

Rank 2 Intersection Types for Local Definitions and Conditional Expressions

FERRUCCIO DAMIANI
Università di Torino

We propose a rank 2 intersection type system with new typing rules for local definitions (let-expressions and letrec-expressions) and conditional expressions (if-expressions and match-expressions). This is a further step towards the use of intersection types in “real” programming languages.

The technique for typing local definitions relies entirely on the principal typing property (i.e. it does not depend on particulars of rank 2 intersection), so it can be applied to any system with principal typings. The technique for typing conditional expressions, which is based on the idea of introducing metrics on types to “limit the use” of the intersection type constructor in the types assigned to the branches of the conditionals, is instead tailored to rank 2 intersection. However, the underlying idea might also be useful for other type systems.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language classifications—*applicative (functional) languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*polymorphism*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*functional constructs, type structure*

General Terms: Algorithms, Theory, Languages

Additional Key Words and Phrases: Type inference, principal typings, polymorphic recursion

1. INTRODUCTION

The Damas/Milner type system [Damas and Milner 1982] is the core of the type systems of modern functional programming languages, like ML [Milner et al. 1997] and Haskell. The fact that this type system is somewhat inflexible¹ has motivated the search for more expressive, but still decidable, type systems (see,

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

Partially supported by IST-2001-33477 DART and MURST Cofin’01 NAPOLI projects. The funding bodies are not responsible for any use that might be made of the results presented here.

Author’s address: Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy; email: damiani@di.unito.it.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

© 2003 ACM 0164-0925/03/0700-0401 \$5.00

for instance, Leivant [1983]; Mycroft [1984]; van Bakel [1993]; Kfoury and Wells [1994]; Damiani and Giannini [1994]; Coppo and Giannini [1995]; Yokouchi [1995]; Jim [1996]; Kfoury and Wells [1999]; Jim [2000]; van Bakel et al. [2000]). *Intersection type* systems [Coppo and Dezani-Ciancaglini 1980; Coppo 1980; Coppo et al. 1981; Barendregt et al. 1983] are particularly interesting since they generally have the *principal pair property* (a.k.a. *principal typing property*),² whose advantages with respect to the *principal type property*³ of the ML type system have been described by Jim [1996] (ML's lack of principal pairs was already pointed out by Damas [1984] in his PhD thesis). In particular, the system of *rank 2 intersection types* [Leivant 1983; van Bakel 1993; Yokouchi 1995; Jim 1996] is able to type all ML programs, has the principal pair property, decidable type inference, and complexity of type inference which is of the same order as in ML.

In this paper we propose a rank 2 intersection type system with new typing rules for (possibly mutually recursive) local definitions and conditional expressions. This is a further step towards the use of intersection types in “real” programming languages.

In order to simplify the exposition, the new typing rules are first introduced through three orthogonal extensions (for local definitions, recursive definitions, and conditional expressions) of a rank 2 intersection type system for a λ -calculus with constants, and then combined into a single system.

For the reader unfamiliar with (rank 2) intersection types, we give an early explanation of what a rank 2 intersection type is. Intersection types are obtained from *simple types* (see, for instance, Hindley [1997]) by adding the *intersection type constructor* \wedge . An expression has type $u_1 \wedge u_2$ (u_1 intersection u_2) if it has both type u_1 and type u_2 . For example, the identity function $\lambda x.x$ has both type $\text{int} \rightarrow \text{int}$ and $\text{bool} \rightarrow \text{bool}$, so it has type $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$. Rank 2 intersection types are types that may contain intersections only to the left of a single arrow. So, for instance, $((\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})) \rightarrow \text{int} \rightarrow \text{int}$ is a rank 2 intersection type,⁴ while $((\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})) \rightarrow (\text{int} \rightarrow \text{int})$ is not a rank 2 intersection type.

1.1 Rank 2 Intersection for Local Definitions

The Curry/Hindley type system [Hindley 1997] assigns the same type to all the uses of an identifier. To overcome this limitation the ML type system [Damas and Milner 1982] considers the following rule to type local definitions:

$$(\text{LETML}) \frac{U \vdash e_0 : u_0 \quad U, x : \forall \vec{\alpha}. u_0 \vdash e : u}{U \vdash \text{let } x = e_0 \text{ in } e : u}$$

where $\vec{\alpha}$ are the free type variables of u_0 that do not occur free in U .

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

³A 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 .

⁴As usual, the arrow type constructor is right associative, so $((\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})) \rightarrow \text{int} \rightarrow \text{int}$ means $((\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})) \rightarrow (\text{int} \rightarrow \text{int})$.

The formula $\forall \vec{\alpha}. u_0$ is a *type scheme*: it represents all types u'_0 that can be obtained from u_0 by substituting all the occurrences of the type variables $\vec{\alpha} = \alpha_1 \cdots \alpha_m$ ($m \geq 0$) with types $u_1 \cdots u_m$ (in this case we say that u'_0 is an *instance* of $\forall \vec{\alpha}. u_0$). The rule (LETML) allows assigning a different instance of the type scheme $\forall \vec{\alpha}. u_0$ to each occurrence of the let-bound identifier x in e .

Systems with rank 2 intersection types can type all the ML typable expressions by handling let-expressions “let $x = e_0$ in e ” as syntactic sugar for “ $(\lambda x. e)e_0$ ”. However this strategy has a drawback: it forces assigning simple types to the *uses* of the locally defined identifier x in e . For instance, the expression⁵

$$\text{let } g = \lambda f. \lambda x. f(f x) \text{ in } g (\lambda y. \lambda z. y) 3 () \text{ true} \quad (1)$$

(which safely computes 3) cannot be typed, since to type (1) it is necessary to assign the rank 2 type

$$\begin{aligned} & ((\text{int} \rightarrow (\text{bool} \rightarrow \text{int})) \wedge ((\text{bool} \rightarrow \text{int}) \rightarrow (\text{unit} \rightarrow \text{bool} \rightarrow \text{int}))) \rightarrow \text{int} \rightarrow \\ & (\text{unit} \rightarrow \text{bool} \rightarrow \text{int}) \end{aligned}$$

to the use of the let-bound identifier g in the body of the let.

In this paper we present a technique that allows assigning 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 (in Section 5.1), the “straightforward” extension of rule (LETML) to rank 2 intersection, that is typing let-expressions let $x = e_0$ in e by associating to the identifier x a *rank 2 type scheme* for e_0 (which is a formula $\forall \vec{\alpha}. v_0$, where v_0 is a rank 2 type and $\vec{\alpha}$ are *some* of the type variables of v_0) is not a good solution. In fact, when e_0 contains free identifiers, it may happen that replacing a subexpression $(\lambda x. e)e_0$ with let $x = e_0$ in e does not preserve typability. To avoid this problem we propose to associate to x a *rank 2 pair scheme* for e_0 (which is a formula $\forall \vec{\alpha}. (A_0, v_0)$, where A_0 is a type environment, v_0 is a rank 2 type, and $\vec{\alpha}$ are *all* the type variables of A_0 and v_0).

It is worth mentioning that this technique relies entirely on the principal pair property (i.e. it does not depend on particulars of rank 2 intersection), so it can be applied to any system with principal pairs.

1.2 Rank 2 Intersection for Recursive Definitions

The typing rule for recursive definitions of the ML type system [Damas and Milner 1982]

$$\text{(FIXML)} \quad \frac{U \vdash \lambda x. e : u \rightarrow u}{U \vdash \text{rec } \{x = e\} : u}$$

allows assigning to a recursive definition $\text{rec } \{x = e\}$ any simple type u that can be assigned to e by associating type u to all the occurrences of x in e .

⁵This example was introduced by Coppo [1980] to illustrate the limitations of the ML type system.

A first example of recursive definition that is not ML-typable is the expression (for a more interesting example we refer to Section 6):

$$\text{rec } \{x = xx\}, \quad (2)$$

where the application xx can not be typed by assuming a simple type for x .

To overcome this limitation Meertens [1983]⁶ and, independently, Mycroft [1984] proposed the rule for *polymorphic recursion*:

$$(\text{FIX}) \frac{U, x : \forall \vec{\alpha}. u \vdash e : \forall \vec{\alpha}. u}{U \vdash \text{rec } \{x = e\} : u}$$

Rule (FIX) allows, for instance, assigning type α to the expression (2). However type inference in presence of rule (FIX) is undecidable [Kfoury et al. 1993; Henglein 1993].

Mycroft [1984] presented a semi-algorithm for polymorphic recursion and suggested the following (decidable) restriction of rule (FIX)⁷:

$$(\text{FIX}') \frac{U \vdash \lambda x_1. \dots \lambda x_m. e' : u_1 \rightarrow \dots \rightarrow u_m \rightarrow u}{U \vdash \text{rec } \{x = e\} : u}$$

where

- (a) e' is obtained from e by renaming all the free occurrences of x with fresh names x_1, \dots, x_m ,
- (b) $\vec{\alpha}$ are the type variables of u that are not free in U , and
- (c) for all $j \in \{1, \dots, m\}$, u_j is an instance of $\forall \vec{\alpha}. u$.

To see how this rule works observe that typing the expression $\text{rec } \{x = e\}$ with rule (FIX') is the same as typing the expression

$$\text{rec } \{x = (\lambda x_1. \dots \lambda x_m. e') \underbrace{x \dots x}_m\}$$

with rule (FIX). Rule (FIX') is of intermediate power between (FIXML) and (FIX). For instance, rule (FIX') allows assigning type α to the expression (2), but does not allow typing the expression

$$(\lambda f. \text{rec } \{x = f(xx)\})(\lambda y. y),$$

that, by using (FIX), can be typed with type α .

Rule (FIX') can be further restricted by replacing the requirement (b) with

- (b1) $\vec{\alpha}$ are the type variables of u that are not free in $U, x_1 : u_1, \dots, x_n : u_n$.

The resulting rule, (FIX1), is of intermediate power between (FIXML) and (FIX'). For instance it does not allow to typing the expression (2), but it allows assigning

⁶The language considered by Meertens [1983], B, has no higher-order function or nested definitions.

⁷A semi-algorithm and a check to arrive at a (terminating) type inference algorithm for the language B was first given by Meertens [1983].

type $\alpha \rightarrow \text{int} \rightarrow \text{int}$ to the expression⁸

$$\text{rec } \{x = \lambda y. \lambda z. \text{if } (z < 3) \text{ then } 1 \text{ else } (x \ 0 \ (z - 1) + x \ \text{false} \ (z - 2))\}, \quad (3)$$

that can not be typed by using (FIXML).

By relying on the principal typing property of the system of rank 2 intersection types Jim [1996] proposed a new typing rule for recursive definitions. Jim’s rule can be understood as the extension to rank 2 intersection of rule (FIX1). As we will see (in Section 6.1) the extended rule is more elegant since the use of intersection avoids the renaming of the occurrences of x in e . The typing rule for recursive definitions considered in this paper is a slight improvement of the rule suggested by Jim [1996].

1.3 Rank 2 Intersection for Conditional Expressions

The ML type system handles an if-expression “if e_0 then e_1 else e_2 ” like the application “ifc $e_0 \ e_1 \ e_2$ ”, where ifc is a constant of type scheme $\forall \alpha. \text{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$. If we apply this strategy to a system with rank 2 intersection types we are forced to assign simple types to the if-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 propose a technique for assigning rank 2 intersection types to conditional expressions. As we will see (in Section 7.2), allowing one to assign to an if-expression if e_0 then e_1 else e_2 any rank 2 type v that can be assigned to both e_1 and e_2 destroys the principal pair (and type) property of the rank 2 intersection type system. To preserve the principal pair property, we introduce two metrics on types to “limit the use” of the intersection type constructor in the type v (assigned to the branches e_1 and e_2 of the if-expression) by “looking at the use” of the arrow type constructor in the principal pairs of e_1 and e_2 .

Although the typing rules presented in the paper are tailored to rank 2 intersection types, the idea of introducing metrics on types to limit the use of intersection in the types assigned to the branches of the conditionals expressions might also be useful for other type systems.

1.4 Organization of the Paper

Section 2 of this paper introduces a small programming language, which can be considered the kernel of functional programming languages like ML and Haskell (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 a rank 2 intersection type system, \vdash_{\wedge_2} , for the “ λ -core” of the language, which does not include local definitions, recursive definitions, and conditional

⁸This expression represents a function that, when applied to an arbitrary value e and to an integer n , ignores the argument e and returns:

— 1, if $n < 1$.

— The n -th number of Fibonacci, otherwise.

expressions. Sections 5, 6, and 7 describe three orthogonal extensions of \vdash_{\wedge_2} :

$\vdash_{\wedge_2}^{\text{Loc}}$, with a typing rule for local definitions,

$\vdash_{\wedge_2}^{\text{Rec}}$, with a typing rule for mutually recursive definitions, and

$\vdash_{\wedge_2}^{\text{Con}}$, with typing rules for conditional expressions (including definitions by pattern matching).

Section 8 presents the type system $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$, which combines the three extensions. Section 9 describes a sound and complete type inference algorithm for $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$. Related work is discussed in Section 10. Proofs of the main theorems can be found in the appendix.

An extended abstract describing a preliminary version of the system presented in this paper appeared as Damiani [2000].

A prototype implementation of the type inference algorithm for $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$ has been developed as a part of the Master thesis of Leporati [2000] (an online demonstration is available at the url <http://lambda.di.unito.it/rank2/>). The prototype is written in O’Caml [O’CAML] and the language accepted is a small subset of O’Caml itself.

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). The constructors include the unique element of type unit, the booleans, the integer numbers, and the constructors for tuples and lists. Let *bf* range over base functions (all unary) and *csⁿ* range over *n*-ary constructors. The syntax of constants (ranged over by *c*) is as follows

$$\begin{aligned} c &::= bf \mid cs^0 \mid cs^2 \mid cs^3 \mid \dots \\ bf &::= \text{not} \mid \text{and} \mid \text{or} \mid + \mid - \mid * \mid / \mid = \mid < \mid \text{fst} \mid \text{snd} \\ cs^0 &::= () \mid \text{true} \mid \text{false} \mid \dots \mid -1 \mid 0 \mid 1 \mid \dots \mid \text{nil} \\ cs^2 &::= \text{tuple}^2 \mid \text{cons} \\ cs^n &::= \text{tuple}^n \quad (n \geq 3) \end{aligned}$$

Sometimes we will use pair as short for tuple².

Expressions (ranged over by *e*) and *patterns* (ranged over by *p*) have the following syntax

$$\begin{aligned} e &::= x \mid c \mid \lambda x.e \mid e_1 e_2 \\ &\quad \mid \text{let } x = e_0 \text{ in } e \\ &\quad \mid \text{rec}_i \{x_1 = e_1, \dots, x_n = e_n\} \\ &\quad \mid \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \mid \text{match } e_0 \text{ with } \{p_1 \Rightarrow e_1 \parallel \dots \parallel p_n \Rightarrow e_n\} \\ p &::= x \mid cs^0 \mid cs^2 p_1 p_2 \mid cs^3 p_1 p_2 p_3 \mid \dots \end{aligned}$$

where *x*, *x*₁, ..., *x*_{*n*} range over identifiers. The construct *rec* allows mutually recursive expression definitions, and the construct *match* allows definitions by pattern matching.

The finite set of the free identifiers of an expression e is denoted by $FV(e)$.

Remark 2.1. (About rec -expressions). A rec -expression $\text{rec}_i \{x_1 = e_1, \dots, x_n = e_n\}$ (where $1 \leq i \leq n$) denotes the i -th expression defined by the mutually recursive definition $\{x_1 = e_1, \dots, x_n = e_n\}$. We consider this kind of expression, instead of the more usual letrec -expression $\text{letrec} \{x_1 = e_1, \dots, x_n = e_n\}$ in e , for convenience in presenting the type system: considering rec -expressions instead of letrec -expressions allows the separation of the issue of typing mutually recursive definitions from the issue of typing local definitions. This is not restrictive since, for the purpose of typechecking, the expression $\text{letrec} \{x_1 = e_1, \dots, x_n = e_n\}$ in e is equivalent to the expression

$$\text{let } x_1 = e'_1 \text{ in } \dots \text{let } x_n = e'_n \text{ in } e$$

where, for $i \in \{1, \dots, n\}$, $e'_i = \text{rec}_i \{x_1 = e_1, \dots, x_n = e_n\}$. A shortcut typing rule for letrec -expression, which avoids the introduction of the n auxiliary expressions e'_i , will be described in Remark 8.2 at the end of Section 8.

