Dom}(D)$, and v is a rank-2 type.

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

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

Rule (RECP) allows to assign to a recursive definition $\mathbf{rec} \{x = e\}$ 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 $\forall \vec{\alpha}. p$, where $\vec{\alpha}$ is the sequence of *all* the type variables occurring in the pair p .

6.2 On the Expressive Power of System \vdash_2^P

System \vdash_2^P 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 $\text{rec } \{f = e\}$, where

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

is non-simple since it defines a function that uses its parameter g with two different non unifiable types. The following \vdash_2^P -typing judgement holds.

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

System \vdash_2^P has more expressive power than the system $\vdash_2 + (\text{REC}) + (\text{RECVAC})$ of Section 5 (and therefore of system \mathbf{P}_2^R [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 $\vdash_2 + (\text{REC}) + (\text{RECVAC}) e : \langle A; v \rangle$, then $\emptyset \vdash_2^P e : \langle A; v \rangle$.*

Theorem 11. *If $\emptyset \vdash e : u$ is Milner-Mycroft derivable, then $\emptyset \vdash_2^P e : \langle \emptyset; u \rangle$.*

6.3 Principal-in- D Pairs and (Un)decidability of \vdash_2^P

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 $p \leq_{\text{spc}} p'$. 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_2^P has the principal-in- D pair property. The following theorem implies that the restriction of \vdash_2^P which uses only simple types, that we will call \vdash_0^P , is undecidable (as is the Milner-Mycroft system [10,15]). We conjecture that also the whole \vdash_2^P is undecidable.

Theorem 13. *Let e be a let-free expression. Then $\emptyset \vdash_0^P e : \langle \emptyset; u \rangle$ iff $\emptyset \vdash e : u$ is Milner-Mycroft derivable.*

6.4 Systems $\vdash_2^{P^{k+}}$ ($k \geq 1$): a Family of Decidable Restrictions of \vdash_2^P

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 $X = \{x_1, \dots, x_n\}$ ($n \geq 0$) let

$$- P_X = \{ \langle A; v \rangle \mid \langle A; v \rangle \text{ is pair such that } \text{Dom}(A) = X \}, \text{ and}$$

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

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

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

$$\text{(RECP}_k) \frac{D, x : p_0 \vdash e : p_1 \cdots D, x : p_{k-1} \vdash e : p_k}{D \vdash \text{rec} \{x = e\} : p_k}$$

where $x \notin D$, $p_0 = \text{bot}_{\text{FV}(\text{rec} \{x=e\})}$, $p_{k-1} = p_k$, and
for all $i \in \{1, \dots, k\}$ p_i is a principal-in- $(D, x : p_{i-1})$ pair for e

(note that $D \vdash_2^{\text{P}^k} e : p$ implies $D \vdash_2^{\text{P}^{k+1}} e : p$). For all $k \geq 1$, system $\vdash_2^{\text{P}^k}$ has the principal-in- D pair property and is decidable — the result follows from Theorem 20 (soundness and completeness of the inference algorithm for $\vdash_2^{\text{P}^k + \text{J}}$, which is an extension of $\vdash_2^{\text{P}^k}$) of Section 6.5. The relation between system $\vdash_2^{\text{P}^k}$ and system \vdash_2^{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 $D \vdash_2^{\text{P}^k} e : p$, then $D \vdash_2^{\text{P}} e : p$.*
2. *If e is $\vdash_2^{\text{P}^k}$ -typable in D and $D \vdash_2^{\text{P}} e : p$, then $D \vdash_2^{\text{P}^k} e : p$.*

Unfortunately, for every $k \geq 1$, system $\vdash_2^{\text{P}^k}$ 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 $\text{rec} \{f = e\}$ of Example 9 in Section 6.2 is non-simple, therefore it is not ML-typable. Since

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

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

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

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

in system \vdash_2^{P} .

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

We will now show that, for all $k \geq 1$, it is quite easy to modify system $\vdash_2^{\text{P}^k}$ 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^{P} -suitable to mean that: the system \vdash_2^{P} obtained from $\vdash_2^{\text{P}^k}$ by replacing rule (RECP) with rule (REC₋)

- is a restriction of system \vdash_2^P (i.e., $D \vdash_2 e : \langle A; v \rangle$ implies $D \vdash_2^P e : \langle A; v \rangle$),
- is decidable, and
- has the principal-in- D pair property.

For instance, for every $k \geq 1$, rule (REC P_k) is \vdash_2^P -suitable. Theorem 14 guarantees that, for all $k \geq 1$, adding to system $\vdash_2^{P_k}$ a \vdash_2^P -suitable rule (REC $_-$) results in a system, denoted by $\vdash_2^{P_k+}$, with both decidability and principal-in- D pair property. So, to extend system $\vdash_2^{P_k}$ to type all the ML typable recursive definitions, we have just to add to system $\vdash_2^{P_k}$ a \vdash_2^P -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 $\vdash_2^{P_k}$, of) the ML rule itself:

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

(producing system $\vdash_2^{P_k+ML}$). Another possibility, is to add the following rule

$$\text{(RECJ)} \frac{D \vdash e : \langle A, x : w; v \rangle}{D \vdash \text{rec} \{x = e\} : \langle A; v \rangle} \text{ where } \text{Gen}((A, x : w), v) \leq_{v2,1} w \text{ and } x \notin D$$

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

Example 16. The expression $\text{rec} \{f = e'\}$ of Example 15, which is not $\vdash_2^{P_k}$ -typable (for all $k \geq 1$), has principal-in- \emptyset pairs $\langle \emptyset; (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \rangle$ and $\langle \emptyset; ((\alpha \rightarrow \alpha) \wedge (\alpha \rightarrow \beta)) \rightarrow (\alpha \wedge \beta) \rightarrow \beta \rangle$ in systems $\vdash_2^{P_1+ML}$ and $\vdash_2^{P_1+J}$, respectively.

6.5 An Inference Algorithm for $\vdash_2^{P_k+J}$ ($k \geq 1$)

The inference algorithm makes use of an algorithm for checking whether the \leq_{spc} relation (see Definition 3) holds and of an algorithm for finding a most general solution to a $\leq_{v2,1}$ -satisfaction problem ($\leq_{v2,1}$ is the relation of Definition 6).

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

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

A $\leq_{v2,1}$ -satisfaction problem [13,12] (see also [6]) is a formula $\exists \vec{\alpha}.P$, where $\vec{\alpha}$ 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 \mathbf{T}_0 types; or 2) an inequality between a $\mathbf{T}_{v2} \cup \mathbf{T}_2$ type and a \mathbf{T}_1 type. A substitution \mathbf{s} is a *solution* to $\exists \vec{\alpha}.P$ if there exists a substitution \mathbf{s}' such that: $\mathbf{s}(\alpha) = \mathbf{s}'(\alpha)$ for all $\alpha \notin \vec{\alpha}$, $\mathbf{s}'(u_1) = \mathbf{s}'(u_2)$ for every equality $(u_1 = u_2) \in P$, and $\mathbf{s}'(vs) \leq_{v2,1} \mathbf{s}'(w)$ (resp. $\mathbf{s}'(\forall \epsilon.v) \leq_{v2,1} \mathbf{s}'(w)$) for every inequality $(vs \leq w) \in P$ (resp. $(v \leq w) \in P$). We will write $\text{MGS}(\exists \vec{\alpha}.P)$ for the set of most general solutions to the $\leq_{v2,1}$ -satisfaction problem $\exists \vec{\alpha}.P$ (a $\leq_{v2,1}$ -satisfaction problem $\exists \vec{\alpha}.P$ 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 $\vdash_2^{\mathbf{P}_k+\mathbf{J}}$ 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 $\vec{\alpha} = \text{FTV}(A) \cup \text{FTV}(v)$, then $\langle \mathbf{s}(A); \mathbf{s}(v) \rangle \in \mathbf{PP}_k(D, x)$.
 - If $x \notin \text{Dom}(D)$ 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 $x \notin \text{FV}(e_0)$ and α is a fresh type variable, then $\langle A; \alpha \rightarrow v \rangle \in \mathbf{PP}_k(D, \lambda x. e_0)$.
 - If $x \in \text{FV}(e_0)$ 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 $\langle \mathbf{s}(A_0 \wedge A_1); \mathbf{s}(v) \rangle \in \mathbf{PP}_k(D, e_0 e_1)$.
 - If $v_0 = u_1 \wedge \dots \wedge u_n \rightarrow v$, for all $i \in \{1, \dots, n\}$ the pairs $\langle A_i; v_i \rangle \in \mathbf{PP}_k(D, e_1)$ are fresh, and $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{v_i \leq u_i \mid i \in \{1, \dots, n\}\})$, then $\langle \mathbf{s}(A_0 \wedge A_1 \wedge \dots \wedge A_n); \mathbf{s}(v) \rangle \in \mathbf{PP}_k(D, e_0 e_1)$.
- If $e = \text{rec } \{x = e_0\}$ and $\langle A; v \rangle \in \mathbf{PP}_k(D, e_0)$, then
 - If $x \notin \text{FV}(e_0)$, then $\langle A; v \rangle \in \mathbf{PP}_k(D, e)$.
 - If $x \in \text{FV}(e_0)$, then
 - * If h is the minimum number in $\{1, \dots, k\}$ such that $p_0 = \text{bot}_{\text{FV}(e)}$, $p_1 \in \mathbf{PP}_k((D, x : p_0), e_0)$, \dots , $p_h \in \mathbf{PP}_k((D, x : p_{h-1}), e_0)$, and $p_h \leq_{\text{spc}} p_{h-1}$, then $p_{h-1} \in \mathbf{PP}_k(D, e)$.
 - * Otherwise (if such an h does not exist), if $A = A'$, $x : w$ and $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{\text{Gen}(A', v) \leq w\})$, 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_{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 $\vdash_2^{\mathbf{P}_k+\mathbf{J}}$, 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 $\vdash_2^{\mathbf{P}^{k+J}}$). For every $k \geq 1$, expression e , and environment D :

(Soundness). If $p \in \mathbf{PP}_k(D, e)$, then $D \vdash_2^{\mathbf{P}^{k+J}} e : p$.

(Completeness). If $D \vdash_2^{\mathbf{P}^{k+J}} e : p'$, 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 $\text{rec}_i \{x_1 = e_1, \dots, x_n = e_n\}$ (where $1 \leq i \leq n$) denote the i -th expression defined by the mutually recursive definition $\{x_1 = e_1, \dots, x_n = e_n\}$.

