# Typing Local Definitions and Conditional Expressions with Rank 2 Intersection
## (Extended Abstract)

Ferruccio Damiani

Dipartimento di Informatica, Università di Torino,
Corso Svizzera 185, 10149 Torino, Italy
damiani@di.unito.it
http://www.di.unito.it/∼damiani

**Abstract.** We introduce a variant of the system of rank 2 intersection types with new typing rules for local definitions (let-expressions and letrec-expressions) and conditional expressions (if-expressions and case-expressions). These extensions are a further step towards the use of intersection types in "real" programming languages.

## 1 Introduction

The Hindley-Milner type system [3] is the core of the type systems of modern functional programming languages, like ML [11], Miranda, Haskell, and Clean. The fact that this type system is somewhat inflexible[1] has motivated the search for more expressive, but still decidable, type systems (see, for instance, [10,14,4,2,8,7,9]). The extensions based on intersection types are particular interesting since they generally have the *principal typing property*[2], whose advantages w.r.t. the *principal type property*[3] of the ML type system have been described in [7]. In particular the system of *rank 2 intersection types* [10,14,15,7] is able to type all ML programs, has the principal typing property, decidable type inference, and complexity of type inference which is of the same order as in ML. The variant of the system of rank 2 intersection types considered by Jim [7] is particularly interesting since it includes a new rule for typing recursive definitions which allows to type some, but not all, examples of polymorphic recursion [12].

In this paper we build on Jim's work [7] and present a new system of rank 2 intersection types, $\vdash_{\wedge_2}^{\mathrm{If,Let,Rec}}$, which allows to give more expressive typings to locally defined identifiers (let-bound and letrec-bound identifiers) and to conditional expressions (we consider only if-expressions, but the technique can be

[1] In particular it does not allow to assign different types to different occurrences of a formal parameter in the body of a function.

[2] A type system has the *principal typing property* if, whenever a term $e$ is typable, there exist a type environment $A$ and a type $v$ representing all possible typings of $e$.

[3] A type system has the *principal type property* if, whenever a term $e$ is typable in a type environment $A$, there exists a type $v$ representing all possible types of $e$ in $A$.

straightforwardly applied to case-expressions). These extensions are a further step towards the use of intersection types in "real" programming languages.

**Better Typings for Local Definitions.** The system of simple types [5] assigns the same type to all the uses of an identifier. To overcome this limitation the Hindley-Milner type system [3] considers a special rule to type local definitions, in this way locally defined identifiers (let-bound identifiers) are handled in a different way from the formal parameters of the functions ($\lambda$-bound identifiers).

In practice let-polymorphism is also used to allow polymorphic use of globally defined identifiers. The key idea is that of handling an expression $e$ which uses globally defined identifiers $x_1, \ldots, x_n$ like the expression $\mathsf{let}\, x_1 = e_1 \,\mathsf{in} \cdots \mathsf{let}\, x_n = e_n \,\mathsf{in}\, e$, in which the definitions of $x_1, \ldots, x_n$ are local (and therefore available). This use of let-polymorphism to deal with global definitions has often been described as an odd feature, since it does not allow to typecheck global definitions in isolation. The problem can be identified with the fact that algorithm $\mathcal{W}$ requires as necessary inputs the type assumptions for the free identifiers of the expression being typed. Some solutions to overcome this limitation have been proposed in the literature (see for instance [1,13]).

Systems with rank 2 intersection types can provide an elegant solution to this problem by relying on the principal typing property, see [7], and handling let-expressions "$\mathsf{let}\, x = e_0 \,\mathsf{in}\, e$" as syntactic sugar for "$(\lambda\, x.e)e_0$". In this way both locally defined and globally defined identifiers are handled as function formal parameters. However this strategy has a drawback: it forces to assign simple types to the uses of locally defined identifiers. For instance, the expression

$$\mathsf{let} \quad g = \lambda\, f.\mathsf{pair}\,(f\, 2)\,(f\,\mathsf{true}) \quad \mathsf{in} \quad g(\lambda\, y.\mathsf{cons}\, y\, \mathsf{nil}) \tag{1}$$

cannot be typed, since to type (1) it is necessary to assign the rank 2 type $((\mathsf{int} \to \mathsf{int\,list}) \wedge (\mathsf{bool} \to \mathsf{bool\,list})) \to (\mathsf{int\,list} \times \mathsf{bool\,list})$ to the locally defined identifier $g$.

In this paper we present a technique that, while preserving the benefits of the principal typing property of the system of rank 2 intersection types, allows to assign rank 2 intersection types to the uses of locally defined identifiers, by exploiting the fact that their definition is indeed available. As we will see, typing let-expressions $\mathsf{let}\ x = e_0\ \mathsf{in}\ e$ by associating to the identifier $x$ the *principal type scheme* of $e_0$ (which is a formula $\forall\overrightarrow{\alpha}.v_0$, where $v_0$ is a rank 2 type and $\overrightarrow{\alpha}$ are *some* of the type variables of $v_0$) is not a good solution, since, when $e_0$ contains free identifies, it may happen that replacing a subexpression $(\lambda\, x.e)e_0$ with $\mathsf{let}\, x = e_0 \,\mathsf{in}\, e$ does not preserve typability. To avoid this problem we will associate to $x$ the *principal pair scheme* of $e_0$ (which is a formula $\forall\overrightarrow{\alpha}.\langle A_0, v_0\rangle$, where $A_0$ is a type environment, $v_0$ is a rank 2 type, and $\overrightarrow{\alpha}$ are *all* the type variables of $A_0$ and $v_0$).

**Better Typings for Conditional Expressions.** The ML type system handles an if-expression "$\mathsf{if}\, e \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$" like the application "$\mathsf{ifc}\, e\, e_1\, e_2$", where ifc is a special constant of principal type scheme $\forall\alpha.\mathsf{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$. If we apply this strategy to a system with rank 2 intersection types we are forced to assign simple types to the conditional expression and to its branches, $e_1$ and $e_2$, and so the additional type information provided by intersection is lost.

In this paper we present a technique that allows to overcome this limitation and to assign rank 2 intersection types to conditional expressions. For simplicity we consider only if-expressions, but the technique can be straightforwardly applied to case-expressions and functions defined by cases. As we will see, allowing to assign to an if-expression $\mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ any rank 2 type $v$ that can be assigned to both $e_1$ and $e_2$ will destroy the principal typing (and type) property of the rank 2 intersection type system. So, to preserve the principal typing property, we will introduce a condition that limits the use of intersection in the type $v$ assigned to the branches $e_1$ and $e_2$ of the if-expression.

**Organization of the Paper.** In Section 2 of this paper we describe a simple programming language, that we call mini-ML, which can be considered the kernel of functional programming languages like ML, Miranda, Haskell, and Clean (the evaluation mechanism, call-by-name or call-by-value, is not relevant for the purpose of typechecking). Section 3 introduces the syntax of our rank 2 intersection types, together with other basic definitions. Section 4 presents the $\vdash^{\mathrm{Rec}}_{\wedge_2}$ type system, which is essentially the extension to mini-ML of the type system $\vdash_{\mathbf{P}^{\mathrm{R}}_2}$ of [7]. In Sections 5 and 6 we describe two new type systems for mini-ML: $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$, which extends the system $\vdash^{\mathrm{Rec}}_{\wedge_2}$ with more powerful rules for typing local definitions (let-expressions and letrec-expressions), and $\vdash^{\mathrm{If,Let,Rec}}_{\wedge_2}$, which extends the system $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$ with a more powerful rule for typing if-expressions.

## 2 The Language Mini-ML

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 inspecting and decomposing data structures. The base functions include some arithmetic operators, and the functions for decomposing pairs ($\mathsf{prj}_1$ and $\mathsf{prj}_2$) and for decomposing and inspecting lists (hd, tl, and null). 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

$$
\begin{aligned}
c &::= bf \mid cs \\
bf &::= \mathsf{not} \mid \mathsf{and} \mid \mathsf{or} \mid + \mid - \mid * \mid / \mid = \mid \cdots \mid \mathsf{prj}_1 \mid \mathsf{prj}_2 \mid \mathsf{hd} \mid \mathsf{tl} \mid \mathsf{null} \\
cs &::= () \mid \mathsf{true} \mid \mathsf{false} \mid \cdots \mid -1 \mid 0 \mid 1 \mid \cdots \mid \mathsf{nil} \mid \mathsf{pair} \mid \mathsf{cons}
\end{aligned}
$$

*Expressions* (ranged over by $e$) have the following syntax

$$
\begin{aligned}
e ::= {} & x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \\
\mid {} & \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid \mathsf{letrec}\, \{x_1 = e_1, \ldots, x_n = e_n\}\, \mathsf{in}\, e
\end{aligned}
$$

where $x$, $x_1$, ..., $x_n$ range over identifiers. The construct $\mathsf{letrec}$ allows mutually recursive expression definitions. Let $\mathrm{FV}(e)$ denote the set of free identifiers of the expression $e$. Expressions are considered syntactically equal modulo renaming of the bound identifiers. In order to simplify the presentation we assume that, in every expression, different bound identifiers have different names and that the names of bound identifiers cannot occur free in the expression (this can be always enforced by suitable renaming of bound identifiers).