3. TYPES, SCHEMES, ENVIRONMENTS, AND \forall -CLOSURE

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.

3.1 Types and Schemes

We will be defining several classes of types. For sake of readability we will omit parens ‘(’ and ‘)’ from the grammars describing these types, but parens are of course needed and used throughout the paper.⁹

The set of *simple types* (\mathbf{T}_0), ranged over by u , is defined by the grammar:

$$u ::= \alpha \mid u_1 \rightarrow u_2 \mid d^0 \mid u_1 d^1 \mid u_1 u_2 d^2 \mid \dots$$

We have *type variables* (ranged over by α), arrow types, and a selection of *parametric datatypes* (ranged over by d^n , where $n \geq 0$ is the number of parameters). The 0-parameter datatypes (also called *ground types*) are: unit (the singleton type), bool (the set of booleans), and int (the set of integers). The other types are list types and n -ary product types ($n \geq 2$).

$$\begin{aligned} d^0 &::= \text{unit} \mid \text{bool} \mid \text{int} \\ d^1 &::= \text{list} \\ d^n &::= \times^n \quad (n \geq 2) \end{aligned}$$

For sake of readability, we will often write $u_1 \times \dots \times u_n$ instead of $u_1 \dots u_n \times^n$.

The constructor \rightarrow is right associative, for example, $u_1 \rightarrow u_2 \rightarrow u_3$ means $u_1 \rightarrow (u_2 \rightarrow u_3)$, and the constructors d^n ($n \geq 1$) bind more tightly than \rightarrow , for example, $u_1 \rightarrow u_2 \text{ list}$ means $u_1 \rightarrow (u_2 \text{ list})$.

The set of *rank 1 intersection types* (\mathbf{T}_1), ranged over by u_i , the set of *rank 2 intersection types* (\mathbf{T}_2), ranged over by v , and the set of *rank 2 intersection*

⁹Indeed, we already adopted this convention when we presented the syntax of the programming language (in Section 2).

schemes (\mathbf{T}_{v_2}), ranged over by vs , are defined as follows

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

where u ranges over the set of simple types \mathbf{T}_0 , $n \geq 1$, and $\vec{\alpha}$ 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 ϵ denote the empty sequence, $\forall \epsilon. v$ is a legal expression, syntactically different from v , so that $\mathbf{T}_2 \cap \mathbf{T}_{v_2} = \emptyset$. The constructor \wedge binds more tightly than \rightarrow , for example, $u_1 \wedge u_2 \rightarrow u_3$ means $(u_1 \wedge u_2) \rightarrow u_3$.

Free and *bound* type variables are defined as usual. For every type $t \in \mathbf{T}_1 \cup \mathbf{T}_2 \cup \mathbf{T}_{v_2}$ let $\text{FTV}(t)$ denote the set of free type variables of t . For every scheme $\forall \vec{\alpha}. v$ it is assumed that $\{\vec{\alpha}\} \subseteq \text{FTV}(v)$. Moreover, schemes are considered equal modulo renaming of bound type variables. We say that a scheme vs is *closed* if $\text{FTV}(vs) = \emptyset$.

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 assume a countable set $\mathbf{T}\mathbf{v}$ of type variables. A *substitution* \mathbf{s} is a function from type variables to simple types which is the identity on all but a finite number of type variables. The *domain* and the *set of free type variables occurring in the range* of a substitution \mathbf{s} are the sets of type variables: $\mathbf{Dom}(\mathbf{s}) = \{\alpha \mid \mathbf{s}(\alpha) \neq \alpha\}$ and $\mathbf{FTVR}(\mathbf{s}) = \cup_{\alpha \in \mathbf{Dom}(\mathbf{s})} \text{FTV}(\mathbf{s}(\alpha))$.

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

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 free variables by simple types, we have that \mathbf{T}_0 , \mathbf{T}_1 , \mathbf{T}_2 , and \mathbf{T}_{v_2} are closed under substitution.

The following definitions 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 3.1. (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 defined by the rules in Figure 1.¹⁰

Note that the relations \leq_1 and \leq_2 are reflexive and transitive.

Definition 3.2. (Instantiation relations $\leq_{v_2,0}$ and $\leq_{v_2,1}$). The instantiation relations $\leq_{v_2,0} (\subseteq \mathbf{T}_{v_2} \times \mathbf{T}_0)$ and $\leq_{v_2,1} (\subseteq \mathbf{T}_{v_2} \times \mathbf{T}_1)$ are defined as follows. For every scheme $\forall \vec{\alpha}. v \in \mathbf{T}_{v_2}$ and for every type

0. $u \in \mathbf{T}_0$, let $\forall \vec{\alpha}. v \leq_{v_2,0} u$ to mean that $u = \mathbf{s}(v)$, for some substitution \mathbf{s} ;
1. $u_1 \wedge \cdots \wedge u_n \in \mathbf{T}_1$, let $\forall \vec{\alpha}. v \leq_{v_2,1} u_1 \wedge \cdots \wedge u_n$ to mean that $\forall \vec{\alpha}. v \leq_{v_2,0} u_i$, for every $i \in \{1, \dots, n\}$.

¹⁰In rule (Ref), the condition that u is not an arrow type (i.e., not a type of the shape $u' \rightarrow u''$) is included for technical convenience only, to get a syntax directed system.

$$\begin{array}{c}
\text{(Ref)} \frac{u \in \mathbf{T}_0 \text{ and } u \text{ is not an arrow type}}{u \leq_2 u} \\
\\
(\wedge) \frac{\{u_1, \dots, u_n\} \supseteq \{u'_1, \dots, u'_m\}}{u_1 \wedge \dots \wedge u_n \leq_1 u'_1 \wedge \dots \wedge u'_m} \quad (\rightarrow) \frac{ui' \leq_1 ui \quad v \leq_2 v'}{ui \rightarrow v \leq_2 ui' \rightarrow v'}
\end{array}$$

Fig. 1. Subtyping relations \leq_1 and \leq_2 .

Example 3.3. For $vs = \forall \alpha_1 \alpha_2 \alpha_3. ((\alpha_1 \rightarrow \alpha_3) \wedge (\alpha_2 \rightarrow \alpha_3)) \rightarrow \alpha_3$, we have (remember that \wedge is idempotent):

- $vs \leq_{v2,0} (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$ (by using the substitution $\mathbf{s}_1 = [\alpha_1, \alpha_2, \alpha_3 := \text{int}]$), and
- $vs \leq_{v2,1} ((\text{int} \rightarrow \text{int}) \rightarrow \text{int}) \wedge ((\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool})$ (by \mathbf{s}_1 as above, and $\mathbf{s}_2 = [\alpha_1, \alpha_2, \alpha_3 := \text{bool}]$).

As stated by the following Lemmas, subtyping and instantiation are closed by substitution.

- LEMMA 3.4. 1. $ui \leq_1 ui'$ implies $\mathbf{s}(ui) \leq_1 \mathbf{s}(ui')$.
 2. $v \leq_2 v'$ implies $\mathbf{s}(v) \leq_2 \mathbf{s}(v')$.

- LEMMA 3.5. 0. $vs \leq_{v2,0} u$ implies $\mathbf{s}(vs) \leq_{v2,0} \mathbf{s}(u)$.
 1. $vs \leq_{v2,1} ui$ implies $\mathbf{s}(vs) \leq_{v2,1} \mathbf{s}(ui)$.

Remark 3.6. (Comparison with the relations \leq_1 , \leq_2 , and $\leq_{v2,1}$ of Jim [1996]). The relations \leq_1 , \leq_2 , and $\leq_{v2,1}$ are the same as Jim [1996]. However, we have defined the relation $\leq_{v2,1}$ in terms of the relation $\leq_{v2,0}$ (see Definition 3.2), while Jim [1996] defines $\leq_{v2,1}$ in terms of a more complex relation $\leq_{v2} (\subseteq \mathbf{T}_{v2} \times \mathbf{T}_{v2})$.

3.2 Environments

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

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

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

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

Definition 3.7. (Rank 0, rank 1, and closed rank 2 environments).

- 0. A *rank 0 environment* U is an environment $\{x_1 : u_1, \dots, x_n : u_n\}$ of simple type assumptions for identifiers.
- 1. A *rank 1 environment* A is an environment $\{x_1 : ui_1, \dots, x_n : ui_n\}$ of rank 1 type assumptions for identifiers.

2. A *closed rank 2 environment* D is an environment $\{x_1 : vs_1, \dots, x_n : vs_n\}$ of *closed rank 2 schemes assumptions* for identifiers.

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

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

and write $A_1 \leq_1 A_2$ to mean that

- $\text{Dom}(A_1) = \text{Dom}(A_2)$,¹¹ and
- for every assumption $x : ui_2 \in A_2$ there is an assumption $x : ui_1 \in A_1$ such that $ui_1 \leq_1 ui_2$.

3.3 \forall -Closure

Given a type $v \in \mathbb{T}_2$ and a set of type variables W , we write $\text{Gen}(W, v)$ for the \forall -closure of v in W , i.e. for the scheme $\forall \vec{\alpha}. v$ where $\{\vec{\alpha}\} = \text{FTV}(v) - W$. For every rank 1 environment A , let $\text{Gen}(A, v)$ be a short for $\text{Gen}(\text{FTV}(A), v)$.

The following lemma will be useful for proving soundness and completeness of the inference algorithm for system $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$ (Theorem 9.13 of Section 9.3).

LEMMA 3.8. *Let $\text{Gen}(A, v) = \forall \vec{\alpha}. v$ and $\{\vec{\alpha}\} \cap (\text{Dom}(\mathbf{s}) \cup \text{FTVR}(\mathbf{s})) = \emptyset$. Then*

$$\mathbf{s}(\text{Gen}(A, v)) \leq_{\forall_2, 1} \mathbf{s}(v) \text{ implies } \text{Gen}(\mathbf{s}(A), \mathbf{s}(v)) \leq_{\forall_2, 1} \mathbf{s}(v).$$

4. SYSTEM \vdash_{\wedge_2} : RANK 2 INTERSECTION TYPES FOR THE “ λ -CORE” OF THE LANGUAGE

In this section we introduce the \vdash_{\wedge_2} type system for the “ λ -core” of the language—we do not consider local definitions, recursive definitions, and conditional expressions. We first present the type inference rules (Section 4.1), then we discuss the role played by the two type environments occurring in the judgements of the system (Section 4.2) and state the principal pair property of the system (Section 4.3).

In Sections 5, 6, and 7, we will extend \vdash_{\wedge_2} with new typing rules for local definitions, recursive definitions, and conditional expressions.

4.1 The Type Inference Rules of \vdash_{\wedge_2}

The type inference system \vdash_{\wedge_2} has judgements of the form

$$D; A \vdash_{\wedge_2} e : v,$$

¹¹The requirement $\text{Dom}(A_1) = \text{Dom}(A_2)$ in the definition of $A_1 \leq_1 A_2$ is “unusual” (going counter to the definitions in other papers). The “usual” definition drops this requirement, thus allowing $\text{Dom}(A_1) \supseteq \text{Dom}(A_2)$. We have added such a requirement since it will simplify the presentation of the type inference system \vdash_{\wedge_2} (in Section 4).

bf	$\mathbf{Typeof}(bf)$	bf	$\mathbf{Typeof}(bf)$
not	$\forall \epsilon. \text{bool} \rightarrow \text{bool}$	fst	$\forall \alpha_1 \alpha_2. \alpha_1 \times \alpha_2 \rightarrow \alpha_1$
and, or	$\forall \epsilon. \text{bool} \times \text{bool} \rightarrow \text{bool}$	snd	$\forall \alpha_1 \alpha_2. \alpha_1 \times \alpha_2 \rightarrow \alpha_2$
+, -, *, /	$\forall \epsilon. \text{int} \times \text{int} \rightarrow \text{int}$		
=, <	$\forall \epsilon. \text{int} \times \text{int} \rightarrow \text{bool}$		

Fig. 2. Types for base functions.

cs	$\mathbf{Typeof}(cs)$	cs	$\mathbf{Typeof}(cs)$
()	$\forall \epsilon. \text{unit}$	nil	$\forall \alpha. \alpha \text{ list}$
true, false	$\forall \epsilon. \text{bool}$	cons	$\forall \alpha. \alpha \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list}$
$\dots - 1, 0, 1, \dots$	$\forall \epsilon. \text{int}$	tuple ⁿ	$\forall \alpha_1 \dots \alpha_n. \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow (\alpha_1 \times \dots \times \alpha_n)$

Fig. 3. Types for constructors.

where

- v is the rank 2 type inferred for e ,
- D is a closed rank 2 environment specifying types for the *globally defined* identifiers¹² (note that, by definition of closed rank 2 environment, $\text{FTV}(D) = \emptyset$), and
- A is a rank 1 environment containing the type assumptions for the free identifiers of e which are not in $\text{Dom}(D)$.

In any valid judgment, we have:

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

System \vdash_{\wedge_2} is meant to be used to infer *pairs* (and not just *types*).¹³ We call *undefined* the identifiers in $\text{Dom}(A)$, since their definition is not available when typechecking the expression e .¹⁴

We say that e is *typable in \vdash_{\wedge_2} with respect to the environment D* if there exist a typing $D; A \vdash_{\wedge_2} e : v$, for some A and v .

Since we distinguish between *global(ly defined)* and *undefined* identifiers, we have two different rules for identifiers (see Figure 4).

The rule for typing constants uses the function **Typeof** (tabulated in Figures 2 and 3), which specifies a closed type scheme for each constant.

¹²E.g. for the identifiers defined in the libraries available to the programmer.

¹³A way to emphasize this aspect is to use typing judgments of the shape $D \vdash_{\wedge_2} e : \langle A, v \rangle$. However, to make the comparison with other type systems easier, we adopt the more usual notation $D; A \vdash_{\wedge_2} e : v$.

¹⁴The possibility of dealing with undefined identifiers allows the support of *incremental type inference* [Aditya and Nikhil 1991] and *smartest recompilation* [Shao and Appel 1993] as outlined in Section 4 of Jim [1996].

$$\begin{array}{l}
(\text{ID}_{\text{global}}) \ D, x : \forall \vec{\alpha}. v; \emptyset \vdash x : \mathbf{s}(v) \quad \text{where } \mathbf{Dom}(\mathbf{s}) = \{\vec{\alpha}\} \\
(\text{ID}_{\text{undefined}}) \ D; \{x : u\} \vdash x : u \quad \text{where } x \notin \text{Dom}(D) \\
(\text{CONST}) \ D; \emptyset \vdash c : \mathbf{s}(v) \quad \text{where } \mathbf{Typeof}(c) = \forall \vec{\alpha}. v \text{ and } \mathbf{Dom}(\mathbf{s}) = \{\vec{\alpha}\} \\
(\text{ABS}) \ \frac{D; A, x' : ui \vdash e[x := x'] : v}{D; A \vdash \lambda x. e : ui \rightarrow v} \quad x \in \text{FV}(e) \text{ and } x' \text{ is fresh} \\
(\text{ABSVAC}) \ \frac{D; A \vdash e : v}{D; A \vdash \lambda x. e : u \rightarrow v} \quad x \notin \text{FV}(e) \text{ and } u \in \mathbf{T}_0 \\
(\text{APP}) \ \frac{D; A \vdash e : u_1 \wedge \dots \wedge u_n \rightarrow v}{D; A + A_1 + \dots + A_n \vdash e e_0 : v} \quad (\forall i \in \{1, \dots, n\}) \ D; A_i \vdash e_0 : u_i \\
(\text{SUB}) \ \frac{D; A_1 \vdash e : v_1 \quad A_2 \leq_1 A_1 \quad v_1 \leq_2 v_2}{D; A_2 \vdash e : v_2}
\end{array}$$

Fig. 4. Type assignment rules for the “ λ -core” of the language (system \vdash_{\wedge_2}).