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

$$\begin{array}{c} D, x_1 : \langle A_1; v_1 \rangle, \dots, x_n : \langle A_n; v_n \rangle \vdash e_1 : \langle A_1; v_1 \rangle \\ \dots \\ D, x_1 : \langle A_1; v_1 \rangle, \dots, x_n : \langle A_n; v_n \rangle \vdash e_n : \langle A_n; v_n \rangle \\ \text{(RECP2)} \frac{D \vdash \text{rec}_{i_0} \{x_1 = e_1, \dots, x_n = e_n\} : \langle A_1 \wedge \dots \wedge A_n; v_{i_0} \rangle}{D \vdash \text{rec}_{i_0} \{x_1 = e_1, \dots, x_n = e_n\} : \langle A_1 \wedge \dots \wedge A_n; v_{i_0} \rangle} \\ \text{where } i_0 \in \{1, \dots, n\}, x_1, \dots, x_n \notin D, \text{ and (for all } i \in \{1, \dots, n\}) \\ \text{Dom}(A_i) = \text{FV}_D(e_i [x_1 := e_1, \dots, x_n := e_n] \cdots [x_1 := e_1, \dots, x_n := e_n]) - \{x_1, \dots, x_n\} \\ \underbrace{\hspace{10em}}_{n-1 \text{ times}} \end{array}$$

The design of decidable systems $\vdash_2^{\mathbf{P}^{2k}}$, $\vdash_2^{\mathbf{P}^{2k+\text{ML}}}$, and $\vdash_2^{\mathbf{P}^{2k+J}}$, corresponding to the decidable systems $\vdash_2^{\mathbf{P}^k}$, $\vdash_2^{\mathbf{P}^{k+\text{ML}}}$, and $\vdash_2^{\mathbf{P}^{k+J}}$, 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 $\text{FV}(e)$ (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

- pair* with *typing* and $\leq_{\mathbf{spc}}$ with a suitable relation for \vdash , that we will call $\leq_{\mathbf{spc}}^{\vdash}$; it is decidable to establish whether a typing t_1 can be specialized to a typing t_2 (i.e, whether $t_1 \leq_{\mathbf{spc}}^{\vdash} t_2$ 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 $\vdash^{\mathbf{P}}$ be the resulting system (which has judgements of the shape $D \vdash^{\mathbf{P}} e : t$).
 3. Prove that, for every set of variables $X = \{x_1, \dots, x_n\}$ ($n \geq 0$) the relation $\leq_{\mathbf{spc}}^{\vdash}$ is a preorder over $\mathbf{P}_X = \{t \mid \text{the typing } t \text{ mentions exactly the variables in } X\}$ and there is a known typing $\mathbf{bot}_X \in \mathbf{P}_X$ such that, for all typings $t \in \mathbf{P}_X$, $\mathbf{bot}_X \leq_{\mathbf{spc}}^{\vdash} t$. For every $k \geq 1$, define the analogous of rule (RECP $_k$) and prove the analogous of Theorem 14. Let $\vdash^{\mathbf{P}^k}$ denote the system obtained from $\vdash^{\mathbf{P}}$ by replacing the analogous of rule (RECP) with the analogous of rule (RECP $_k$). System $\vdash^{\mathbf{P}^k}$ 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 $\leq_{\mathbf{spc}}^{\vdash}$).
 4. If necessary, add to $\vdash^{\mathbf{P}^k}$ a $\vdash^{\mathbf{P}}$ -*suitable* (the analogous for $\vdash^{\mathbf{P}}$ of the notion of $\vdash_2^{\mathbf{P}}$ -*suitable* given in Section 6.4) rule (REC $_-$). Let $\vdash^{\mathbf{P}^k+}$ 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 $\vdash^{\mathbf{P}^k+}$ (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 $\vdash_0^{\mathbf{P}}$, 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, $\vdash_0^{\mathbf{P}^k+\text{ML}}$, are of intermediate power between the let-free fragments of the ML system and of the Milner-Mycroft system — we believe that the systems $\vdash_0^{\mathbf{P}^k+\text{ML}}$ correspond to the family of abstract interpreters described in [8].

Further work includes, on the one side, designing more expressive decidable extensions of systems $\vdash_2^{\mathbf{P}^k+\text{J}}$ (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 \mathbf{P} [14], and System E [2].

Acknowledgments. I thank Viviana Bono, Sebastien Carlier, Mario Coppo, Joe Hallet, Assaf Kfoury, Emiliano Leporati, Peter Møller Neergaard, and Joe Wells for discussions on the subject of this paper. I also thank the TLCA'05

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 fonctionnelle et elimination des coupures dans l'arithmetique 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.

A Proofs

A.1 Preliminary Definitions

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 *domain* and the *set of free type variables occurring in the range* of a substitution \mathbf{s} are the sets of type variables: $\mathbf{Dom}(\mathbf{s}) = \{\alpha \mid \mathbf{s}(\alpha) \neq \alpha\}$ and $\mathbf{FTVR}(\mathbf{s}) = \cup_{\alpha \in \mathbf{Dom}(\mathbf{s})} \mathbf{FTV}(\mathbf{s}(\alpha))$. Substitutions will be denoted by $[\alpha_1 := u_1, \dots, \alpha_n := u_n]$ ($n \geq 0$); the empty substitution will be denoted by $[\]$.

The *composition* of two substitutions \mathbf{s}_1 and \mathbf{s}_2 is the substitution, denoted by $\mathbf{s}_1 \circ \mathbf{s}_2$, such that $\mathbf{s}_1 \circ \mathbf{s}_2(\alpha) = \mathbf{s}_1(\mathbf{s}_2(\alpha))$, for all type variables α . We say that \mathbf{s} is *more general* than \mathbf{s}' , written $\mathbf{s} \leq \mathbf{s}'$, if there is a substitution \mathbf{s}'' such that $\mathbf{s}' = \mathbf{s}'' \circ \mathbf{s}$. A substitution is *idempotent* if $\mathbf{s} = \mathbf{s} \circ \mathbf{s}$ (i.e. if $\mathbf{Dom}(\mathbf{s}) \cap \mathbf{FTVR}(\mathbf{s}) = \emptyset$).

Given two *environments* (i.e., sets of assumptions $x : t$ for variables, as defined in Section 3) T_1 and T_2 , we will write

$T_1 \uplus T_2$ to denote the environment $T_1 \cup T_2$ under the assumption that $\mathbf{Dom}(T_1) \cap \mathbf{Dom}(T_2) = \emptyset$.

$T_1 \oplus T_2$ to denote the environment $T_1 \cup T_2$ under the assumption that $x : t_1 \in T_1$ and $x : t_2 \in T_2$ imply $t_1 = t_2$.

Let $T|_X$ denote the *restriction of the environment T to the set of variables X* , which is the environment $\{x : t \in T \mid x \in X\}$.

Definition 21 (ML-type, ML-scheme, and ML environments).

1. An *ML-type environment* U is an environment $\{x_1 : u_1, \dots, x_n : u_n\}$ of simple type assumptions for variables.
2. An *ML-scheme environment* S is an environment $\{x_1 : t_1, \dots, x_n : t_n\}$ of ML-scheme assumptions for variables. I.e., each t_i is of the shape $\forall \vec{\alpha}^i. u_i$ for some simple type u_i and sequence of type variables $\{\vec{\alpha}^i\} \subseteq \mathbf{FTV}(u_i)$.
3. An *ML environment* B is an environment of the shape $U \uplus S$ where U is an ML-type environment and S is an ML-scheme environment.

Typing Rules for let-expressions and if-expression. As explained at the end of Section 4.1, in the paper we have omitted the typing rules for local definitions and conditional expressions for system \vdash_2 (and its extension outlined in Section 5) and system \vdash_2^P (and its decidable restrictions outlined in Section 6.4). Since some of the proofs refer to these rules, we give them in Figs 4 and 5 (where ifc is a special constant of type $\text{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$).

$$\text{(LETSIMPLE)} \frac{\vdash (\lambda x. e_1) e_0 : p}{\vdash \text{let } x = e_0 \text{ in } e_1 : p} \quad \text{(IFSIMPLE)} \frac{\vdash \text{ifc } e_0 \ e_1 \ e_2 : p}{\vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : p}$$

Fig. 4. Typing rules for local definitions and conditional expressions (system \vdash_2)

$$\text{(LETSIMPLE)} \frac{D \vdash (\lambda x. e_1) e_0 : p}{D \vdash \text{let } x = e_0 \text{ in } e_1 : p} \quad \text{(IFSIMPLE)} \frac{D \vdash \text{ifc } e_0 \ e_1 \ e_2 : p}{D \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : p}$$

Fig. 5. Typing rules for local definitions and conditional expressions (system \vdash_2^P)

A Syntax-directed Version of the Milner-Mycroft System. Let \vdash' denote the syntax-directed version of the Milner-Mycroft system in Fig. 6 (which is essentially the system $\vdash_{MM'}$ given in Section 4.1 of [10]).

Lemma 22. *If $B \vdash' e : u$, then $s(B) \vdash' e : s(u)$ for all substitutions s .*

Proof. By structural induction on \vdash' derivations.

Lemma 23. *If $B \vdash' e : u$, then $B' \vdash' e : u$ for all B' such that $B' \supseteq B$.*

Proof. By structural induction on \vdash' derivations.

Lemma 24. *If $B \vdash' e : u$, then $B' \vdash' e : u$ for all B' such that $B' \subseteq B$ and $\text{Dom}(B') \supseteq \text{FV}(e)$.*

Proof. By structural induction on \vdash' derivations.

We will write $B \vdash'' e : u$ to mean that $B \vdash' e : u$ with $\text{Dom}(B) = \text{FV}(e)$.

System \vdash_0^P . The typing rules of system \vdash_0^P (the restriction of system \vdash_2^P which uses only simple types) are given in Fig. 7.

A.2 Proof of Theorem 10

Restatement of Theorem 10. *If $\vdash_2 + (\text{REC}) + (\text{RECVAC}) \ e : \langle A; v \rangle$, then $\emptyset \vdash_2^P \ e : \langle A; v \rangle$.*

Proof. For sake of readability, let \vdash^J be short for $\vdash_2 + (\text{REC}) + (\text{RECVAC})$. The proof is by structural induction on \vdash^J -derivations. Let the derivation $\vdash^J \ e : \langle A; v \rangle$ end by rule:

(VAR). Then $e = x$, $A = \{x : u\}$, and $v = u$, for some variable x and simple type u . By rule (VAR), we have that $\emptyset \vdash x : \langle \{x : u\}; u \rangle$.

