Principal Typings and True Rank 2 Intersection Typable Recursive Definitions

Ferruccio Damiani*

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

Abstract. We propose new typing rules for assigning rank 2 intersection types to (possibly mutually) recursive definitions. A major achievement of the new rules over previous proposals is that they allow to type also *true rank 2 intersection typable* recursive definitions (i.e., recursive definitions that have a rank 2 intersection type and no simple type). A notable feature of these rules is that they rely entirely on *principal typings*, so they can be added to any system with principal typings.

1 Introduction

The Hindley-Milner type system [3], which is the core of the type systems of modern functional programming languages (like ML and Haskell), has several limitations that prevent safe programs from being typed. In particular, it does not allow to assign different types to different occurrences of a formal parameter in the body of a function. To overcome these limitations, various extensions of the Hindley-Milner system based on *universal types* [5,16], *intersection types* [2,1], *recursive types*, and combinations of them, have been proposed in the literature.

Systems with intersection types are particularly interesting since they generally have *principal typings* [10,19] (a.k.a. *principal pairs*). Intersection types are obtained from *simple types* [8] by adding the *intersection type constructor* $\wedge$. A term has type $u_1 \wedge u_2$ (u_1 intersection u_2) if it has both type u_1 and type u_2. For example, the identity function $\lambda x.x$ has both type $\mathsf{int} \to \mathsf{int}$ and $\mathsf{bool} \to \mathsf{bool}$, so it has type $(\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{bool} \to \mathsf{bool})$.

The system of *rank 2 intersection types* [13,17,20] is able to type all ML programs, has principal typings, decidable type inference, and the complexity of type inference is of the same order as in ML. Rank 2 intersection types are types that may contain intersections only to the left of a single arrow. Therefore, for instance, $((\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{bool} \to \mathsf{bool})) \to \mathsf{int} \to \mathsf{int}$ is a rank 2 intersection type (as usual, the arrow type constructor is right associative), while $(((\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{bool} \to \mathsf{bool})) \to \mathsf{int}) \to \mathsf{int}$ is not a rank 2 intersection type.

The problem of typing standard programming language constructs like recursive definitions, local definitions, and conditional expressions without losing

* Partially supported by IST-2001-33477 DART project. The founding bodies are not responsible for any use that might be made of the results presented here.

the extra information provided by rank 2 intersection is more difficult than one might expect (see, e.g., [9,10,18,4,6]).

In [4] we introduced typing rules for assigning rank 2 intersection types to local definitions and conditional expressions. In this paper we propose a new typing rule for assigning rank 2 intersection types to (possibly mutually) recursive definitions. The resulting rank 2 intersection type system has more expressive power than the Milner-Mycroft system [15] (see also [14]), in the sense that the set of typable terms increases, and types express better the behaviour of terms. Our system is undecidable, as it is the Milner-Mycroft system [7,12]. However, by relying on principal typings, we will define various decidable restrictions of the rule. We say that a term e is *true rank 2 intersection typable* if it has a rank 2 type and no simple type (the fact that e has no simple types implies that it has no rank 1 types). A major achievement over previous proposals for typing recursive definitions in presence of rank 2 intersection [9,10,18,4] is that the new rules allow to type also true rank 2 intersection typable recursive definitions. A notable feature of these rules is that they rely entirely on principal typings, so they can be added to any system with principal typings.

Organization of the Paper. Section 2 introduces a small programming language, which can be considered the kernel of functional programming languages like ML and Haskell. Section 3 introduces the syntax of rank 2 intersection types, together with other basic definitions. Section 4 presents the rank 2 intersection type system ($\vdash_2$) for the rec-free subset of the language. Section 5 extends $\vdash_2$ with the new typing rules for assigning rank 2 types to non-mutually recursive definitions and Section 6 adapts the results to mutually recursive definitions.

The appendices contain the proofs of the main results and some examples. More examples and an on-line demonstration of a prototype implementation of the type inference algorithm are available at the url http://lambda.di.unito.it/recp2.

2 A Small ML-like Language

We consider two classes of *constants*: *constructors* for denoting base values (integer, booleans) and building data structures, and *base functions* for denoting operations on base values and for decomposing data structures. The base functions include some arithmetic and logical operators, and the functions for decomposing pairs (fst and snd) and lists (null, hd and tl). The constructors include the unique element of type unit, the booleans, the integer numbers, and the constructors for pairs and lists. Let bf range over base functions (all unary) and cs range over constructors. The syntax of constants (ranged over by c) is as follows:

$$
\begin{array}{l}
c \ ::= bf \mid cs \\
bf ::= \mathsf{not} \mid \mathsf{and} \mid \mathsf{or} \mid + \mid - \mid * \mid / \mid = \mid < \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{null} \mid \mathsf{hd} \mid \mathsf{tl} \\
cs ::= () \mid \mathsf{true} \mid \mathsf{false} \mid \cdots \mid -1 \mid 0 \mid 1 \mid \cdots \mid \mathsf{pair} \mid \mathsf{nil} \mid \mathsf{cons}
\end{array}
$$

Expressions (ranged over by e) have the following syntax:

$$
e ::= x \mid c \mid \lambda\, x.e \mid e_1\, e_2 \mid \mathsf{let}\, x = e_0\, \mathsf{in}\, e \mid \mathsf{rec}_i\, \{x_1 = e_1, \ldots, x_n = e_n\} \mid \mathsf{if}\, e_0\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2,
$$

where $n \geq 1$ and x, x_1, ..., x_n range over variables. The construct rec allows mutually recursive expression definitions. The finite set of the free variables of an expression e is denoted by $\mathrm{FV}(e)$.

3 Basic Definitions

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

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

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

where $n \geq 1$. We have *type variables* (ranged over by α), arrow types, and a selection of *ground types* and *parametric datatypes*. The ground types are: unit (the singleton type), bool (the set of booleans), and int (the set of integers). The other types are pair types and list types. Note that $\mathbf{T}_0 = \mathbf{T}_1 \cap \mathbf{T}_2$.

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

We assume a countable set of type variables. A *substitution* $\mathbf{s}$ is a function from type variables to simple types which is the identity on all but a finite number of type variables. The application of a substitution $\mathbf{s}$ to a type t, denoted by $\mathbf{s}(t)$, is defined as usual. Note that, since substitutions replace type variables by simple types, we have that $\mathbf{T}_0$, $\mathbf{T}_1$ and $\mathbf{T}_2$ are closed under substitution.

An *environment* T is a set $\{x_1 : t_1, \ldots, x_n : t_n\}$ of assumptions for variables such that every variable x_i ($1 \leq i \leq n$) can occur at most once in T. The expression $\mathrm{Dom}(T)$ denotes the *domain* of T, which is the set $\{x_1, \ldots, x_n\}$. We write

T_1, T_2 for the environment $T_1 \cup T_2$ where it is assumed that $\mathrm{Dom}(T_1) \cap \mathrm{Dom}(T_2) = \emptyset$, and $T, x : t$ as short for $T, \{x : t\}$.

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

The application of a substitution $\mathbf{s}$ to an environment T, denoted by $\mathbf{s}(T)$, is defined as usual.

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

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

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

The following subtyping relations are fairly standard.

Definition 2 (Subtyping relations $\leq_1$ and $\leq_2$). *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*

$$(\wedge)\ \frac{\{u_1, \ldots, u_n\} \supseteq \{u'_1, \ldots, u'_m\}}{u_1 \wedge \cdots \wedge u_n \leq_1 u'_1 \wedge \cdots \wedge u'_m} \qquad (\textsc{Ref})\ \frac{u \in \mathbf{T}_0}{u \leq_2 u} \qquad (\rightarrow)\ \frac{w' \leq_1 w \quad v \leq_2 v'}{w \rightarrow v \leq_2 w' \rightarrow v'}$$

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

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

The relations $\leq_1$ and $\leq_2$ are reflexive, transitive, and closed under substitution.

A *pair* is a formula $\langle A; v\rangle$ where A is a rank 1 environment and v is a rank 2 type. The type variables occurring in a pair are (implicitly) universally quantified.[2]

Definition 3 (Pair specialization relation $\leq_{\mathbf{spc}}$). *A pair $\langle A; v\rangle$* can be specialized to *$\langle A'; v'\rangle$ (notation $\langle A; v\rangle \leq_{\mathbf{spc}} \langle A'; v'\rangle$) if $A' \leq_1 \mathbf{s}(A)$ and $\mathbf{s}(v) \leq_2 v'$, for some substitution $\mathbf{s}$.*

Example 1. We have $\langle\{y : \beta\}; \alpha \rightarrow \beta\rangle \leq_{\mathbf{spc}} \langle\{y : \gamma\}; ((\gamma \rightarrow \gamma) \wedge \gamma) \rightarrow \gamma\rangle$.

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

Proposition 1 (Decidability of $\leq_{\mathbf{spc}}$). *There is an algorithm that, for every $\langle A; v\rangle$ and $\langle A'; v'\rangle$, decides whether $\langle A; v\rangle \leq_{\mathbf{spc}} \langle A'; v'\rangle$ holds.*

Definition 4 (Pair environments). *A* pair environment *D is an environment $\{x_1 : \langle A_1; v_1\rangle, \ldots, x_n : \langle A_n; v_n\rangle\}$ of pair assumptions for variables such that $\mathrm{Dom}(D) \cap \mathrm{FVR}(D) = \emptyset$. The expression $\mathrm{FVR}(D)$ denotes the* set of variables occurring in the range of *D, which is the set $\cup_{x:\langle A; v\rangle \in D}\mathrm{Dom}(A)$. Let $\mathrm{FV}(D)$ denote $\mathrm{Dom}(D) \cup \mathrm{FVR}(D)$.*

[1] The requirement $\mathrm{Dom}(A) = \mathrm{Dom}(A')$ in the definition of $A \leq_1 A'$ is "unusual" (going counter to the definitions in other papers). The "usual" definition drops this requirement, thus allowing $\mathrm{Dom}(A) \supseteq \mathrm{Dom}(A')$. We have added such a requirement since it will simplify the presentation of the type system (in Section 4).