We have two rules for typing an abstraction $\lambda x.e$, corresponding to the two cases $x \in \text{FV}(e)$ and $x \notin \text{FV}(e)$. Note that rule (ABS) renames the occurrences of the bound identifier x before typing e (the condition “ x' is fresh” is equivalent to “ x' does not occur in D and $\text{FV}(e)$ ”). This renaming is done to prevent “name-clashes” with the identifiers in the environment D (this will be further explained in Section 4.2.3).¹⁵

The rule for typing function application, (APP), allows using different typing for each expected type of the argument.

The only non-structural rule is (SUB), which allows strengthening the type assumptions in the rank 1 environment and weakening the rank 2 type inferred for the expression (for instance, without rule (SUB) it would not be possible to assign type $(\alpha_1 \wedge \alpha_2) \rightarrow \alpha_1$ to the identity function $\lambda x.x$).

4.2 About Environments and Renaming in Rule (ABS)

The two environments in the typing judgements of system \vdash_{\wedge_2} and the renaming in rule (ABS) will play an essential role in the design the new typing rules for local definitions (see Section 5), recursive definitions (see Section 6), and conditional expressions (see Section 7). In this section we try to clarify their role in system \vdash_{\wedge_2} .

4.2.1 The Environment A is “Relevant”. A notable feature of system \vdash_{\wedge_2} is a “relevant” treatment of the type environment A : for any typing $D; A \vdash_{\wedge_2} e : v$,

- $\text{Dom}(A) \subseteq \text{FV}(e)$, and
- the type environment A has been “built-up” from the type environments of the typings of the subexpression of e .

4.2.2 The Environment D is for “Defined” Identifiers. The environment D stores the closed rank 2 scheme assumptions for the globally defined identifiers.

¹⁵Remember that, in any valid judgement $D; A \vdash_{\wedge_2} e : v$, we have $\text{Dom}(D) \cap \text{Dom}(A) = \emptyset$.

4.2.3 *The Renaming in Rule (ABS) is Done for “Preserving Typability under Alpha Conversion”.* The renaming in rule (ABS) prevents “name clashes” with the identifiers occurring in environment D . It allows, for instance, typing the expression $k(\lambda k.\text{fst } k)$, with respect to the environment $D = \{k : \forall \alpha_1 \alpha_2. \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1\}$. In fact we have:

- (1) $D; \emptyset \vdash_{\wedge_2} \text{fst} : (\beta_1 \times \beta_2) \rightarrow \beta_1$, by axiom (CONST), with $\mathbf{s} = [\alpha_1 := \beta_1, \alpha_2 := \beta_2]$,
- (2) $D; \{k' : \beta_1 \times \beta_2\} \vdash_{\wedge_2} k' : \beta_1 \times \beta_2$, by axiom (ID_{undefined}),
- (3) $D; \{k' : \beta_1 \times \beta_2\} \vdash_{\wedge_2} \text{fst } k' : \beta_1$, from hypotheses (1) and (2), by rule (APP),
- (4) $D; \emptyset \vdash_{\wedge_2} \lambda k.\text{fst } k : (\beta_1 \times \beta_2) \rightarrow \beta_1$, from hypothesis (3), by rule (ABS), (note that renaming the λ -bound identifier k into the fresh identifier k' prevents a “name-clash” with the global identifier k),
- (5) $D; \emptyset \vdash_{\wedge_2} k : ((\beta_1 \times \beta_2) \rightarrow \beta_1) \rightarrow \gamma \rightarrow (\beta_1 \times \beta_2) \rightarrow \beta_1$, by axiom (ID_{global}), with $\mathbf{s} = [\alpha_1 := (\beta_1 \times \beta_2) \rightarrow \beta_1, \alpha_2 := \gamma]$, and
- (6) $D; \emptyset \vdash_{\wedge_2} k(\lambda k.\text{fst } k) : \gamma \rightarrow (\beta_1 \times \beta_2) \rightarrow \beta_1$, from hypotheses (5) and (4), by rule (APP).

Without the renaming performed by rule (ABS) in point (4), it would be not possible to type the expression $k(\lambda k.\text{fst } k)$, with respect to the environment D (that contains an assumption for the identifier k).

Let $\vdash_{\wedge_2}^{\text{Naive}}$ be the system obtained from \vdash_{\wedge_2} by replacing rule (ABS) with the following rule

$$\text{(ABSNAIVE)} \quad \frac{D; A, x : ui \vdash e : v}{D; A \vdash \lambda x.e : ui \rightarrow v} \quad x \in \text{FV}(e)$$

System $\vdash_{\wedge_2}^{\text{Naive}}$ is not able to type the expression $k(\lambda k.\text{fst } k)$, with respect to the environment D , while it is able to type the expression $k(\lambda h.\text{fst } h)$, which is equal, modulo *alpha conversion* (i.e., renaming of bound identifiers). So, in system $\vdash_{\wedge_2}^{\text{Naive}}$, typability is not preserved under alpha conversion, while it is preserved in system \vdash_{\wedge_2} (that uses rule (ABS)).

4.3 Principal Pairs for \vdash_{\wedge_2}

The type derivations of system \vdash_{\wedge_2} are closed by substitution (remember that $\text{FTV}(D) = \emptyset$).

LEMMA 4.1. (*Substitutivity property for \vdash_{\wedge_2}*). *If $D; A \vdash_{\wedge_2} e : v$, then $D; \mathbf{s}(A) \vdash_{\wedge_2} e : \mathbf{s}(v)$, for every substitution \mathbf{s} .*

PROOF. By induction on the structure of derivations, using Lemma 3.4 for rule (SUB). \square

The following definition of principal pair takes into account the presence of the environment D .

Definition 4.2. (Pairs, instantiation, and principal pairs for \vdash_{\wedge_2}).

- (1) A pair $\langle A, v \rangle$ is a *pair for e with respect to D* if $D; A \vdash_{\wedge_2} e : v$.
- (2) A pair $\langle A', v' \rangle$ is an *instance of a pair $\langle A, v \rangle$* if there is a substitution \mathbf{s} such that $\text{Dom}(\mathbf{s}) = \text{FTV}(A) \cup \text{FTV}(v)$, $\mathbf{s}(v) \leq_2 v'$ and $A' \leq_1 \mathbf{s}(A)$.

- (3) A pair for e with respect to D is *principal* if any pair for e with respect to D is an instance of it.

If $\langle A, v \rangle$ is a principal pair for e with respect to D we say that $D; A \vdash_{\wedge_2} e : v$ is a *principal typing for e with respect to D* .

The type system \vdash_{\wedge_2} has the principal pair property.

THEOREM 4.3. (*Principal pair property for \vdash_{\wedge_2}*). *If e is typable in \vdash_{\wedge_2} with respect to D , then it has a principal pair with respect to D .*

The proof of the Theorem 4.3 is similar to the proof of Theorem 8.4 of Section 8 (the principal pair property for system $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$, which is an extension of \vdash_{\wedge_2}).

5. SYSTEM $\vdash_{\wedge_2}^{\text{Loc}}$: RANK 2 INTERSECTION FOR LOCAL DEFINITIONS

In this section we extend the system \vdash_{\wedge_2} to deal with let-expressions. We first motivate and introduce the notion of *pair scheme* (Section 5.1). Then we present the new type system (Section 5.2).

5.1 From type schemes to pair schemes

The simplest way to extend system \vdash_{\wedge_2} to deal with local definitions is to treat let-expressions $\text{let } x = e_0 \text{ in } e$ as syntactic sugar for the application $(\lambda x. e)e_0$. This corresponds to introducing the following rule:

$$\text{(LETSUGAR)} \quad \frac{D; A \vdash (\lambda x. e)e_0 : v}{D; A \vdash \text{let } x = e_0 \text{ in } e : v}$$

Rule (LETSUGAR) allows typing all the let-expressions that can be typed by the ML type system, but it does not allow assigning rank 2 types to the uses of x in e . To overcome this limitation, Jim [1996] suggests typing let-expressions $\text{let } x = e_0 \text{ in } e$ by:

- inferring a *rank 2 type scheme* vs_0 for the expression e_0 , and
- associating a different *rank 2 type*, instance of vs_0 , to each use of x in e .

The following rule implements this strategy by storing in the environment D the assumptions of the rank 2 type schemes for the let-bound identifiers (note that now the environment D can contain free type variables).¹⁶

$$\text{(LETTs)} \quad \frac{D; A_0 \vdash e_0 : v_0 \quad D, x' : \text{Gen}(\text{FTV}(D) \cup \text{FTV}(A_0), v_0); A_1 \vdash e[x:=x'] : v}{D; A \vdash \text{let } x = e_0 \text{ in } e : v}$$

$$\text{where } x' \text{ is fresh and } A = \begin{cases} A_1, & \text{if } x \in \text{FV}(e) \\ A_0 + A_1, & \text{otherwise} \end{cases}$$

The system $\vdash_{\wedge_2}^{\text{T}s}$ which uses this rule to type let-expressions allows, for instance, assigning type int to the expression (1) of Section 1. However it has an

¹⁶In rule (LETTs), the condition “ x' is fresh” is equivalent to “ x' does not occur in D , $\text{FV}(e_0)$, and $\text{FV}(e)$ ”. The renaming of the bound identifier x is done just for preserving typability under alpha conversion (see the explanation for rule (ABS), in Section 4.2.3).

unpleasant feature: for some e_0 and e such that $\text{FV}(e_0) \neq \emptyset$, replacing $(\lambda x.e)e_0$ with $\text{let } x = e_0 \text{ in } e$ may not preserve typability, as the following example shows.

Example 5.1. (Weakness of rule (LETTS)). System $\vdash_{\wedge_2}^{\text{Ts}}$ is not able to type the expression $x'x'$ with respect to the environment $\{x' : \forall \epsilon.\alpha\}$, so it is not able to type the expressions $\lambda y.(\text{let } x = y \text{ in } xx)$.

Instead, as shown below, \vdash_{\wedge_2} (and so also $\vdash_{\wedge_2}^{\text{Ts}}$) allows assigning type $((\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1) \rightarrow \alpha_2$ to the expression $\lambda y.((\lambda x.xx)y)$. So $\lambda y.(\text{let } x = y \text{ in } xx)$ can be typed by using the rule (LETUGAR) introduced at the beginning of the section.

- (1) $\emptyset; \{x' : \alpha_1 \rightarrow \alpha_2\} \vdash_{\wedge_2} x' : \alpha_1 \rightarrow \alpha_2$, by rule (ID_{undefined}),
- (2) $\emptyset; \{x' : \alpha_1\} \vdash_{\wedge_2} x' : \alpha_1$, by rule (ID_{undefined}),
- (3) $\emptyset; \{x' : (\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1\} \vdash_{\wedge_2} x'x' : \alpha_2$, from hypotheses (1) and (2), by rule (APP),
- (4) $\emptyset; \emptyset \vdash_{\wedge_2} \lambda x.xx : ((\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1) \rightarrow \alpha_2$, from hypothesis (3), by rule (ABS),
- (5) $\emptyset; \{y' : \alpha_1 \rightarrow \alpha_2\} \vdash_{\wedge_2} y' : \alpha_1 \rightarrow \alpha_2$, by rule (ID_{undefined}),
- (6) $\emptyset; \{y' : \alpha_1\} \vdash_{\wedge_2} y' : \alpha_1$, by rule (ID_{undefined}),
- (7) $\emptyset; \{y' : (\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1\} \vdash_{\wedge_2} (\lambda x.xx)y' : \alpha_2$, from hypotheses (4), (5), and (6), by rule (APP),
- (8) $\emptyset; \emptyset \vdash_{\wedge_2} \lambda y.(\lambda x.xx)y : ((\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1) \rightarrow \alpha_2$, from hypothesis (7), by rule (ABS).

This unpleasant feature of system $\vdash_{\wedge_2}^{\text{Ts}}$ is due to the fact that rule (LETTS) associates to the let-bound identifier x a *type scheme* which, in general, can not express the principal typing of the expression e_0 . To overcome this limitation we introduce the notion of *pair scheme*.

Definition 5.2. (Pair schemes). A *pair scheme* ps is a formula $\forall \vec{\alpha}. \langle A, v \rangle$ where A is a rank 1 environment, v is a rank 2 type, and $\vec{\alpha} = \text{FTV}(A) \cup \text{FTV}(v)$.

5.2 The Type Inference Rules of $\vdash_{\wedge_2}^{\text{Loc}}$

We propose typing let-expressions $\text{let } x = e_0 \text{ in } e$ by:

- inferring a *pair scheme* $\forall \vec{\alpha}. \langle A_0, v_0 \rangle$ for the expression e_0 , and
- associating a different *rank 2 typing*, instance of $\forall \vec{\alpha}. \langle A_0, v_0 \rangle$, to each use of the *local(ly defined)* identifier x in e .

The rules in Figure 5 implement this strategy: rule (LETPs) stores in the environment D the pair scheme assumptions for the let-bound identifier¹⁷ and rule (ID_{local}) associates a new rank 2 *typing* to each use of the let-bound identifier.

System $\vdash_{\wedge_2}^{\text{Loc}}$ extends \vdash_{\wedge_2} with the rules (ID_{local}) and (LETPs) of Figure 5. The new system extends $\vdash_{\wedge_2}^{\text{Ts}}$ and is such that $D; A \vdash_{\wedge_2}^{\text{Loc}} (\lambda x.e)e_0 : v$ implies $D; A \vdash_{\wedge_2}^{\text{Loc}} \text{let } x = e_0 \text{ in } e : v$, for all expressions e_0 and e . For instance, it allows assigning

¹⁷In rule (LETPs), the condition “ x' is fresh” is equivalent to “ x' does not occur in D , $\text{FV}(e_0)$, and $\text{FV}(e)$ ”. Again, the renaming of the bound identifier x is done just for preserving typability under alpha conversion.

$$\begin{array}{c}
(\text{ID}_{\text{local}}) \quad D, x : \forall \vec{\alpha}. \langle A, v \rangle; \mathbf{s}(A) \vdash x : \mathbf{s}(v) \quad \text{where } \mathbf{Dom}(\mathbf{s}) = \{ \vec{\alpha} \} \\
(\text{LETPs}) \quad \frac{D; A_0 \vdash e_0 : v_0 \quad D, x' : \forall \vec{\alpha}. \langle A_0, v_0 \rangle; A_1 \vdash e[x := x'] : v}{D; A \vdash \text{let } x = e_0 \text{ in } e : v} \\
\text{where } x' \text{ is fresh, } \{ \vec{\alpha} \} = \text{FTV}(A_0) \cup \text{FTV}(v_0), \text{ and } A = \begin{cases} A_1, & \text{if } x \in \text{FV}(e) \\ A_0 + A_1, & \text{otherwise} \end{cases}
\end{array}$$

Fig. 5. Typing rules for local definitions (system $\vdash_{\wedge_2}^{\text{Loc}}$).

type $((\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1) \rightarrow \alpha_2$ to the expression $\lambda y. (\text{let } x = y \text{ in } xx)$ of Example 5.1. In fact we have:

- (1) $\emptyset; \{y' : \alpha\} \vdash_{\wedge_2}^{\text{Loc}} y' : \alpha$, by rule $(\text{ID}_{\text{undefined}})$,
- (2) $\{x' : \forall \alpha. \langle \{y' : \alpha\}, \alpha \rangle; \{y' : \alpha_1 \rightarrow \alpha_2\} \vdash_{\wedge_2}^{\text{Loc}} x' : \alpha_1 \rightarrow \alpha_2$, by rule $(\text{ID}_{\text{local}})$, with $\mathbf{s} = [\alpha := \alpha_1 \rightarrow \alpha_2]$,
- (3) $\{x' : \forall \alpha. \langle \{y' : \alpha\}, \alpha \rangle; \{y' : \alpha_1\} \vdash_{\wedge_2}^{\text{Loc}} x' : \alpha_1$, by rule $(\text{ID}_{\text{local}})$, with $\mathbf{s} = [\alpha := \alpha_1]$,
- (4) $\{x' : \forall \alpha. \langle \{y' : \alpha\}, \alpha \rangle; \{y' : (\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1\} \vdash_{\wedge_2}^{\text{Loc}} x'x' : \alpha_2$, from hypotheses (2) and (3), by rule (APP) ,
- (5) $\emptyset; \{y' : (\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1\} \vdash_{\wedge_2}^{\text{Loc}} (\text{let } x = y' \text{ in } xx) : \alpha_2$, from hypotheses (1) and (4), by rule (LETPs) ,
- (6) $\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Loc}} \lambda y. (\text{let } x = y \text{ in } xx) : ((\alpha_1 \rightarrow \alpha_2) \wedge \alpha_1) \rightarrow \alpha_2$, from hypothesis (5), by rule (ABS) .