## 3 Types, Schemes, and Environments

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.

**Types and Schemes.** The language of *simple types* ($\mathbf{T}_0$), ranged over by $u$, is defined by the grammar: $u ::= \alpha \mid \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid u \to u \mid u \times u \mid u\,\mathsf{list}$. We have type variables (ranged over by $\alpha$) and a selection of *ground* types and *composite* types. The *ground* types are $\mathsf{unit}$ (the singleton type), $\mathsf{bool}$ (the set of booleans), and $\mathsf{int}$ (the set of integers). The composite types are product and list types.

The language of *rank 1 intersection types* ($\mathbf{T}_1$), ranged over by $ui$, the language of *rank 2 intersection types* ($\mathbf{T}_2$), ranged over by $v$, and the language of *rank 2 intersection schemes* ($\mathbf{T}_{\forall 2}$), ranged over by $vs$, are defined as follows

$$
\begin{array}{ll}
ui ::= u_1 \wedge \cdots \wedge u_n & \text{(rank 1 types, i.e. intersections of simple types)} \\
v \ ::= u \mid ui \to v & \text{(rank 2 types)} \\
vs ::= \forall \overrightarrow{\alpha}.v & \text{(rank 2 schemes)}
\end{array}
$$

where $u$ ranges over the set of simple types $\mathbf{T}_0$, $n \geq 1$, and $\overrightarrow{\alpha}$ is a finite (possibly empty) sequence of type variables $\alpha_1 \cdots \alpha_m$ $(m \geq 0)$. Note that $\mathbf{T}_0 = \mathbf{T}_1 \cap \mathbf{T}_2$. Let $\epsilon$ denote the empty sequence. We consider $\forall \epsilon.v \neq v$, so $\mathbf{T}_2 \cap \mathbf{T}_{\forall 2} = \emptyset$.

*Free* and *bound* type variables are defined as usual. For every type $t \in \mathbf{T}_1 \cup \mathbf{T}_2 \cup \mathbf{T}_{\forall 2}$ let $\mathrm{FTV}(t)$ denote the set of free type variables of $t$. We say that a scheme $vs$ is *closed* if $\mathrm{FTV}(vs) = \emptyset$.

To simplify the presentation we adopt the following syntactic convention: we consider $\wedge$ to be associative, commutative, and idempotent. Modulo this convention any type in $\mathbf{T}_1$ can be considered as a set of types in $\mathbf{T}_0$. We also assume that for every scheme $\forall \overrightarrow{\alpha}.v$ we have that $\{\overrightarrow{\alpha}\} \subseteq \mathrm{FTV}(v)$.

A *substitution* $\mathbf{s}$ is a mapping from type variables to simple types which is the identity on all but a finite number of type variables. The domain, $\mathrm{Dom}(\mathbf{s})$, of a substitution $\mathbf{s}$ is the set of type variables $\{\alpha \mid \mathbf{s}(\alpha) \neq \alpha\}$. We use $\mathbf{s}_{\{\overrightarrow{\alpha}\}}$ to range over substitutions whose domain is a subset of $\{\overrightarrow{\alpha}\}$. Note that, since substitutions replace free variables by simple types, we have that $\mathbf{T}_0$, $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_{\forall 2}$ are closed under substitution.

The following definition are fairly standard. Note that we keep a clear distinction between *subtyping* and *instantiation* relations, and we do not introduce a subtyping relation between rank 2 schemes.

**Definition 1 (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 inductively defined as follows*

- $u \leq_2 u$, *if* $u \in \mathbf{T}_0$
- $u_1 \wedge \cdots \wedge u_n \leq_1 u'_1 \wedge \cdots \wedge u'_m$, *if* $\{u_1, \ldots, u_n\} \supseteq \{u'_1, \ldots, u'_m\}$
- $ui \to v \leq_2 ui' \to v'$, *if* $ui' \leq_1 ui$ *and* $v \leq_2 v'$.

**Definition 2 (Instantiation relations $\leq_{\forall 2,0}$, $\leq_{\forall 2,1}$ and $\leq_{\forall 2,2}$).** *The instantiation relations $\leq_{\forall 2,0}$ ($\subseteq \mathbf{T}_{\forall 2} \times \mathbf{T}_0$), $\leq_{\forall 2,1}$ ($\subseteq \mathbf{T}_{\forall 2} \times \mathbf{T}_1$), and $\leq_{\forall 2,2}$ ($\subseteq \mathbf{T}_{\forall 2} \times \mathbf{T}_2$) are defined as follows. For every scheme $\forall \overrightarrow{\alpha}.v \in \mathbf{T}_{\forall 2}$ and for every type*

0. $u \in \mathbf{T}_0$, *we write* $\forall \overrightarrow{\alpha}.v \leq_{\forall 2,0} u$ *if* $u = \mathbf{s}_{\{\overrightarrow{\alpha}\}}(v)$, *for some substitution* $\mathbf{s}_{\{\overrightarrow{\alpha}\}}$;
1. $u_1 \wedge \cdots \wedge u_n \in \mathbf{T}_1$, *we write* $\forall \overrightarrow{\alpha}.v \leq_{\forall 2,1} u_1 \wedge \cdots \wedge u_n$ *if* $\forall \overrightarrow{\alpha}.v \leq_{\forall 2,0} u_i$, *for every* $i \in \{1, \ldots, n\}$;
2. $v' \in \mathbf{T}_2$, *we say that* $v'$ is an instance of $\forall \overrightarrow{\alpha}.v$, *and write* $\forall \overrightarrow{\alpha}.v \leq_{\forall 2,2} v'$, *if* $\mathbf{s}_{\{\overrightarrow{\alpha}\}}(v) \leq_2 v'$, *for some substitution* $\mathbf{s}_{\{\overrightarrow{\alpha}\}}$.

For example, for $vs = \forall \alpha_1\alpha_2\alpha_3.((\alpha_1 \to \alpha_3) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_3$, we have (remember that $\wedge$ is idempotent) $vs \leq_{\forall 2,0} (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ (by using the substitution $\mathbf{s}_1 = [\alpha_1, \alpha_2, \alpha_3 := \mathsf{int}]$) and $vs \leq_{\forall 2,1} ((\mathsf{int} \to \mathsf{int}) \to \mathsf{int}) \wedge ((\mathsf{bool} \to \mathsf{bool}) \to \mathsf{bool})$ (by $\mathbf{s}_1$ as above, and $\mathbf{s}_2 = [\alpha_1, \alpha_2, \alpha_3 := \mathsf{bool}]$). We also have $\forall \alpha.\alpha \to \alpha \leq_{\forall 2,2} (\alpha_1 \wedge \alpha_2) \to \alpha_1$ (by $\mathbf{s} = [\alpha := \alpha_1]$ and $\leq_2$).

**Type Environments.** A *type environment* $T$ is a set $\{x_1 : t_1, \ldots, x_n : t_n\}$ of type assumptions for identifiers such that every identifier $x$ can occur at most once in $T$. We write $\mathrm{Dom}(T)$ for $\{x_1, \ldots, x_n\}$ and $T, x : t$ for the environment $T \cup \{x : t\}$ where it is assumed that $x \notin \mathrm{Dom}(T)$. In particular:

- a *rank 1 type environment* $A$ is an environment $\{x_1 : ui_1, \ldots, x_n : ui_n\}$ of rank 1 type assumptions for identifiers, and
- a *rank 2 scheme environment* $B$ is an environment $\{x_1 : vs_1, \ldots, x_n : vs_n\}$ of *closed* rank 2 schemes assumptions for identifiers.

For every type $v \in \mathbf{T}_2$ and type environment $T$ we write $\mathrm{Gen}(T, v)$ for the $\forall$-*closure of* $v$ *in* $T$, i.e. for the scheme $\forall \overrightarrow{\alpha}.v$ where $\{\overrightarrow{\alpha}\} = \mathrm{FTV}(v) - \mathrm{FTV}(T)$.