[2] A way to emphasize this fact is to use, as done in [4], formulas of the shape $\forall\overrightarrow{\alpha}.\langle A; v\rangle$, where $\overrightarrow{\alpha} = \alpha_1 \ldots \alpha_m$ is the sequence of *all* the type variables occurring in the pair $\langle A; v\rangle$. In this paper, in order to simplify the notation, we prefer to leave the universal quantification implicit.

bf	**Typeof**(bf)	cs	**Typeof**(cs)
not	$\mathsf{bool} \to \mathsf{bool}$	()	unit
and, or	$\mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$	true, false	bool
$+, -, *, /$	$\mathsf{int} \times \mathsf{int} \to \mathsf{int}$	$\cdots -1, 0, 1, \cdots$	int
$=, <$	$\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$	pair	$\alpha_1 \to \alpha_2 \to (\alpha_1 \times \alpha_2)$
fst	$\alpha_1 \times \alpha_2 \to \alpha_1$	nil	$\alpha\,\mathsf{list}$
snd	$\alpha_1 \times \alpha_2 \to \alpha_2$	cons	$\alpha \to \alpha\,\mathsf{list} \to \alpha\,\mathsf{list}$
null	$\alpha\,\mathsf{list} \to \mathsf{bool}$		
hd	$\alpha\,\mathsf{list} \to \alpha$		
tl	$\alpha\,\mathsf{list} \to \alpha\,\mathsf{list}$		

Fig. 1. Types for base functions and constructors

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

In this section we introduce the type system for the "rec-free" fragment of the language (i.e., the fragment without recursive definitions).

4.1 The Type Inference Rules of $\vdash_2$

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

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

We call *undefined* the variables in $\mathrm{Dom}(A)$, since their definition is not available when typing the expression e. In any valid judgement we have $\mathrm{Dom}(D) \cap \mathrm{Dom}(A) = \emptyset$. Moreover, the environment A is *relevant*:

> for any typing judgement $D \vdash_2 e : \langle A; v\rangle$ it holds that $\mathrm{Dom}(A) = \mathrm{FV}_D(e)$, where $\mathrm{FV}_D(e) = (\mathrm{FV}(e) - \mathrm{Dom}(D)) \cup \mathrm{FVR}(D|_{\mathrm{FV}(e)})$ is the set of the *free variables of the expression e in the pair environment D*.

The typing rules of system $\vdash_2$ are given in Fig. 2. System $\vdash_2$ is (modulo change of notation) a subsystem of the system presented in Section 8 of [4].

Rule (Spc), which is the only non-structural rule, allows to specialize the pair inferred for the expression. It models the fact that all the type variables occurring in a pair are (implicitly) universally quantified.

$$(\textsc{Spc})\ \frac{D \vdash e : \langle A;\, v\rangle}{D \vdash e : \langle A';\, v'\rangle} \quad \text{where } \langle A;\, v\rangle \leq_{\mathbf{spc}} \langle A';\, v'\rangle$$

$$(\textsc{Const})\ D \vdash c : \langle \emptyset;\, v\rangle \quad \text{where } v = \mathbf{Typeof}(c)$$

$$(\textsc{Var}_{\text{defined}})\ D,\, x : \langle A;\, v\rangle \vdash x : \langle A;\, v\rangle$$

$$(\textsc{Var}_{\text{undefined}})\ D \vdash x : \langle \{x : u\};\, u\rangle \quad \text{where } x \notin \mathrm{Dom}(D) \text{ and } u \in \mathbf{T}_0$$

$$(\textsc{Abs})\ \frac{D \vdash e : \langle A, x : w;\, v\rangle}{D \vdash \lambda x.e : \langle A;\, w \to v\rangle} \quad \text{where } x \notin \mathrm{FV}(D) \text{ and } x \in \mathrm{FV}(e)$$

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

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

$$(\textsc{LetP})\ \frac{D \vdash e_0 : \langle A_0;\, v_0\rangle \qquad D,\, x : \langle A_0;\, v_0\rangle \vdash e : \langle A;\, v\rangle}{D \vdash \mathsf{let}\, x = e_0\, \mathsf{in}\, e : \langle A_0 + A;\, v\rangle} \quad \text{where } x \notin \mathrm{FV}(D) \text{ and } x \notin \mathrm{FV}(e_0)$$

$$(\textsc{IfSimple})\ \frac{D \vdash e_0 : \langle A_0;\, \mathsf{bool}\rangle \qquad D \vdash e_1 : \langle A_1;\, u\rangle \qquad D \vdash e_2 : \langle A_2;\, u\rangle}{D \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \langle A_0 + A_1 + A_2;\, u\rangle}$$

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

The rule for typing constants, (Const), uses the function **Typeof** (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 $\langle \emptyset;\, \mathbf{Typeof}(c)\rangle$.

Since we distinguish between *defined* and *undefined* variables, we have two different rules for variables: ($\textsc{Var}_{\text{defined}}$) and ($\textsc{Var}_{\text{undefined}}$).

We have two rules for typing an abstraction $\lambda x.e$, (Abs) and (AbsVac), corresponding to the two cases $x \in \mathrm{FV}(e)$ and $x \notin \mathrm{FV}(e)$. Note that, by rule ($\textsc{Var}_{\text{undefined}}$), 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 rule for typing let-expressions, (LetP), stores in the environment D the pair assumption for the let-bound variable. Therefore, by rules ($\textsc{Var}_{\text{defined}}$) and (Spc), it is possible to assign a distinct specialization of that pair to each occurrence of the let-bound variable.

The typing rule for conditional expressions, (IfSimple), forces to assign the same simple type to both the branches of the conditional expression. Therefore it does not allow to type true rank 2 intersection typable conditional expressions. Instead, the paper [4] presents a more powerful rule, which allow to type true rank 2 intersection typable conditional expressions. For sake of readability we have not considered this rule in the present paper, but it can be used without problems.

4.2 Principal-in-D Pairs for $\vdash_2$

In this section we introduce the notion of *principal-in-D pairs*. We first give a *system-dependent* definition (i.e., involving syntactic operations on the types of the particular system considered) and prove that $\vdash_2$ is decidable and has principal-in-D pairs. Then we introduce an (equivalent) *system-independent* definition (inspired by Wells' system-independent definition of *principal typings* [19]).

The system-dependent definition of principal-in-D pairs is as follows.

Definition 5 (Principal-in-D pairs — system-dependent). *A pair* $\langle A;\, v\rangle$ *is* principal-in-D for a term e *if* $D \vdash e : \langle A;\, v\rangle$*, and if* $D \vdash e : \langle A';\, v'\rangle$ *implies* $\langle A;\, v\rangle \leq_{\mathbf{spc}} \langle A';\, v'\rangle$.

System $\vdash_2$ has principal-in-D pairs and is decidable.

Theorem 1 (Decidability of system $\vdash_2$). *There is an algorithm that, for every expression e and environment D, decides whether e is $\vdash_2$-typable in D and, if so, returns a principal-in-D pair for e.*

The system-independent definition of principal-in-D pairs is as follows.

Definition 6 (Principal-in-D pairs — system-independent). *A pair* $\langle A_1;\, v_1\rangle$ *is* at least as strong as $\langle A_2;\, v_2\rangle$ *(notation* $\langle A_1;\, v_1\rangle \leq_{\vdash} \langle A_2;\, v_2\rangle$*) if, for every term* e*,* $\emptyset \vdash e : \langle A_1;\, v_1\rangle$ *implies* $\emptyset \vdash e : \langle A_2;\, v_2\rangle$*. A pair* $\langle A;\, v\rangle$ *is* principal-in-D for a term e *if* $D \vdash e : \langle A;\, v\rangle$ *, and if* $D \vdash e : \langle A';\, v'\rangle$ *implies* $\langle A;\, v\rangle \leq_{\vdash} \langle A';\, v'\rangle$.

The system-dependent and the system-independent definitions are equivalent.

Lemma 1. $\langle A_1;\, v_1\rangle \leq_{\mathbf{spc}} \langle A_2;\, v_2\rangle$ *iff* $\langle A_1;\, v_1\rangle \leq_{\vdash_2} \langle A_2;\, v_2\rangle$.

Theorem 2. *In system $\vdash_2$, a pair $\langle A;\, v\rangle$ is principal-in-D for e according to Definition 5 iff it is principal-in-D for e according to Definition 6.*

Remark 1 (Principal-in-D typings). According to Wells [19]:

- 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 triple $\langle D; A; v\rangle$).
- Let $\vdash$ be a type system with judgements $\vdash e : t$. A typing t_1 is *at least as strong as* t_2 (notation $t_1 \leq_{\vdash} t_2$) if, for every term e, $\vdash e : t_1$ implies $\vdash e : t_2$. A typing t is *principal for a term* e if $\vdash e : t$, and if $\vdash e : t'$ implies $t \leq_{\vdash} t'$.