We write $\text{Dom}_{\text{global}}(D)$ for $\{x \mid x : vs \in D \text{ and } vs \text{ is a closed rank 2 type scheme}\}$ and $\text{Dom}_{\text{local}}(D)$ for $\{x \mid x : ps \in D \text{ and } ps \text{ is a pair scheme}\}$. The expression $\text{FVR}(D)$ denotes the *set of identifiers occurring in the range of D* , which is the set $\bigcup_{x : \forall \vec{\alpha}. \langle A, v \rangle \in \text{Dom}(D)} \text{Dom}(A)$.

In any valid judgement

$$D; A \vdash_{\wedge_2}^{\text{Loc}} e : v,$$

we have:

- $\{\text{Dom}_{\text{global}}(D), \text{Dom}_{\text{local}}(D)\}$ is a partition of $\text{Dom}(D)$,
- $\text{Dom}(D) \cap \text{Dom}(A) = \emptyset$ and $\text{Dom}(D) \cap \text{FVR}(D) = \emptyset$,
- $\text{Dom}(A) = (\text{FV}(e) - \text{Dom}(D)) \cup \text{FVR}(D|_{\text{FV}(e)})$, and
- $\text{FV}(e) = \text{Dom}(D|_{\text{FV}(e)}) \cup \text{Dom}(A|_{\text{FV}(e)})$.

Rules $(\text{ID}_{\text{local}})$ and (LETPs) destroy the property

$$\text{Dom}(A) \subseteq \text{FV}(e)$$

of $\vdash_{\wedge_2}^{\text{Loc}}$. In fact, in system $\vdash_{\wedge_2}^{\text{Loc}}$, $\text{Dom}(A)$ may contain identifiers that occur in $\text{FVR}(D|_{\text{FV}(e)})$ and do not occur in $\text{FV}(e)$. Note, however, that we still have that system $\vdash_{\wedge_2}^{\text{Loc}}$ does a “relevant” treatment of the type environment A (see Section 4.2.1), since

$$\text{Dom}(A) \subseteq \text{FV}(e) \cup \text{FVR}(D|_{\text{FV}(e)}).$$

The system $\vdash_{\wedge_2}^{\text{Loc}}$ has the substitutivity property and the principal pair property. Principal pairs with respect to an environment D are defined as for \vdash_{\wedge_2}

(see Definition 4.2), just remember that, now, the environment D can contain pair schemes.

For an example of application of rule (LETPs) where the principal typing of the let-bound identifier contains an intersection, consider the expression

$$\begin{aligned} &\text{let } twice = \lambda f.\lambda x.f (f x) \text{ in} \\ &\text{let } tolist = \lambda y.\text{cons } y \text{ nil in} \\ &twice \ tolist \end{aligned}$$

that cannot be typed by the ML type system. The type system $\vdash_{\wedge_2}^{Loc}$ can assign the following principal pair schemes to the let-bound identifiers of the expression:

$$\begin{aligned} twice &: \forall \alpha \beta \gamma. \langle \emptyset, ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma)) \rightarrow \alpha \rightarrow \gamma \rangle \\ tolist &: \forall \delta. \langle \emptyset, \delta \rightarrow \delta \text{ list} \rangle \end{aligned}$$

and the principal pair $\langle \emptyset, \vartheta \rightarrow \vartheta \text{ list list} \rangle$ to the expression $twice \ tolist$.

The expression¹⁸

$$\begin{aligned} &\text{let } selfApply2 = \lambda z.(z z) z \text{ in} \\ &\text{let } apply = \lambda f.\lambda x.f x \text{ in} \\ &\text{let } reverseApply = \lambda y.\lambda g.g y \text{ in} \\ &\text{let } id = \lambda w.w \text{ in} \\ &\text{pair } (selfApply2 \ apply \ \text{not true}) \ (selfApply2 \ reverseApply \ id \ \text{false not}) \end{aligned}$$

safely computes pair false true. However, it cannot be typed by the ML type system. System $\vdash_{\wedge_2}^{Loc}$ instead allows typing it by assigning the following principal pair schemes to the let-bound identifiers of the expression:

$$\begin{aligned} selfApply2 &: \forall \alpha \beta \gamma. \langle \emptyset, ((\alpha \rightarrow \beta \rightarrow \gamma) \wedge \alpha \wedge \beta) \rightarrow \gamma \rangle \\ apply &: \forall \alpha \beta. \langle \emptyset, (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \rangle \\ reverseApply &: \forall \alpha \beta. \langle \emptyset, \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \rangle \\ id &: \forall \alpha. \langle \emptyset, \alpha \rightarrow \alpha \rangle \end{aligned}$$

Remark 5.3. (About the renaming in rules (ABS) and (LETPs)). The renaming in rules (ABS) and (LETPs) prevents “name-clashes” with the identifiers occurring in (the domain and in the range of) the environment D , thus preserving typability under alpha conversion. In Section 4.2.3 we have illustrated this point through an example with system \vdash_{\wedge_2} . We now present another example, illustrating how things become more “subtle” with system $\vdash_{\wedge_2}^{Loc}$, where the environment D can contain pair schemes.

Let $\vdash_{\wedge_2}^{LocNaive}$ be the system obtained from $\vdash_{\wedge_2}^{Loc}$ by replacing rule (ABS) with the rule (ABSNATIVE) presented at the end of Section 4.2.3. Take the expression

$$e_1 = (\lambda x.\text{let } y = x + 1 \text{ in } \lambda z.\text{pair } y \ (\text{not } z)) \ 4 \ \text{true}.$$

¹⁸This example, suggested by Joe Wells, is essentially a rearrangement of a program that Pawel Urzyczyn [1997] proved to be not typable in F_ω . Indeed, Urzyczyn proved the untypability of the expression $(\lambda x.z(x(\lambda f u.f u))(x(\lambda v g.g v)))(\lambda y.y y y)$, that cannot be typed in our system, but can be typed by using intersection at rank 3 (so it can be typed by the algorithm given in Kfoury and Wells [1999]).

We have $\emptyset; \emptyset \vdash_{\wedge_2}^{\text{LocNaive}} e_1 : \text{int} \times \text{bool}$ (indeed the expression e_1 is ML-typable), while the expression

$$e_2 = (\lambda x. \text{let } y = x + 1 \text{ in } \lambda x. \text{pair } y \text{ (not } x)) 4 \text{ true}$$

(which is equal, modulo alpha conversion) cannot be typed by $\vdash_{\wedge_2}^{\text{LocNaive}}$. Consider, in fact, the following fragment of derivation:

- (1) $\emptyset; \{x : \text{int}\} \vdash_{\wedge_2}^{\text{LocNaive}} x + 1 : \text{int}$, from axioms ($\text{ID}_{\text{undefined}}$) and (CONST), by rule (APP),
- (2) $\{y' : \forall \epsilon. \langle \{x : \text{int}\}, \text{int} \rangle\}; \{x : \text{int}\} \vdash_{\wedge_2}^{\text{LocNaive}} y' : \text{int}$, by axiom (ID_{local}),
- (3) $\{y' : \forall \epsilon. \langle \{x : \text{int}\}, \text{int} \rangle\}; \{x : \text{int}\} \vdash_{\wedge_2}^{\text{LocNaive}} \text{pair } y' : \text{bool} \rightarrow (\text{int} \times \text{bool})$, from axiom (CONST) and hypothesis (2), by rule (APP),
- (4) $\{y' : \forall \epsilon. \langle \{x : \text{int}\}, \text{int} \rangle\}; \{x : \text{bool}\} \vdash_{\wedge_2}^{\text{LocNaive}} \text{not } x : \text{bool}$, from axioms (CONST) and ($\text{ID}_{\text{undefined}}$), by rule (APP),
- (5) $\{y' : \forall \epsilon. \langle \{x : \text{int}\}, \text{int} \rangle\}; \{x : \text{int} \wedge \text{bool}\} \vdash_{\wedge_2}^{\text{LocNaive}} \text{pair } y' (\text{not } x) : \text{int} \times \text{bool}$, from hypothesis (3) and (4), by rule (APP),
- (6) $\{y' : \forall \epsilon. \langle \{x : \text{int}\}, \text{int} \rangle\}; \emptyset \vdash_{\wedge_2}^{\text{LocNaive}} \lambda x. \text{pair } y' (\text{not } x) : (\text{int} \wedge \text{bool}) \rightarrow (\text{int} \times \text{bool})$, from hypothesis (5), by rule ($\text{ABS}_{\text{NAIVE}}$).

By using system $\vdash_{\wedge_2}^{\text{LocNaive}}$ there is no way to assign to $\lambda x. \text{pair } y' (\text{not } x)$ the type $\text{bool} \rightarrow (\text{int} \times \text{bool})$, and so there is no way to type e_2 .

The problem that occurred in the example above is that, in the rank 1 environment $\{x : \text{int} \wedge \text{bool}\}$ inferred for the expression $\text{pair } y' (\text{not } x)$ in step (5), there has been a “name-clash” between the assumption $x : \text{int}$ for the identifier x occurring in the definition of the let-bound identifier y (which is bound by the leftmost λ in e_2) and the assumption $x : \text{bool}$ for the identifier x occurring in the expression $\text{pair } y' (\text{not } x)$ (which is bound by the rightmost λ in e_2).

The system $\vdash_{\wedge_2}^{\text{Loc}}$ (in which typability is preserved under alpha conversion) prevents “name-clashes” in typing the expression e_2 . In fact we have:

- (1) $\emptyset; \{x' : \text{int}\} \vdash_{\wedge_2}^{\text{Loc}} x' + 1 : \text{int}$, from axioms ($\text{ID}_{\text{undefined}}$) and (CONST), by rule (APP),
- (2) $\{y' : \forall \epsilon. \langle \{x' : \text{int}\}, \text{int} \rangle\}; \{x' : \text{int}\} \vdash_{\wedge_2}^{\text{Loc}} y' : \text{int}$, by axiom (ID_{local}),
- (3) $\{y' : \forall \epsilon. \langle \{x' : \text{int}\}, \text{int} \rangle\}; \{x' : \text{int}\} \vdash_{\wedge_2}^{\text{Loc}} \text{pair } y' : \text{bool} \rightarrow (\text{int} \times \text{bool})$, from axiom (CONST) and hypothesis (2), by rule (APP),
- (4) $\{y' : \forall \epsilon. \langle \{x' : \text{int}\}, \text{int} \rangle\}; \{x'' : \text{bool}\} \vdash_{\wedge_2}^{\text{Loc}} \text{not } x'' : \text{bool}$, by rules (CONST), ($\text{ID}_{\text{undefined}}$), and (APP),
- (5) $\{y' : \forall \epsilon. \langle \{x' : \text{int}\}, \text{int} \rangle\}; \{x' : \text{int}, x'' : \text{bool}\} \vdash_{\wedge_2}^{\text{Loc}} \text{pair } y' (\text{not } x'') : \text{int} \times \text{bool}$, from hypotheses (3) and (4), by rule (APP),
- (6) $\{y' : \forall \epsilon. \langle \{x' : \text{int}\}, \text{int} \rangle\}; \{x' : \text{int}\} \vdash_{\wedge_2}^{\text{Loc}} \lambda x. \text{pair } y' (\text{not } x) : \text{bool} \rightarrow (\text{int} \times \text{bool})$, from hypothesis (5), by rule (ABS),
- (7) $\emptyset; \{x' : \text{int}\} \vdash_{\wedge_2}^{\text{Loc}} \text{let } y = x' + 1 \text{ in } \lambda x. \text{pair } y (\text{not } x) : \text{bool} \rightarrow (\text{int} \times \text{bool})$, from hypotheses (1) and (6), by rule (LETPS), and
- (8) $\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Loc}} \lambda x. \text{let } y = x + 1 \text{ in } \lambda x. \text{pair } y (\text{not } x) : \text{int} \rightarrow \text{bool} \rightarrow (\text{int} \times \text{bool})$, from hypotheses (7), by rule (ABS),

- (9) $\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Loc}} (\lambda x. \text{let } y = x + 1 \text{ in } \lambda x. \text{pair } y (\text{not } x)) \ 4 : \text{bool} \rightarrow (\text{int} \times \text{bool})$, from hypotheses (8) and axiom (CONST), by rule (APP),
- (10) $\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Loc}} e_2 : \text{int} \times \text{bool}$, from hypotheses (9) and axiom (CONST), by rule (APP).

6. SYSTEM $\vdash_{\wedge_2}^{\text{Rec}}$: RANK 2 INTERSECTION FOR MUTUALLY RECURSIVE DEFINITIONS

In this section we extend \vdash_{\wedge_2} to deal with rec-expressions. We first present the typing rule for recursive definition of Jim [1996] (Section 6.1). Then we present the new type system (Section 6.2).

6.1 Jim’s Rule for Recursive Definitions

The rule for typing recursive definitions proposed by Jim [1996] is (modulo change of notation¹⁹):

$$\text{(REC)} \frac{D; A \vdash \lambda x. e : ui \rightarrow v \quad \text{Gen}((A, x : ui), v) \leq_{v2,1} ui}{D; A \vdash \text{rec } \{x = e\} : v}$$

This rule can be understood as the straightforward extension to rank 2 intersection of the rule (FIX1) presented in Section 1.2. A first example of recursive definition that can be typed by (REC) (or (FIX1)) and cannot be typed by (FIXML) is the expression (3) of Section 1.2. Another example (taken from Jim [1996]) is the recursive definition:

$$\text{rec } \{x = (\lambda y. \lambda z. z)(xx)\},$$

that by using (REC) (or (FIX1)) can be typed with principal type $\alpha \rightarrow \alpha$, but cannot be typed by (FIXML).

Note that, when $x \notin \text{FV}(e)$, rule (REC) requires that the rank 2 type v assigned to e must be such that $\text{Gen}((A, x : ui), v) \leq_{v2,1} ui$, for some $ui \in \mathbf{T}_0$ (see rule (ABSVAC) of Figure 4). So, for instance, the term $\text{rec } \{x = \lambda f. \lambda y. f(fy)\}$ can be typed with principal type $((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma)) \rightarrow \alpha \rightarrow \gamma$,²⁰ while the term $\text{rec } \{x = \lambda y. yy\}$ cannot be typed.²¹ This anomaly has been pointed out in Jim [1995] where a solution to the problem is described: modify rule (REC) not to require $\text{Gen}((A, x : ui), v) \leq_{v2,1} ui$ when $x \notin \text{FV}(e)$.

6.2 The Type Inference Rules of $\vdash_{\wedge_2}^{\text{Rec}}$

System $\vdash_{\wedge_2}^{\text{Rec}}$ extends \vdash_{\wedge_2} with the rule (REC2) of Figure 6.²² This extension preserves the substitutivity property and the principal pair property.

¹⁹We still give to the rule the original name used in Jim [1996], but we adapt the rule to fit in the type assignment system \vdash_{\wedge_2} .

²⁰Since $\forall \alpha \beta \gamma. ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma)) \rightarrow \alpha \rightarrow \gamma \leq_{v2,1} (\delta \rightarrow \delta) \rightarrow \delta \rightarrow \delta$.

²¹Since $\lambda y. yy$ has no rank 0 types.