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

$$\begin{array}{l}\{x : ui_1 \wedge ui_2 \mid x : ui_1 \in A_1 \text{ and } x : ui_2 \in A_2\} \\ \cup \{x : ui_1 \in A_1 \mid x \notin \mathrm{Dom}(A_2)\} \cup \{x : ui_2 \in A_2 \mid x \notin \mathrm{Dom}(A_1)\}\ ,\end{array}$$

and write $A_1 \leq_1 A_2$ to mean that $\mathrm{Dom}(A_1) = \mathrm{Dom}(A_2)$ and for every assumption $x : ui_2 \in A_2$ there is an assumption $x : ui_1 \in A_1$ such that $ui_1 \leq_1 ui_2$.

## 4 System $\vdash^{\mathbf{Rec}}_{\wedge_2}$: Jim's $\vdash_{\mathbf{P}_2^{\mathrm{R}}}$ Type System Revised

In this section we introduce the $\vdash^{\mathrm{Rec}}_{\wedge_2}$ type system for mini-ML. System $\vdash^{\mathrm{Rec}}_{\wedge_2}$ is essentially an extension to mini-ML of the type system $\vdash_{\mathbf{P}_2^{\mathrm{R}}}$ of [7] (the language considered in [7] is a $\lambda$-calculus without constants enriched with letrec-expressions). Then, in Sections 5 and 6, we will extend $\vdash^{\mathrm{Rec}}_{\wedge_2}$ with new typing rules for local definitions and conditional expressions.

**The Type Inference Rules.** The type inference system $\vdash^{\mathrm{Rec}}_{\wedge_2}$ has judgements of the form $A; B \vdash^{\mathrm{Rec}}_{\wedge_2} e : v$, where $B$ is a rank 2 environment specifying closed $\mathbf{T}_{\forall 2}$ types for library identifiers[4], and $A$ is a rank 1 environment containing the type assumptions for the remaining free identifiers of $e$. So $\mathrm{FV}(e) \subseteq \mathrm{Dom}(A \cup B)$, $\mathrm{Dom}(A) \cap \mathrm{Dom}(B) = \emptyset$, and $\mathrm{Dom}(A) = \mathrm{FV}(e) - \mathrm{Dom}(B)$[5]. Note that (by definition of rank 2 scheme environment) $\mathrm{FTV}(B) = \emptyset$.

We say that *$e$ is typable in $\vdash^{\mathrm{Rec}}_{\wedge_2}$ w.r.t. the library environment $B$* if there exist a typing $A; B \vdash^{\mathrm{Rec}}_{\wedge_2} e : v$, for some $A$ and $v$.

The type inference rules are presented in Fig. 2. The rule for typing constants uses the function Typeof (tabulated in Fig. 1) which assigns a closed scheme to each constant. Since, by definition, $\mathrm{Dom}(A)$ contains exactly the assumptions for the free non-library identifiers of the expression $e$ being typed, we have two rules for typing an abstraction $\lambda\, x.e$, corresponding to the two cases $x \in \mathrm{FV}(e)$ and $x \notin \mathrm{FV}(e)$. The rule for typing function application, (App), allows to use a different typing for each expected type of the argument. The rule for typing if-expressions handles an expression $\mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2$ like the application $\mathsf{ifc}\; e\; e_1\; e_2$, where $\mathsf{ifc}$ is a special constant of type $\forall \alpha.\mathsf{bool} \to \alpha \to \alpha \to \alpha$. A let-expression, $\mathsf{let}\, x = e_0\; \mathsf{in}\; e$, is considered as syntactic sugar for the application $(\lambda\, x.e)e_0$.

The rule for typing letrec-expressions, $\mathsf{letrec}\,\{x_1 = e_1, \ldots, x_n = e_n\}\,\mathsf{in}\, e$, introduces auxiliary expressions of the form $\mathsf{rec}_i\,\{x_1 = e_1, \ldots, x_n = e_n\}$, for $1 \le i \le n$. These auxiliary expressions are introduced just for convenience in presenting the type system ($\mathsf{rec}_i\,\{x_1 = e_1, \ldots, x_n = e_n\}$ is simply a short for $\mathsf{letrec}\,\{x_1 = e_1, \ldots, x_n = e_n\}\,\mathsf{in}\, x_i$).

The only non-structural rule is (Sub), which allows to assume less specific types for the free non-library identifiers and to assign more specific types to expressions (for instance, without rule (Sub) it would not be possible to assign type $(\alpha_1 \wedge \alpha_2) \to \alpha_1$ to the identity function $\lambda\, x.x$). The operations of $\forall$-introduction and $\forall$-elimination are embedded in the structural rules.

**Comparison with the System $\vdash_{\mathbf{P}_2^{\mathrm{R}}}$.** Besides the presence of constants, if-expressions, and let-expressions, the main differences between $\vdash^{\mathrm{Rec}}_{\wedge_2}$ and $\vdash_{\mathbf{P}_2^{\mathrm{R}}}$ are the presence of the library environment $B$ (which is not present in $\vdash_{\mathbf{P}_2^{\mathrm{R}}}$, although its use has been suggested in [7]) and the improved typing rules for recursive

[4] I.e. for the identifiers defined in the libraries available to the programmer.

[5] The fact that the environment $A$ is *relevant* (i.e., $x \in \mathrm{Dom}(A)$ implies $x \in \mathrm{FV}(e)$) is used in rule (Rec) of Fig. 2 (as explained at the end of Section 4).

| $c$ | Typeof($c$) | $c$ | Typeof($c$) |
|---|---|---|---|
| () | $\forall\epsilon.\mathsf{unit}$ | pair | $\forall\alpha_1\alpha_2.\alpha_1 \to \alpha_2 \to (\alpha_1 \times \alpha_2)$ |
| true, false | $\forall\epsilon.\mathsf{bool}$ | $\mathsf{prj}_i$ | $\forall\alpha_1\alpha_2.(\alpha_1 \times \alpha_2) \to \alpha_i$ |
| not | $\forall\epsilon.\mathsf{bool} \to \mathsf{bool}$ | nil | $\forall\alpha.\alpha\,\mathsf{list}$ |
| and, or | $\forall\epsilon.\mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$ | cons | $\forall\alpha.\alpha \to \alpha\,\mathsf{list} \to \alpha\,\mathsf{list}$ |
| $\cdots -1,\, 0,\, 1, \cdots$ | $\forall\epsilon.\mathsf{int}$ | hd | $\forall\alpha.\alpha\,\mathsf{list} \to \alpha$ |
| $+, -, *, /$ | $\forall\epsilon.\mathsf{int} \times \mathsf{int} \to \mathsf{int}$ | tl | $\forall\alpha.\alpha\,\mathsf{list} \to \alpha\,\mathsf{list}$ |
| $=, <, \cdots$ | $\forall\epsilon.\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$ | null | $\forall\alpha.\alpha\,\mathsf{list} \to \mathsf{bool}$ |

**Fig. 1.** Types for constants

definitions. For instance we have that $\mathsf{rec}_1\,\{x_1 = \lambda\, y.yy\}$ can be typed in $\vdash^{\mathrm{Rec}}_{\wedge_2}$: $\emptyset;\emptyset \vdash^{\mathrm{Rec}}_{\wedge_2} \mathsf{rec}_1\,\{x_1 = \lambda\, y.yy\} : ((\alpha_1 \to \alpha_2) \wedge \alpha_1) \to \alpha_2$, while it is not typable in $\vdash_{\mathbf{P}^{\mathrm{R}}_2}$. This is due to the fact that, when typing a (possibly mutually) recursive definition $\mathsf{rec}_{i_0}\,\{x_1 = e_1, \ldots, x_n = e_n\}$, the rules of $\vdash_{\mathbf{P}^{\mathrm{R}}_2}$ require that (also for those $i \in \{1, \ldots, n\}$ such that $x_i \notin \cup_{j\in\{1,\ldots,n\}}\mathrm{FV}(e_j)$) the rank two type $v_i$ assigned to $e_i$ must be such that $\mathrm{Gen}(A,\, v_i) \leq_{\forall 2,1} u_i$, for some simple type $u_i$. This anomaly has been pointed out in [6] where it is also described a solution to the problem in the case of a single recursive definition ($\mathsf{rec}_1\{x_1 = e_1\}$): if $x_1 \notin \mathrm{FV}(e_1)$ then do not require that $\mathrm{Gen}(A,\, v_1) \leq_{\forall 2,1} u_1$. System $\vdash^{\mathrm{Rec}}_{\wedge_2}$ generalizes this idea to mutually recursive definitions: the constraint $\mathrm{Gen}(A,\, v_i) \leq_{\forall 2,1} ui_i$ is enforced only for those $i$ such that $x_i \in \cup_{j\in\{1,\ldots,n\}}\mathrm{FV}(e_j)$.