Let $\vdash_2^-$ be the subsystem of $\vdash_2$ obtained by removing rules ($\textsc{Var}_{\text{defined}}$) and ($\textsc{LetP}$), by removing the side condition "$x \notin \mathrm{FV}(e)$" from rules ($\textsc{Var}_{\text{undefined}}$), ($\textsc{Abs}$), ($\textsc{AbsVac}$), and by removing the environment D from the typing judgment. In system $\vdash_2^-$ a typing is a pair $\langle A;\, v\rangle$. System $\vdash_2^-$ has principal typings. Extending system $\vdash_2^-$ by adding the environment D (which is a set of typing assumption for variables) could be seen as taking in input a system with principal typings and yielding as output a system without principal typings and

with *principal-in-D typings*. This technique, which is the key component of our typing rule for let-expressions (and, as we will see in Section 5, for the new typing rules for rec-expressions presented in this paper), relies entirely on principal typings, because it does not depends on the details of the particular system considered. Therefore, it can be applied to any system with principal typings. Note that *principal-in-$\emptyset$ typings* have all the good properties of principal typings (described, for instance, in [10,19]).

5 Typing Non-Mutually Recursive Definitions

This section presents the new typing rules for recursive definitions $\mathsf{rec}\,\{x = e\}$.

5.1 A New Typing Rule for Recursive Definitions

The following typing rule:

$$(\textsc{RecP})\ \frac{D, x : \langle A;\, v\rangle \vdash e : \langle A;\, v\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle A;\, v\rangle}$$
$$\text{where } x \notin \mathrm{FV}(D) \text{ and } \mathrm{Dom}(A) = \mathrm{FV}_D(e) - \{x\}$$

allows to assign to a recursive definition $\mathsf{rec}\,\{x = e\}$ any pair $\langle A;\, v\rangle$ that can be assigned to e by assuming the pair $\langle A;\, v\rangle$ itself for x.

System $\vdash_2$+(RecP) allows to type true rank 2 intersection typable recursive definitions, as illustrated by the following example.

Example 2. The true rank 2 intersection typable expression $\mathsf{rec}\,\{f = e\}$, where

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

has $\vdash_2$+(RecP)-typing judgement

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

System $\vdash_2$+(RecP) has more expressive power than the Milner-Mycroft system [15] (see also [14]), in the sense that the set of typable terms increases, and types express better the behaviour of terms. System $\vdash_2$+(RecP) is undecidable, as it is the Milner-Mycroft system [7,12]. Decidability can be recovered by allowing to declare pairs for the recursively defined variables (that is, to write recursive definitions of the shape $\mathsf{rec}\,\{x : \langle A;\, v\rangle = e\}$) and replacing rule (RecP) with the following rule

$$(\textsc{RecDecP})\ \frac{D, x : \langle A;\, v\rangle \vdash e : \langle A;\, v\rangle}{D \vdash \mathsf{rec}\,\{x : \langle A;\, v\rangle = e\} : \langle A;\, v\rangle}$$
$$\text{where } x \notin \mathrm{FV}(D) \text{ and } \mathrm{Dom}(A) = \mathrm{FV}_D(e) - \{x\}$$

System $\vdash_2$+(RecDecP) has principal-in-D pairs (in the sense of Definition 5) and is decidable.

Theorem 3 (Decidability of system $\vdash_2$+(RecDecP)). *There is an algorithm that, for every expression e (containing typing declarations for all the* rec*-bound variables) and environment D, decides whether e is $\vdash_2$+(RecDecP)-typable in D and, if so, returns a principal-in-D pair for e.*

Also for system $\vdash_2$+(RecDecP) the system-dependent and the system-independent definition of principal-in-D pairs are equivalent.

5.2 Comparison with Jim's typing rules for recursive definitions

In this section we briefly recall the typing rules for not mutually recursive definitions proposed by Jim [9,10] (see also [4]). All these rules are strictly less powerful than rule (RecP).

Rank 2 intersection type schemes, ranged over by vs, are defined as follows

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

where $\overrightarrow{\alpha}$ is a finite (possibly empty) sequence of type variables $\alpha_1 \cdots \alpha_m$ and $v \in \mathbf{T}_2$. Given a type $v \in \mathbf{T}_2$ and a rank 1 environment A, we write $\mathrm{Gen}(A, v)$ for the $\forall$-*closure of* v *in* A, that is the rank 2 scheme $\forall\overrightarrow{\alpha}.v$, where $\overrightarrow{\alpha}$ is the sequence of the type variables in $\mathrm{FTV}(v) - \cup_{x:w\in A}\mathrm{FTV}(w)$.

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

Example 3. We have (remember that $\wedge$ is idempotent): $\forall\alpha_1\alpha_2.((\alpha_1 \to \alpha_3) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_3 \leq_{\forall 2,1} ((\mathsf{int} \to \alpha_3) \to \alpha_3) \wedge ((\mathsf{bool} \to \alpha_3) \to \alpha_3)$.

Jim [9] proposed the following rules for typing recursive definitions:[3]

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

$$\text{(RecVac)}\ \frac{D \vdash e : \langle A;\ v\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle A;\ v\rangle} \quad \text{where } x \notin \mathrm{FV}(D) \text{ and } x \notin \mathrm{FV}(e)$$

These two rules corresponds to the two cases $x \in \mathrm{FV}(e)$ and $x \notin \mathrm{FV}(e)$. Note that the rule for non-vacuous recursion,(Rec), requires that the rank 2 type v assigned to $\mathsf{rec}\,\{x = e\}$ must be such that $\mathrm{Gen}((A, x : w),\ v) \leq_{\forall 2,1} w$ (where w is a rank 1 type), so it does not allow to type true rank 2 typable recursive definitions. System $\vdash_2$+(Rec)+(RecVac) has principal-in-D pairs and is decidable (see, e.g., [4]).

As pointed out by Jim [10], rule (Rec) might be generalized along the lines of Mycroft's rule (Fix′) [15]: just replace the condition $\mathrm{Gen}((A, x : w),\ v) \leq_{\forall 2,1} w$ by the condition $\mathrm{Gen}(A,\ v) \leq_{\forall 2,1} w$. Let us call (Rec′) the generalized rule. Rule (Rec′) is strictly more powerful than rule (Rec) (see Section 3 of [10] for an example) but, again, it does not allow to type true rank 2 typable recursive definitions.

Jim [9] proposed also the following rule:[3]

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

The decidability of $\vdash_2$+(RecInt) is an open question [9]. Note that also this rule does not allow to type true rank 2 typable recursive definitions.

[3] We still give to these rules the original name used in [9], but we adapt them to fit in the type assignment system $\vdash_2$.

5.3 Decidable restrictions of rule (RecP)

For every finite (possibly empty) set of variables $X = \{x_1, \ldots, x_n\}$ $(n \geq 0)$ let o_X denote any expression of principal-in-D pair $\langle \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}; \alpha \rangle$, where the type variables $\alpha_1, \ldots, \alpha_n, \alpha$ are all distinct. We can take, for instance, $o_X = \mathsf{fst}\,(\mathsf{pair}\,(\mathsf{hd}\,\mathsf{nil})\,(\mathsf{pair}\,x_1(\cdots(\mathsf{pair}\,x_{n-1}\,x_n)\cdots)))$.

For every natural number k, variable x, and expression e, let $\mathbf{LET}^k(x, e)$ denote the expression

$$\mathsf{let}\ x = o_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e\})}\ \mathsf{in}\ \underbrace{\mathsf{let}\ x = e\ \mathsf{in}\ \cdots\ \mathsf{let}\ x = e\ \mathsf{in}}_{k\ \text{times}}\ e.$$

By relying on principal-in-D pairs, we can design, for every $k \geq 0$, the following decidable restriction of rule (RecP):

$$(\mathrm{RecP}_k)\ \frac{D \vdash \mathbf{LET}^k(x, e) : \langle A;\, v \rangle \qquad D \vdash \mathbf{LET}^{k+1}(x, e) : \langle A;\, v \rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle A;\, v \rangle}$$

where $\langle A;\, v \rangle$ is a principal-in-D pair for both $\mathbf{LET}^k(x, e)$ and $\mathbf{LET}^{k+1}(x, e)$.

For every $k \geq 0$, system $\vdash_2 + (\mathrm{RecP}_k)$ has principal-in-D pairs (in the sense of Definition 5) and is decidable.

Theorem 4 (Decidability of system $\vdash_2 + (\mathrm{RecP}_k)$). *For every $k \geq 0$, there is an algorithm that, for every expression e and environment D, decides whether e is $\vdash_2 + (\mathrm{RecP}_k)$-typable in D and, if so, returns a principal-in-D pair for e.*

Also for system $\vdash_2 + (\mathrm{RecP}_k)$ the system-dependent and the system-independent definition of principal-in-D pairs are equivalent. The relation between system $\vdash_2 + (\mathrm{RecP}_k)$ and system $\vdash_2 + (\mathrm{RecP})$ is stated by the following theorem.