²²In rule (REC2), the condition “ x'_1, \dots, x'_n are fresh” is equivalent to “ x'_1, \dots, x'_n are distinct and do not occur in $D, \text{FV}(e_1), \dots, \text{FV}(e_n)$ ”. Again, the renaming of the bound identifiers x_1, \dots, x_n is done just for preserving typability under alpha conversion.

$$\begin{array}{c}
(\forall i \in \{1, \dots, n\}) \ D; A_i \vdash e_i[x_1 := x'_1] \cdots [x_n := x'_n] : v_i \\
(\text{REC2}) \ \frac{(\forall j \in \{j_1, \dots, j_m\}) \ \text{Gen}(A_1 + \cdots + A_n, v_j) \leq_{v2,1} ui_j}{D; A \vdash \text{rec}_{i_0} \{x_1 = e_1, \dots, x_n = e_n\} : v_{i_0}} \\
\text{where } x'_1, \dots, x'_n \text{ are fresh,} \\
A_1 + \cdots + A_n = A, \ x'_{j_1} : ui_{j_1}, \dots, x'_{j_m} : ui_{j_m}, \\
\text{Dom}(A) \cap \{x'_1, \dots, x'_n\} = \emptyset, \ \text{and} \\
i_0 \in \{1, \dots, n\}
\end{array}$$

Fig. 6. Typing rule for recursive definitions (system $\vdash_{\wedge_2}^{\text{Rec}}$).

By observing that the rank 1 environment A is “relevant” (as it is pointed out in Section 4.2.1) we can note that rule (REC2) applies “for free” to mutually recursive definitions Jim’s extension of rule (REC) described at the end of Section 6.1. The constraint $\text{Gen}(A_1 + \cdots + A_n, v_j) \leq_{v2,1} ui_j$ is enforced only for those j such that $x_j \in \cup_{i \in \{1, \dots, n\}} \text{FV}(e_i)$.²³

Remark 6.1. (Comparison with Mycroft’s (FIX’) rule). There are many interesting examples of polymorphic recursion that can be typed by rule (FIX’) and cannot be typed by rule (REC2). Consider for instance the following O’Caml program²⁴

```

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

let rec collect t = match t with
  EMPTY      -> []
  | NODE(n,t1) -> n::flatmap collect (collect t1);;

```

where `'a tree` is a polymorphic tree type and `flatmap` is the standard list manipulation function of principal type $(\text{'a} \rightarrow \text{'b list}) \rightarrow \text{'a list} \rightarrow \text{'b list}$. The function `collect`, which collects all the labels in an `'a tree` and returns them in an `'a list`, is not typable by (REC2) but can be typed (with principal type $\text{'a tree} \rightarrow \text{'a list}$) by rule (FIX’).

As pointed out by Jim [1996], rule (REC2) can be easily generalized along the lines of rule (FIX’): just replace the condition

$$(\forall j \in \{j_1, \dots, j_m\}) \ \text{Gen}(A_1 + \cdots + A_n, v_j) \leq_{v2,1} ui_j$$

by

$$(\forall j \in \{j_1, \dots, j_m\}) \ \text{Gen}(A, v_j) \leq_{v2,1} ui_j.$$

²³The rule for mutually recursive definitions proposed in Jim [1996, 1995] enforces the constraint $\text{Gen}(A_1 + \cdots + A_n, v_j) \leq_{v2,1} ui_j$ for all $j \in \{1, \dots, n\}$. So rule (REC2) is a slight generalization of that rule.

²⁴This example is taken from Jim [1996], it originally comes from the ML mailing list, and has arisen in practice. Here we use O’Caml syntax since this is the syntax accepted by our prototype implementation of the type inference algorithm (see Section 1.4).

$$\begin{array}{l}
(\text{IDPAT}) \quad \{x : u\} \triangleright x : u \\
(\text{CSPAT}) \quad \frac{(\forall i \in \{1, \dots, n\}) \quad U_i \triangleright p_i : u'_i}{\emptyset, U_1, \dots, U_n \triangleright cs^n p_1 \cdots p_n : u'} \\
\text{where } \quad n \geq 0, \\
\quad \mathbf{Typeof}(cs^n) = \forall \vec{\alpha}. u_1 \rightarrow \cdots \rightarrow u_n \rightarrow u, \text{ and} \\
\quad \mathbf{s}(u_1 \rightarrow \cdots \rightarrow u_n \rightarrow u) = u'_1 \rightarrow \cdots \rightarrow u'_n \rightarrow u'
\end{array}$$

Fig. 7. Type assignment rules for patterns.

The resulting system still has principal typing and decidable type inference. For simplicity, we do not consider this extension.²⁵

7. SYSTEM $\vdash_{\wedge_2}^{\text{Con}}$: RANK 2 INTERSECTION FOR CONDITIONAL EXPRESSIONS

In this section we extend system \vdash_{\wedge_2} to deal with if-expressions and match-expressions. We first introduce the typing rules for patterns (Section 7.1) and the notions of \rightarrow -index and \wedge -index of a type (Section 7.2). Then we present the new type system (Section 7.3).

7.1 System \triangleright : Types for Patterns

The type inference system \triangleright (see Figure 7) has judgements of the form

$$U \triangleright p : u,$$

where

- u is the rank 0 type inferred for the pattern p , and
- U is a rank 0 environment containing type assumption for all and only the free identifiers occurring in the pattern (i.e., $\text{Dom}(U) = \text{FV}(p)$).

The type derivations of system \triangleright are closed by substitution and the system has the principal pair property. This follows straightforwardly from the well-known substitutivity property and principal pair property of the Curry/Hindley type system (see, for instance, Chapter 3 of Hindley [1997]).

LEMMA 7.1. (*Substitutivity property for \triangleright*). *If $U \triangleright p : u$, then $\mathbf{s}(U) \triangleright p : \mathbf{s}(u)$, for every substitution \mathbf{s} .*

Definition 7.2. (Pairs, instantiation, and principal pairs for \triangleright).

- (1) A pair $\langle U, u \rangle$ is a *pair for p* if $U \triangleright p : u$.
- (2) A pair $\langle U', u' \rangle$ is an *instance of a pair $\langle U, u \rangle$* if there is a substitution \mathbf{s} such that $\text{Dom}(\mathbf{s}) = \text{FTV}(U) \cup \text{FTV}(u)$, $\mathbf{s}(u) = u'$ and $U' = \mathbf{s}(U)$.
- (3) A pair for p is *principal* if any pair for p is an instance of it.

²⁵The corresponding inference algorithm requires a termination check and a iteration to compute the fixpoint, along the line of the inference algorithm for the system with (Fix') described by Mycroft [1984].

LEMMA 7.3. (*Principal pair property for \triangleright*). *If p is typable in \triangleright , then it has a principal pair.*

7.2 Introducing \wedge -Indexes and \rightarrow -Indexes

The simplest way to extend system \vdash_{\wedge_2} to deal with conditional expressions is to treat them as in ML. This amounts to introducing the following typing rules (note that rule (MATCHSIMPLE) uses the system \triangleright to type patterns)²⁶

$$\text{(IFSIMPLE)} \frac{D; A \vdash e : \text{bool} \quad D; A_1 \vdash e_1 : u \quad D; A_2 \vdash e_2 : u}{D; A + A_1 + A_2 \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : u}$$

$$\text{(MATCHSIMPLE)} \frac{\begin{array}{c} D; A_0 \vdash e_0 : u_0 \\ (\forall i \in \{1, \dots, n\}) U_i \triangleright p'_i : u_0 \\ D; A_i, \{y : u' \in U_i \mid y \in \text{FV}(e'_i)\} \vdash e'_i : u \end{array}}{D; A_0 + A_1 + \dots + A_n \vdash \text{match } e_0 \text{ with } \{p_1 \Rightarrow e_1, \dots, p_n \Rightarrow e_n\} : u}$$

where $(\forall i \in \{1, \dots, n\}) \{x_{i,1}, \dots, x_{i,m_i}\} = \text{FV}(p_i)$,
 $x'_{i,1}, \dots, x'_{i,m_i}$ are fresh
 $p'_i = p_i[x_{i,1} := x'_{i,1}] \cdots [x_{i,m_i} := x'_{i,m_i}]$, and
 $e'_i = e_i[x_{i,1} := x'_{i,1}] \cdots [x_{i,m_i} := x'_{i,m_i}]$

that assign to a conditional expression the unique rank 0 type inferred for its branches.

Rule (IFSIMPLE) does not allow assigning rank 2 types to the branches of if-expressions. So it does not allow typing expressions that have a “true rank 2 type” (i.e., a type in $\mathbf{T}_2 - \mathbf{T}_0$) and no rank 0 type, as illustrated by the following example.

Example 7.4. Take the expressions $e_1 = \lambda f . f(\text{pair } 1 \ 2)$ and $e_2 = \lambda g . g(\text{pair } 1 \ \text{true})$. We have

$$\begin{array}{l} \emptyset; \emptyset \vdash_{\wedge_2} e_1 : u_1, \text{ where } u_1 = ((\text{int} \times \text{int}) \rightarrow \alpha) \rightarrow \alpha, \text{ and} \\ \emptyset; \emptyset \vdash_{\wedge_2} e_2 : u_2, \text{ where } u_2 = ((\text{int} \times \text{bool}) \rightarrow \alpha) \rightarrow \alpha. \end{array}$$

Let $v = (((\text{int} \times \text{int}) \rightarrow \alpha) \wedge ((\text{int} \times \text{bool}) \rightarrow \alpha)) \rightarrow \alpha$. Since both $u_1 \leq_2 v$ and $u_2 \leq_2 v$ (v is, indeed, the least upper bound of u_1 and u_2 with respect to \leq_2), we have $\emptyset; \emptyset \vdash_{\wedge_2} e_1 : v$ and $\emptyset; \emptyset \vdash_{\wedge_2} e_2 : v$. However, the expression $e = \lambda z . \text{if } z \text{ then } e_1 \text{ else } e_2$ cannot be typed by system \vdash_{\wedge_2} extended with rule (IFSIMPLE).

We may consider replacing rule (IFSIMPLE) by the following rule

$$\text{(IFSTRONG)} \frac{D; A \vdash e_0 : \text{bool} \quad D; A_1 \vdash e_1 : v \quad D; A_2 \vdash e_2 : v}{D; A + A_1 + A_2 \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : v}$$

which allows assigning a rank 2 type to the branches of an if-expression. By using rule (IFSTRONG) we can assign the type $\text{bool} \rightarrow v$ to the expression

²⁶In rule (MATCHSIMPLE), the condition “ $x_{i,1}, \dots, x_{i,m_i}$ ($1 \leq i \leq n$) are fresh” is equivalent to “ $x_{1,1}, \dots, x_{1,m_1}, \dots, x_{n,1}, \dots, x_{n,m_n}$ are distinct and do not occur in $D, \text{FV}(e_0), \text{FV}(p_1), \dots, \text{FV}(p_n), \text{FV}(e_1), \dots, \text{FV}(e_n)$ ”. Again, the renaming of the bound identifiers $x_{i,1}, \dots, x_{i,m_i}$ ($1 \leq i \leq n$) is done just for preserving typability under alpha conversion.

$e = \lambda z. \text{if } z \text{ then } e_1 \text{ else } e_2$ introduced at the end of Example 7.4. However the resulting system, $\vdash_{\wedge_2}^{\text{Strong}}$, has neither the principal pair property nor the principal type property, as the following example shows.

Example 7.5. (Rule (IFSTRONG) destroys the principal pair property of \vdash_{\wedge_2}). Let $e' = \lambda z. \lambda x_1. \lambda x_2. \text{if } z \text{ then } x_1 \text{ else } x_2$. We have

- $\emptyset; \{z' : \text{bool}, x'_1 : \alpha, x'_2 : \alpha\} \vdash_{\wedge_2}^{\text{Strong}} \text{if } z' \text{ then } x'_1 \text{ else } x'_2 : \alpha$, and
- $\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Strong}} e' : \text{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$.

We also have

- $\emptyset; \{z' : \text{bool}, x'_1 : u_1, x'_2 : u_2\} \vdash_{\wedge_2}^{\text{Strong}} \text{if } z' \text{ then } x'_1 \text{ else } x'_2 : v$, and
- $\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Strong}} e' : \text{bool} \rightarrow u_1 \rightarrow u_2 \rightarrow v$,

where u_1, u_2 , and v are as in Example 7.4.

The fact that both $\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Strong}} e' : \text{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ and $\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Strong}} e' : \text{bool} \rightarrow u_1 \rightarrow u_2 \rightarrow v$ shows that in $\vdash_{\wedge_2}^{\text{Strong}}$ there is no principal type for e' : the “natural candidate” is $u = \text{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$, but there exists no substitution \mathbf{s} such that $\mathbf{s}(u) \leq_2 \text{bool} \rightarrow u_1 \rightarrow u_2 \rightarrow v$.

This problem is due to the fact that, by definition, substitutions map type variables into simple types (see Section 3.1).

In order to restore the principal pair property we will restrict rule (IFSTRONG) by adding some condition, involving the principal pairs of e_1 and e_2 with respect to D , that must be satisfied by the rank 2 type v .

As a first attempt, we restrict rule (IFSTRONG) with the condition:

- (C1) $v = \mathbf{s}(v')$, $v'_1 \leq_2 v'$, and $v'_2 \leq_2 v'$,
 where $\langle A'_i, v'_i \rangle$ is a principal pair for e_i with respect to D ($1 \leq i \leq 2$).

Consider the types u_1, u_2 , and v introduced in Example 7.4. The resulting rule, (IFC1),

- allows assigning type $\text{bool} \rightarrow v$ to the expression e of Example 7.4,
- allows assigning type $\text{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ to the expression e' of Example 7.5, and
- prevents assigning the type $\text{bool} \rightarrow u_1 \rightarrow u_2 \rightarrow v$ to e' . In fact, we have that (modulo renaming of type variables) the principal pairs for x'_i are of the shape $\langle \{x'_i : \beta_i\}, \beta_i \rangle$ ($1 \leq i \leq 2$) and, for any type v' , if $\beta_1 \leq_2 v'$ and $\beta_2 \leq_2 v'$, then $\beta_1 = \beta_2 = v' = \beta$, for some type variable β . So, since there is no substitution \mathbf{s} such that $v = \mathbf{s}(\beta)$, the type v does not satisfy the condition (C1).

However, rule (IFC1) has a serious drawback: it does not allow typing all the expressions that can be typed by using rule (IFSIMPLE), as shown by the following example.

Example 7.6. (Rule (IFC1) does not extend rule (IFSIMPLE)). System \vdash_{\wedge_2} extended with rule (IFSIMPLE) allows assigning type $u_0 = \text{bool} \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ to the expression $e_0 = \lambda z. \lambda x. \text{if } z \text{ then } x \text{ else } \lambda y. y$. Instead, e_0 cannot be typed

$$\begin{aligned}
& \mathbf{Ind}^\wedge(u) = 0, \text{ for } u \in \mathbf{T}_0 \\
& \mathbf{Ind}^\wedge(ui \rightarrow v) = 1 + \mathbf{Ind}^\wedge(v), \text{ for } ui \rightarrow v \in \mathbf{T}_2 - \mathbf{T}_0 \\
\\
& \mathbf{Ind}^\rightarrow(\text{unit}) = \mathbf{Ind}^\rightarrow(\text{bool}) = \mathbf{Ind}^\rightarrow(\text{int}) = \mathbf{Ind}^\rightarrow(u_1 \times u_2) = \mathbf{Ind}^\rightarrow(u \text{ list}) = 0 \\
& \mathbf{Ind}^\rightarrow(ui \rightarrow v) = 1 + \mathbf{Ind}^\rightarrow(v)
\end{aligned}$$

Fig. 8. The functions $\mathbf{Ind}^\wedge(v)$ and $\mathbf{Ind}^\rightarrow(v)$.

by system \vdash_{\wedge_2} extended with rule (IFC1). In fact (modulo renaming of type variables), the principal pairs for x' are of the shape $\langle \{x' : \beta\}, \beta \rangle$, the principal pairs for $\lambda y. y$ are of the shape $\langle \emptyset, (\gamma_1 \wedge \dots \wedge \gamma_n) \rightarrow \gamma_1 \rangle$ ($n \geq 1$), and there is no type v' such that $\beta \leq_2 v'$ and $(\gamma_1 \wedge \dots \wedge \gamma_n) \rightarrow \gamma_1 \leq_2 v'$. So the condition (C1) cannot be satisfied.²⁷

Our second attempt avoids any explicit reference to the notion of substitution. It is based on the idea of restricting rule (IFSTRONG) by “limiting the use” of the intersection type constructor in the type v by “looking at the use” of the arrow type constructor in the principal pairs of e_1 and e_2 . To do this we introduce two metrics on types: the \wedge -index and the \rightarrow -index of a type.