$\text{(CON)} \frac{B \vdash c : \mathbf{s}(\mathbf{type}(c))}{\text{where } \mathbf{Dom}(\mathbf{s}) = \text{FTV}(\mathbf{type}(c)) \text{ and } \mathbf{s}(\mathbf{type}(c)) \in \mathbf{T}_0}$	$\text{(VAR}_\lambda) \frac{B, x : u \vdash x : u}{\text{where } u \in \mathbf{T}_0}$
$\text{(APP)} \frac{B \vdash e : u_0 \rightarrow u \quad B \vdash e_0 : u_0}{B \vdash e e_0 : u}$	$\text{(ABS)} \frac{B, x : u_0 \vdash e : u}{B \vdash \lambda x. e : u_0 \rightarrow u}$
$\text{(VAR}_{\text{let,rec}}) \frac{B, x : \forall \vec{\alpha}. u \vdash x : \mathbf{s}(u)}{\text{where } \mathbf{Dom}(\mathbf{s}) = \{\vec{\alpha}\}}$	$\text{(RECM)} \frac{B, x : \forall \vec{\alpha}. u \vdash e : u}{B \vdash \text{rec } \{x = e\} : \mathbf{s}(u)}$ <p style="text-align: center; margin: 0;"> where $\vec{\alpha} = \text{FTV}(u) - \text{FTV}(B)$ and $\mathbf{Dom}(\mathbf{s}) = \{\vec{\alpha}\}$ </p>
$\text{(LET)} \frac{B \vdash e_0 : u_0 \quad B, x : \text{Gen}(B, u_0) \vdash e_1 : u_1}{B \vdash \text{let } x = e_0 \text{ in } e_1 : u_1}$	$\text{(IF)} \frac{B \vdash \text{ifc } e_0 e_1 e_2 : u}{B \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : u}$

Fig. 6. Syntax-directed typing rules for the Milner-Mycroft system

(REC). Then $e = \text{rec } \{x = e_0\}$ for some e_0 and $\vdash^J e_0 : \langle A, x : w; v \rangle$ for some w such that $\text{Gen}((A, x : w), v) \leq_{\forall 2,1} w$. By induction, $\emptyset \vdash e_0 : \langle A, x : w; v \rangle$. Let $w = u_1 \wedge \dots \wedge u_n$ ($n \geq 1$). Since $\text{Gen}((A, x : w), v) \leq_{\forall 2,1} u_1 \wedge \dots \wedge u_n$ implies $\langle A; v \rangle \leq_{\text{SPC}} \langle A; u_i \rangle$ for all $i \in \{1, \dots, n\}$, we can transform the derivation of $\emptyset \vdash e_0 : \langle A, x : w; v \rangle$ into a derivation of $\{x : \langle A; v \rangle\} \vdash e_0 : \langle A; v \rangle$ just by replacing each use of the axiom (VAR) for x , that must be of the shape

$$\emptyset \vdash x : \langle \{x : u_i\}; u_i \rangle, \text{ for some } i \in \{1, \dots, n\},$$

into an use of the axiom (VARP):

$$\{x : \langle A; v \rangle\} \vdash x : \langle A; v \rangle,$$

followed by an application of rule (SPC). Therefore, by rule (RECP), we can derive $\emptyset \vdash e : \langle A; v \rangle$.

(RECVAC). Then $e = \text{rec } \{x = e_0\}$, with $x \notin \text{FV}(e_0)$, and $\vdash^J e_0 : \langle A; v \rangle$. By induction, $\emptyset \vdash e_0 : \langle A; v \rangle$ which implies $\{x : \langle A; v \rangle\} \vdash e_0 : \langle A; v \rangle$. Therefore, by rule (RECP), we can derive $\emptyset \vdash e : \langle A; v \rangle$.

(CON). Immediate.

(SPC), (APP), (ABS), (ABSVAC), (LETSIMPLE), or (IFSIMPLE). Straightforward by induction.

A.3 Proof of Theorem 11

Lemma 25. *Let $D \vdash_2^P e : \langle U \uplus A; u \rangle$ with $D = \{x_1 : \langle U; u_1 \rangle, \dots, x_n : \langle U; u_n \rangle\}$ for some u_1, \dots, u_n ($n \geq 0$). If U' is an ML-type environment such that $\text{Dom}(U') \cap \text{Dom}(D) = \emptyset$, $\text{Dom}(U') \cap \text{Dom}(U) = \emptyset$, and $\text{Dom}(U') \cap \text{Dom}(A) = \emptyset$ and*

$$D_0 = \{y : \langle U \uplus U'; u_0 \rangle \mid y : \langle U; u_0 \rangle \in D\}.$$

Then

$$\begin{array}{c}
\text{(SPC)} \frac{D \vdash e : p}{D \vdash e : \mathbf{s}(p)} \quad \text{where } \mathbf{Dom}(\mathbf{s}) = \text{FTV}(p) \quad \text{(CON)} D \vdash c : \langle \emptyset; u \rangle \quad \text{where } u = \mathbf{type}(c) \quad \text{(VAR)} D \vdash x : \langle \{x : u\}; u \rangle \\
\text{where } u \in \mathbf{T}_0 \quad \text{and } x \notin \text{Dom}(D) \\
\text{(APP)} \frac{D \vdash e : \langle U; u_0 \rightarrow u \rangle \quad D \vdash e_0 : \langle U_0; u_0 \rangle}{D \vdash e e_0 : \langle U \oplus U_0; u \rangle} \\
\text{(ABS)} \frac{D \vdash e : \langle U, x : u_0; u \rangle}{D \vdash \lambda x. e : \langle U; u_0 \rightarrow u \rangle} \quad \text{where } x \notin D \quad \text{(ABSVAC)} \frac{D \vdash e : \langle A; u \rangle}{D \vdash \lambda x. e : \langle U; u_0 \rightarrow u \rangle} \\
\text{where } x \notin \text{FV}(e), u_0 \in \mathbf{T}_0, \text{ and } x \notin D \\
\text{(VARP)} D, x : p \vdash x : p \quad \text{(RECP)} \frac{D, x : \langle U; u \rangle \vdash e : \langle U; u \rangle}{D \vdash \mathbf{rec} \{x = e\} : \langle U; u \rangle} \\
\text{where } \text{Dom}(U) = \text{FV}_D(\mathbf{rec} \{x = e\}) \text{ and } x \notin D \\
\text{(IFSIMPLE)} \frac{D \vdash \text{ifc } e_0 e_1 e_2 : p}{D \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : p}
\end{array}$$

Fig. 7. Typing rules of system \vdash_0^P

1. $\text{Dom}(D_0) \cap \text{FV}(e) = \emptyset$ implies $D_0 \vdash_2^P e : \langle U \uplus A; u \rangle$, and
2. $\text{Dom}(D_0) \cap \text{FV}(e) \neq \emptyset$ implies $D_0 \vdash_2^P e : \langle U \uplus U' \uplus A; u \rangle$.

Proof. Straightforward, by structural induction on \vdash_2^P -derivations.

Recall the systems \vdash' and \vdash'' introduced in Section A.1.

Lemma 26. *If $B \vdash'' e : u$ and $B = U \uplus L \uplus R$ for some ML-type environment U and ML-scheme environments L, R . Then $D \vdash_2^P e : \langle U \uplus A; u \rangle$ where*

- (C1) $D = \{x : \langle U; u_0 \rangle \mid x : \forall \vec{\alpha}. u_0 \in R\}$, and
- (C2) $\text{Dom}(A) = \text{Dom}(L)$ and if $x : u_1 \wedge \dots \wedge u_n \in A$ then there exist an assumption $x : \forall \vec{\alpha}. u_0 \in L$ and substitutions $\mathbf{s}_1, \dots, \mathbf{s}_n$ such that (for all $i \in \{1, \dots, n\}$) $\mathbf{Dom}(\mathbf{s}_i) = \{\vec{\alpha}\}$ and $\mathbf{s}_i(u_0) = u_i$.

Proof. The proof is by structural induction on \vdash' -derivations. Let the derivation $U \uplus L \uplus R \vdash'' e : u$ end by rule:

(VAR $_\lambda$). Then $e = x$, $U = \{x : u\}$, and $L = R = \emptyset$ for some variable x . By rule (VAR), we have that $\emptyset \vdash_2^P x : \langle \{x : u\}; u \rangle$.

(VAR $_{\text{let}, \text{rec}}$). Then $e = x$, $U = \emptyset$, and $L \uplus R = \{x : \forall \vec{\alpha}. u'\}$, for some variable x and scheme $\forall \vec{\alpha}. u'$ such that $\mathbf{s}(u') = u$, where \mathbf{s} is a substitution such that $\mathbf{Dom}(\mathbf{s}) = \{\vec{\alpha}\}$. We consider two cases.

1. Let $L = \{x : \forall \vec{\alpha}. u'\}$. Then, by rule (VAR), we have that $\emptyset \vdash_2^P x : \langle \{x : u\}; u \rangle$.
2. Let $R = \{x : \forall \vec{\alpha}. u'\}$. Then, by rule (VARP), we have that $\{x : \langle \emptyset; u \rangle\} \vdash_2^P x : \langle \emptyset; u \rangle$.

(RECM). Then $e = \text{rec } \{x = e_0\}$ and $U \uplus L \uplus (R_0, x : \forall \vec{\alpha}. u') \vdash'' e_0 : u'$ for $\forall \vec{\alpha}. u' = \text{Gen}((U \uplus L \uplus R_0), u)$ such that $\mathbf{s}(u') = u$, where \mathbf{s} is a substitution such that $\text{Dom}(\mathbf{s}) = \{\vec{\alpha}\}$. By induction, $D \vdash_2^P e_0 : \langle U \uplus A; u' \rangle$ where D and A satisfy conditions (C1) and (C2), respectively. Since condition (C1) is satisfied we have $D = D_0, x : \langle U; u' \rangle$. Therefore, by rule (RECP), we get $D \vdash_2^P e : \langle U \uplus A; u' \rangle$ and, by rule (SPC), we get $D \vdash_2^P e : \langle U \uplus \mathbf{s}(A); u \rangle$, where D and $\mathbf{s}(A)$ satisfy conditions (C1) and (C2), respectively.

(LET). Then $e = \text{let } x = e_0 \text{ in } e_1, U \uplus L \uplus R_0 \vdash' e_0 : u_0$, and $U \uplus (L, x : \forall \vec{\alpha}. u_0) \uplus R_0 \vdash' e_1 : u$ where $\forall \vec{\alpha}. u_0 = \text{Gen}((U \uplus L \uplus R_0), u)$. We have two different cases.