Theorem 5. *For every $k \geq 0$:*

1. *If $D \vdash_2 + (\mathrm{RecP}_k)\ e : \langle A;\, v \rangle$, then $D \vdash_2 + (\mathrm{RecP})\ e : \langle A;\, v \rangle$.*
2. *If e is $\vdash_2 + (\mathrm{RecP}_k)$-typable in D and $D \vdash_2 + (\mathrm{RecP})\ e : \langle A;\, v \rangle$, then $D \vdash_2 + (\mathrm{RecP}_k)\ e : \langle A;\, v \rangle$.*

Let (Rule) be any typing rule for rec-expressions such that system $\vdash_2 + (\mathrm{Rule})$ is decidable, has principal-in-D pairs, and $D \vdash_2 + (\mathrm{Rule})\ e : \langle A;\, v \rangle$ implies $D \vdash_2 + (\mathrm{RecP})\ e : \langle A;\, v \rangle$. By relying on Theorem 5, for every $k \geq 0$, we can combine rule (RecP_k) and rule (Rule) while preserving decidability and principal-in-D pairs. The combined rule, let us call it $(\mathrm{RecP}_k^{\mathrm{Rule}})$, is defined as follows:

> if $\mathsf{rec}\,\{x = e\}$ can be typed by using rule (RecP_k) then apply rule (RecP_k), otherwise apply rule (Rule).

Rule $(\mathrm{RecP}_k^{\mathrm{Rule}})$ extends both (RecP_k) and (Rule), and is a restriction of (RecP). For instance, for every $k \geq 0$, rule (RecP_k) and rule (Rec) (the decidable restriction of rule (RecP) presented in Section 5.2) are incomparable (see Example 4 below). So, for every $k \geq 0$, system $\vdash_2 + (\mathrm{RecP}_k^{\mathrm{Rec}})$ is an extension of both $\vdash_2 + (\mathrm{RecP}_k)$ and $\vdash_2 + (\mathrm{Rec})$.

Example 4. (Rule (RecP_k) and rule (Rec) are incomparable) The expression $\mathsf{rec}\,\{f = e\}$ of Example 2 is true rank 2 intersection typable, therefore it is not $\vdash_2 +(\text{Rec})$-typable. Since both the expressions "$\mathsf{let}\ f = o_{\{y\}}\ \mathsf{in}\ e$" and "$\mathsf{let}\ f = o_{\{y\}}\ \mathsf{in}\ \mathsf{let}\ f = e\ \mathsf{in}\ e$" are $\vdash_2+(\text{RecP}_0)$-typable with principal-in-$\emptyset$ pair:

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

this is a principal-in-$\emptyset$ pair for $\mathsf{rec}\,\{f = e\}$ in system $\vdash_2+(\text{RecP}_k)$ and also in system $\vdash_2+(\text{RecP}_k^{\text{Rec}})$ (for all $k \geq 0$).

The expression $\mathsf{rec}\,\{f = e'\}$, where $e' = \lambda\, g\, y.\, \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ y\ \mathsf{else}\ g(f\, g\, y)$, is not $\vdash_2+(\text{RecP}_k)$-typable (for any $k \geq 0$) — in fact, for every $k \geq 0$, the expression $\mathbf{LET}^k(f, e')$ has principal-in-$\emptyset$ pair:

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

Since $\mathsf{rec}\,\{f = e'\}$ is $\vdash_2+(\text{Rec})$-typable with principal-in-$\emptyset$ pair $\langle \emptyset;\ ((\alpha \to \alpha) \wedge (\alpha \to \beta)) \to (\alpha \wedge \beta) \to \beta\rangle$, this is a principal-in-$\emptyset$ pair for $\mathsf{rec}\,\{f = e'\}$ also in system $\vdash_2+(\text{RecP}_k^{\text{Rec}})$ (for all $k \geq 0$). Note that $\mathsf{rec}\,\{f = e'\}$ is $\vdash_2+(\text{RecP})$-typable with pair $\langle \emptyset;\ ((\alpha \to \gamma) \wedge (\gamma \to \alpha) \wedge (\alpha \to \beta)) \to (\alpha \wedge \beta \wedge \gamma) \to \beta\rangle$.

6 Typing Mutually Recursive Definitions

The results presented in Section 5 can be straightforwardly extended to mutually recursive definitions $\mathsf{rec}_i\,\{x_1 = e_1, \ldots, x_n = e_n\}$.

The extension of rule (RecP) to mutually recursive definitions is:

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

where $i_0 \in \{1, \ldots, n\}$, $x_1, \ldots, x_n \notin \text{FV}(D)$, and (for all $i \in \{1, \ldots, n\}$)

$$\text{Dom}(A_i) = \text{FV}_D(e_i \underbrace{[x_1 := e_1, \ldots, x_n := e_n] \cdots [x_1 := e_1, \ldots, x_n := e_n]}_{n-1 \text{ times}}) - \{x_1, \ldots, x_n\}$$

The design of decidable restrictions (RecDecP2), (RecP2_k) and ($\text{RecP2}_k^{\text{Rule}}$) of rule (RecP2), corresponding to the decidable restrictions (RecDecP), (RecP_k) and ($\text{RecP}_k^{\text{Rule}}$) of rule (RecP), is straightforward. The resulting systems have principal-in-D pairs.

7 Related Work and Conclusion

The literature related to the present work has been partially quoted through the paper. We conclude by briefly discussing two more papers. The paper [11] proposes type systems that combine universal types, recursive types, and object types. One of these systems, called $\Lambda_2^{\mu,\mathbf{fix},-}$, combines rank 2 universal and recursive types, is decidable, and allows to type (with rank 2 universal types) some

examples of true rank 2 intersection typable recursive definitions. A drawback of the systems presented in [11] is that, as clearly explained in Section 1.3 of [11], these systems do not have principal typings (and neither principal types). The paper [18] presents a system with rank 2 universal and intersection types for a term graph rewriting language with sharing (corresponding to let-expressions) and cycles (corresponding to rec-expressions). The system has principal typings but is not able to type true rank 2 intersection typable recursive definitions.

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. 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.
3. L. M. M. Damas and R. Milner. Principal type schemas for functional programs. In *POPL'82*, pages 207–212. ACM, 1982.
4. F. Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Trans. Prog. Lang. Syst.*, 25(4):401–451, 2003.
5. J. Y. Girard. *Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur*. PhD thesis, Université Paris VII, 1972.
6. J. J. Hallett and A. J. Kfoury. Programming examples needing polymorphic recursion. Technical report, Department of Computer Science, Boston University, 2003.
7. F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Prog. Lang. Syst.*, 15(2):253–289, 1993.
8. R. Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, London, 1997.
9. T. Jim. Rank 2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, LCS, Massachusetts Institute of Technology, 1995.
10. T. Jim. What are principal typings and what are they good for? In *POPL'96*, pages 42–53. ACM, 1996.
11. A. J. Kfoury and S M. Pericas-Geertsen. Type inference for recursive definitions. In *LICS'99*, pages 119–128. IEEE, 1999.
12. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Prog. Lang. Syst.*, 15(2):290–311, 1993.
13. D. Leivant. Polymorphic Type Inference. In *POPL'83*, pages 88–98. ACM, 1983.
14. L. Meertens. Incremental polymorphic type checking in B. In *POPL'83*, pages 265–275. ACM, 1983.
15. A. Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *International Symposium on Programming*, LNCS 167, pages 217–228. Springer, 1984.
16. J. C. Reynolds. Towards a Theory of Type Structure. In *Colloque sur la Programmation*, LNCS 19. Springer, 1974.
17. S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Katholieke Universiteit Nijmegen, 1993.
18. S. van Bakel. Rank 2 types for term graph rewriting. In *TIP'02*, volume 75 of *ENTCS*. Elsevier, 2003.
19. J.B. Wells. The essence of principal typings. In *ICALP'02*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.
20. H. Yokouchi. Embedding a Second-Order Type System into an Intersection Type System. *Information and Computation*, 117:206–220, 1995.

A Proofs

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.

Substitutions will be denoted by $[\alpha_1 := u_1, \ldots, \alpha_n := u_n]$ $(n \geq 0)$; 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.

A.1 Proof of Proposition 1

The algorithms MS_i $(0 \leq i \leq 2)$, defined in Fig. 3, take a pair of types $t, t' \in \mathbf{T}_i$ such that $\mathrm{FTV}(t) \cap \mathrm{FTV}(t') = \emptyset$ and return a set of substitutions, $\mathrm{MS}_i(t, t')$, 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 2. *1. Let $u, u' \in \mathbf{T}_i$ and $\mathrm{FTV}(u) \cap \mathrm{FTV}(u') = \emptyset$. Then*
$\mathrm{MS}_0(u, u') = \{\mathbf{s} \mid \mathrm{Dom}(\mathbf{s}) = \mathrm{FTV}(u) - \mathrm{FTV}(u')$ *and* $\mathbf{s}(u) = u'\}$.
2. Let $w, w' \in \mathbf{T}_i$ and $\mathrm{FTV}(w) \cap \mathrm{FTV}(w') = \emptyset$. Then
$\mathrm{MS}_1(w, w') = \{\mathbf{s} \mid \mathrm{Dom}(\mathbf{s}) = \mathrm{FTV}(w) - \mathrm{FTV}(w')$ *and* $w' \leq_1 \mathbf{s}(w)\}$.
3. Let $v, v' \in \mathbf{T}_i$ and $\mathrm{FTV}(v) \cap \mathrm{FTV}(v') = \emptyset$. Then
$\mathrm{MS}_2(v, v') = \{\mathbf{s} \mid \mathrm{Dom}(\mathbf{s}) = \mathrm{FTV}(v) - \mathrm{FTV}(v')$ *and* $\mathbf{s}(v) \leq_2 v'\}$.