Definition 7.7. (\wedge -index and \rightarrow -index of a type). For every type $v \in \mathbf{T}_2$ the \wedge -index of v , $\mathbf{Ind}^\wedge(v)$, and the \rightarrow -index of v , $\mathbf{Ind}^\rightarrow(v)$, are the natural numbers defined in Figure 8 (note that $\mathbf{Ind}^\wedge(v) \leq \mathbf{Ind}^\rightarrow(v)$).²⁸

The fundamental properties of the metrics \mathbf{Ind}^\wedge and \mathbf{Ind}^\rightarrow are expressed by the following propositions.

- PROPOSITION 7.8. (1) If $\mathbf{Ind}^\wedge(v) = p$ and $\mathbf{Ind}^\rightarrow(v) = q$ ($q \geq p \geq 0$), then v is of the form $v = ui_1 \rightarrow \dots \rightarrow ui_p \rightarrow u_1 \rightarrow \dots \rightarrow u_{q-p} \rightarrow u$ for some $ui_1, \dots, ui_p \in \mathbf{T}_1$ and $u_1, \dots, u_{q-p}, u \in \mathbf{T}_0$, with $ui_p \notin \mathbf{T}_0$ and u not an arrow type.
- (2) For every substitution \mathbf{s} , $\mathbf{Ind}^\wedge(v) \geq \mathbf{Ind}^\wedge(\mathbf{s}(v))$ and $\mathbf{Ind}^\rightarrow(v) \leq \mathbf{Ind}^\rightarrow(\mathbf{s}(v))$.²⁹
- (3) If $v \leq_2 v'$ then $\mathbf{Ind}^\wedge(v) \leq \mathbf{Ind}^\wedge(v')$ and $\mathbf{Ind}^\rightarrow(v) = \mathbf{Ind}^\rightarrow(v')$.

- PROOF. (1) By induction on the definition of \mathbf{Ind}^\rightarrow and \mathbf{Ind}^\wedge (Definition 7.7).
(2) By structural induction of v , using the fact that any substitution \mathbf{s} maps type variables to rank 0 types.
(3) By induction on the definition of \leq_2 (Definition 3.1). \square

²⁷We may consider weakening the condition (C1) by applying the substitution \mathbf{s} directly on v'_1 and v'_2 and just requiring v to be an upper bound of $\mathbf{s}(v'_1)$ and $\mathbf{s}(v'_2)$. However, some restriction on the substitution \mathbf{s} must be specified, since the condition

(C0) $v = v'$, where $\mathbf{s}(v'_i) \leq_2 v'$ and $\langle A'_i, v'_i \rangle$ is a principal typing for e_i with respect to D ($1 \leq i \leq 2$) is satisfied by any type v that can be assigned to both e_1 and e_2 (see Definition 4.2), i.e., it does not restrict rule (IFSTRONG).

²⁸Remember 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 .

²⁹Remember that a substitution \mathbf{s} may transform a type $ui \in \mathbf{T}_1 - \mathbf{T}_0$ into a type $\mathbf{s}(ui) \in \mathbf{T}_0$ (take, for instance, $ui = \beta \wedge \gamma$ and $\mathbf{s} = [\beta := \alpha, \gamma := \alpha]$).

PROPOSITION 7.9. *For every expression e and environment D , if $\langle A, v \rangle$ is a principal pair for e with respect to D , then*

$$\mathbf{Ind}^\rightarrow(v) = \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v') \mid D; A' \vdash e : v'\}.$$

PROOF. By Definition 4.2, observing that (by Proposition 7.8.(2)) $\mathbf{Ind}^\rightarrow(v) \leq \mathbf{Ind}^\rightarrow(\mathbf{s}(v))$, for every substitution \mathbf{s} , and that (by Proposition 7.8.(3)) $v \leq_2 v'$ implies $\mathbf{Ind}^\rightarrow(v) = \mathbf{Ind}^\rightarrow(v')$. \square

We can now formulate the condition:

$$\text{(C2)} \quad \mathbf{Ind}^\wedge(v) \leq \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v'_1), \mathbf{Ind}^\rightarrow(v'_2)\},$$

where $\langle A'_i, v'_i \rangle$ is a principal pair for e_i with respect to D ($1 \leq i \leq 2$).

that produces a rule, (IFC2), which retains the principal pair property, is of intermediate power between rule (IFSIMPLE) and (IFSTRONG), and “successfully handles” the expressions considered in the above examples: it allows assigning type $\text{bool} \rightarrow v$ to the expression e of Example 7.4, it allows assigning type $\text{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ to the expression e' of Example 7.5, and it prevents assigning to e' the type $\text{bool} \rightarrow u_1 \rightarrow u_2 \rightarrow v$. Indeed, we can do even better, as “suggested” by the following example.

Example 7.10. (A limitation of rule (IFC2)). The expression

$$e'' = \lambda z. (\lambda x. \text{if } z \text{ then } x \text{ else } e_2) e_1,$$

where $e_1 = \lambda f. f(\text{pair } 1 \ 2)$ and $e_2 = \lambda g. g(\text{pair } 1 \ \text{true})$ are the expressions introduced in Example 7.4, is not typable by rule (IFC2). In fact, the type $v = (((\text{int} \times \text{int}) \rightarrow \alpha) \wedge ((\text{int} \times \text{bool}) \rightarrow \alpha)) \rightarrow \alpha$ considered in Example 7.4 is such that

$$\mathbf{Ind}^\wedge(v) = 1 \not\leq 0 = \mathbf{Min}\{0, 1\} = \mathbf{Min}\{\mathbf{Ind}^\rightarrow(\beta), \mathbf{Ind}^\rightarrow(((\text{int} \times \text{bool}) \rightarrow \alpha) \rightarrow \alpha)\}$$

(where $\langle \{x' : \beta\}, \beta \rangle$ is a principal pair for x' and $\langle \emptyset, ((\text{int} \times \text{bool}) \rightarrow \alpha) \rightarrow \alpha \rangle$ is a principal pair for e_2). So, condition (IFC2) cannot be satisfied.

In order to be able to assign type $\text{bool} \rightarrow v$ to the expression e'' of Example 7.10, we weaken the condition (C2) into the following condition:

$$\text{(C3)} \quad \mathbf{Ind}^\wedge(v) \leq \mathbf{Max}\{\mathbf{Ind}^\rightarrow(v'_1), \mathbf{Ind}^\rightarrow(v'_2)\},$$

where $\langle A'_i, v'_i \rangle$ is a principal pair for e_i with respect to D ($1 \leq i \leq 2$).

that produces a rule which extends rule (IFC2) while retaining the principal pair property.

Note that the condition (C3) can be rephrased to avoid any explicit reference to principal pairs. In fact, in a system with the principal pair property, we have that (by Proposition 7.9) the condition (C3) is equivalent to the condition

$$\text{(Cif)} \quad \mathbf{Ind}^\wedge(v) \leq \mathbf{Max}\{\mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_1) \mid D; A'_1 \vdash e_1 : v_1\}, \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_2) \mid D; A'_2 \vdash e_2 : v_2\}\},$$

that does not mention principal pairs.

$$\begin{array}{c}
\text{(IFI)} \quad \frac{D; A \vdash e_0 : \text{bool} \quad D; A_1 \vdash e_1 : v \quad D; A_2 \vdash e_2 : v}{D; A + A_1 + A_2 \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : v} \\
\text{Ind}^\wedge(v) \leq \mathbf{Max}\{ \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_1) \mid D; A'_1 \vdash e_1 : v_1\}, \\
\mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_2) \mid D; A'_2 \vdash e_2 : v_2\} \} \\
\text{(MATCHI)} \quad \frac{D; A_0 \vdash e_0 : u_0 \quad (\forall i \in \{1, \dots, n\}) \quad U_i \triangleright p'_i : u_0 \quad D; A_i, \{y : u \in U_i \mid y \in \text{FV}(e'_i)\} \vdash e'_i : v}{D; A_0 + A_1 + \dots + A_n \vdash \text{match } e_0 \text{ with } \{p_1 \Rightarrow e_1, \dots, p_n \Rightarrow e_n\} : v} \\
\text{where } (\forall i \in \{1, \dots, n\}) \quad \{x_{i,1}, \dots, x_{i,m_i}\} = \text{FV}(p_i), \\
x'_{i,1}, \dots, x'_{i,m_i} \text{ are fresh} \\
p'_i = p_i[x_{i,1} := x'_{i,1}] \dots [x_{i,m_i} := x'_{i,m_i}], \text{ and} \\
e'_i = e_i[x_{i,1} := x'_{i,1}] \dots [x_{i,m_i} := x'_{i,m_i}]
\end{array}$$

Fig. 9. Typing rules for conditional expressions (system $\vdash_{\wedge_2}^{\text{Con}}$).

7.3 The Type Inference Rules of $\vdash_{\wedge_2}^{\text{Con}}$

System $\vdash_{\wedge_2}^{\text{Con}}$ extends \vdash_{\wedge_2} with rules (IFI) and (MATCHI) of Figure 9.³⁰ This extension preserves the substitutivity property and the principal pair property.

Rule (IFI) is the restriction of rule (IFSTRONG) of Section 7.2 with the condition (C_{if}) presented at the end of Section 7.2. Rule (MATCHI) applies the same idea to match-expressions.

Example 7.11. (Use of rule (IFI)). By using rule (IFI) instead of (IFSIMPLE), it is possible to assign type $\text{bool} \rightarrow v$ to the expression $e = \lambda z. \text{if } z \text{ then } e_1 \text{ else } e_2$ of Example 7.4. In fact, since the type $v = ((\text{int} \times \text{int}) \rightarrow \alpha \wedge (\text{int} \times \text{bool}) \rightarrow \alpha) \rightarrow \alpha$ is such that

$$\mathbf{Ind}^\wedge(v) = 1 \leq 1 = \mathbf{Max}\{1, 1\} = \mathbf{Max}\{\mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_1) \mid \emptyset; A'_1 \vdash e_1 : v_1\}, \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_2) \mid \emptyset; A'_2 \vdash e_2 : v_2\}\},$$

condition (C_{if}) is satisfied. So, by rule (IFI), we can derive $\emptyset; \{z' : \text{bool}\} \vdash_{\wedge_2}^{\text{Con}}$ if z' then e_1 else $e_2 : v$ and then, by rule (ABS), we can obtain

$$\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Con}} e : \text{bool} \rightarrow v,$$

which is, indeed, a principal typing of e .

Also the expression $e'' = \lambda z. (\lambda x. \text{if } z \text{ then } x \text{ else } e_2) e_1$ (where $e_1 = \lambda f. f(\text{pair } 1 \ 2)$ and $e_2 = \lambda g. g(\text{pair } 1 \ \text{true})$) of Example 7.10, is typable by $\vdash_{\wedge_2}^{\text{Con}}$. In fact, we have

$$\mathbf{Ind}^\wedge(v) = 1 \leq 1 = \mathbf{Max}\{0, 1\} = \mathbf{Max}\{\mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_1) \mid \emptyset; A'_1 \vdash x' : v_1\}, \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_2) \mid \emptyset; A'_2 \vdash e_2 : v_2\}\}.$$

³⁰In rule (MATCHI), as in rule (MATCHSIMPLE), the renaming of the bound identifiers $x_{i,1}, \dots, x_{i,m_i}$ ($1 \leq i \leq n$) is done just for preserving typability under alpha conversion.

So, by rule (IFI), $\emptyset; \{z' : \text{bool}, x' : ((\text{int} \times \text{int}) \rightarrow \alpha) \rightarrow \alpha\} \vdash_{\wedge_2}^{\text{Con}} \text{if } z' \text{ then } x' \text{ else } e_2 : v$ and, by rules (ABS), (APP), and (ABS), we get

$$\emptyset; \emptyset \vdash_{\wedge_2}^{\text{Con}} e'' : \text{bool} \rightarrow v,$$

which is a principal typing of e'' .

Instead, it is not possible to assign type $\text{bool} \rightarrow u_1 \rightarrow u_2 \rightarrow v$ to the expression $e' = \lambda z. \lambda x_1. \lambda x_2. \text{if } z \text{ then } x_1 \text{ else } x_2$ of Example 7.5. In fact, since

$$\mathbf{Ind}^\wedge(v) = 1 \not\leq 0 = \mathbf{Max}\{0, 0\} = \mathbf{Max}\{\mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_1) \mid \emptyset; A'_1 \vdash x'_1 : v_1\}, \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_2) \mid \emptyset; A'_2 \vdash x'_2 : v_2\}\},$$

rule (IFI) cannot be used to assign type v to $\text{if } z' \text{ then } x'_1 \text{ else } x'_2$.

For an example of application of rule (MATCHI), consider the expression

```
( λ l . match l with
    nil           ⇒ λ f . λ x . f ( f x )
  || cons h r    ⇒ λ w . λ z . r
)
nil
(λ y . cons y nil)
5.
```

It safely computes $\text{cons } 5 \text{ nil}$ nil, however, it cannot be typed by the ML type system. Instead, the $\vdash_{\wedge_2}^{\text{Con}}$ type system can assign the two principal pairs

$$\langle \emptyset, ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma)) \rightarrow \alpha \rightarrow \gamma \rangle$$

$$\langle \{r' : \varphi \text{ list}\}, \delta \rightarrow \vartheta \rightarrow \varphi \text{ list} \rangle$$

to the first and to the second branch of the match-expression (respectively), principal pair

$$\langle \{l' : \varphi \text{ list}\}, ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \varphi \text{ list})) \rightarrow \alpha \rightarrow \varphi \text{ list} \rangle$$

to the match-expression, principal pair

$$\langle \emptyset, \varphi \text{ list} \rightarrow ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \varphi \text{ list})) \rightarrow \alpha \rightarrow \varphi \text{ list} \rangle$$

to the function $(\lambda l. \dots)$, and principal pair $\langle \emptyset, \text{int list list} \rangle$ to the whole expression.

8. SYSTEM $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$: PUTTING THE THREE EXTENSIONS TOGETHER

In this section we present a type system, that combines the three extensions of \vdash_{\wedge_2} presented in Sections 5, 6, and 7. We first show that, because of a “subtle” interaction between the rules for typing local definitions and the rules for typing conditional expressions, the combination of $\vdash_{\wedge_2}^{\text{Loc}}$ and $\vdash_{\wedge_2}^{\text{Con}}$ does not have the principal pair property. Then we present the new type system (Section 8.2) and its principal pair property (Section 8.3).

8.1 A Subtle Problem

Extending \vdash_{\wedge_2} with both the rules in Figure 5 and the rules in Figure 9 does not preserve the principal pair property, as the following example illustrates.

Example 8.1. (The interaction between (LETPs) and (IFI) destroys the principal pair property). Consider the expression

$$e''' = \lambda z. \lambda y. \lambda x_2. \text{let } x_1 = y \text{ in if } z \text{ then } x_1 \text{ else } x_2$$

In the system \vdash_{\wedge_2} extended with the rules in Figures 5 and 9 we have:

- $\{x'_1 : \forall \alpha. (\{y' : \alpha\}, \alpha)\}; \{z' : \text{bool}, y' : \alpha', x'_2 : \alpha'\} \vdash \text{if } z' \text{ then } x'_1 \text{ else } x'_2 : \alpha'$,
- $\emptyset; \{z' : \text{bool}, y' : \alpha', x'_2 : \alpha'\} \vdash \text{let } x_1 = y' \text{ in if } z' \text{ then } x_1 \text{ else } x'_2 : \alpha'$, and
- $\emptyset; \emptyset \vdash e''' : \text{bool} \rightarrow \alpha' \rightarrow \alpha' \rightarrow \alpha'$.