**Principal Typings for $\vdash^{\mathbf{Rec}}_{\wedge_2}$.** The type system $\vdash^{\mathrm{Rec}}_{\wedge_2}$ has the principal typing property. The following definition and theorem are a formulation for $\vdash^{\mathrm{Rec}}_{\wedge_2}$ (keeping in account the presence of the library environment $B$) of an analogous result for $\vdash_{\mathbf{P}^{\mathrm{R}}_2}$ presented in [7].

**Definition 3 (Principal typings for $\vdash^{\mathrm{Rec}}_{\wedge_2}$).** *A typing $A';B \vdash^{\mathrm{Rec}}_{\wedge_2} e : v'$ is an* instance *of a typing $A;B \vdash^{\mathrm{Rec}}_{\wedge_2} e : v$ if there is a substitution* $\mathbf{s}$ *such that* $\mathrm{Dom}(\mathbf{s}) = \mathrm{FTV}(A) \cup \mathrm{FTV}(v)$*,* $\mathbf{s}(v) \leq_2 v'$ *and* $A' \leq_1 \mathbf{s}(A)$*.*
*A typing $A;B \vdash^{\mathrm{Rec}}_{\wedge_2} e : v$ is a* principal typing for $e$ w.r.t. $B$ *if any other typing of $e$ w.r.t. $B$ is an instance of it.*

**Theorem 1 (Principal typing property for $\vdash^{\mathrm{Rec}}_{\wedge_2}$).** *If $e$ is typable in $\vdash^{\mathrm{Rec}}_{\wedge_2}$ w.r.t. $B$, then it has a principal typing w.r.t. $B$.*

## 5 The System $\vdash^{\mathbf{Let,Rec}}_{\wedge_2}$: Better Typings for Local Definitions

Rule (LetSugar) of $\vdash^{\mathrm{Rec}}_{\wedge_2}$ prevents us to assign rank 2 types to the uses of local definitions. The following rule, which allows to store the rank 2 type schemes

$$(\textsc{Id}_1)\ \{x : u\}; B \vdash x : u \quad x \notin \mathrm{Dom}(B) \qquad (\textsc{Id}_2)\ \emptyset; B, x : \forall\overrightarrow{\alpha}.v \vdash x : \mathbf{s}_{\{\overrightarrow{\alpha}\}}(v)$$

$$(\textsc{Con})\ \emptyset; B \vdash c : \mathbf{s}_{\{\overrightarrow{\alpha}\}}(v) \quad \mathrm{Typeof}(c) = \forall\overrightarrow{\alpha}.v$$

$$(\textsc{Abs})\ \frac{A, x : ui; B \vdash e : v}{A; B \vdash \lambda x.e : ui \to v} \qquad (\textsc{AbsVac})\ \frac{A; B \vdash e : v}{A; B \vdash \lambda x.e : u \to v}\ x \notin \mathrm{Dom}(A \cup B)$$

$$(\textsc{App})\ \frac{A; B \vdash e : u_1 \wedge \cdots \wedge u_n \to v \quad (\forall i \in \{1,\ldots,n\})\ A_i; B \vdash e_0 : v_i\ \mathrm{Gen}(A_i, v_i) \leq_{\forall 2,0} u_i}{A + A_1 + \cdots + A_n; B \vdash ee_0 : v}$$

$$(\textsc{IfSimple})\ \frac{A; B \vdash e : \mathsf{bool} \quad A_1; B \vdash e_1 : u \quad A_2; B \vdash e_2 : u}{A + A_1 + A_2; B \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : u}$$

$$(\textsc{LetSugar})\ \frac{A; B \vdash (\lambda x.e)e_0 : v}{A; B \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e : v}$$

$$(\textsc{Letrec})\ \frac{A; B \vdash (\lambda x_1.\cdots.\lambda x_n.e)e'_1 \cdots e'_n : v}{A; B \vdash \mathsf{letrec}\,\{x_1 = e_1, \ldots, x_n = e_n\}\ \mathsf{in}\ e : v}$$
where, for $i \in \{1, \ldots, n\}$, $e'_i = \mathsf{rec}_i\,\{x_1 = e_1, \ldots, x_n = e_n\}$

$$(\textsc{Rec})\ \frac{(\forall i \in \{1, \ldots, n\})\ A_i; B \vdash e_i : v_i \quad (\forall j \in \{j_1, \ldots, j_m\})\ \mathrm{Gen}(A, v_j) \leq_{\forall 2,1} ui_j}{A_0; B \vdash \mathsf{rec}_{i_0}\,\{x_1 = e_1, \ldots, x_n = e_n\} : v_{i_0}}$$
where $\{x_{j_1}, \ldots, x_{j_m}\} = \{x_1, \ldots, x_n\} \cap (\cup_{1 \leq i \leq n} \mathrm{FV}(e_i))$,
$A = A_0, x_{j_1} : ui_{j_1}, \ldots, x_{j_m} : ui_{j_m} = A_1 + \cdots + A_n$, and $i_0 \in \{1, \ldots, n\}$

$$(\textsc{Sub})\ \frac{A_1; B \vdash e : v_1 \quad A_2 \leq_1 A_1 \quad v_1 \leq_2 v_2}{A_2; B \vdash e : v_2}$$

**Fig. 2.** Type assignment rules (system $\vdash^{\mathrm{Rec}}_{\wedge_2}$)

inferred for local definitions in the environment $B$, has been suggested in [7] to overcome this limitation.

$$(\textsc{LetWeak})\ \frac{A_0; B \vdash e_0 : v_0 \quad A; B, x : \mathrm{Gen}(A_0 \cup B, v_0) \vdash e' : v}{A_0 + A; B \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e : v}$$

However the system $\vdash^{\mathrm{LetWeak}}_{\wedge_2}$ which uses such a rule to type let-expressions[6] has an unpleasant feature: for some $e_0$ and $e$ such that $\mathrm{FV}(e_0) \neq \emptyset$, replacing $(\lambda x.e)e_0$ with $\mathsf{let}\ x = e_0\ \mathsf{in}\ e$ may not preserve typability, as the following example shows.

*Example 1.* We have that $\{y : (\alpha_1 \to \alpha_2) \wedge \alpha_1\}; \emptyset \vdash^{\mathrm{LetWeak}}_{\wedge_2} (\lambda x.xx)y : \alpha_2$ and so $\emptyset; \emptyset \vdash^{\mathrm{LetWeak}}_{\wedge_2} \lambda y.((\lambda x.xx)y) : ((\alpha_1 \to \alpha_2) \wedge \alpha_1) \to \alpha_2$.
Instead $\lambda y.(\mathsf{let}\ x = y\ \mathsf{in}\ xx)$ cannot be typed in $\vdash^{\mathrm{LetWeak}}_{\wedge_2}$.

[6] The rank 2 type $v_0$ may contain free type variables (which are not allowed to occur in the library environment in the judgements of $\vdash^{\mathrm{Rec}}_{\wedge_2}$). So, if we want to use rule (LetWeak) instead of (LetSugar), we have to replace, in the type inference rules of Fig. 2, every occurrence of $\mathrm{Gen}(A, v)$ by $\mathrm{Gen}(A \cup B, v)$.

This problem is due to the fact that, like the ML type system does, rule (LetWeak) associates to each let-bound identifier a *type scheme* which, in general, cannot express the principal typing of the body, $e_0$, of the local definition. To overcome this limitation we introduce the notions of *pair scheme* and *pair environment*.

**Definition 4 (Pair schemes and pair environments).** *A* pair scheme *$p$ is a formula $\forall\overrightarrow{\alpha}.\langle A, v\rangle$ where $A$ is a rank 1 environment, $v$ is a rank 2 type, and $\overrightarrow{\alpha} = \mathrm{FTV}(A) \cup \mathrm{FTV}(v)$.*
*A* pair environment *$L$ is an environment $\{x_1 : \forall\overrightarrow{\alpha}^1.\langle A_1, v_1\rangle, \ldots, x_n : \forall\overrightarrow{\alpha}^n.\langle A_n, v_n\rangle\}$ of pair scheme assumptions for identifiers.*

**The New Typing Rules.** The new typing rules for local definitions allow to associate to each locally defined identifier a pair scheme representing the principal typing of its definition. The new type system uses an additional pair environment for locally defined identifiers (let-bound and letrec-bound identifiers), i.e. it has judgements of the form $A; B; L \vdash^{\mathrm{Let,Rec}}_{\wedge_2} e : v$, where $\mathrm{FV}(e) \subseteq \mathrm{Dom}(A \cup B \cup L)$, the domains of the three environments $A$, $B$, $L$, are pairwise disjoint, and $\mathrm{Dom}(A) = \mathrm{FV}(e) - \mathrm{Dom}(B \cup L)$.