– Let $x \in \text{FV}(e_1)$. Then, by Lemma 24, $U' \uplus L' \uplus R'_0 \vdash'' e_0 : u_0$ and $U'' \uplus (L'', x : \forall \vec{\alpha}. u_0) \uplus R''_0 \vdash'' e_1 : u$, where $U' \oplus U'' = U, L' \oplus L'' = L$, and $R' \oplus R'' = R$. By induction, $D' \vdash_2^P e_0 : \langle U' \uplus A'; u' \rangle$ and $D'' \vdash_2^P e_1 : \langle U'' \uplus (A'', x : w); u \rangle$, where conditions (C1) and (C2) are satisfied. By rule (SPC), $D' \vdash_2^P e_0 : \langle U' \uplus \mathbf{s}_i(A'); u_i \rangle$, for all $i \in \{1, \dots, n\}$. By Lemma 25 we have that

- If $\text{Dom}(D) \cap \text{Dom}(\text{FV}(e_0)) = \emptyset$. Then $D \vdash_2^P e_1 : \langle U' \uplus U'' \uplus (A'', x : w); u \rangle$ and $D \vdash_2^P e_0 : \langle U' \uplus A'; u_i \rangle$, for all $i \in \{1, \dots, n\}$, where $D = \{y : \langle U; u'' \rangle \mid y : \langle U'; u'' \rangle \in D' \text{ or } y : \langle U''; u'' \rangle \in D''\}$. Therefore, by rules (ABS) and (APP), $D \vdash_2^P (\lambda x. e_1) e_0 : \langle U \uplus A'' \wedge \mathbf{s}_1(A') \wedge \dots \wedge \mathbf{s}_n(A'); u \rangle$, where D and $A'' \wedge \mathbf{s}_1(A') \wedge \dots \wedge \mathbf{s}_n(A')$ satisfy conditions (C1) and (C2), respectively.
- If $\text{Dom}(D) \cap \text{Dom}(\text{FV}(e_0)) \neq \emptyset$. Then $D \vdash_2^P e_1 : \langle U' \uplus U'' \uplus (A'', x : w); u \rangle$ and $D \vdash_2^P e_0 : \langle U' \uplus U'' \uplus A'; u_i \rangle$, for all $i \in \{1, \dots, n\}$, where $D = \{y : \langle U; u'' \rangle \mid y : \langle U'; u'' \rangle \in D' \text{ or } y : \langle U''; u'' \rangle \in D''\}$. Then, we can conclude as in the case $\text{Dom}(D) \cap \text{Dom}(\text{FV}(e_0)) = \emptyset$.

– The case $x \notin \text{FV}(e_1)$ is similar to the case $x \in \text{FV}(e_1)$ above.

(CON). Immediate.

(APP), (ABS), or (IF). Straightforward by induction, using Lemma 24 and Lemma 25.

Restatement of Theorem 11. *If $\emptyset \vdash e : u$ is Milner-Mycroft derivable, then $\emptyset \vdash_2^P e : \langle \emptyset; u \rangle$.*

Proof. Immediate by Lemma 26 and Lemma 24.

A.4 Proof of Theorem 13

Recall the system \vdash' introduced in Section A.1.

Lemma 27. *Let e be a let-free expression. If $B \vdash' e : u$ and $B = U \uplus R$ for some ML-type environment U and ML-scheme environment R . Then $D \vdash_0^P e : \langle U; u \rangle$, where $D = \{x : \langle U; u_0 \rangle \mid x : \forall \vec{\alpha}. u_0 \in R\}$.*

Proof. Similar to the proof of Lemma 26.

Lemma 28. *Let e be a let-free expression. If $D \vdash_0^P e : \langle U; u \rangle$, with $D = \{x_1 : \langle U_1; u_1 \rangle, \dots, x_n : \langle U_n; u_n \rangle\}$ ($n \geq 0$), and for all $i \in \{1, \dots, n\}$ $\langle \bar{U}_i; \bar{u}_i \rangle$ has been obtained from $\langle U_i; u_i \rangle$ by renaming all its type variables in such a way that*

- the sets $\text{FTV}(\langle U; u \rangle)$, $\text{FTV}(\langle \bar{U}_1; \bar{u}_1 \rangle)$, \dots , $\text{FTV}(\langle \bar{U}_n; \bar{u}_n \rangle)$ are pairwise disjoint, and
- the sets $\bigoplus_{1 \leq i \leq n} \text{FTV}(\langle \bar{U}_i; \bar{u}_i \rangle)$ and $\bigoplus_{1 \leq i \leq n} \text{FTV}(\langle U_i; u_i \rangle)$ are disjoint.

Then for all substitutions \bar{s} such that

(C) $\text{Dom}(\bar{s}) = \text{FTV}(U) \cup (\bigcup_{1 \leq i \leq n} \text{FTV}(\bar{U}_i))$ and $\bar{s}(U) \oplus (\bigoplus_{1 \leq i \leq n} \bar{s}(\bar{U}_i))$ is defined

it holds that $(\bar{s}(U) \oplus (\bigoplus_{1 \leq i \leq n} \bar{s}(\bar{U}_i))) \uplus R \vdash' e : \bar{s}(u)$ with $R = \{x_1 : \text{Gen}(\bar{s}(\bar{U}_1), \bar{s}(\bar{u}_1)), \dots, x_n : \text{Gen}(\bar{s}(\bar{U}_n), \bar{s}(\bar{u}_n))\}$.

Proof. The proof is by structural induction on \vdash_0^P -derivations. Assume that $D \vdash_0^P e : \langle U; u \rangle$, with $D = \{x_1 : \langle U_1; u_1 \rangle, \dots, x_n : \langle U_n; u_n \rangle\}$, and that \bar{s} is a substitution that satisfies condition (C). Let the derivation $D \vdash_0^P e : \langle U; u \rangle$ end by rule:

- (SPC). Then $\langle U; u \rangle = \mathbf{s}(\langle U'; u' \rangle)$ for some \mathbf{s} and $\langle U'; u' \rangle$ such that $D \vdash_0^P e : \langle U'; u' \rangle$ and $\text{Dom}(\mathbf{s}) = \text{FTV}(\langle U'; u' \rangle)$. Let $\mathbf{s}' = \bar{s} \circ \mathbf{s}$. Assume, without loss of generality, that $\text{FTV}(\langle U'; u' \rangle)$ is disjoint from $\text{FTV}(\langle U; u \rangle) \cup (\bigcup_{1 \leq i \leq n} \text{FTV}(\langle \bar{U}_i; \bar{u}_i \rangle))$. Since $\text{Dom}(\mathbf{s})$ is disjoint from $\text{FTV}(\langle U; u \rangle) \cup (\bigcup_{1 \leq i \leq n} \text{FTV}(\langle \bar{U}_i; \bar{u}_i \rangle))$ we have that $\mathbf{s}'(U) \oplus (\bigoplus_{1 \leq i \leq n} \mathbf{s}'(\bar{U}_i))$ is defined. Therefore, by induction, $(\mathbf{s}'(U) \oplus (\bigoplus_{1 \leq i \leq n} \mathbf{s}'(\bar{U}_i))) \uplus R' \vdash' e : \mathbf{s}'(u)$, with $R' = \{x_1 : \text{Gen}(\mathbf{s}'(\bar{U}_1), \mathbf{s}'(\bar{u}_1)), \dots, x_n : \text{Gen}(\mathbf{s}'(\bar{U}_n), \mathbf{s}'(\bar{u}_n))\}$. So, by (\star) , we can conclude that $R' = R$ and $(\bar{s}(U') \oplus (\bigoplus_{1 \leq i \leq n} \bar{s}(\bar{U}_i))) \uplus R \vdash' e : \bar{s}(u')$.
- (VAR). Then $e = x$ and $U = \{x : u\}$, for some variable $x \notin \text{Dom}(D)$. By rule (VAR_λ) , we have that $(\{x : \bar{s}(u)\} \oplus (\bigoplus_{1 \leq i \leq n} \bar{s}(\bar{U}_i))) \uplus R \vdash' x : \bar{s}(u)$.
- (VARP). Then $e = x_h$ and $x_h : \langle U_h; u_h \rangle \in D$ with $\langle U; u \rangle = \langle U_h; u_h \rangle$ for some $h \in \{1, \dots, n\}$. Assume, without loss of generality, that $h = 1$. By rule $(\text{VAR}_{\text{let}, \text{rec}})$, we have that $(\bar{s}(U) \oplus (\bigoplus_{1 \leq i \leq n} \bar{s}(\bar{U}_i))) \uplus R \vdash' e : \bar{s}(u)$ with $R = \{x_1 : \text{Gen}(\bar{s}(\bar{U}_1), \bar{s}(\bar{u}_1)), \dots, x_n : \text{Gen}(\bar{s}(\bar{U}_n), \bar{s}(\bar{u}_n))\}$, $\bar{s}(\bar{U}) = \bar{s}(\bar{U}_1)$, $\text{Gen}(\bar{s}(\bar{U}), \bar{s}(\bar{u})) = \text{Gen}(\bar{s}(\bar{U}_1), \bar{s}(\bar{u}_1)) = \forall \bar{\alpha}. u'$, and $\mathbf{s}(u') = \bar{s}(u)$, for some substitution \mathbf{s} that performs just a renaming of the type variables $\bar{\alpha}$.
- (RECP). Then $e = \text{rec } \{x = e_0\}$ for some e_0 and $D, x : \langle U; u \rangle \vdash_0^P e_0 : \langle U; u \rangle$. Let $\langle \bar{U}; \bar{u} \rangle$ be obtained from $\langle U; u \rangle$ by renaming all its type variables in such a way that $\text{FTV}(\langle \bar{U}; \bar{u} \rangle)$ is disjoint from $\text{FTV}(\langle U; u \rangle) \cup (\bigcup_{1 \leq i \leq n} \text{FTV}(\langle \bar{U}_i; \bar{u}_i \rangle))$. Let \mathbf{s}_0 be a substitution such that $\text{Dom}(\mathbf{s}_0) = \text{FTV}(\bar{U})$ and $\mathbf{s}_0(\bar{U}) = U$, and let $\mathbf{s}' = \bar{s} \circ \mathbf{s}_0$. It holds that $\text{Dom}(\mathbf{s}_0)$ is disjoint from $\text{FTV}(\langle U; u \rangle) \cup (\bigcup_{1 \leq i \leq n} \text{FTV}(\langle \bar{U}_i; \bar{u}_i \rangle))$. The fact that condition (C) holds implies that $(\mathbf{s}'(\bar{U}) \oplus (\bigoplus_{1 \leq i \leq n} \mathbf{s}'(\bar{U}_i))) \oplus \bar{s}'(\bar{U}))$ is defined. Therefore, by induction, $((\mathbf{s}'(\bar{U}) \oplus (\bigoplus_{1 \leq i \leq n} \mathbf{s}'(\bar{U}_i))) \oplus \bar{s}'(\bar{U})) \uplus (R, x : \text{Gen}(\mathbf{s}'(\bar{U}), \bar{s}(\bar{u}_1))) \vdash' e_0 : \mathbf{s}'(u)$. Since $\text{Gen}(\mathbf{s}'(\bar{U}), \mathbf{s}'(\bar{u})) = \forall \bar{\alpha}. \mathbf{s}'(\bar{u})$ and $\bar{\alpha} = \text{FTV}(\mathbf{s}'(\bar{u})) - \text{FTV}(\mathbf{s}'(\bar{U}))$. By rule (RECMM) we get $(\mathbf{s}'(\bar{U}) \oplus (\bigoplus_{1 \leq i \leq n} \mathbf{s}'(\bar{U}_i))) \uplus R \vdash' e : \mathbf{s}'(u)$. Therefore, by $(\star\star)$, we can conclude that $(\bar{s}(\bar{U}) \oplus (\bigoplus_{1 \leq i \leq n} \bar{s}(\bar{U}_i))) \uplus R \vdash' e : \bar{s}(u)$.
- (APP). Then $e = e' e_0$, $D \vdash_0^P e' : \langle U'; u_0 \rightarrow u \rangle$, and $D \vdash_0^P e_0 : \langle U_0; u_0 \rangle$, where $U' \uplus U_0 = U$. The fact that condition (C) holds implies

- the restriction \bar{s}' of \bar{s} to the domain $\text{FTV}(U') \cup (\cup_{1 \leq i \leq n} \text{FTV}(\bar{U}_i))$ is such that $\bar{s}'(U') \oplus (\oplus_{1 \leq i \leq n} \bar{s}'(\bar{U}_i))$ is defined, and
- the restriction \bar{s}_0 of \bar{s} to the domain $\text{FTV}(U_0) \cup (\cup_{1 \leq i \leq n} \text{FTV}(\bar{U}_i))$ is such that $\bar{s}_0(U_0) \oplus (\oplus_{1 \leq i \leq n} \bar{s}_0(\bar{U}_i))$ is defined.