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

Restatement of Proposition 1 (Decidability of $\leq_{\mathbf{spc}}$). *There is an algorithm that, for every $\langle A; v\rangle$ and $\langle A'; v'\rangle$, decides whether $\langle A; v\rangle \leq_{\mathbf{spc}} \langle A'; v'\rangle$ holds.*

Proof. By Lemma 2, observing that, for all pairs $\langle\{x_1 : w_1, \ldots, x_n : w_n\}; v\rangle$ and $\langle\{x_1 : w'_1, \ldots, x_n : w'_n\}; v'\rangle$, we have that $\langle\{x_1 : w_1, \ldots, x_n : w_n\}; v\rangle \leq_{\mathbf{spc}} \langle\{x_1 : w'_1, \ldots, x_n : w'_n\}; v'\rangle$ if and only if $\langle\emptyset; w_1 \to \cdots \to w_n \to v\rangle \leq_{\mathbf{spc}} \langle\emptyset; w'_1 \to \cdots \to w'_n \to v'\rangle$ if and only if $\mathrm{MS}_2(w_1 \to \cdots \to w_n \to v, w'_1 \to \cdots \to w'_n \to v') \neq \emptyset$.

A.2 Proof of Theorem 1

Restatement of Theorem 1 (Decidability of system $\vdash_2$). *There is an algorithm that, for every expression e and environment D, decides whether e is $\vdash_2$-typable in D and, if so, returns a principal-in-D pair for e.*

Proof. System $\vdash_2$ is (modulo change of notation) a subsystem of the system $\vdash_{\wedge_2}^{\mathrm{Loc,Rec,Con}}$ presented in Section 8 of [4]. A sound and complete inference algorithm for system $\vdash_2$ can be straightforwardly obtained from the inference algorithm for $\vdash_{\wedge_2}^{\mathrm{Loc,Rec,Con}}$ presented in Section 9 of [4].

$$
\begin{array}{l}
\text{(For all } i \in \{0,1,2\}) \quad \mathrm{MS}_i(t,t') = \mathrm{MS}'_i(t,t',\mathrm{FTV}(t')), \text{ where} \\
\\
\begin{array}{rl}
\mathrm{MS}'_0(u,u,W) &= \{[\,]\} \\
\mathrm{MS}'_0(\alpha,u,W) &= \{[\alpha := u]\}, \text{ if } \alpha \notin W \\
\mathrm{MS}'_0(u_1 \to u_2, u'_1 \to u'_2, W) &= \{\mathbf{s}_2 \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in \mathrm{MS}'_0(u_1,u'_1,W) \\
&\qquad \text{and } \mathbf{s}_2 \in \mathrm{MS}'_0(\mathbf{s}_1(u_2),u'_2,W)\} \\
\mathrm{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 \mathrm{MS}'_0(u_1,u'_1,W) \\
&\qquad \text{and } \mathbf{s}_2 \in \mathrm{MS}'_0(\mathbf{s}_1(u_2),u'_2,W)\} \\
\mathrm{MS}'_0(u\,\mathsf{list}, u'\,\mathsf{list}, W) &= \mathrm{MS}'_0(u,u',W) \\
\mathrm{MS}'_0(u,u',W) &= \emptyset, \text{ otherwise}
\end{array} \\
\\
\begin{array}{rl}
\mathrm{MS}'_1(u_1 \wedge \cdots \wedge u_m, u'_1 \wedge \cdots \wedge u'_n, W) &= \{\mathbf{s}_m \circ \cdots \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in \mathrm{MS}'_0(u_1,u'_{i_1},W), \\
&\qquad \mathbf{s}_2 \in \mathrm{MS}'_0(\mathbf{s}_1(u_2),u'_{i_2},W), \\
&\qquad \ldots, \\
&\qquad \mathbf{s}_m \in \mathrm{MS}'_0(\mathbf{s}_{m-1} \circ \cdots \circ \mathbf{s}_1(u_m),u'_{i_m},W), \\
&\qquad \text{and } i_1,\ldots,i_m \in \{1,\ldots,n\}\}
\end{array} \\
\\
\begin{array}{rl}
\mathrm{MS}'_2(u,u',W) &= \mathrm{MS}'_0(u,u',W) \\
\mathrm{MS}'_2(\alpha, w \to v, W) &= \{\mathbf{s}_2 \circ \mathbf{s}_1 \circ [\alpha := \alpha_1 \to \alpha_2] \mid \mathbf{s}_1 \in \mathrm{MS}'_1(\alpha_1,w,W) \\
&\qquad \text{and } \mathbf{s}_2 \in \mathrm{MS}'_2(\alpha_2,v,W)\}, \\
&\quad \text{if } \alpha \notin W, \text{ for some fresh } \alpha_1 \text{ and } \alpha_2 \\
\mathrm{MS}'_2(w \to v, w' \to v', W) &= \{\mathbf{s}_2 \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in \mathrm{MS}'_1(w,w',W) \\
&\qquad \text{and } \mathbf{s}_2 \in \mathrm{MS}'_2(\mathbf{s}_1(v),v',W)\} \\
\mathrm{MS}'_2(v,v',W) &= \emptyset, \text{ otherwise}
\end{array}
\end{array}
$$

Fig. 3. Algorithms MS_0, MS_1, and MS_2

A.3 Proof of Lemma 1 and Theorem 2

The clauses in Fig. 4 define, for every type $v \in \mathbf{T}_2$, a term e^v which characterizes the type v in the sense given by the following lemma.

Lemma 3. *1.* $\emptyset \vdash_2 e^v : \langle \emptyset; v \rangle$.
2. $\emptyset \vdash_2 e^v : \langle \emptyset; v_1 \rangle$ *implies* $\langle \emptyset; v \rangle \leq_{\mathbf{spc}} \langle \emptyset; v_1 \rangle$.

Proof. Let $\mathrm{FTV}(v) = \{\alpha_1,\ldots,\alpha_m\}$ ($m \geq 0$). Let v', w', and u' be subexpressions of the type v.

1. We have that:
   (a) $\mathrm{FTV}(u') = \{\alpha_{i_1},\ldots,\alpha_{i_q}\}$ implies $\mathrm{FV}(a^{y:u'}) = \{y, x_{i_1},\ldots,x_{i_q}\}$ and $\emptyset \vdash_2 a^{y:u'} : \langle \{y : u', x_{i_1} : \alpha_{i_1},\ldots,x_{i_q} : \alpha_{i_q}\}; u' \rangle$.
   (b) $\mathrm{FTV}(w') = \{\alpha_{i_1},\ldots,\alpha_{i_q}\}$ and $w' = u_1 \wedge \cdots \wedge u_r$ ($r \geq 1$) imply $\mathrm{FV}(ai^{y:w'}) = \{y, x_{i_1},\ldots,x_{i_q}\}$ and $\emptyset \vdash_2 ai^{y:w'} : \langle \{y : w', x_{i_1} : \alpha_{i_1},\ldots,x_{i_q} : \alpha_{i_q}\}; (u_1 \times (\cdots \times u_r)\cdots) \rangle$.
   (c) $\mathrm{FTV}(v') = \{\alpha_{i_1},\ldots,\alpha_{i_q}\}$ implies $\mathrm{FV}(b^{v'}) = \{x_{i_1},\ldots,x_{i_q}\}$ and $\emptyset \vdash_2 b^{v'} : \langle \{x_{i_1} : \alpha_{i_1},\ldots,x_{i_q} : \alpha_{i_q}\}; v' \rangle$.

(The proof is straightforward by induction on the definition of b^v, $ai^{y:w}$, and $a^{y:u}$ in Fig. 4.)
Therefore $\emptyset \vdash_2 b^v : \langle \{x_1 : \alpha_1, \ldots, x_m : \alpha_m\}; v \rangle$ and (by rules (Const), ($\text{Var}_{\text{defined}}$), (Abs), and (App)) $\emptyset \vdash_2 e^v : \langle \emptyset; v \rangle$.

2. Let $\sharp_\alpha(v')$ denote the number of occurrences of the type variable α in v' (similarly for w' and u'). We have that:
   (a) $\text{FTV}(u') = \{\alpha_{i_1}, \ldots, \alpha_{i_q}\}$ and, for all $j \in \{1, \ldots, q\}$, $\sharp_{\alpha_{i_j}}(u') = k_j$ imply that

   $$\langle \{y : u'', x_{i_1} : \alpha_{i_1,1} \wedge \cdots \wedge \alpha_{i_1,k_1}, \ldots, x_{i_q} : \alpha_{i_q,1} \wedge \cdots \wedge \alpha_{i_q,k_q}\}; u'' \rangle,$$

   where
   – the type variables $\alpha_{i_1,1}, \cdots, \alpha_{i_1,k_1}, \cdots, \alpha_{i_q,1}, \cdots, \alpha_{i_q,k_q}$ are all distinct and do not occur in u', and
   – u'' is obtained from u' by replacing, for all $j \in \{1, \ldots, q\}$ and $h \in \{1, \ldots, k_j\}$, the h-h occurrence of α_{i_j} in u' by $\alpha_{i_j,h}$,

   is a principal-in-$\emptyset$ pair for $a^{y:u'}$.
   (b) $\text{FTV}(w') = \{\alpha_{i_1}, \ldots, \alpha_{i_q}\}$, $w' = u_1 \wedge \cdots \wedge u_r$ $(r \geq 1)$ and, for all $j \in \{1, \ldots, q\}$, $\sharp_{\alpha_{i_j}}(w') = k_j$ imply that

   $$\langle \{y : w'', x_{i_1} : \alpha_{i_1,1} \wedge \cdots \wedge \alpha_{i_1,k_1}, \ldots, x_{i_q} : \alpha_{i_q,1} \wedge \cdots \wedge \alpha_{i_q,k_q}\}; (u'_1 \times (\cdots \times u'_r) \cdots) \rangle,$$