We say that *$e$ is typable in $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$ w.r.t. the library environment $B$ and the local environment $L$* if there exist a typing $A; B; L \vdash^{\mathrm{Let,Rec}}_{\wedge_2} e : v$, for some $A$ and $v$.

The type inference rules for system $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$ are in Fig. 3. There are two rules for typing a let-expression, $\mathsf{let}\, x = e_0 \,\mathsf{in}\, e$, corresponding to the two cases $x \in \mathrm{FV}(e)$ and $x \notin \mathrm{FV}(e)$. The key rule is the first one, (LetNew), which uses the local environment $L$ to store a pair scheme ($\forall\overrightarrow{\alpha}.\langle A_0, v_0\rangle$) representing the typings of the local definition $x = e_0$. Then the rule (Id$_3$) allows to associate a new typing to each use of a locally defined identifier. The new rule for typing letrec-expressions, (LetrecNew), simply relies on rule (LetNew). All the remaining rules ignore the local environment $L$ and behave as the corresponding rules of system $\vdash^{\mathrm{Rec}}_{\wedge_2}$.

The system $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$ extends both $\vdash^{\mathrm{Rec}}_{\wedge_2}$ and $\vdash^{\mathrm{LetWeak}}_{\wedge_2}$, and is such that $A; B; L \vdash^{\mathrm{Let,Rec}}_{\wedge_2} (\lambda x.e)e_0 : v$ implies $A; B; L \vdash^{\mathrm{Let,Rec}}_{\wedge_2} \mathsf{let}\, x = e_0 \,\mathsf{in}\, e : v$, for all expressions $e_0$ and $e$. For instance (considering the expression in Example 1) we have

1. $\{y : \alpha\}; \emptyset; \emptyset \vdash^{\mathrm{Let,Rec}}_{\wedge_2} y : \alpha$, by rule (Id$_1$),
2. $\{y : \alpha_1 \to \alpha_2\}; \emptyset; \{x : \forall\alpha.\langle\{y : \alpha\}, \alpha\rangle\} \vdash^{\mathrm{Let,Rec}}_{\wedge_2} x : \alpha_1 \to \alpha_2$, by rule (Id$_3$), with $\mathsf{s} = [\alpha := \alpha_1 \to \alpha_2]$,
3. $\{y : \alpha_1\}; \emptyset; \{x : \forall\alpha.\langle\{y : \alpha\}, \alpha\rangle\} \vdash^{\mathrm{Let,Rec}}_{\wedge_2} x : \alpha_1$, by rule (Id$_3$), with $\mathsf{s} = [\alpha := \alpha_1]$,
4. $\{y : (\alpha_1 \to \alpha_2) \wedge \alpha_1\}; \emptyset; \{x : \forall\alpha.\langle\{y : \alpha\}, \alpha\rangle\} \vdash^{\mathrm{Let,Rec}}_{\wedge_2} xx : \alpha_2$, from hypotheses (2) and (3), by rule (App),
5. $\{y : (\alpha_1 \to \alpha_2) \wedge \alpha_1\}; \emptyset; \emptyset \vdash^{\mathrm{Let,Rec}}_{\wedge_2} (\mathsf{let}\, x = y \,\mathsf{in}\, xx) : \alpha_2$, from hypotheses (1) and (4), by rule (LetNew),

(Id$_1$) $\{x : u\}; B; L \vdash x : u \quad$ where $x \notin \mathrm{Dom}(B \cup L)$ and $\mathrm{Dom}(B) \cap \mathrm{Dom}(L) = \emptyset$

(Id$_2$) $\emptyset; B,\ x : \forall\overrightarrow{\alpha}.v; L \vdash x : \mathbf{s}_{\{\overrightarrow{\alpha}\}}(v) \quad$ where $(\mathrm{Dom}(B) \cup \{x\}) \cap \mathrm{Dom}(L) = \emptyset$

(Id$_3$) $\mathbf{s}_{\{\overrightarrow{\alpha}\}}(A); B; L,\ x : \forall\overrightarrow{\alpha}.\langle A, v\rangle \vdash x : \mathbf{s}_{\{\overrightarrow{\alpha}\}}(v)$ where $\mathrm{Dom}(B) \cap (\mathrm{Dom}(L) \cup \{x\}) = \emptyset$

(Con) $\emptyset; B; L \vdash c : \mathbf{s}_{\{\overrightarrow{\alpha}\}}(v) \quad$ where $\mathrm{Dom}(B) \cap \mathrm{Dom}(L) = \emptyset$ and $\mathrm{Typeof}(c) = \forall\overrightarrow{\alpha}.v$

(Abs) $\dfrac{A,\ x : ui; B; L \vdash e : v}{A; B; L \vdash \lambda\, x.e : ui \to v}$ (AbsVac) $\dfrac{A; B; L \vdash e : v}{A; B; L \vdash \lambda\, x.e : u \to v}\ x \notin \mathrm{Dom}(A \cup B)$

(App) $\dfrac{A;B;L \vdash e : u_1 \wedge \cdots \wedge u_n \to v \quad (\forall i \in \{1,\ldots,n\})\ A_i;B;L \vdash e_0 : v_i \quad \mathrm{Gen}(A_i, v_i) \leq_{\forall 2,0} u_i}{A + A_1 + \cdots + A_n; B; L \vdash ee_0 : v}$

(IfSimple) $\dfrac{A; B; L \vdash e : \mathsf{bool} \quad A_1; B; L \vdash e_1 : u \quad A_2; B; L \vdash e_2 : u}{A + A_1 + A_2; B; L \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : u}$

(LetNew) $\dfrac{A_0; B; L \vdash e_0 : v_0 \quad A; B; L,\ x : \forall\overrightarrow{\alpha}.\langle A_0, v_0\rangle \vdash e : v}{A; B; L \vdash \mathsf{let}\, x = e_0\ \mathsf{in}\ e : v}\ x \in \mathrm{FV}(e)$
where $\{\overrightarrow{\alpha}\} = \mathrm{FTV}(A_0) \cup \mathrm{FTV}(v_0)$

(LetVac) $\dfrac{A_0; B; L \vdash e_0 : v_0 \quad A; B; L \vdash e : v}{A_0 + A; B; L \vdash \mathsf{let}\, x = e_0\ \mathsf{in}\ e : v}\ x \notin \mathrm{FV}(e)$

(LetrecNew) $\dfrac{A; B; L \vdash \mathsf{let}\, x_1 = e'_1\ \mathsf{in}\ \cdots \mathsf{let}\, x_n = e'_n\ \mathsf{in}\ e : v}{A; B; L \vdash \mathsf{letrec}\, \{x_1 = e_1, \ldots, x_n = e_n\}\ \mathsf{in}\ e : v}$
where, for $i \in \{1, \ldots, n\}$, $e'_i = \mathsf{rec}_i\ \{x_1 = e_1, \ldots, x_n = e_n\}$

(Rec) $\dfrac{(\forall i \in \{1, \ldots, n\})\ A_i; B; L \vdash e_i : v_i \quad (\forall j \in \{j_1, \ldots, j_m\})\ \mathrm{Gen}(A, v_j) \leq_{\forall 2,1} ui_j}{A_0; B; L \vdash \mathsf{rec}_{i_0}\ \{x_1 = e_1, \ldots, x_n = e_n\} : v_{i_0}}$
where $\{x_{j_1}, \ldots, x_{j_m}\} = \{x_1, \ldots, x_n\} \cap (\cup_{1 \leq i \leq n} \mathrm{FV}(e_i))$,
$A = A_0,\ x_{j_1} : ui_{j_1}, \ldots, x_{j_m} : ui_{j_m} = A_1 + \cdots + A_n$, and $i_0 \in \{1, \ldots, n\}$

(Sub) $\dfrac{A_1; B; L \vdash e : v_1 \quad A_2 \leq_1 A_1 \quad v_1 \leq_2 v_2}{A_2; B; L \vdash e : v_2}$

**Fig. 3.** Type assignment rules (system $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$)

6. $\emptyset; \emptyset; \emptyset \vdash^{\mathrm{Let,Rec}}_{\wedge_2} \lambda\, y.(\mathsf{let}\, x = y\, \mathsf{in}\, xx) : ((\alpha_1 \to \alpha_2) \wedge \alpha_1) \to \alpha_2$, from hypothesis (5), by rule (Abs).