Therefore, by induction,

- $\bar{s}'(U') \oplus (\oplus_{1 \leq i \leq n} \bar{s}'(\bar{U}_i)) \uplus R_0 \vdash' e' : \mathbf{s}'(u_0 \rightarrow u)$ with $R' = R$, and
- $\bar{s}_0(U_0) \oplus (\oplus_{1 \leq i \leq n} \bar{s}_0(\bar{U}_i)) \uplus R' \vdash' e_0 : \mathbf{s}_0(u_0)$ with $R_0 = R$.

By Lemma 23 we have that

- $\bar{s}(U) \oplus (\oplus_{1 \leq i \leq n} \bar{s}(\bar{U}_i)) \uplus R \vdash' e' : \mathbf{s}'(u_0 \rightarrow u)$, and
- $\bar{s}(U_0) \oplus (\oplus_{1 \leq i \leq n} \bar{s}(\bar{U}_i)) \uplus R \vdash' e_0 : \mathbf{s}(u_0)$.

So, by rule (APP), we get $(\bar{s}(U) \oplus (\oplus_{1 \leq i \leq n} \bar{s}(\bar{U}_i))) \uplus R \vdash' e : \bar{s}(u)$.

(CON). Immediate.

(ABS), (ABSVAC), or (IFSIMPLE). Straightforward by induction.

Restatement of Theorem 13. *Let e be a let-free expression. Then $\emptyset \vdash_0^P e : \langle \emptyset; u \rangle$ iff $\emptyset \vdash e : u$ is Milner-Mycroft derivable.*

Proof. Immediate by Lemma 27 and Lemma 28 (observing that, for $D = \emptyset$ and $U = \emptyset$, condition (C) in Lemma 28 is satisfied by the empty substitution).

A.5 Proof of Theorem 17

The algorithms MS_i ($0 \leq i \leq 2$), defined in Fig. 8, take a pair of types $t, t' \in \mathbf{T}_i$ such that $\text{FTV}(t) \cap \text{FTV}(t') = \emptyset$ and return a set of substitutions, $\text{MS}_i(t, t')$, that we call the “set of *matching substitutions* on t against t' ”. The fundamental property of the algorithms MS_i is given by the following lemma.

Lemma 29. *1. Let $u, u' \in \mathbf{T}_i$ and $\text{FTV}(u) \cap \text{FTV}(u') = \emptyset$. Then*

$$\text{MS}_0(u, u') = \{\mathbf{s} \mid \text{Dom}(\mathbf{s}) = \text{FTV}(u) - \text{FTV}(u') \text{ and } \mathbf{s}(u) = u'\}.$$

2. Let $w, w' \in \mathbf{T}_i$ and $\text{FTV}(w) \cap \text{FTV}(w') = \emptyset$. Then

$$\text{MS}_1(w, w') = \{\mathbf{s} \mid \text{Dom}(\mathbf{s}) = \text{FTV}(w) - \text{FTV}(w') \text{ and } w' \leq_1 \mathbf{s}(w)\}.$$

3. Let $v, v' \in \mathbf{T}_i$ and $\text{FTV}(v) \cap \text{FTV}(v') = \emptyset$. Then

$$\text{MS}_2(v, v') = \{\mathbf{s} \mid \text{Dom}(\mathbf{s}) = \text{FTV}(v) - \text{FTV}(v') \text{ and } \mathbf{s}(v) \leq_2 v'\}.$$

Proof. By induction on the definition of MS_i ($0 \leq i \leq 2$), using Definition 3.

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

Proof. By Lemma 29, observing that, for all pairs $\langle \{x_1 : w_1, \dots, x_n : w_n\}; v \rangle$ and $\langle \{x_1 : w'_1, \dots, x_n : w'_n\}; v' \rangle$, we have that $\langle \{x_1 : w_1, \dots, x_n : w_n\}; v \rangle \leq_{\text{SPC}} \langle \{x_1 : w'_1, \dots, x_n : w'_n\}; v' \rangle$ if and only if $\langle \emptyset; w_1 \rightarrow \dots \rightarrow w_n \rightarrow v \rangle \leq_{\text{SPC}} \langle \emptyset; w'_1 \rightarrow \dots \rightarrow w'_n \rightarrow v' \rangle$ if and only if $\text{MS}_2(w_1 \rightarrow \dots \rightarrow w_n \rightarrow v, w'_1 \rightarrow \dots \rightarrow w'_n \rightarrow v') \neq \emptyset$.

(For all $i \in \{0, 1, 2\}$) $MS_i(t, t') = MS'_i(t, t', \text{FTV}(t'))$, where

$$\begin{aligned}
MS'_0(u, u, W) &= \{[]\} \\
MS'_0(\alpha, u, W) &= \{[\alpha := u]\}, \text{ if } \alpha \notin W \\
MS'_0(u_1 \rightarrow u_2, u'_1 \rightarrow u'_2, W) &= \{\mathbf{s}_2 \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in MS'_0(u_1, u'_1, W) \\
&\quad \text{and } \mathbf{s}_2 \in MS'_0(\mathbf{s}_1(u_2), u'_2, W)\} \\
MS'_0(u_1 \times u_2, u'_1 \times u'_2, W) &= \{\mathbf{s}_2 \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in MS'_0(u_1, u'_1, W) \\
&\quad \text{and } \mathbf{s}_2 \in MS'_0(\mathbf{s}_1(u_2), u'_2, W)\} \\
MS'_0(u \text{ list}, u' \text{ list}, W) &= MS'_0(u, u', W) \\
MS'_0(u, u', W) &= \emptyset, \text{ otherwise} \\
MS'_1(u_1 \wedge \dots \wedge u_m, u'_1 \wedge \dots \wedge u'_n, W) &= \{\mathbf{s}_m \circ \dots \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in MS'_0(u_1, u'_{i_1}, W), \\
&\quad \mathbf{s}_2 \in MS'_0(\mathbf{s}_1(u_2), u'_{i_2}, W), \\
&\quad \dots, \\
&\quad \mathbf{s}_m \in MS'_0(\mathbf{s}_{m-1} \circ \dots \circ \mathbf{s}_1(u_m), u'_{i_m}, W), \\
&\quad \text{and } i_1, \dots, i_m \in \{1, \dots, n\}\} \\
MS'_2(u, u', W) &= MS'_0(u, u', W) \\
MS'_2(\alpha, w \rightarrow v, W) &= \{\mathbf{s}_2 \circ \mathbf{s}_1 \circ [\alpha := \alpha_1 \rightarrow \alpha_2] \mid \mathbf{s}_1 \in MS'_1(\alpha_1, w, W) \\
&\quad \text{and } \mathbf{s}_2 \in MS'_2(\alpha_2, v, W)\}, \\
&\quad \text{if } \alpha \notin W, \text{ for some fresh } \alpha_1 \text{ and } \alpha_2 \\
MS'_2(w \rightarrow v, w' \rightarrow v', W) &= \{\mathbf{s}_2 \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in MS'_1(w, w', W) \\
&\quad \text{and } \mathbf{s}_2 \in MS'_2(\mathbf{s}_1(v), v', W)\} \\
MS'_2(v, v', W) &= \emptyset, \text{ otherwise}
\end{aligned}$$

Fig. 8. Algorithms MS_0 , MS_1 , and MS_2

A.6 Proof of Theorem 18

The results presented in this section are due to Jim [13,12].

Definition 30 (Most general solutions of a $\leq_{\forall 2,1}$ -satisfaction problem).

1. A substitution \mathbf{s} is a *most general solution (mgs)* of a $\leq_{\forall 2,1}$ -satisfaction problem $\exists \vec{\alpha}.P$ if it satisfies the following conditions:⁵
 - \mathbf{s} is a solution of $\exists \vec{\alpha}.P$,
 - $\mathbf{s} \leq \mathbf{s}'$, for all solutions \mathbf{s}' of $\exists \vec{\alpha}.P$,
 - \mathbf{s} is idempotent, and
 - $\text{Dom}(\mathbf{s}) \subseteq \text{FTV}(\exists \vec{\alpha}.P)$.
2. We write $\mathbf{MGS}(\exists \vec{\alpha}.P)$ for the (possibly empty) set of the most general solutions of $\exists \vec{\alpha}.P$.

⁵ The last two conditions are included for technical convenience only. Indeed they can be eliminated: a $\leq_{\forall 2,1}$ -satisfaction problem has a solution that satisfies the first two conditions if and only if it has a solution that satisfies all the four conditions.

We say that two $\leq_{\forall 2,1}$ -satisfaction problems are *equivalent* if they have the same solutions.

An *unification problem* is a $\leq_{\forall 2,1}$ -satisfaction problem which involves only equalities.⁶

Lemma 31. *Every $\leq_{\forall 2,1}$ -satisfaction problem is equivalent to an unification problem. In particular, there is an algorithm that, given a $\leq_{\forall 2,1}$ -satisfaction problem, either proves that it can not be satisfied or transforms it into an equivalent unification problem.*

Proof. Following [13,12] (see also [6]), we prove the lemma by providing the transformation algorithm. We first introduce a set of transformation rules of the form

$$t \leq w \Rightarrow \exists \vec{\alpha}.P \quad (\text{where } t \in \mathbf{T}_2 \cup \mathbf{T}_{\forall 2}).$$

Each rule transforms an inequality in a $\leq_{\forall 2,1}$ -satisfaction problem. The transformation rules are in Fig. 9. Note that, for each transformation rule $t \leq w \Rightarrow \exists \vec{\alpha}.P$, we have that

$$\text{FTV}(t) \cup \text{FTV}(w) = \text{FTV}(\exists \vec{\alpha}.P). \quad (1)$$

In particular, the type variables $\vec{\alpha}$ are either fresh type variables introduced by the transformation rule (i.e., they do not appear in the left-hand side) or are the bound variables of t (when $t \in \mathbf{T}_{\forall 2}$).