   where
   – the type variables $\alpha_{i_1,1}, \cdots, \alpha_{i_1,k_1}, \cdots, \alpha_{i_q,1}, \cdots, \alpha_{i_q,k_q}$ are all distinct and do not occur in w', and
   – $w'' = u'_1 \wedge \cdots \wedge u'_r$ is obtained from w' by replacing, for all $j \in \{1, \ldots, q\}$ and $h \in \{1, \ldots, k_j\}$, the h-h occurrence of α_{i_j} in w' by $\alpha_{i_j,h}$,

   is a principal-in-$\emptyset$ pair for $ai^{y:w'}$.
   (c) $\text{FTV}(v') = \{\alpha_{i_1}, \ldots, \alpha_{i_q}\}$ and, for all $j \in \{1, \ldots, q\}$, $\sharp_{\alpha_{i_j}}(u') = k_j$ imply that

   $$\langle \{x_{i_1} : \alpha_{i_1,1} \wedge \cdots \wedge \alpha_{i_1,k_1}, \ldots, x_{i_q} : \alpha_{i_q,1} \wedge \cdots \wedge \alpha_{i_q,k_q}\}; v'' \rangle,$$

   where
   – the type variables $\alpha_{i_1,1}, \cdots, \alpha_{i_1,k_1}, \cdots, \alpha_{i_q,1}, \cdots, \alpha_{i_q,k_q}$ are all distinct and do not occur in v', and
   – v'' is obtained from v' by replacing, for all $j \in \{1, \ldots, q\}$ and $h \in \{1, \ldots, k_j\}$, the h-h occurrence of α_{i_j} in v' by $\alpha_{i_j,h}$,

   is a principal-in-$\emptyset$ pair for $b^{v'}$.

(The proof is straightforward by induction on the definition of e^v in Fig. 4, by using the definition of the inference algorithm for $\vdash_2$ mentioned in the proof of Theorem 1.)
Therefore, by using the definition of the inference algorithm for $\vdash_2$ mentioned in the proof of Theorem 1, we have that $\langle \{y_1 : w_1, \ldots, y_n : w_n\}; \alpha_1 \to \cdots \to \alpha_m \to u \rangle$ is a principal-in-$\emptyset$ pair for $((\lambda y.y)(\lambda x_1 \ldots x_m.b^v y_1 \cdots y_n))$ and $\langle \emptyset; v \rangle$ is a principal-in-$\emptyset$ pair for e^v (according to Definition 5).

For all $v \in \mathbf{T}_2$ with $\mathrm{FTV}(v) = \{\alpha_1, \ldots, \alpha_m\}$ $(m \geq 0)$ and $v = w_1 \to \cdots \to w_n \to u$ $(n \geq 0$ and u is not an arrow type),

$$e^v = \lambda y_1 \ldots y_n.((\lambda y.y)(\lambda x_1 \ldots x_m.b^v y_1 \cdots y_n))\underbrace{(\mathsf{hd}\,\mathsf{nil}) \cdots (\mathsf{hd}\,\mathsf{nil})}_{m \text{ times}}$$

where the term b^v, with $\mathrm{FV}(b^v) = \{x_1, \ldots, x_m\}$, is inductively defined as follows

$$\begin{array}{l}
b^{\mathsf{bool}} = \mathsf{true} \\
b^{\mathsf{int}} = 0 \\
b^{\alpha_i} = x_i \ (1 \leq i \leq m) \\
b^{u_1 \times u_2} = \mathsf{pair}\ b^{u_1}\ b^{u_2} \\
b^{u\ \mathsf{list}} = \mathsf{cons}\ b^u\ \mathsf{nil} \\
b^{w \to v} = \lambda y.(\lambda z.b^v)ai^{y:w} \\
\\
ai^{y:u} = a^{y:u} \\
ai^{y:u \wedge w} = \mathsf{pair}\ a^{y:u}\ ai^{y:w} \\
\\
a^{y:u} = \mathsf{hd}\,(\mathsf{cons}\ b^u\ (\mathsf{cons}\ y\ \mathsf{nil}))
\end{array}$$

Fig. 4. Characteristic term e^v

Restatement of Lemma 1. $\langle A_1; v_1\rangle \leq_{\mathbf{spc}} \langle A_2; v_2\rangle$ *iff* $\langle A_1; v_1\rangle \leq_{\vdash_2} \langle A_2; v_2\rangle$.

Proof. – **(only if).** Let $\langle A_1; v_1\rangle \leq_{\mathbf{spc}} \langle A_2; v_2\rangle$. Then, for every term e, we have that, by rule (Spc), $\emptyset \vdash_2 e : \langle A_1; v_1\rangle$ implies $\emptyset \vdash_2 e : \langle A_2; v_2\rangle$. Therefore $\langle A_1; v_1\rangle \leq_{\vdash_2} \langle A_2; v_2\rangle$.

– **(if).** Let $\langle A_1; v_1\rangle \leq_{\vdash_2} \langle A_2; v_2\rangle$. Then $\mathrm{Dom}(A_1) = \mathrm{Dom}(A_2)$. Let $A_1 = \{x_1 : w_1, \ldots, x_n : w_n\}$, $A_2 = \{x_1 : w'_1, \ldots, x_n : w'_n\}$, $v'_1 = w_1 \to \cdots \to w_n \to v_1$, and $v'_2 = w'_1 \to \cdots \to w'_n \to v_2$. Since
  1. $\langle A_1; v_1\rangle \leq_{\vdash_2} \langle A_2; v_2\rangle$ implies $\langle \emptyset; v'_1\rangle \leq_{\vdash_2} \langle \emptyset; v'_2\rangle$, and
  2. $\langle \emptyset; v'_1\rangle \leq_{\mathbf{spc}} \langle \emptyset; v'_2\rangle$ implies $\langle A_1; v_1\rangle \leq_{\mathbf{spc}} \langle A_2; v_2\rangle$,

  it is enough to show that $\langle \emptyset; v'_1\rangle \leq_{\vdash_2} \langle \emptyset; v'_2\rangle$ implies $\langle \emptyset; v'_1\rangle \leq_{\mathbf{spc}} \langle \emptyset; v'_2\rangle$.
  Let $\langle \emptyset; v'_1\rangle \leq_{\vdash_2} \langle \emptyset; v'_2\rangle$. Since, by Lemma 3.1, $\emptyset \vdash_2 e^{v'_1} : v'_1$ holds we have that $\emptyset \vdash_2 e^{v'_1} : v'_2$ holds. Therefore, by Lemma 3.2, $\langle \emptyset; v'_1\rangle \leq_{\mathbf{spc}} \langle \emptyset; v'_2\rangle$.

Restatement of Theorem 2. *In system $\vdash_2$, a pair $\langle A; v\rangle$ is principal-in-D for e according to Definition 5 iff it is principal-in-D for e according to Definition 6.*

Proof. Immediate by Lemma 1.

A.4 Proof of Theorem 3

Restatement of Theorem 3 (Decidability of system $\vdash_2$+(RecDecP)). *There is an algorithm that, for every expression e (containing typing declarations for all the* rec*-bound variables) and environment D, decides whether e is $\vdash_2$ +(RecDecP)-typable in D and, if so, returns a principal-in-D pair for e.*

Proof. A sound and complete inference algorithm for system $\vdash_{\wedge_2}+(\textsc{RecDecP})$ can be obtained by extending the inference algorithm for $\vdash_2$ (see Section A.2) to handle expressions of the shape $\mathsf{rec}\,\{x : \langle A;\, v\rangle = e\}$ by:

- inferring a principal-in-$(D, x : \langle A;\, v\rangle)$ pair $\langle A';\, v'\rangle$ for e, and
- checking whether $\langle A';\, v'\rangle \leq_{\mathbf{spc}} \langle A;\, v\rangle$ holds, by using the algorithm for the specialization relation $\leq_{\mathbf{spc}}$ (see Section A.1).

A.5 Proof of Theorem 4

Restatement of Theorem 4 (Decidability of system $\vdash_2+(\textsc{RecP}_k)$). *For every $k \geq 0$, there is an algorithm that, for every expression e and environment D, decides whether e is $\vdash_2+(\textsc{RecP}_k)$-typable in D and, if so, returns a principal-in-D pair for e.*

Proof. A sound and complete inference algorithm for system $\vdash_2+(\textsc{RecP}_k)$ can be straightforwardly obtained by extending the inference algorithm for $\vdash_2$ (see Section A.2) to handle expressions of the shape $\mathsf{rec}\,\{x = e\}$ by:

- inferring a principal-in-D pair $\langle A;\, v\rangle$ for $\mathbf{LET}^k(x, e)$ and a principal-in-D pair $\langle A';\, v'\rangle$ for $\mathbf{LET}^{k+1}(x, e)$, and
- checking whether $\langle A';\, v'\rangle \leq_{\mathbf{spc}} \langle A;\, v\rangle$ holds, by using the algorithm for the specialization relation $\leq_{\mathbf{spc}}$ (see Section A.1).

A.6 Proof of Theorem 5

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

- $\mathsf{P}_X = \{\langle A;\, v\rangle \mid A \text{ is a rank 1 environment, } v \in \mathbf{T}_2, \text{ and } \mathrm{Dom}(A) = X\}$, and
- $\mathsf{bot}_X = \langle\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\};\, \alpha\rangle$, where the type variables $\alpha_1, \ldots, \alpha_n, \alpha$ are all distinct.