We also have

- $\{x'_1 : \forall \beta \gamma. (\{y' : \beta \rightarrow \gamma\}, \beta \rightarrow \gamma)\}; \{z' : \text{bool}, y' : \beta' \rightarrow \gamma', x'_2 : \delta \rightarrow \gamma'\} \vdash$
if z' then x'_1 else $x'_2 : \beta' \wedge \delta \rightarrow \gamma'$,
- $\emptyset; \{z' : \text{bool}, y' : \beta' \rightarrow \gamma', x'_2 : \delta \rightarrow \gamma'\} \vdash \text{let } x_1 = y' \text{ in if } z' \text{ then } x_1 \text{ else } x'_2 :$
 $\beta' \wedge \delta \rightarrow \gamma'$, and
- $\emptyset; \emptyset \vdash e''' : \text{bool} \rightarrow (\beta' \rightarrow \gamma') \rightarrow (\delta \rightarrow \gamma') \rightarrow \beta' \wedge \delta \rightarrow \gamma'$.

The fact both $\emptyset; \emptyset \vdash e''' : \text{bool} \rightarrow \alpha' \rightarrow \alpha' \rightarrow \alpha'$ and $\emptyset; \emptyset \vdash e''' : \text{bool} \rightarrow (\beta' \rightarrow \gamma') \rightarrow (\delta \rightarrow \gamma') \rightarrow \beta' \wedge \delta \rightarrow \gamma'$ shows (by the same argument used at the end of Example 7.5) that the system \vdash_{\wedge_2} extended with the rules in Figures 5 and 9 does not have the principal pair property.

This problem is due to the fact that, since the condition in rule (IFI) considers the \rightarrow -indexes of rank 2 types that can be assigned to the branches with respect to the *given* environment D (see Figure 9), the minimum \rightarrow -index associated to the then-branch (the let-bound identifier x_1) is computed on a subset of the types of the expression y associated to x_1 .³¹

8.2 The Type Inference Rules of $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$

The principal pair property for the combination of $\vdash_{\wedge_2}^{\text{Loc}}$ and $\vdash_{\wedge_2}^{\text{Con}}$ can be restored by forcing one to assume types of minimum \rightarrow -index for let-bound identifiers. This is exactly what rule (LETPSI) in Figure 10 does: it is the restriction of rule (LETPs) in Figure 5 requiring that, when typing a let-expression $\text{let } x = e_0 \text{ in } e$, the pair scheme $\forall \vec{\alpha}. (A_0, v_0)$ assumed for the locally defined identifier x must satisfy the condition

$$(C_{\text{let}}) \quad \mathbf{Ind}^{\rightarrow}(v_0) = \mathbf{Min}\{\mathbf{Ind}^{\rightarrow}(v'_0) \mid D; A'_0 \vdash e_0 : v'_0\}.$$

For instance, by using rule (LETPSI) instead of (LETPs), it is not possible to assign type $\text{bool} \rightarrow (\beta' \rightarrow \gamma') \rightarrow (\delta \rightarrow \gamma') \rightarrow \beta' \wedge \delta \rightarrow \gamma'$ to the term e''' of Example 8.1.

System $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$ is the extension of \vdash_{\wedge_2} with the typing rules in Figure 10, where rule (MATCHI) uses the type inference system for patterns \triangleright (see Figure 7). Since rules (REC2), (IFI), and (MATCHI) are exactly the corresponding rules of $\vdash_{\wedge_2}^{\text{Rec}}$ (in Figure 6) and of $\vdash_{\wedge_2}^{\text{Con}}$ (in Figure 9), the system $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$ extends both $\vdash_{\wedge_2}^{\text{Rec}}$ and $\vdash_{\wedge_2}^{\text{Con}}$. Moreover, in spite of the fact that rule (LETPSI) in Figure 10 is a

³¹The subset represented by the pair scheme assumed for x_1 in the given environment D .

$$\begin{array}{c}
 (\text{ID}_{\text{local}}) \quad D, x : \forall \vec{\alpha}. \langle A, v \rangle; \mathbf{s}(A) \vdash x : \mathbf{s}(v) \quad \text{where } \mathbf{Dom}(\mathbf{s}) = \{\vec{\alpha}\} \\
 \\
 \begin{array}{c}
 D; A_0 \vdash e_0 : v_0 \\
 \mathbf{Ind}^\rightarrow(v_0) = \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v'_0) \mid D; A'_0 \vdash e_0 : v'_0\} \\
 D, x' : \forall \vec{\alpha}. \langle A_0, v_0 \rangle; A_1 \vdash e[x := x'] : v \\
 (\text{LETPSI}) \quad \frac{}{D; A \vdash \text{let } x = e_0 \text{ in } e : v}
 \end{array} \\
 \text{where } x' \text{ is fresh, } \{\vec{\alpha}\} = \text{FTV}(A_0) \cup \text{FTV}(v_0), \text{ and } A = \begin{cases} A_1, & \text{if } x \in \text{FV}(e) \\ A_0 + A_1, & \text{otherwise} \end{cases} \\
 \\
 \begin{array}{c}
 (\forall i \in \{1, \dots, n\}) \quad D; A_i \vdash e_i[x_1 := x'_1] \cdots [x_n := x'_n] : v_i \\
 (\forall j \in \{j_1, \dots, j_m\}) \quad \text{Gen}(A_1 + \cdots + A_n, v_j) \leq_{\forall 2,1} u_{ij} \\
 (\text{REC2}) \quad \frac{}{D; A \vdash \text{rec}_{i_0} \{x_1 = e_1, \dots, x_n = e_n\} : v_{i_0}} \\
 \text{where } x'_1, \dots, x'_n \text{ are fresh,} \\
 A_1 + \cdots + A_n = A, x'_{j_1} : u_{ij_1}, \dots, x'_{j_m} : u_{ij_m}, \\
 \text{Dom}(A) \cap \{x'_1, \dots, x'_n\} = \emptyset, \quad \text{and} \\
 i_0 \in \{1, \dots, n\}
 \end{array} \\
 \\
 \begin{array}{c}
 D; A \vdash e_0 : \text{bool} \quad D; A_1 \vdash e_1 : v \quad D; A_2 \vdash e_2 : v \\
 \mathbf{Ind}^\wedge(v) \leq \mathbf{Max}\{ \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_1) \mid D; A'_1 \vdash e_1 : v_1\}, \\
 \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v_2) \mid D; A'_2 \vdash e_2 : v_2\} \} \\
 (\text{IF1}) \quad \frac{}{D; A + A_1 + A_2 \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : v}
 \end{array} \\
 \\
 \begin{array}{c}
 D; A_0 \vdash e_0 : u_0 \\
 (\forall i \in \{1, \dots, n\}) \quad U_i \triangleright p'_i : u_0 \\
 D; A_i, \{y : u \in U_i \mid y \in \text{FV}(e'_i)\} \vdash e'_i : v \\
 \mathbf{Ind}^\wedge(v) \leq \mathbf{Max}(\cup_{1 \leq i \leq n} \mathbf{Min}\{ \mathbf{Ind}^\rightarrow(v_i) \mid \\
 D; A'_0 \vdash e_0 : u'_0, \\
 U'_i \triangleright p'_i : u'_0, \text{ and} \\
 D; A'_i, \{y : u \in U'_i \mid y \in \text{FV}(e'_i)\} \vdash e'_i : v_i \}) \\
 (\text{MATCH1}) \quad \frac{}{D; A_0 + A_1 + \cdots + A_n \vdash \text{match } e_0 \text{ with } \{p_1 \Rightarrow e_1, \dots, p_n \Rightarrow e_n\} : v} \\
 \text{where } (\forall i \in \{1, \dots, n\}) \quad \{x_{i,1}, \dots, x_{i,m_i}\} = \text{FV}(p_i), \\
 x'_{i,1}, \dots, x'_{i,m_i} \text{ are fresh} \\
 p'_i = p_i[x_{i,1} := x'_{i,1}] \cdots [x_{i,m_i} := x'_{i,m_i}], \text{ and} \\
 e'_i = e_i[x_{i,1} := x'_{i,1}] \cdots [x_{i,m_i} := x'_{i,m_i}]
 \end{array}
 \end{array}$$

Fig. 10. Typing rules for local definitions, recursive definitions, and conditional expressions (system $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$).

restriction of rule (LETPs) in Figure 5,³² we have that $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$ also extends $\vdash_{\wedge_2}^{\text{Loc}}$. This follows from the fact that, by Proposition 7.9, any principal typing $D; A_0 \vdash_{\wedge_2}^{\text{Loc,Rec,Con}} e_0 : v_0$ satisfies the condition (\mathbf{C}_{let}).

Remark 8.2. (A shortcut typing rule for letrec-expression). As explained in Remark 2.1 at the end of Section 2, $\text{rec}_i \{x_1 = e_1, \dots, x_n = e_n\}$ (where $1 \leq i \leq n$), have been considered instead of the more usual letrec-expressions, $\text{letrec} \{x_1 = e_1, \dots, x_n = e_n\} \text{ in } e$, since they allow separating the

³²Rule (ID_{local}), instead, is exactly the rule in Figure 5.

$$\begin{array}{c}
(\forall i \in \{1, \dots, n\}) \quad D; A_i \vdash e_i[x_1 := x'_1] \cdots [x_n := x'_n] : v_i \\
(\forall j \in \{j_1, \dots, j_m\}) \quad \text{Gen}(A_1 + \cdots + A_n, v_j) \leq_{\forall 2,1} ui_j \\
\text{(LETRECPSI)} \frac{D, x_1 : \forall \alpha^{\vec{1}}. \langle A_0, v_1 \rangle, \dots, x_n : \forall \alpha^{\vec{n}}. \langle A_0, v_n \rangle; A' \vdash e[x_1 := x'_1] \cdots [x_n := x'_n] : v}{D; A \vdash \text{letrec } \{x_1 = e_1, \dots, x_n = e_n\} \text{ in } e : v}
\end{array}$$

where x'_1, \dots, x'_n are fresh,

$$\begin{array}{l}
A_1 + \cdots + A_n = A_0, x'_{j_1} : ui_{j_1}, \dots, x'_{j_m} : ui_{j_m}, \\
\text{Dom}(A_0) \cap \{x'_1, \dots, x'_n\} = \emptyset, \\
(\forall i \in \{1, \dots, n\}) \quad \{\alpha^i\} = \text{FTV}(A_0) \cup \text{FTV}(v_i), \\
\{x'_{h_1}, \dots, x'_{h_q}\} = \{x'_1, \dots, x'_n\} - \text{Dom}(A'), \quad \text{and} \\
A = A' + A_{h_1} + \cdots + A_{h_q} \\
(\forall i \in \{1, \dots, n\}) \quad (\mathbf{Ind}^{\rightarrow}(v_1), \dots, \mathbf{Ind}^{\rightarrow}(v_n)) = \\
\quad \mathbf{Min}\{ (\mathbf{Ind}^{\rightarrow}(v'_1), \dots, \mathbf{Ind}^{\rightarrow}(v'_n)) \mid \\
\quad \quad D; A'_1 \vdash e_1[x_1 := x'_1] \cdots [x_n := x'_n] : v'_1, \\
\quad \quad \vdots \\
\quad \quad D; A'_n \vdash e_n[x_1 := x'_1] \cdots [x_n := x'_n] : v'_n, \\
\quad \quad A'_1 + \cdots + A'_n = A'_0, x'_{j_1} : ui'_{j_1}, \dots, x'_{j_m} : ui'_{j_m}, \\
\quad \quad (\forall j \in \{j_1, \dots, j_m\}) \quad \text{Gen}(A'_1 + \cdots + A'_n, v'_j) \leq_{\forall 2,1} ui'_j \}
\end{array}$$

Fig. 11. Shortcut typing rule for letrec-expressions (system $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$).

issue of typing mutually recursive definitions from the issue of typing local definitions.

Even though, for the purpose of typechecking, the expression $\text{letrec } \{x_1 = e_1, \dots, x_n = e_n\} \text{ in } e$ is equivalent to the expression

$$\text{let } x_1 = e'_1 \text{ in } \cdots \text{let } x_n = e'_n \text{ in } e$$

where $e'_i = \text{rec}_i \{x_1 = e_1, \dots, x_n = e_n\}$ ($i \in \{1, \dots, n\}$), it is better, for the design of practical inference algorithm, to introduce a shortcut typing rule for letrec-expressions, which avoids the introduction of the n auxiliary rec-expressions e'_i and does not require typing each expression e_i n times.

The rule (LETRECPSI) in Figure 11 is a suitable shortcut typing rule for letrec-expression in system $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$.³³ However, to keep the proofs simpler, we have not considered rule (LETRECPSI) in the type inference algorithm presented in Section 9.

8.3 Principal Pairs for $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$

The system $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$ has the substitutivity property.

LEMMA 8.3. (*Substitutivity property for $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$*). *If $D; A \vdash_{\wedge_2}^{\text{Loc, Rec, Con}} e : v$, then $D; \mathbf{s}(A) \vdash_{\wedge_2}^{\text{Loc, Rec, Con}} e : \mathbf{s}(v)$, for every substitution \mathbf{s} .*

PROOF By induction on the structure of derivations, using Proposition 7.8.(2) to deal with rule (IfI), and Lemma 7.1 and Proposition 7.8.(2) to deal with rule (MATCHI). \square

³³In rule (LETRECPSI), the minimum of the set of tuples of indexes is taken with respect to pointwise ordering. The condition “ x'_1, \dots, x'_n are fresh” is equivalent to “ x'_1, \dots, x'_n are distinct and do not occur in $D, \text{FV}(e_1), \dots, \text{FV}(e_n)$, and $\text{FV}(e)$ ”. Again, the renaming of the bound identifiers x_1, \dots, x_n is done just for preserving typability under alpha conversion.

Principal pairs for $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$ are defined as for \vdash_{\wedge_2} (see Definition 4.2). The following result holds.

THEOREM 8.4. (*Principal pair property for $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$*). *If e is typable in $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$ with respect to D , then it has a principal pair with respect to D .*

PROOF. The result follows from Theorem 9.13 (soundness and completeness of the inference algorithm for $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$) of Section 9. \square

9. A SOUND AND COMPLETE INFERENCE ALGORITHM FOR $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$

In this section we present an algorithm that, for any expression e and environment D ,

- reports a failure, if e can not be typed by $\vdash_{\wedge_2}^{\text{Loc,Rec,Con}}$ with respect to D , and
- computes a principal pair for e with respect to D , otherwise.

Until now, to simplify the presentation, we have considered \wedge to be associative, commutative, and idempotent. In this section we do not rely on this syntactic convention.

9.1 An Extension of Jim's Subtype Satisfaction Algorithm

Following Jim [1996, 1995] we say that a $\leq_{v_2,1}$ -satisfaction problem is a formula $\exists \vec{\alpha}.P$, where

- $\vec{\alpha}$ is a (possibly empty) sequence of type variables occurring free in P , and
- P is a set in which every element is
 - an equality between \mathbf{T}_0 types, or
 - an inequality between a \mathbf{T}_2 type and a \mathbf{T}_1 type, or
 - an inequality between a \mathbf{T}_{v_2} type and a \mathbf{T}_1 type.

A substitution \mathbf{s} is a *solution* to $\exists \vec{\alpha}.P$ if there exists a substitution \mathbf{s}' such that

- $\mathbf{s}(\alpha) = \mathbf{s}'(\alpha)$ for all $\alpha \notin \vec{\alpha}$,
- $\mathbf{s}'(vs) \leq_{v_2,1} \mathbf{s}'(ui)$ for every inequality $(vs \leq ui) \in P$, and
- $\mathbf{s}'(u_1) = \mathbf{s}'(u_2)$ for every equality $(u_1 = u_2) \in P$.

Definition 9.1. (Most general solutions of a $\leq_{v_2,1}$ -satisfaction problem).