The transformation algorithm is specified as a rewrite relation on problems, \Longrightarrow , defined by the rule:

$$(\Longrightarrow) \frac{vs \leq w \Rightarrow \exists \vec{\alpha}.P}{\exists \vec{\beta}.(P' \uplus \{vs \leq w\}) \Longrightarrow \exists \vec{\beta} \uplus \vec{\alpha}.(P' \cup P)}$$

where the operator “ \uplus ” denotes disjoint union (this implies that the variables $\vec{\alpha}$ must be fresh).

To see that the rewriting relation \Longrightarrow describes an algorithm which proves the lemma, observe that:

1. every rewriting step (corresponding to the application of one of the rules in Fig. 9) transforms a $\leq_{\forall 2,1}$ -satisfaction problem into another $\leq_{\forall 2,1}$ -satisfaction problem,
2. every rewriting step preserves the set of solutions (note that the condition (1) above is necessary to ensure this preservation),
3. every inequality matches the left-hand side of at most one rule, and
4. repeated application of these rules must terminate.⁷ This is proved by defining a metric $|\cdot|$ on problems for which each rewriting step is strictly decreasing. The size of inequalities will be defined by structural induction. By point

⁶ An unification problem can be solved by Robinson’s algorithm (see, for instance, Chapter 3 of [11]), which can decide whether it is solvable, and, if so, return a most general solution.

⁷ In the original termination proof given in [13,12] there is an error. Here we are following the termination proof given in [14].

$$\begin{aligned}
u_0 \leq u &\Rightarrow \exists \epsilon. \{u_0 = u\} \\
&\text{where } u_0, u \in \mathbf{T}_0 \text{ and } u_0 \text{ is not an arrow type} \\
(w \rightarrow v) \leq \alpha &\Rightarrow \exists \alpha_1 \alpha_2. \{\alpha_1 \leq w, v \leq \alpha_2, \alpha = \alpha_1 \rightarrow \alpha_2\} \\
&\text{where } \alpha_1, \alpha_2 \text{ are fresh} \\
(w \rightarrow v) \leq (u_1 \rightarrow u_2) &\Rightarrow \exists \epsilon. \{u_1 \leq w, v \leq u_2\} \\
t \leq (u_1 \wedge \dots \wedge u_n) &\Rightarrow \exists \epsilon. \{t \leq u_1, \dots, t \leq u_n\} \\
&\text{where } t \in \mathbf{T}_2 \cup \mathbf{T}_{\forall 2}, n \geq 2, \text{ and } u_1, \dots, u_n \in \mathbf{T}_0 \\
(\forall \vec{\alpha}. v) \leq u &\Rightarrow \exists \vec{\alpha}. \{v \leq u\} \\
&\text{where } u \in \mathbf{T}_0 \text{ and } \{\vec{\alpha}\} \cap \text{FTV}(u) = \emptyset
\end{aligned}$$

Fig. 9. Transformation rules for $\leq_{\forall 2,1}$ -satisfaction

(3), it is sufficient to consider the cases given by the left-hand sides of the rules in Fig. 9. For the base case we set $|u_0 \leq u| = 1$. Every other case is simply defined so that the size of the left-hand side of a rule is greater than the sum of the of the inequalities appearing on the corresponding right-hand side. For example, define

- $|(w \rightarrow v) \leq \alpha| = |\alpha_1 \leq w| + |v \leq \alpha_2| + 1$,
- $|(w \rightarrow v) \leq (u_1 \rightarrow u_2)| = |u_1 \leq w| + |v \leq u_2| + 1$,
- $|t \leq (u_1 \wedge \dots \wedge u_n)| = |t \leq u_1| + \dots + |t \leq u_n| + 1$, and
- $|(\forall \vec{\alpha}. v) \leq u| = |v \leq u| + 1$.

The function $|\cdot|$ is defined by structural induction (since types on right-hand sides either appear as syntactic subtypes on left-hand sides, or are type variables) and, by construction, gives a decreasing metric on problems.

Normal forms are either unification problems (i.e. do not contain inequalities) or contain at least an inequality of the form $w \rightarrow v \leq u_0$ where u_0 is a simple type which is neither a type variable nor an arrow type (such an inequality is clearly not satisfiable).

Restatement of 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.*

Proof. Immediate, by Lemma 31.

A.7 Proof of Theorem 20

Lemma 32. *For every $k \geq 1$, expression e , and environment D , if $\langle A; v \rangle \in \mathbf{PP}_k(D, e)$, then*

1. $\text{Dom}(A) = (\text{FV}(e) - \text{Dom}(D)) \cup \text{VR}(D|_{\text{FV}(e)})$, and

2. $\langle A'; v' \rangle \in \mathbf{PP}_k(D, e)$ if and only if there is a bijection $\mathbf{s} : \mathbf{T}\mathbf{v} \rightarrow \mathbf{T}\mathbf{v}$ such that $\mathbf{s}(A) = A'$ and $\mathbf{s}(v) = v'$.

Proof. By induction on Definition 19.

Recall the definitions of \mathbf{P}_X and \mathbf{bot}_X from Section 6.4. Let $\leq_{\mathbf{spc}}^{\mathbf{err}}$ be the extension of $\leq_{\mathbf{spc}}$ to the set $\mathbf{P}_X^{\mathbf{err}} = \mathbf{P}_X \cup \{\mathbf{err}\}$, where \mathbf{err} is a new element (not belonging to \mathbf{P}_X) such that, for all $p \in \mathbf{P}_X^{\mathbf{err}}$, $p \leq_{\mathbf{spc}}^{\mathbf{err}} \mathbf{err}$. Let $\equiv_{\mathbf{spc}}^{\mathbf{err}}$ be the equivalence relation on $\mathbf{P}_X^{\mathbf{err}}$ induced by $\leq_{\mathbf{spc}}^{\mathbf{err}}$ and let

- $\mathbf{P}_X^{\mathbf{err}}$ be the quotient set $\mathbf{P}_X^{\mathbf{err}} / \equiv_{\mathbf{spc}}^{\mathbf{err}}$,
- $\leq_{\mathbf{spc}}$ be the partial order relation over $\mathbf{P}_X^{\mathbf{err}}$ induced by the preorder relation $\leq_{\mathbf{spc}}^{\mathbf{err}}$ over $\mathbf{P}_X^{\mathbf{err}}$,
- $\mathbf{bot}_X = [\mathbf{bot}_X]_{\equiv_{\mathbf{spc}}^{\mathbf{err}}}$, and
- $\mathbf{err} = [\mathbf{err}]_{\equiv_{\mathbf{spc}}^{\mathbf{err}}}$.

For every $k \geq 1$, pair environment D , and expression $\mathbf{rec}\{x = e\}$, define the function $\mathcal{F}_{\vdash_2^{\mathbf{P}_k + \mathbf{J}}}^{D, \mathbf{rec}\{x=e\}} : \mathbf{P}_{\mathbf{FV}_D(\mathbf{rec}\{x=e\})}^{\mathbf{err}} \rightarrow \mathbf{P}_{\mathbf{FV}_D(\mathbf{rec}\{x=e\})}^{\mathbf{err}}$ such that, for all $[p]_{\equiv_{\mathbf{spc}}^{\mathbf{err}}} \in \mathbf{P}_{\mathbf{FV}_D(\mathbf{rec}\{x=e\})}$,

$$\mathcal{F}_{\vdash_2^{\mathbf{P}_k + \mathbf{J}}}^{D, \mathbf{rec}\{x=e\}}([p]_{\equiv_{\mathbf{spc}}^{\mathbf{err}}}) = \begin{cases} [p']_{\equiv_{\mathbf{spc}}^{\mathbf{err}}}, & \text{for some } p' \in \mathbf{PP}_k(D, \mathbf{rec}\{x = e\}), \\ & \text{if } \mathbf{PP}_k(D, \mathbf{rec}\{x = e\}) \neq \emptyset \\ \mathbf{err}, & \text{otherwise} \end{cases}$$

and

$$\mathcal{F}_{\vdash_2^{\mathbf{P}_k + \mathbf{J}}}^{D, \mathbf{rec}\{x=e\}}(\mathbf{err}) = \mathbf{err}.$$

The function $\mathcal{F}_{\vdash_2^{\mathbf{P}_k + \mathbf{J}}}^{D, \mathbf{rec}\{x=e\}}$ is monotone w.r.t. $\leq_{\mathbf{spc}}$, i.e., for all $\mathbf{p}, \mathbf{p}' \in \mathbf{P}_{\mathbf{FV}_D(\mathbf{rec}\{x=e\})}^{\mathbf{err}}$,

if $\mathbf{p} \leq_{\mathbf{spc}} \mathbf{p}'$ then $\mathcal{F}_{\vdash_2^{\mathbf{P}_k + \mathbf{J}}}^{D, \mathbf{rec}\{x=e\}}(\mathbf{p}) \leq_{\mathbf{spc}} \mathcal{F}_{\vdash_2^{\mathbf{P}_k + \mathbf{J}}}^{D, \mathbf{rec}\{x=e\}}(\mathbf{p}')$ — the proof is straightforward by using the definition of the inference algorithm \mathbf{PP}_k (Definition 19). Therefore we have that, if for some $h \geq 0$

$$\underbrace{\mathcal{F}_{\vdash_2^{\mathbf{P}_k + \mathbf{J}}}^{D, \mathbf{rec}\{x=e\}}(\dots \mathcal{F}_{\vdash_2^{\mathbf{P}_k + \mathbf{J}}}^{D, \mathbf{rec}\{x=e\}}(\mathbf{bot}_{\mathbf{FV}(\mathbf{rec}\{x=e\})}) \dots)}_{h \text{ times}}$$

is a fixed point of $\mathcal{F}_{\vdash_2^{\mathbf{P}_k + \mathbf{J}}}^{D, \mathbf{rec}\{x=e\}}$, then it is the minimum fixed point (and pre-fixed point).

Restatement of Theorem 20 (Soundness and completeness of \mathbf{PP}_k for $\vdash_2^{\mathbf{P}_k + \mathbf{J}}$). For every $k \geq 1$, expression e , and environment D :

(Soundness). If $p \in \mathbf{PP}_k(D, e)$, then $D \vdash_2^{\mathbf{P}_k + \mathbf{J}} e : p$.

(Completeness). If $D \vdash_2^{\mathbf{P}_k + \mathbf{J}} e : p'$, then $p \leq_{\mathbf{spc}} p'$ for some $p \in \mathbf{PP}_k(D, e)$.