The relation $\leq_{\mathbf{spc}}$ is a preorder over P_X and, for all pairs $t \in \mathsf{P}_X$, $\mathsf{bot}_X \leq_{\mathbf{spc}} t$. Let $\leq^{\mathsf{err}}_{\mathbf{spc}}$ be the extension of $\leq_{\mathbf{spc}}$ to the set $\mathsf{P}^{\mathsf{err}}_X = \mathsf{P}_X \cup \{\mathsf{err}\}$, where err is a new element (not belonging to P_X) such that, for all $p \in \mathsf{P}^{\mathsf{err}}_X$, $p \leq^{\mathsf{err}}_{\mathbf{spc}} \mathsf{err}$. Let $=^{\mathsf{err}}_{\mathbf{spc}}$ be the equivalence relation on $\mathsf{P}^{\mathsf{err}}_X$ induced by $\leq^{\mathsf{err}}_{\mathbf{spc}}$ and let

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

For every pair environment D and expression $\mathsf{rec}\,\{x = e\}$, define the function $\mathcal{F}^{D,\mathsf{rec}\,\{x=e\}}_{\vdash_2} : \mathbf{P}^{\mathbf{err}}_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e\})} \to \mathbf{P}^{\mathbf{err}}_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e\})}$ such that, for all $[t]^{=^{\mathsf{err}}_{\mathbf{spc}}} \in$

$\mathbf{P}_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e\})}$,

$$\mathcal{F}^{D,\mathsf{rec}\,\{x=e\}}_{\vdash_2}([t]^{=^{\mathsf{err}}_{\mathbf{spc}}}) = \begin{cases} [t']^{=^{\mathsf{err}}_{\mathbf{spc}}}, & \text{where } t' \text{ is a principal-in-}(D, x:t) \text{ pair for } e \text{ in } \vdash_2, \\ & \text{if } e \text{ is } \vdash_2\text{-typable in } D, x:t \\ \mathbf{err}, & \text{otherwise} \end{cases}$$

and

$$\mathcal{F}^{D,\mathsf{rec}\,\{x=e\}}_{\vdash_2}(\mathbf{err}) \quad = \mathbf{err}.$$

The function $\mathcal{F}^{D,\mathsf{rec}\,\{x=e\}}_{\vdash_2}$ is monotone w.r.t. $\preceq_{\mathbf{spc}}$, i.e., for all $\mathbf{p}, \mathbf{p}' \in \mathbf{P}^{\mathbf{err}}_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e\})}$, if $\mathbf{p} \preceq_{\mathbf{spc}} \mathbf{p}'$ then $\mathcal{F}^{D,\mathsf{rec}\,\{x=e\}}_{\vdash_2}(\mathbf{p}) \preceq_{\mathbf{spc}} \mathcal{F}^{D,\mathsf{rec}\,\{x=e\}}_{\vdash_2}(\mathbf{p}')$. (The proof is straightforward by using the definition of the inference algorithm for $\vdash_2$ mentioned in the proof of Theorem 1.)

Restatement of Theorem 5. *For every $k \geq 0$:*

1. *If $D \vdash_2+(\mathrm{RecP}_k)\ e : \langle A;\, v\rangle$, then $D \vdash_2+(\mathrm{RecP})\ e : \langle A;\, v\rangle$.*
2. *If e is $\vdash_2+(\mathrm{RecP}_k)$-typable in D and $D \vdash_2+(\mathrm{RecP})\ e : \langle A;\, v\rangle$, then $D \vdash_2 +(\mathrm{RecP}_k)\ e : \langle A;\, v\rangle$.*

Proof. Assume, without loss of generality, that $e = \mathsf{rec}\,\{x = e'\}$ for some expression e' not containing rec-expressions.

1. Let $D \vdash_2+(\mathrm{RecP}_k)\ \mathsf{rec}\,\{x = e'\} : \langle A;\, v\rangle$. The derivation of the judgement $D \vdash_2+(\mathrm{RecP}_k)\ \mathsf{rec}\,\{x = e'\} : \langle A;\, v\rangle$ must end with exactly one application of rule (RecP_k) and some (possibly 0) applications if rule (Spc). Therefore we have $D \vdash_2 \mathbf{LET}^k(x, e') : \langle A';\, v'\rangle$, $D \vdash_2 \mathbf{LET}^{k+1}(x, e') : \langle A';\, v'\rangle$, and $\langle A';\, v'\rangle \leq_{\mathbf{spc}} \langle A;\, v\rangle$, where $\langle A';\, v'\rangle$ is a principal-in-D pair for both $\mathbf{LET}^k(x, e')$ and $\mathbf{LET}^{k+1}(x, e')$. By structural induction on $\vdash_2$ derivations $D, x : \langle A';\, v'\rangle \vdash_2 e' : \langle A';\, v'\rangle$, therefore $D \vdash_2+(\mathrm{RecP})\ e : \langle A';\, v'\rangle$ (by rule (RecP)), and $D \vdash_2+(\mathrm{RecP})\ e : \langle A;\, v\rangle$ (by rule (Spc)).
2. Let $\mathsf{rec}\,\{x = e'\}$ be $\vdash_2+(\mathrm{RecP}_k)$-typable in D and $D \vdash_2+(\mathrm{RecP})\ \mathsf{rec}\,\{x = e'\} : \langle A;\, v\rangle$.
   - Let
   $$\langle A_0;\, v_0\rangle = \underbrace{\mathcal{F}^{D,\mathsf{rec}\,\{x=e'\}}_{\vdash_2}(\,\cdots\,\mathcal{F}^{D,\mathsf{rec}\,\{x=e'\}}_{\vdash_2}}_{k \text{ times}}(\mathbf{bot}_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e'\})})\,\cdots\,).$$
   The hypothesis that $\mathsf{rec}\,\{x = e'\}$ is $\vdash_2+(\mathrm{RecP}_k)$-typable in D implies that
   $$D \vdash_2+(\mathrm{RecP}_k)\ \mathsf{rec}\,\{x = e'\} : \langle A_0;\, v_0\rangle.$$
   - By structural induction on $\vdash_2+(\mathrm{RecP})$ derivations, $D \vdash_2+(\mathrm{RecP})\ \mathsf{rec}\,\{x = e'\} : \langle A;\, v\rangle$ implies that $D, x : \langle A;\, v\rangle \vdash_2 e' : \langle A';\, v'\rangle$, for some $\langle A';\, v'\rangle$ such that $\langle A';\, v'\rangle \leq_{\mathbf{spc}} \langle A;\, v\rangle$. Therefore $[\langle A;\, v\rangle]^{=^{\mathsf{err}}_{\mathbf{spc}}}$ is a pre-fixed point of $\mathcal{F}^{D,\mathsf{rec}\,\{x=e'\}}_{\vdash_2}$ (i.e. $\mathcal{F}^{D,\mathsf{rec}\,\{x=e'\}}_{\vdash_2}([\langle A;\, v\rangle]^{=^{\mathsf{err}}_{\mathbf{spc}}}) \preceq_{\mathbf{spc}} [\langle A;\, v\rangle]^{=^{\mathsf{err}}_{\mathbf{spc}}}$).

   Since $\mathcal{F}^{D,\mathsf{rec}\,\{x=e'\}}_{\vdash_2}$ is monotone w.r.t. $\preceq_{\mathbf{spc}}$, $[\langle A_0;\, v_0\rangle]^{=^{\mathsf{err}}_{\mathbf{spc}}}$ is the minimum pre-fixed point (and fixed point) of $\mathcal{F}^{D,\mathsf{rec}\,\{x=e'\}}_{\vdash_2}$. Therefore $\langle A_0;\, v_0\rangle \leq_{\mathbf{spc}} \langle A;\, v\rangle$ and, by rule (Spc) , $D \vdash_2+(\mathrm{RecP}_k)\ \mathsf{rec}\,\{x = e'\} : \langle A;\, v\rangle$.

B Examples

The examples presented in this appendix have been checked by using a prototype implementation of the type inference algorithm for $\vdash_2$+(RecDecP2) + (RecP2$_k^{\text{Rec2}}$), where

- rule (RecDecP2) is the typing rule mentioned in Section 6, and
- rule (RecP2$_k^{\text{Rec2}}$) is the combination of rule (RecP2$_k$), mentioned in Section 6, and of rule (Rec2). Rule (Rec2) is the typing rule for mutual recursion proposed in [4]; it is based on the combination of rules (Rec) and (RecVac) of Section 5.2 and it slightly improves the rule for mutual recursion proposed in [10] which is based on rule (Rec) only.

The examples in Appendix B.1 are true rank 2 intersection typable, so they can not be typed by the Milner-Mycroft system [15].

The examples in Appendix B.2 can be typed by the Milner-Mycroft system but can not be typed by systems $\Lambda_2^{\mu,\mathbf{fix},-}$ [11] and $\vdash_2$+(Rec2) [4].

All the examples and an on-line demonstration of the prototype are available at the url http://lambda.di.unito.it/recp2/.

Remark 2 (About the prototype implementation). The prototype is written in O'Caml. The language accepted is a small subset of Caml (including datatype declarations and definitions by pattern matching) where the let rec declaration accepts as optional parameter a natural number k which specifies to use

- rule (RecP2$_k^{\text{Rec2}}$) if enclosed in braces, and
- a combination of rules (RecP2$_k$) and (RecDecP2) if enclosed in square brackets (in this case at least one of the mutually recursively defined variables must not have a declared pair scheme).

When square brackets without an enclosed natural number are supplied rule (RecDecP2) is used (in this case all the mutually recursively defined variables must have a declared pair scheme) and when no optional parameter is supplied rule (Rec2) is used.