The following example shows another application of rule (LetNew).

*Example 2.* The expression $e = (\ \mathsf{let}\ \ g = \lambda\, f\, x.f(fx)\ \ \mathsf{in}\ \ g(\lambda\, y.\mathsf{cons}\, y\, \mathsf{nil})\ )$ cannot be typed neither in ML nor by system $\vdash^{\mathrm{Rec}}_{\wedge_2}$. With system $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$, instead, we have

1. $\emptyset; \emptyset; \emptyset \vdash^{\mathrm{Let,Rec}}_{\wedge_2} \lambda\, f\, x.f(fx) : ((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3$,
2. $\emptyset; \emptyset; \emptyset \vdash^{\mathrm{Let,Rec}}_{\wedge_2} \lambda\, y.\mathsf{cons}\, y\, \mathsf{nil} : \alpha \to (\alpha\, \mathsf{list})$,

3. $\emptyset;\emptyset;\{g : \forall\alpha_1\alpha_2\alpha_3.\langle\emptyset, ((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3\rangle\} \vdash_{\wedge_2}^{\mathrm{Let,Rec}} g : ((\alpha' \to (\alpha'\,\mathsf{list})) \wedge ((\alpha'\,\mathsf{list}) \to (\alpha'\,\mathsf{list\,list}))) \to \alpha' \to (\alpha'\,\mathsf{list\,list})$, by rule (Id$_3$), with $\mathbf{s} = [\alpha_1 := \alpha',\ \alpha_2 := \alpha'\,\mathsf{list},\ \alpha_3 := \alpha'\,\mathsf{list\,list}]$,
4. $\emptyset;\emptyset;\{g : \forall\alpha_1\alpha_2\alpha_3.\langle\emptyset, ((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3\rangle\} \vdash_{\wedge_2}^{\mathrm{Let,Rec}} g(\lambda\, y.\mathsf{cons}\, y\,\mathsf{nil}) : \alpha' \to (\alpha'\,\mathsf{list\,list})$, from hypotheses (2) and (3), by rule (App),
5. $\emptyset;\emptyset;\emptyset \vdash_{\wedge_2}^{\mathrm{Let,Rec}} e : \alpha' \to (\alpha'\,\mathsf{list\,list})$, from hypotheses (1) and (4), by rule (LetNew).

Also the expression (1) of Section 1 can be typed with $\vdash_{\wedge_2}^{\mathrm{Let,Rec}}$.

**Principal Typings for $\vdash_{\wedge_2}^{\mathbf{Let,Rec}}$.** The following definition and theorem generalize the corresponding result of Section 5 by keeping in account the presence of the local environment $L$.

**Definition 5 (Principal typings for $\vdash_{\wedge_2}^{\mathrm{Let,Rec}}$).** *A typing $A';B;L \vdash_{\wedge_2}^{\mathrm{Let,Rec}} e : v'$ is an* instance *of a typing $A;B;L \vdash_{\wedge_2}^{\mathrm{Let,Rec}} e : v$ if there is a substitution* $\mathbf{s}$ *such that* $\mathrm{Dom}(\mathbf{s}) = \mathrm{FTV}(A) \cup \mathrm{FTV}(v)$*,* $\mathbf{s}(v) \leq_2 v'$ *and* $A' \leq_1 \mathbf{s}(A)$*.*
*A typing $A;B;L \vdash_{\wedge_2}^{\mathrm{Let,Rec}} e : v$ is a* principal typing for $e$ w.r.t. $B$ and $L$ *if any other typing of $e$ w.r.t. $B$ and $L$ is an instance of it. When $L = \emptyset$ we say that $A;B;\emptyset \vdash_{\wedge_2}^{\mathrm{Let,Rec}} : v$ is a* principal typing for $e$ w.r.t. $B$.

**Theorem 2 (Principal typing property for $\vdash_{\wedge_2}^{\mathrm{Let,Rec}}$).** *If $e$ is typable in $\vdash_{\wedge_2}^{\mathrm{Let,Rec}}$ w.r.t. $B$ and $L$, then it has a principal typing w.r.t. $B$ and $L$.*

## 6 The System $\vdash_{\wedge_2}^{\mathbf{If,Let,Rec}}$: Better Typings for if-Expressions

The rule (IfSimple) of $\vdash_{\wedge_2}^{\mathrm{Rec}}$ and $\vdash_{\wedge_2}^{\mathrm{Let,Rec}}$ seems overly restrictive: it does not allow to assign rank 2 types to the branches of if-expressions. We may think to replace that rule by the following rule

$$(\textsc{IfStrong})\ \frac{A;B;L \vdash e : \mathsf{bool} \quad A_1;B;L \vdash e_1 : v \quad A_2;B;L \vdash e_2 : v}{A + A_1 + A_2;B;L \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : v}$$

which allows to assign a rank 2 type to the branches of an if-expression. However the resulting system, $\vdash_{\wedge_2}^{\mathrm{IfStrong}}$, does not have neither the principal typing property nor the principal type property, as the following example shows.

*Example 3.* Take the expressions $e_1 = \lambda\, f\,.f3$, $e_2 = \lambda\, g\,.\mathsf{prj}_1(g(\mathsf{pair}\,1\,4))$, and $e_0 = \mathsf{if}\ z\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$. We have

– $\emptyset;\emptyset;\emptyset \vdash_{\wedge_2}^{\mathrm{Let,Rec}} e_1 : v_1$, *where* $v_1 = (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$, *and*
– $\emptyset;\emptyset;\emptyset \vdash_{\wedge_2}^{\mathrm{Let,Rec}} e_2 : v_2$, *where* $v_2 = ((\mathsf{int} \times \mathsf{int}) \to (\mathsf{int} \times \mathsf{int})) \to \mathsf{int}$,

but the expression $e_0$ cannot be typed by using $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$. Instead with system $\vdash^{\mathrm{IfStrong}}_{\wedge_2}$ we have: $\{z : \mathsf{bool}\}; \emptyset; \emptyset \vdash^{\mathrm{IfStrong}}_{\wedge_2} e_0 : v_0$, where $v_0 = ((\mathsf{int} \to \mathsf{int}) \wedge ((\mathsf{int} \times \mathsf{int}) \to (\mathsf{int} \times \mathsf{int}))) \to \mathsf{int}$ is the least upper bound of $v_1$ and $v_2$ w.r.t. $\leq_2$. Also the expressions $e'_0 e_2$ and $e''_0 e_1 e_2$, where $e'_0 = \lambda\, x.\mathsf{if}\; z\, \mathsf{then}\; e_1\; \mathsf{else}\; x$ and $e''_0 = \lambda\, x_1\, x_2.\mathsf{if}\; z\, \mathsf{then}\; x_1\; \mathsf{else}\; x_2$, are not typable by $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$ and typable by $\vdash^{\mathrm{IfStrong}}_{\wedge_2}$. For instance we have

- $\{z : \mathsf{bool}\}; \emptyset; \emptyset \vdash^{\mathrm{IfStrong}}_{\wedge_2} e'_0 : v'_0$, where $v'_0 = (\alpha \to \mathsf{int}) \to ((\mathsf{int} \to \mathsf{int}) \wedge \alpha) \to \mathsf{int}$ (and also $\{z : \mathsf{bool}\}; \emptyset; \emptyset \vdash^{\mathrm{IfStrong}}_{\wedge_2} e'_0 : v_1 \to v_2 \to v_0$, so $\{z : \mathsf{bool}\}; \emptyset; \emptyset \vdash^{\mathrm{IfStrong}}_{\wedge_2} e'_0 e_2 : v_0$), and
- $\{z : \mathsf{bool}\}; \emptyset; \emptyset \vdash^{\mathrm{IfStrong}}_{\wedge_2} e''_0 : v_1 \to v_2 \to v_0$, (so $\{x : \mathsf{bool}\}; \emptyset; \emptyset \vdash^{\mathrm{IfStrong}}_{\wedge_2} e''_0\, e_1\, e_2 : v_0$).

Note that in $\vdash^{\mathrm{IfStrong}}_{\wedge_2}$ there is no principal type for $e''_0$ (the "natural candidate" is $u = \alpha \to \alpha \to \alpha$, but there exists no substitution $\mathbf{s}$ such that $\mathbf{s}(u) \leq_2 v_1 \to v_2 \to v_0$). This problem is due to the fact that the rank 2 schemes cannot express the fact that, because of rules (Sub) and (IfStrong), a type $v$ can be assigned to an if-expression $\mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2$ if and only if it is the upper bound w.r.t. $\leq_2$ of a pair of types $v_1$ and $v_2$ that can be inferred for $e_1$ and $e_2$, respectively.