Proof. Both soundness and completeness by structural induction on e .

$e = x$.

(Soundness). We have to consider two cases.

- If $x : \langle A_0; v_0 \rangle \in D$ with $\vec{\alpha} = \text{FTV}(A_0) \cup \text{FTV}(v_0)$, then $\langle \mathbf{s}(A_0); \mathbf{s}(v_0) \rangle \in \mathbf{PP}_k(D, x)$, for some renaming \mathbf{s} of $\vec{\alpha}$. We have $D \vdash x : \langle A_0; v_0 \rangle$ by rule (VARP) and $D \vdash x : \langle \mathbf{s}(A_0); \mathbf{s}(v_0) \rangle$ by rule (SPC).
- If $x \notin \text{Dom}(D)$, use rule (VAR).

(Completeness). Again, we have to consider two cases.

- If $x : \langle A_0; v_0 \rangle \in D$, then the derivation must end by $l \geq 0$ applications of rule (SPC) preceded by an application of rule (VARP).
- If $x \notin \text{Dom}(D)$, then the derivation must end by $l \geq 0$ applications of rule (SPC) preceded by an application of rule (VAR).

In both the cases the proof is immediate.

$e = c$.

(Soundness). Use rule (CON).

(Completeness). Immediate, since the derivation must end by $l \geq 0$ applications of rule (SPC) preceded by an application of rule (CON).

$e = \lambda x.e_0$.

(Soundness). We have to consider two cases.

- If $x \in \text{FV}(e_0)$, then $\langle A, x : w; v_0 \rangle \in \mathbf{PP}_k(D, e_0)$ and $\langle A; w \rightarrow v_0 \rangle \in \mathbf{PP}_k(D, \lambda x.e_0)$. By induction, $D \vdash e_0 : \langle A, x : w; v_0 \rangle$, and by rule (ABS) we get:

$$D \vdash \lambda x.e_0 : \langle A; w \rightarrow v_0 \rangle.$$

- If $x \notin \text{FV}(e_0)$, then $\langle A; v_0 \rangle \in \mathbf{PP}_k(D, e_0)$ and $\langle A; \alpha \rightarrow v_0 \rangle \in \mathbf{PP}_k(D, \lambda x.e_0)$ for some fresh type variable α . By induction, $D \vdash e_0 : \langle A; v_0 \rangle$, and by rule (ABSVAC) we get:

$$D \vdash \lambda x.e_0 : \langle A; \alpha \rightarrow v_0 \rangle.$$

(Completeness). Let $p' = \langle A'; v' \rangle$. Again, we have to consider two cases.

- If $x \in \text{FV}(e_0)$, then the derivation must end by $l \geq 0$ applications of rule (SPC) preceded by an application of rule (ABS). Assume, without loss of generality, that $l = 0$. We have $v' = w' \rightarrow v'_0$ and $D \vdash e_0 : \langle A', x : w'; v'_0 \rangle$. By induction there exist a fresh pair $\langle \langle A, x : w \rangle; v_0 \rangle \in \mathbf{PP}_k(D, e_0)$ such that

$$\langle \langle A, x : w \rangle; v_0 \rangle \leq_{\mathbf{spc}} \langle A', x : w'; v'_0 \rangle.$$

So $\langle A; w \rightarrow v_0 \rangle \in \mathbf{PP}_k(D, \lambda x.e_0)$ is the desired pair.

- If $x \notin \text{FV}(e_0)$, then the derivation must end by $l \geq 0$ applications of rule (SPC) preceded by an application of rule (ABSVAC). Assume, without loss of generality, that $l = 0$. We have $v' = u \rightarrow v'_0$ and $D \vdash e_0 : \langle A'; v'_0 \rangle$. By induction there exist a fresh pair $\langle A; v_0 \rangle \in \mathbf{PP}_k(D, e_0)$ such that

$$\langle A; v_0 \rangle \leq_{\mathbf{spc}} \langle A'; v'_0 \rangle.$$

So $\langle A; \alpha \rightarrow v_0 \rangle \in \mathbf{PP}_k(D, \lambda x.e_0)$, where α is a fresh type variable, is the desired pair.

$e = e_0 e_1$.

(Soundness). We have $\langle A_0; v_0 \rangle \in \mathbf{PP}_k(D, e_0)$ and, by induction, $D \vdash e_0 : \langle A_0; v_0 \rangle$.

- If v_0 is a type variable α , then there exists a fresh pair $\langle A_1; v_1 \rangle \in \mathbf{PP}_k(D, e_1)$ such that $p = \langle \mathbf{s}(A_0 \wedge A_1); \mathbf{s}(\alpha_2) \rangle$, where $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{v_1 \leq \alpha_1, \alpha = \alpha_1 \rightarrow \alpha_2\})$ for fresh type variables α_1 and α_2 . By induction, $D \vdash e_1 : \langle A_1; v_1 \rangle$. By rule (SPC)

$$D \vdash e_0 : \langle \mathbf{s}(A_0); \mathbf{s}(\alpha) \rangle, \text{ and}$$

$$D \vdash e_1 : \langle \mathbf{s}(A_1); \mathbf{s}(v_1) \rangle.$$

Then, by rule (APP), we have:

$$D \vdash e_0 e_1 : p.$$

- If $v_0 = u_1 \wedge \dots \wedge u_n \rightarrow v'$, then there exist fresh pairs $\langle A_i; v_i \rangle \in \mathbf{PP}_k(D, e_1)$ (for all $i \in \{1, \dots, n\}$), $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{v_i \leq u_i \mid i \in \{1, \dots, n\}\})$, $p = \langle \mathbf{s}(A_0 \wedge A_1 \wedge \dots \wedge A_n); \mathbf{s}(v') \rangle$. By induction and rule (SPC)

$$D \vdash e_0 : \langle \mathbf{s}(A_0); \mathbf{s}(u_1 \wedge \dots \wedge u_n \rightarrow v') \rangle, \text{ and}$$

$$D \vdash e_i : \langle \mathbf{s}(A_i); \mathbf{s}(v_i) \rangle \text{ (for all } i \in \{1, \dots, n\} \text{)}.$$

Then, by rule (APP), we have:

$$D \vdash e_0 e_1 : p.$$

(Completeness). Let $p' = \langle A'; v' \rangle$. The derivation must end by $l \geq 0$ applications of rule (SPC) preceded by an application of rule (APP). Assume, without loss of generality, that $l = 0$. We have $D \vdash e_0 : \langle A'_0; u'_1 \wedge \dots \wedge u'_n \rightarrow v' \rangle$, $D \vdash e_1 : \langle A'_i; u'_i \rangle$ ($\forall i \in \{1, \dots, n\}$), and $A' = A'_0 \wedge A'_1 \wedge \dots \wedge A'_n$. By induction both $\mathbf{PP}_k(D, e_0)$ and $\mathbf{PP}_k(D, e_1)$ are not empty and, by Lemma 32.(2), it is enough to consider the following two cases on the structure of the pairs $\langle A_0; v_0 \rangle \in \mathbf{PP}_k(D, e_0)$.

- $v_0 = \alpha$ (a type variable). By induction $\langle A_0; v_0 \rangle \leq_{\mathbf{SPC}} p'$, i.e. (by definition of $\leq_{\mathbf{SPC}}$) there exists a substitution \mathbf{s}_0 such that

$$A'_0 \leq_1 \mathbf{s}_0(A_0) \text{ and } \mathbf{s}_0(\alpha) \leq_2 u'_1 \wedge \dots \wedge u'_n \rightarrow v'.$$

By definition of substitution, $\mathbf{s}_0(\alpha) \in \mathbf{T}_0$, so we have that $\mathbf{s}_0(\alpha) = u'_i \rightarrow u$ for some $i \in \{1, \dots, n\}$ and $u \leq_2 v'$. Assume, without loss of generality, that $i = 1$.

By induction and Lemma 32.(2) there is a fresh pair $\langle A_1; v_1 \rangle \in \mathbf{PP}_k(D, e_1)$ and a substitution \mathbf{s}_1 such that

$$A'_1 \leq_1 \mathbf{s}_1(A_1) \text{ and } \mathbf{s}_1(v_1) \leq_2 u'_1.$$

Let $P = \{v_1 \leq \alpha_1, \alpha = \alpha_1 \rightarrow \alpha_2\}$, where α_1, α_2 are fresh. The substitution $\mathbf{s}' = \mathbf{s}_0 \circ \mathbf{s}_1 \circ \{\alpha_1 := u'_1, \alpha_2 := u\}$ is a solution of the satisfaction problem $\exists \epsilon. P$ and is such that

$$A' \leq_1 \mathbf{s}_0(A_0) \wedge \mathbf{s}_1(A_1) = \mathbf{s}'(A_0 \wedge A_1), \text{ and}$$

$$\mathbf{s}'(\alpha_2) = u \leq_2 v'.$$

So there is $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon.P)$ such that

$$p = \langle \mathbf{s}(A_0 \wedge A_1); \mathbf{s}(\alpha_2) \rangle \in \mathbf{PP}_k(D, e_0 e_1)$$

and $\mathbf{s} \leq \mathbf{s}'$. Therefore $p \leq_{\mathbf{SPC}} p'$ holds.

- $v_0 = u_1 \wedge \cdots \wedge u_m \rightarrow v$. By induction there exists a substitution \mathbf{s}_0 such that

$$A'_0 \leq_1 \mathbf{s}_0(A_0) \text{ and}$$

$$\mathbf{s}_0(v) = u''_1 \wedge \cdots \wedge u''_n \rightarrow v'' \leq_2 u'_1 \wedge \cdots \wedge u'_m \rightarrow v' \quad (n \leq m).$$

Assume, without loss of generality, that $u''_i = u'_i$ for $i \in \{1, \dots, n\}$. By induction and Lemma 32.(2), for all $j \in \{1, \dots, n\}$, there are fresh pairs $\langle A_j; v_j \rangle \in \mathbf{PP}_k(D, e_1)$ and substitutions \mathbf{s}_j such that

$$A'_j \leq_1 \mathbf{s}_j(A_j) \text{ and } \mathbf{s}_j(v_j) = u'_j.$$

Let $P = \{v_j \leq u_j \mid j \in \{1, \dots, n\}\}$. The substitution $\mathbf{s}' = \mathbf{s}_0 \circ \mathbf{s}_1 \circ \cdots \circ \mathbf{s}_n$ is a solution of the satisfaction problem $\exists \epsilon.P$ and is such that $A'' = \mathbf{s}_0(A_0) \wedge \mathbf{s}_1(A_1) \wedge \cdots \wedge \mathbf{s}_n(A_n) = \mathbf{s}'(A_0 \wedge A_1 \wedge \cdots \wedge A_n)$, and $\mathbf{s}'(v) = v''$. So there is $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon.P)$ such that

$$p = \langle \mathbf{s}(A_0 \wedge A_1 \wedge \cdots \wedge A_n); \mathbf{s}(v) \rangle \in \mathbf{PP}_k(D, e_0 e_1)$$

and $\mathbf{s} \leq \mathbf{s}'$. Therefore $p \leq_{\mathbf{SPC}} p'$ holds.

$$e = \text{rec } \{x = e_0\}.$$

(Soundness). We have three cases.