B.1 True Rank 2 Intersection Typable Examples

In this section we present examples of $\vdash_2$+(RecP)-typable recursive definitions that are true rank 2 intersection typable and, therefore, cannot be typed by the Milner-Microft system [15], by Jim's rules for recursion [9,10] (outlined in Section 5.2), or by rule (Rec2) of [4].

Example 5 (Transposition of a matrix). The following O'Caml program[4] computes the transpose of a matrix given as a list of rows (the matrix is represented by a list of equally length lists):

[4] This example is taken from [11] (see also [6]). Here we use O'Caml syntax since this is the syntax accepted by our prototype implementation of the type inference algorithm.

```
let rec map f l = match l with [] -> [] | h :: t -> (f h) :: (map f t)
in
let rec mapTwo f lol = match lol with []      -> []
                                    | [] :: _ -> []
                                    | l -> (f hd l) :: (mapTwo f (f tl l))
in
    mapTwo map [[1;2];[3;4]]
```

The functions `hd`, `tl` and `map` are the standard list manipulating functions of types $\alpha\,\mathsf{list} \to \alpha$, $\alpha\,\mathsf{list} \to \alpha\,\mathsf{list}$ and $(\alpha \to \beta) \to \alpha\,\mathsf{list} \to \beta\,\mathsf{list}$, respectively.

If typable, this program would safely compute `[[1;3];[2;4]]`. However, the function `mapTwo` is true rank 2 intersection tybable, therefore the Hindley-Milner system, the Milner/Microft system, Jim's rule for recursion, and rule (REC2) are not able to type it.

After appropriate adjustment to the syntax of O'Caml:

- System $\vdash_2$+(RECP) allows to infer pair

$$\langle \emptyset;\ \Big(\big((\alpha\,\mathsf{list} \to \alpha) \to \beta\,\mathsf{list\,list} \to \gamma\big) \wedge \big((\delta\,\mathsf{list} \to \delta\,\mathsf{list}) \to \beta\,\mathsf{list\,list} \to \beta\,\mathsf{list\,list}\big)\Big) \to \beta\,\mathsf{list\,list} \to \gamma\,\mathsf{list}\rangle$$

  for the function `mapTwo` and pair $\langle \emptyset;\ \mathsf{int\,list\,list}\rangle$ for the whole program.[5]
- System $\Lambda_2^{\mu,\mathbf{fix},-}$ allows (see Section 5 of [11]) to infer the rank 2 universal type

$$\forall\alpha.\Big(\forall\beta.(\alpha\,\mathsf{list} \to \beta) \to \alpha\,\mathsf{list\,list} \to \beta\,\mathsf{list}\Big) \to \alpha\,\mathsf{list\,list} \to \alpha\,\mathsf{list\,list}$$

  for the function `mapTwo` and type $\mathsf{int\,list\,list}$ for the whole program.
- For every $k \geq 0$, the function `mapTwo` is not $\vdash_2$+(RECP$_k$)-typable. We are currently investigating a decidable extension of (RECP$_k$) that allows to type the function `mapTwo`.

Example 6. Consider the O'Caml program:

```
let toList x= [x]
in
let rec mapPair f l1 l2 = match l1 with
                            []       -> []
                          | h1 :: t1 -> match l2 with
                                          []       -> []
                                        | h2 :: t2 -> ((f h1),(g h2)) :: (mapPair f t1 t2)
in
   mapPair toList [1;2] [true;false]
```

where the functions `toList` and `mapPair` are Hindley-Milner-typable with principal types $\alpha \to \alpha\,\mathsf{list}$ and $(\alpha \to \beta) \to \alpha\,\mathsf{list} \to \alpha\,\mathsf{list} \to (\beta \times \beta)\,\mathsf{list}$, respectively.[6]

If typable, this program would safely compute `[([1],[true]);([2],[false])]`. However, it is not possible to type the expression `mapPair toList [1;2] [true;false]`

[5] This has been checked with the prototype, by using rule (RECDECP2).

[6] Therefore, the function `mapPair` is *not* a true rank 2 intersection typable term.

by assigning a simple type to `mapPair`,[7] so the Hindley-Milner system, the Milner-Microft system, Jim's rule for recursion, and rule (REC2) are not able to type it.

After appropriate adjustment to the syntax of O'Caml:

- System $\vdash_2+(\text{REC}\mathrm{P}_0)$ allows to infer principal pair

$$\langle \emptyset;\ \Big((\alpha_1 \to \beta_1) \wedge (\alpha_2 \to \beta_2)\Big) \to \alpha_1\,\mathsf{list} \to \alpha_2\,\mathsf{list} \to (\beta_1 \times \beta_2)\rangle$$

  for the function `mapPair` and principal-in-$\emptyset$ pair $\langle \emptyset;\ (\mathsf{int} \times \mathsf{bool})\,\mathsf{list}\rangle$ for the whole program.
- We conjecture that system $\Lambda_2^{\mu,\mathbf{fix},-}$ allows to infer the rank 2 universal type

$$\forall\alpha_1\alpha_2\beta_1\beta_2.(\forall\alpha\beta.\alpha \to \beta) \to \alpha_1\,\mathsf{list} \to \alpha_2\,\mathsf{list} \to (\beta_1 \times \beta_2)$$

  for the function `mapPair` and type $(\mathsf{int} \times \mathsf{bool})\,\mathsf{list}$ for the whole program.

B.2 Milner-Mycroft-typable Examples

In this section we show that system $\vdash_2+(\text{REC}\mathrm{P2}_0)$ allows to type well-known examples of recursive definitions that can be typed by the Milner-Microft system [15] but not by the system $\Lambda_2^{\mu,\mathbf{fix},-}$ of [11], by Jim's rule for mutual recursion [9,10], or by rule (REC2) of [4].

Example 7 (Simultaneous definition of `map`*,* `squarelist` *and* `complement`*).* The following O'Caml program is an example[8] of simultaneous recursive definition that is Milner-Mycroft typable and not Hindley-Milner typable:

```
let rec map f l = match l with []     -> []
                             | h :: t -> (f h) :: (map f t)
    and squarelist l = map (fun x -> x * x) l
    and complement l = map (fun x -> not x) l
```

After appropriate adjustment to the syntax of O'Caml:

- System $\vdash_2+(\text{REC}\mathrm{P2}_0)$ allows to infer principal-in-$\emptyset$ pairs $\langle \emptyset;\ (\alpha \to \beta) \to \alpha\,\mathsf{list} \to \beta\,\mathsf{list}\rangle$, $\langle \emptyset;\ \mathsf{int}\,\mathsf{list} \to \mathsf{int}\,\mathsf{list}\rangle$ and $\langle \emptyset;\ \mathsf{bool}\,\mathsf{list} \to \mathsf{bool}\,\mathsf{list}\rangle$ to the functions `map`, `squarelist` and `complement`, respectively.
- The example can not be typed by system $\Lambda_2^{\mu,\mathbf{fix},-}$ (as explained in Section 6 of [11]), by Jim's rules for mutual recursion, by rule (REC2), and by Mycroft's rule (FIX′) [15] (which is a decidable restriction of rule (FIX)).

Example 8 (The function `collect` *on the datatype* $\alpha\,\mathsf{tree}$*).* The following O'Caml program is an example[9] of recursive definition that is Milner-Mycroft typable (indeed, it can be typed also by Mycroft's rule (FIX′) [15] and by rule (REC′) outlined in Section 5.2) and not Hindley-Milner-typable.

[7] We can say that "the use of `mapPair`" is true rank 2 intersection typable.

[8] Taken from [15].

[9] This example is taken from [10], it originally comes from the ML mailing list, and has arisen in practice.

```
type 'a tree  =  EMPTY | NODE of 'a * ('a tree) tree

let rec append l1 l2 = match l1 with []     -> l2
                                   | h :: t -> h :: (append t l2)
in
let rec flatmap f l = match l with []     -> []
                                 | h :: t -> append (f h) (flatmap f t)
in
let rec collect t = match t with
                    EMPTY      -> []
                  | NODE(n,t1) -> n :: flatmap collect (collect t1)
in
    collect (NODE(1,NODE(NODE(2,EMPTY),EMPTY)))
```

The type $\alpha\,\mathsf{tree}$ is a polymorphic tree type, while `append` and `flatmap` are the standard list manipulation functions of types $\alpha\,\mathsf{list} \to \alpha\,\mathsf{list} \to \alpha\,\mathsf{list}$ and $(\alpha \to \beta\,\mathsf{list}) \to \alpha\,\mathsf{list} \to \beta\,\mathsf{list}$, respectively.

The function `collect` collects all the labels in an $\alpha\,\mathsf{tree}$ and returns them in an $\alpha\,\mathsf{list}$. If typable, this program would safely compute the list `[1;2]`.

After appropriate adjustment to the syntax of O'Caml:

- System $\vdash_2+(\textsc{RecP}2_0)$ allows to infer principal pair $\langle \emptyset;\ \alpha\,\mathsf{tree} \to \alpha\,\mathsf{list}\rangle$ for the function `collect` and principal-in-$\emptyset$ pair $\langle \emptyset;\ \mathsf{int}\,\mathsf{list}\rangle$ for the whole program.
- This example cannot be typed by Jim's rules for recursion (as explained in Section 3 of [10]) and by rule (Rec2). We conjecture that it cannot be typed also by system $\Lambda_2^{\mu,\mathbf{fix},-}$ of [11].