In order to preserve the principal typing property of $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$ we restrict rule (IfStrong) by limiting the use of intersection in the type assigned to the branches of an if-expression. The condition that we will use to restrict rule (IfStrong) is based on the notions of *$\wedge$-index and $\to$-index of a type* and of *index of an expression.*

**Definition 6 ($\wedge$-index and $\to$-index of a type).** *For every type $v \in \mathbf{T}_2$ the* $\wedge$-index of $v$, $\mathbf{Ind}^{\wedge}(v)$, *and the* $\to$-index of $v$, $\mathbf{Ind}^{\to}(v)$, *are the natural numbers defined in Fig. 4 (note that $\mathbf{Ind}^{\wedge}(v) \leq \mathbf{Ind}^{\to}(v)$).*

The fundamental properties of the metrics $\mathbf{Ind}^{\wedge}$ and $\mathbf{Ind}^{\to}$ are expressed by the following proposition[7].

**Proposition 1.** *1. If $\mathbf{Ind}^{\wedge}(v) = p$ and $\mathbf{Ind}^{\to}(v) = p + q$ $(p, q \geq 0)$, then $v$ is of the form $v = ui_1 \to \cdots \to ui_p \to u_1 \to \cdots \to u_q \to u$ for some $ui_1, \cdots, ui_p \in \mathbf{T}_1$, $u_1, \cdots, u_q, u \in \mathbf{T}_0$, $ui_p \notin \mathbf{T}_0$, and $u$ not of the form $u' \to u''$.*
*2. For every substitution $\mathbf{s}$, $\mathbf{Ind}^{\wedge}(v) \geq \mathbf{Ind}^{\wedge}(\mathbf{s}(v))$ and $\mathbf{Ind}^{\to}(v) \leq \mathbf{Ind}^{\to}(\mathbf{s}(v))$.*
*3. If $v \leq_2 v'$ then $\mathbf{Ind}^{\wedge}(v) \leq \mathbf{Ind}^{\wedge}(v')$ and $\mathbf{Ind}^{\to}(v) = \mathbf{Ind}^{\to}(v')$.*

**Definition 7 (Index of an expression).** *An* index environment I *is an environment $\{x_1 : i_1, \ldots, x_n : i_n\}$ of natural number (index) assumptions for identifiers. For every expression $e$ and index environment* I *such that* $\mathrm{FV}(e) \subseteq \mathrm{Dom(I)}$, *the* index of $e$ in I, $\mathbf{Ind}(e, \mathrm{I})$, *is the natural number defined by the clauses in Fig. 5.*

[7] Remember that $\wedge$ is idempotent, so, for any $u \in \mathbf{T}_0$, the type $u \wedge u$ is considered to be an element of $\mathbf{T}_0$ .

$\mathbf{Ind}^{\wedge}(u) = 0,\;$ for $u \in \mathbf{T}_0$
$\mathbf{Ind}^{\wedge}(ui \to v) = 1 + \mathbf{Ind}^{\wedge}(v),\;$ for $ui \to v \in \mathbf{T}_1 - \mathbf{T}_0$

$\mathbf{Ind}^{\to}(\mathsf{unit}) = \mathbf{Ind}^{\to}(\mathsf{bool}) = \mathbf{Ind}^{\to}(\mathsf{int}) = \mathbf{Ind}^{\to}(u_1 \times u_2) = \mathbf{Ind}^{\to}(u\,\mathsf{list}) = 0$
$\mathbf{Ind}^{\to}(ui \to v) = 1 + \mathbf{Ind}^{\to}(v)$

**Fig. 4.** The functions $\mathbf{Ind}^{\wedge}(v)$ and $\mathbf{Ind}^{\to}(v)$

The fundamental property of the metric **Ind** is given by the following proposition (the proof is by structural induction on $e$, the only non-trivial case is the computation of the index of the auxiliary expressions $\mathsf{rec}_i\,\{x_1 = e_1, \ldots, x_n = e_n\}$).

**Proposition 2.** *If* $A; B; L \vdash^{\mathrm{Let,Rec}}_{\wedge_2} e : v$ *then* $\mathbf{Ind}(e, \{x : 0 \mid x \in \mathrm{FV}(e)\}) \leq \mathbf{Ind}^{\to}(v)$.

This implies that every $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$-typable expressions $e$ has an index and, if $\mathbf{Ind}(e, \{x : 0 \mid x \in \mathrm{FV}(e)\}) = i$, then $e$ is a function that can accept at least $i$ arguments[8]. The indexes of the *open* subexpressions of a *closed* expression $e$ are computed by associating index 0 to formal parameters of functions, and the index of the corresponding definition to locally defined identifiers. For instance: $\mathbf{Ind}(y, \{y : 0\}) = 0$, $\mathbf{Ind}(\lambda y.y, \emptyset) = 1$, and $\mathbf{Ind}(g, \{g : 1\}) = 1$, so $\mathbf{Ind}(\mathsf{let}\, g = (\lambda y.y)\,\mathsf{in}\, g, \emptyset) = 1$. For an example involving mutually recursive definitions, take the auxiliary expression $e = \mathsf{rec}_1\,\{x_1 = \lambda w.x_2(w + 1),\; x_2 = \lambda yz.\mathsf{if}\,(y > z)\,\mathsf{then}\,1\,\mathsf{else}\,z * (x_1 yz)$. We have $\mathbf{Ind}(e, \emptyset) = 2$ (note that this requires two iterations of the while-loop in Fig. 5). We remark that the clauses in Fig. 5 are just a specification, and do not represent an efficient algorithm for computing the index of an expression.

**The New Typing Rules.** Let (IfNew) be the restriction of rule (IfStrong) requiring that the rank 2 type, say $v$, assigned to the branches of an if-expression, $\mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2$, must satisfy the condition

$$\mathbf{Ind}^{\wedge}(v) \leq \mathbf{Ind}(\mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2, \{x : 0 \mid x \in \mathrm{FV}(\mathsf{if}\; e\; \mathsf{then}\; e_1\; \mathsf{else}\; e_2)\}).$$

By using rule (IfNew) instead of (IfSimple), it is possible to assign types $v_0$, $v'$ and $v_0$ to the expressions $e_0$, $e_0'$ and $e_0' e_2$ of Example 3, respectively. Instead, it is not possible to assign type $v_1 \to v_2 \to v_0$ to the expression $e_0''$, so the expression $e_0''\; e_1\; e_2$ cannot be typed.

In order to compute more accurate indexes for expressions involving free library and local identifiers we introduce indexed environments, which allow to store the indexes of library and local definitions.

[8] This does not hold for non-$\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$-typable expressions (unless we restrict to expressions not containing conditionals). Take for instance $e = \lambda x\,.\mathsf{if}\; x\; \mathsf{then}\; 0\; \mathsf{else}\; \lambda y\,.y$. We have $\mathbf{Ind}(e, \emptyset) = 2$, but $e\; \mathsf{true}$ is not a function.

$$
\begin{aligned}
\mathbf{Ind}(c, \mathrm{I}) &= \mathbf{Ind}^{\rightarrow}(\mathrm{Typeof}(c)) \\
\mathbf{Ind}(x, \mathrm{I}) &= i \ \text{ if } x : i \in \mathrm{I} \\
\mathbf{Ind}(\lambda\, x.e, \mathrm{I}) &= \mathbf{Ind}(e, \mathrm{I} \cup \{x : 0\}) + 1 \\
\mathbf{Ind}(e_1\, e_2, \mathrm{I}) &= \begin{cases} 0 & \text{if } \mathbf{Ind}(e_1, \mathrm{I}) = 0 \\ \mathbf{Ind}(e_1, \mathrm{I}) - 1 & \text{if } \mathbf{Ind}(e_1, \mathrm{I}) \geq 1 \end{cases} \\
\mathbf{Ind}(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \mathrm{I}) &= \mathrm{Max}(\mathbf{Ind}(e_1, \mathrm{I}), \mathbf{Ind}(e_2, \mathrm{I})) \\
\mathbf{Ind}(\mathsf{let}\, x = e_0\ \mathsf{in}\ e, \mathrm{I}) &= \mathbf{Ind}(e, \mathrm{I} \cup \{x : \mathbf{Ind}(e_0, \mathrm{I})\}) \\
\mathbf{Ind}(\mathsf{letrec}\ \{x_1 = e_1, \ldots, x_n = e_n\}\ \mathsf{in}\ e, \mathrm{I}) &= \mathbf{Ind}(e, \mathrm{I} \cup \{x_1 : \mathbf{Ind}(e'_1, \mathrm{I}), \ldots, x_n : \mathbf{Ind}(e'_n, \mathrm{I})\}) \\
&\qquad \text{where, for } i \in \{1, \ldots, n\},\ e'_i = \mathsf{rec}_i\ \{x_1 = e_1, \ldots, x_n = e_n\} \\
\mathbf{Ind}(\mathsf{rec}_i\ \{x_1 = e_1, \ldots, x_n = e_n\}, \mathrm{I}) &= \text{begin}
\end{aligned}
$$

```
$(k_1, \ldots, k_n) := (0, \ldots, 0)$;
$(j_1, \ldots, j_n) := (\mathbf{Ind}(e_1, \mathrm{I}), \ldots, \mathbf{Ind}(e_n, \mathrm{I}))$;
while $(j_1, \ldots, j_n) \neq (k_1, \ldots, k_n)$ do begin
    $(k_1, \ldots, k_n) := (j_1, \ldots, j_n)$;
    $\mathrm{I}' := \mathrm{I} \cup \{x_1 : j_1, \ldots, x_n : j_n\}$;
    $(j_1, \ldots, j_n) :=$
    $(\mathbf{Ind}(e_1, \mathrm{I}'), \ldots, \mathbf{Ind}(e_n, \mathrm{I}'))$ end
return $j_i$
end
```