1. If $x \notin \text{FV}(e_0)$, then we have $p \in \mathbf{PP}_k(D, e_0)$ and, by induction, both $D \vdash e_0 : p$ and p principal-in- D for e_0 . Therefore, by rule (RECP_k), we get

$$D \vdash e : p.$$

2. If $x \in \text{FV}(e_0)$ and h is the minimum number in $\{1, \dots, k\}$ such that $p_0 = \text{bot}_{\text{FV}(e)}$, $p_1 \in \mathbf{PP}_k((D, x : p_0), e_0)$, \dots , $p_h \in \mathbf{PP}_k((D, x : p_{h-1}), e_0)$, and $p_h \leq_{\mathbf{SPC}} p_{h-1}$. Then, by induction, $D, x : p_0 \vdash e_0 : p_1$, \dots , $D, x : p_{h-1} \vdash e_0 : p_h$, and for all $i \in \{1, \dots, h\}$ p_i principal-in- $(D, x : p_{i-1})$ for e_0 . Since $\mathcal{F}_{\vdash_2^{\text{PP}_k}}^{D, e}$ is monotone, it holds that $p_{h-1} \leq_{\mathbf{SPC}} p_h$, which implies that p_{h-1} principal-in- $(D, x : p_{h-1})$ for e_0 . By applying rule (SPC), we also have $D, x : p_{h-1} \vdash e_0 : p_{h-1}$. Therefore, we can apply (RECP_k) to get

$$D \vdash e : p_{h-1}.$$

3. Otherwise (if $x \in \text{FV}(e_0)$ and such an h does not exist), we have $\langle A, x : w; v_0 \rangle \in \mathbf{PP}_k(D, e_0)$ and $p = \langle \mathbf{s}(A); \mathbf{s}(v_0) \rangle$, where $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{ \text{Gen}(A', v) \leq w \})$. By induction, $D \vdash e_0 : \langle A, x : w; v_0 \rangle$.

Therefore, by rule (SPC) we get $D \vdash e_0 : \langle \mathbf{s}(A, x : w); \mathbf{s}(v_0) \rangle$ and, by rule (RECJ) we get

$$D \vdash e : p.$$

(Completeness). Let $p' = \langle A'; v' \rangle$. Again, we have different cases.

1. If $x \notin \text{FV}(e_0)$, then the derivation must end by $l \geq 0$ applications of rule (SPC) preceded by an application of rule (RECP_k). Assume, without loss of generality, that $l = 0$. We have $D, x : p' \vdash e_0 : p'$ and, since $x \notin \text{FV}(e_0)$, $D \vdash e_0 : p'$. Therefore, by induction, $p \leq_{\text{SPC}} p'$ for some $p \in \mathbf{PP}_k(D, e_0)$.
2. If $x \in \text{FV}(e_0)$, then the derivation must end by $l \geq 0$ applications of rule (SPC) preceded by either an application of rule (RECP_k) or an application of rule (RECJ). Assume, without loss of generality, that $l = 0$.

(a) Let the derivation end by an application of rule (RECP_k). Then $D, x : p_0 \vdash e : p_1, \dots, D, x : p_{k-1} \vdash e : p_k$, where $x \notin D$, $p_0 = \text{bot}_{\text{FV}(e)}$, $p_{k-1} = p_k$, and for all $i \in \{1, \dots, k\}$ p_i is a principal-in- $(D, x : p_{i-1})$ pair for e_0 . By induction, there are pairs $p'_0 = \text{bot}_{\text{FV}(e)}$, $p'_1 \in \mathbf{PP}_k((D, x : p_0), e_0), \dots, p'_k \in \mathbf{PP}_k((D, x : p_{k-1}), e_0)$ such that for all $i \in \{1, \dots, k\}$ p'_i is a principal-in- $(D, x : p'_{i-1})$ pair for e_0 . Let h is the minimum number in $\{1, \dots, k\}$ such that $p'_h \leq_{\text{SPC}} p'_{h-1}$, i.e., $[p_{h-1}]^{\text{err}}_{\text{SPC}}$ is the minimum fixed point of $\mathcal{F}_{\vdash_2^{\text{P}_k+J}}^{D,e}$. We have that $p'_{h-1} \in \mathbf{PP}_k(D, e_0)$. Moreover, by Definition 12, we have that for all $i \in \{1, \dots, k\}$ $[p_i]^{\text{err}}_{\text{SPC}} = [p'_i]^{\text{err}}_{\text{SPC}}$ and, since $[p_{h-1}]^{\text{err}}_{\text{SPC}}$ is the minimum fixed point of $\mathcal{F}_{\vdash_2^{\text{P}_k+J}}^{D,e}$, $p'_{h-1} \leq_{\text{SPC}} p'_h$. So $[p_{h-1}]^{\text{err}}_{\text{SPC}} = [p_h]^{\text{err}}_{\text{SPC}} = \dots = [p_k]^{\text{err}}_{\text{SPC}}$ and p_{h-1} is the desired pair.

(b) Let the derivation end by an application of rule (RECJ). We have $(\star) D \vdash e_0 : \langle A', x : w'; v' \rangle$ with $\text{Gen}(\langle A', x : w' \rangle, v') \leq_{\forall 2,1} w'$. By induction, there is a pair $\langle A, x : w; v \rangle \in \mathbf{PP}_k(D, e_0)$ such that

$$(\star\star) \langle A, x : w; v \rangle \leq_{\text{SPC}} \langle A', x : w'; v' \rangle.$$

We have now to consider two cases.

- i. If h is the minimum number in $\{1, \dots, k\}$ such that $p_0 = \text{bot}_{\text{FV}(e)}$, $p_1 \in \mathbf{PP}_k((D, x : p_0), e_0), \dots, p_h \in \mathbf{PP}_k((D, x : p_{h-1}), e_0)$, and $p_h \leq_{\text{SPC}} p_{h-1}$, then $p_{h-1} \in \mathbf{PP}_k(D, e)$. By induction, p_{h-1} is a principal-in- $(D, x : p_{h-1})$ pair for e_0 . Observe that (\star) implies that $D, x : \langle A'; v' \rangle \vdash e_0 : \langle A'; v' \rangle$, i.e., $[\langle A'; v' \rangle]^{\text{err}}_{\text{SPC}}$ is a fixed point of $\mathcal{F}_{\vdash_2^{\text{P}_k+J}}^{D,e}$. So, since $[p_{h-1}]^{\text{err}}_{\text{SPC}}$ is the minimum fixed point of $\mathcal{F}_{\vdash_2^{\text{P}_k+J}}^{D,e}$, we have that $p_{h-1} \leq_{\text{SPC}} \langle A'; v' \rangle$.
- ii. Otherwise (if such an h does not exist), observe that $(\star\star)$ means that there is a substitution \mathbf{s}' such that

$$A' \leq_1 \mathbf{s}'(A) \text{ and } \mathbf{s}'(v) \leq_2 v'.$$

The substitution \mathbf{s}' is a solution to the satisfaction problem $\exists \epsilon.P$, where $P = \{\text{Gen}((A, x : w), v) \leq w\}$. So there is $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon.P)$ such that

$$p = \langle \mathbf{s}(A); \mathbf{s}(v) \rangle \in \mathbf{PP}_k(D, e)$$

and $\mathbf{s} \leq \mathbf{s}'$. Therefore $p \leq_{\mathbf{spc}} p'$ holds.

A.8 Proof of Theorem 14

Restatement of Theorem 14. *For every $k \geq 1$:*

1. *If $D \vdash_2^{\text{P}^k} e : p$, then $D \vdash_2^{\text{P}} e : p$.*
2. *If e is $\vdash_2^{\text{P}^k}$ -typable in D and $D \vdash_2^{\text{P}} e : p$, then $D \vdash_2^{\text{P}^k} e : p$.*

Proof. 1. Immediate.

2. Let $\text{rec}\{x = e'\}$ be $\vdash_2^{\text{P}^k}$ -typable in D and $D \vdash_2^{\text{P}} \text{rec}\{x = e'\} : \langle A; v \rangle$. The hypothesis that $\text{rec}\{x = e'\}$ is $\vdash_2^{\text{P}^k}$ -typable in D implies (by structural induction on $\vdash_2^{\text{P}^k}$ derivations) that

$$D \vdash_2^{\text{P}^k} \text{rec}\{x = e'\} : \langle A_0; v_0 \rangle$$

where

$$\langle A_0; v_0 \rangle = \underbrace{\mathcal{F}_{\vdash_2^{\text{P}^k}}^{D, \text{rec}\{x=e'\}}(\dots \mathcal{F}_{\vdash_2^{\text{P}^k}}^{D, \text{rec}\{x=e'\}}(\mathbf{bot}_{\text{FV}_D(\text{rec}\{x=e'\})}) \dots)}_{k \text{ times}}$$

is the minimum fixed point of $\mathcal{F}_{\vdash_2^{\text{P}^k}}^{D, \text{rec}\{x=e'\}}$ (the analogous for $\vdash_2^{\text{P}^k}$ of the functional $\mathcal{F}_{\vdash_2^{\text{P}^k + \text{J}}}^{D, \text{rec}\{x=e'\}}$ defined in Section A.7). By structural induction on \vdash_2^{P} derivations, $D \vdash_2^{\text{P}} \text{rec}\{x = e'\} : \langle A; v \rangle$ implies that $D, x : \langle A; v \rangle \vdash_2^{\text{P}} e' : \langle A; v \rangle$. Therefore $[\langle A; v \rangle]_{\text{spc}}^{\text{err}}$ is a fixed point of $\mathcal{F}_{\vdash_2^{\text{P}^k}}^{D, \text{rec}\{x=e'\}}$ (i.e., $\mathcal{F}_{\vdash_2^{\text{P}^k}}^{D, \text{rec}\{x=e'\}}([\langle A; v \rangle]_{\text{spc}}^{\text{err}}) = [\langle A; v \rangle]_{\text{spc}}^{\text{err}}$). Since $[\langle A_0; v_0 \rangle]_{\text{spc}}^{\text{err}}$ is the minimum fixed point of $\mathcal{F}_{\vdash_2^{\text{P}^k}}^{D, \text{rec}\{x=e'\}}$ we have that $\langle A_0; v_0 \rangle \leq_{\mathbf{spc}} \langle A; v \rangle$ and, by rule (SPC), we can conclude that $D \vdash_2^{\text{P}^k} \text{rec}\{x = e'\} : \langle A; v \rangle$.