**Fig. 5.** The function $\mathbf{Ind}(e, \mathrm{I})$

**Definition 8 (Indexed environments).** *An* indexed rank 2 scheme environment *$B$ is an environment $\{x_1 : (i_1, \forall\overrightarrow{\alpha}^1.v_1), \ldots, x_n : (i_n, \forall\overrightarrow{\alpha}^n.v_n)\}$ of index and closed rank 2 scheme assumptions for identifiers such that, for every $j \in \{1, \ldots, n\}$, $i_j \leq \mathbf{Ind}^{\rightarrow}(v_j)$.*
*An* indexed pair environment *$L$ is an environment $\{x_1 : (i_1, \forall\overrightarrow{\alpha}^1.\langle A_1, v_1\rangle), \ldots, x_n : (i_n, \forall\overrightarrow{\alpha}^n.\langle A_n, v_n\rangle)\}$ of index and pair scheme assumptions for identifiers such that, for every $j \in \{1, \ldots, n\}$, $i_j \leq \mathbf{Ind}^{\rightarrow}(v_j)$.*

For every environment $T$ containing rank 1, indexed rank 2, and indexed pair assumptions define $\mathbf{Ind}(T) = \{x : 0 \mid x : ui \in T\} \cup \{x : i \mid x : (i, v) \in T\} \cup \{x : i \mid x : (i, p) \in T\}$. Let $\vdash^{\mathrm{If,Let,Rec}}_{\wedge_2}$ be the extension of $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$ which uses (in the typing judgements of Fig. 3) library indexed environments and local indexed environments, uses (see Fig. 6) rule (IfNew$'$) instead of (IfSimple) and rules (LetNew$'$), (Id$'_2$), and (Id$'_3$) instead of the corresponding rules of $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$. Rule (LetNew$'$) computes and stores the indexes of local definitions in the local environment in order to allow to compute more accurate indexes, while rules (Id$'_2$) and (Id$'_3$) simply ignore the indexes and behave as the corresponding rules of $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$. Rephrase of Proposition 2 holds also for system $\vdash^{\mathrm{If,Let,Rec}}_{\wedge_2}$.

**Proposition 3.** *If $A; B; L \vdash^{\mathrm{If,Let,Rec}}_{\wedge_2} e : v$ then $\mathbf{Ind}(e, \mathbf{Ind}(A \cup B \cup L)) \leq \mathbf{Ind}^{\rightarrow}(v)$.*

This guarantees that the indexes required in rules (IfNew$'$) and (LetNew$'$) in Fig. 6 (which involve only typable expressions) are always defined.

$$(\textsc{IfNew}')\ \frac{A; B; L \vdash e : \mathsf{bool} \quad A_1; B; L \vdash e_1 : v \quad A_2; B; L \vdash e_2 : v \quad \mathbf{Ind}^{\wedge}(v) \leq \mathbf{Ind}(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \mathbf{Ind}((A + A_1 + A_2) \cup B \cup L))}{A + A_1 + A_2; B; L \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : v}$$

$$(\textsc{LetNew}')\ \frac{A_0; B; L \vdash e_0 : v_0 \quad A; B; L,\ x : (i_0, \forall\overrightarrow{\alpha}.\langle A_0, v_0\rangle) \vdash e : v}{A; B; L \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e : v}\ x \in \mathrm{FV}(e)$$
where $i_0 = \mathbf{Ind}(e_0, \mathbf{Ind}(A_0 \cup B \cup L))$ and $\{\overrightarrow{\alpha}\} = \mathrm{FTV}(A_0) \cup \mathrm{FTV}(v_0)$

$(\textsc{Id}_2')\ \emptyset; B,\ x : (i, \forall\overrightarrow{\alpha}.v); L \vdash x : \mathbf{s}_{\{\overrightarrow{\alpha}\}}(v)$ where $(\mathrm{Dom}(B) \cup \{x\}) \cap \mathrm{Dom}(L) = \emptyset$

$(\textsc{Id}_3')\ \mathbf{s}_{\{\overrightarrow{\alpha}\}}(A); B; L, x : (i, \forall\overrightarrow{\alpha}.\langle A, v\rangle) \vdash x : \mathbf{s}\{\overrightarrow{\alpha}\}(v)$ where $\mathrm{Dom}(B) \cap (\mathrm{Dom}(L) \cup \{x\}) = \emptyset$

**Fig. 6.** Typing rules for if-expressions and indexes manipulation (system $\vdash^{\mathrm{If,Let,Rec}}_{\wedge_2}$)

**Principal Typings and Type Inference for $\vdash^{\mathbf{If,Let,Rec}}_{\wedge_2}$.** Principal typings w.r.t. a library environment $B$ and a local environment $L$ are defined as for $\vdash^{\mathrm{Let,Rec}}_{\wedge_2}$ (see Definition 5). The following result holds.

**Theorem 3 (Principal typing property for $\vdash^{\mathrm{If,Let,Rec}}_{\wedge_2}$).** *If $e$ is typable in $\vdash^{\mathrm{If,Let,Rec}}_{\wedge_2}$ w.r.t. $B$ and $L$, then it has a principal typing w.r.t. $B$ and $L$.*

System $\vdash^{\mathrm{If,Let,Rec}}_{\wedge_2}$ admits a complete inference algorithm (not included in this paper) that, for any expression $e$, library environment $B$, and local environment $L$ such that $\mathrm{Dom}(B) \cap \mathrm{Dom}(L) = \emptyset$, computes a principal typing for $e$ w.r.t. $B$ and $L$.

**Acknowledgements.** I thank Mario Coppo, Paola Giannini, and the referees of earlier versions on this paper, for suggestions to improve the presentation.

## References

1. S. Aditya and R. Nikhil. Incremental polymorphism. In *POPL'93*, LNCS 523, pages 379–405. Springer–Verlag, 1991.
2. M. Coppo and P. Giannini. Principal Types and Unification for Simple Intersection Types Systems. *Information and Computation*, 122(1):70–96, 1995.
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 and P. Giannini. A Decidable Intersection Type System based on Relevance. In *TACS'94*, LNCS 789, pages 707–725. Springer–Verlag, 1994.
5. R. Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, London, 1997.
6. T. Jim. Rank 2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, LCS, Massachusetts Institute of Technology, 1995.
7. T. Jim. What are principal typings and what are they good for? In *POPL'96*, pages 42–53. ACM, 1996.

8. A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order lambda -calculus. In *LISP and Functional Programming '94*. ACM, 1994.
9. A. J. Kfoury and J. B. Wells. Principality and Decidable Type Inference for Finite-Rank Intersection Types. In *POPL'99*. ACM, 1999.
10. D. Leivant. Polymorphic Type Inference. In *POPL'83*. ACM, 1983.
11. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. MIT press, 1997.
12. A. Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *International Symposium on Programming*, LNCS 167, pages 217–228. Springer–Verlag, 1984.
13. Z. Shao and A. W. Appel. Smartest recompilation. In *POPL'93*, pages 439–450. ACM, 1993.
14. S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Katholieke Universiteit Nijmegen, 1993.
15. H. Yokouchi. Embedding a Second-Order Type System into an Intersection Type System. *Information and Computation*, 117:206–220, 1995.