- (1) A substitution \mathbf{s} is a *most general solution (mgs)* of a $\leq_{v_2,1}$ -satisfaction problem $\exists \vec{\alpha}.P$ if it satisfies the following conditions:³⁴
 - \mathbf{s} is a solution of $\exists \vec{\alpha}.P$,
 - $\mathbf{s} \leq \mathbf{s}'$, for all solutions \mathbf{s}' of $\exists \vec{\alpha}.P$,
 - \mathbf{s} is idempotent, and
 - $\mathbf{Dom}(\mathbf{s}) \subseteq \text{FTV}(\exists \vec{\alpha}.P)$.
- (2) We write $\mathbf{MGS}(\exists \vec{\alpha}.P)$ for the (possibly empty) set of the most general solutions of $\exists \vec{\alpha}.P$.

³⁴The last two conditions are included for technical convenience only. Indeed they can be eliminated: a $\leq_{v_2,1}$ -satisfaction problem has a solution that satisfies the first two conditions if and only if it has a solution that satisfies all four conditions.

$$\begin{aligned} \mathbf{Lub}_{\leq_2}(u, u) &= u, \text{ if } u \in \mathbf{T}_0 \text{ and } u \text{ is not an arrow type} \\ \mathbf{Lub}_{\leq_2}(u_1 \rightarrow v_1, u_2 \rightarrow v_2) &= u_1 \wedge u_2 \rightarrow \mathbf{Lub}_{\leq_2}(v_1, v_2) \end{aligned}$$

Fig. 12. The partial function $\mathbf{Lub}_{\leq_2} : \mathbf{T}_2 \times \mathbf{T}_2 \rightarrow \mathbf{T}_2$.

THEOREM 9.2. *If a $\leq_{v_2,1}$ -satisfaction problem is solvable, then there is a most general solution for it. In particular, there is an algorithm that decides, for any $\leq_{v_2,1}$ -satisfaction problem, whether it is solvable, and, if so, returns a most general solution.*

PROOF. See Appendix A.1. \square

9.2 A Lub Satisfaction Algorithm

Definition 9.3. (Least upper bound with respect to \leq_2). Given two rank 2 types $v_1, v_2 \in \mathbf{T}_2$ the *least upper bound (lub) with respect to \leq_2* of v_1 and v_2 is a rank 2 type v such that

- (1) $v_1 \leq_2 v, v_2 \leq_2 v$, and
- (2) for every $v' \in \mathbf{T}_2$ if $v_1 \leq_2 v'$ and $v_2 \leq_2 v'$ then $v \leq_2 v'$.

It is straightforward to check that, for every $v_1, v_2 \in \mathbf{T}_2$, if the lub with respect to \leq_2 exists then it is unique (modulo \cong_2).³⁵ Moreover, the lub operator, which is a partial function $\mathbf{T}_2 \times \mathbf{T}_2 \rightarrow \mathbf{T}_2$, is idempotent, commutative, and associative (modulo \cong_2). As illustrated by the following lemmas the lub operator admits an inductive characterization and commutes with substitution.

LEMMA 9.4. (Inductive characterization the lub with respect to \leq_2). Let $\mathbf{Lub}_{\leq_2} : \mathbf{T}_2 \times \mathbf{T}_2 \rightarrow \mathbf{T}_2$ be the partial function defined by the clauses in Figure 12. For every $v_1, v_2 \in \mathbf{T}_2$,

- (1) if $\mathbf{Lub}_{\leq_2}(v_1, v_2)$ is defined, then it is the lub with respect to \leq_2 of v_1 and v_2 , and
- (2) if $\mathbf{Lub}_{\leq_2}(v_1, v_2)$ is not defined, then no upper bound with respect to \leq_2 of v_1 and v_2 exists.

PROOF. By induction on the definition of \mathbf{Lub}_{\leq_2} (see Figure 12). \square

Observe that Lemma 9.4 assures that, for all $v_1, v_2 \in \mathbf{T}_2$, if there exists $v \in \mathbf{T}_2$ such that $v_1 \leq_2 v$ and $v_2 \leq_2 v$, then $\mathbf{Lub}_{\leq_2}(v_1, v_2)$ is defined.

LEMMA 9.5. (\mathbf{Lub}_{\leq_2} commutes with substitution). If $v = \mathbf{Lub}_{\leq_2}(v_1, v_2)$, then $\mathbf{s}(v) \cong_2 \mathbf{Lub}_{\leq_2}(\mathbf{s}(v_1), \mathbf{s}(v_2))$.

PROOF. By induction on the definition of \mathbf{Lub}_{\leq_2} (see Figure 12). \square

³⁵The relation $\cong_2 (\subseteq \mathbf{T}_2 \times \mathbf{T}_2)$ is the equivalence induced by \leq_2 , i.e., $v_1 \cong_2 v_2$ means that both $v_1 \leq_2 v_2$ and $v_2 \leq_2 v_1$.

For every $v_1, \dots, v_n \in \mathbf{T}_2$ ($n \geq 1$) define

$$\mathbf{Lub}_{\leq 2}^n(v_1, \dots, v_n) = \begin{cases} v_1, & \text{if } n = 1 \\ \mathbf{Lub}_{\leq 2}(\mathbf{Lub}_{\leq 2}^{n-1}(v_1, \dots, v_{n-1}), v_n), & \text{if } n > 1 \end{cases}$$

A $\mathbf{Lub}_{\leq 2}$ -satisfaction problem is a pair $\langle j, Q \rangle$ where

- Q is a finite non-empty set of rank 2 types, and
- $j \in \{0, \dots, \mathbf{Max}\{\mathbf{Ind}^{\rightarrow}(v) \mid v \in Q\}\}$.

A substitution \mathbf{s} is a *solution* to $\langle j, \{v_1, \dots, v_n\} \rangle$ if $\mathbf{Ind}^{\wedge}(\mathbf{Lub}_{\leq 2}^n(\mathbf{s}(v_1), \dots, \mathbf{s}(v_n))) \leq j$.

Definition 9.6. (Most general solutions of a $\mathbf{Lub}_{\leq 2}$ -satisfaction problem).

- (1) A substitution \mathbf{s} is a *most general solution (mgs)* of a $\mathbf{Lub}_{\leq 2}$ -satisfaction problem $\langle j, Q \rangle$ if it satisfies the following conditions³⁶:
 - \mathbf{s} is a solution of $\langle j, Q \rangle$,
 - $\mathbf{s} \leq \mathbf{s}'$, for all solutions \mathbf{s}' of $\langle j, Q \rangle$,
 - \mathbf{s} is idempotent, and
 - $\mathbf{Dom}(\mathbf{s}) \subseteq \mathbf{FTV}(Q)$.
- (2) We write $\mathbf{MGS}(\langle j, Q \rangle)$ for the (possibly empty) set of the most general solutions of $\langle j, Q \rangle$.

THEOREM 9.7. *If a $\mathbf{Lub}_{\leq 2}$ -satisfaction problem is solvable, then there is a most general solution for it. In particular, there is an algorithm that decides, for any $\mathbf{Lub}_{\leq 2}$ -satisfaction problem, whether it is solvable, and, if so, returns a most general solution.*

PROOF. See Appendix A.2. \square

9.3 An Inference Algorithm for $\vdash_{\wedge 2}^{\text{Loc, Rec, Con}}$

In the following definitions we will write “the pair $\langle A, v \rangle$ is fresh” as short for “the type variables in the pair $\langle A, v \rangle$ are fresh”. Moreover, for a rigorous treatment of new type variables, we assume that, whenever “ $\mathbf{s} \in \mathbf{MGS}(\exists \vec{\alpha}. P)$ ” (or “ $\mathbf{s} \in \mathbf{MGS}(\langle j, Q \rangle)$ ”) occurs in a mathematical context, \mathbf{s} is chosen so that it does not interfere with “current” type variables, that is, we assume that $\mathbf{FTVR}(\mathbf{s}) \cap W = \emptyset$, where $W \uplus \mathbf{FTV}(\exists \vec{\alpha}. P)$ ($W \uplus \mathbf{FTV}(Q)$) is the set of the type variables present in the context.³⁷

9.3.1 Inference Algorithm for \triangleright . The inference algorithm is presented by defining the function **PAT** that, for every pattern p , provides an inductive characterization of the set of principal pairs for p .

Definition 9.8. (The function **PAT**). For every pattern p , the set $\mathbf{PAT}(p)$ is defined by structural induction on p .

³⁶As for Definition 9.1, the last two conditions are included for technical convenience only. Indeed they can be eliminated: a $\mathbf{Lub}_{\leq 2}$ -satisfaction problem has a solution that satisfies the first two conditions if and only if it has a solution that satisfies all four conditions.

³⁷The operator \uplus denotes disjoint union.

- If $p = x$, then $\langle \{x : \alpha\}, \alpha \rangle \in \mathbf{PAT}(x)$.
- If $p = cs^n p_1 \cdots p_n$, $\mathbf{Typeof}(cs^n) = \forall \vec{\alpha}. u_1 \rightarrow \cdots \rightarrow u_n \rightarrow u$, the substitution \mathbf{s} is a fresh renaming of $\vec{\alpha}$, for all $i \in \{1, \dots, n\}$ the pairs $\langle U_i, u_i \rangle \in \mathbf{PAT}(p_i)$ are fresh, for all $j, l \in \{1, \dots, n\}$ $j \neq l$ implies $\text{Dom}(U_j) \cap \text{Dom}(U_l) = \emptyset$, and $\mathbf{s}' \in \mathbf{MGS}(\exists \epsilon. \{u'_1 = \mathbf{s}(u_1), \dots, u'_n = \mathbf{s}(u_n)\})$, then

$$\langle \mathbf{s}'(U_1 \cup \cdots \cup U_n), \mathbf{s}'(\mathbf{s}(u)) \rangle \in \mathbf{PAT}(cs^n p_1 \cdots p_n).$$

For every pattern p , the set $\mathbf{PAT}(p)$ is an equivalence class of pairs modulo renaming of the type variables in a pair.

LEMMA 9.9. *For every pattern p , if $\langle U, u \rangle \in \mathbf{PAT}(p)$, then*

- (1) $\text{Dom}(U) = \text{FV}(p)$, and
- (2) $\langle U', u' \rangle \in \mathbf{PAT}(p)$ if and only if there is a bijection $\mathbf{s} : \mathbf{Tv} \rightarrow \mathbf{Tv}$ such that $\mathbf{s}(U) = U'$ and $\mathbf{s}(u) = u'$.

PROOF. By induction on Definition 9.8. \square

Indeed Definition 9.8 specifies an inference algorithm: to perform type inference on a pattern p simply follow the definition of $\mathbf{PAT}(p)$, choosing fresh type variables and using the \leq_2 -satisfaction algorithm as necessary.

LEMMA 9.10. *(Soundness and completeness of \mathbf{PAT} for \triangleright). For every pattern p :*

- (Soundness). *If $\langle U, u \rangle \in \mathbf{PAT}(p)$, then $U \triangleright p : u$.*
- (Completeness). *If $U \triangleright p : u'$, then there are a pair $\langle U, u \rangle \in \mathbf{PAT}(p)$ and a substitution \mathbf{s} such that: $\mathbf{s}(U) = U'$ and $\mathbf{s}(u) = u'$.*

PROOF. By structural induction on p (note that system \triangleright is syntax directed). \square

9.3.2 *An Inference Algorithm for $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$.* The inference algorithm is presented by defining the function \mathbf{PP} which, for every expression e and environment D , returns a set of pairs $\mathbf{PP}(D, e)$ such that

- if $\mathbf{PP}(D, e) = \emptyset$, then e can not be typed by $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$ with respect to D , and
- every element of $\mathbf{PP}(D, e)$ is a principal pair for e with respect to D .³⁸

Definition 9.11. (The function \mathbf{PP}). For every expression e and environment D , the set $\mathbf{PP}(D, e)$ is defined by structural induction on e .

- If $e = x$, then
 - If $x : \forall \vec{\alpha}. \langle A, v \rangle \in D$ and the substitution \mathbf{s} is a fresh renaming of $\vec{\alpha}$, then $\langle \mathbf{s}(A), \mathbf{s}(v) \rangle \in \mathbf{PP}(D, x)$.
 - If $x : \forall \vec{\alpha}. v \in D$ and the substitution \mathbf{s} is a fresh renaming of $\vec{\alpha}$, then $\langle \emptyset, \mathbf{s}(v) \rangle \in \mathbf{PP}(D, x)$.
 - If $x \notin \text{Dom}(D)$ and α is a type variable, then $\langle \{x : \alpha\}, \alpha \rangle \in \mathbf{PP}(D, x)$.

³⁸The set $\mathbf{PP}(D, e)$ does not contain all the principal pairs for e with respect to D . For instance, $\langle \emptyset, (\alpha_1 \wedge \alpha_2) \rightarrow \alpha_1 \rangle$ is a principal pair for $\lambda x. x$ with respect to \emptyset , but $\langle \emptyset, (\alpha_1 \wedge \alpha_2) \rightarrow \alpha_1 \rangle \notin \mathbf{PP}(\emptyset, \lambda x. x)$.

- If $e = c$, $\mathbf{Typeof}(c) = \forall \vec{\alpha}. v$, and the substitution \mathbf{s} is a fresh renaming of $\vec{\alpha}$, then $\langle \emptyset, \mathbf{s}(v) \rangle \in \mathbf{PP}(D, c)$.
- If $e = \lambda x.e_0$, x' is fresh and $\langle A, v \rangle \in \mathbf{PP}(D, e_0[x := x'])$, then
 - If $x \notin \mathbf{FV}(e_0)$ and α is a fresh type variable, then

$$\langle A, \alpha \rightarrow v \rangle \in \mathbf{PP}(D, \lambda x.e_0)$$
 - If $x \in \mathbf{FV}(e_0)$ and $A = A'$, $x' : ui$, then

$$\langle A', ui \rightarrow v \rangle \in \mathbf{PP}(D, \lambda x.e_0)$$
- If $e = e_0e_1$ and $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$, then
 - If $v_0 = \alpha$ (a type variable), α_1 and α_2 are fresh type variables, $\langle A_1, v_1 \rangle \in \mathbf{PP}(D, e_1)$ is fresh, and $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{v_1 \leq \alpha_1, \alpha = \alpha_1 \rightarrow \alpha_2\})$, then

$$\langle \mathbf{s}(A_0 + A_1), \mathbf{s}(v_0) \rangle \in \mathbf{PP}(D, e_0e_1)$$
 - If $v_0 = u_1 \wedge \dots \wedge u_n \rightarrow v$, for all $i \in \{1, \dots, n\}$ the pairs $\langle A_i, v_i \rangle \in \mathbf{PP}(D, e_1)$ are fresh, and $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{v_i \leq u_i \mid i \in \{1, \dots, n\}\})$, then

$$\langle \mathbf{s}(A_0 + A_1 + \dots + A_n), \mathbf{s}(v_0) \rangle \in \mathbf{PP}(D, e_0e_1)$$
- If $e = \text{let } x = e_0 \text{ in } e_1$, $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$, $\vec{\alpha} = \mathbf{FTV}(A_0) \cup \mathbf{FTV}(v_0)$, x' is fresh, and $\langle A_1, v_1 \rangle \in \mathbf{PP}(D, x' : \forall \vec{\alpha}. \langle A_0, v_0 \rangle)$, $e_1[x := x']$ is fresh, then
 - If $x \in \mathbf{FV}(e_1)$, then

$$\langle A_1, v_1 \rangle \in \mathbf{PP}(D, \text{let } x = e_0 \text{ in } e_1)$$
 - If $x \notin \mathbf{FV}(e_1)$, then

$$\langle A_0 + A_1, v_1 \rangle \in \mathbf{PP}(D, \text{let } x = e_0 \text{ in } e_1)$$
- If $e = \text{rec}_{i_0} \{x_1 = e_1, \dots, x_n = e_n\}$, x'_1, \dots, x'_n are fresh, for all $i \in \{1, \dots, n\}$ the pairs $\langle A_i, v_i \rangle \in \mathbf{PP}(D, e_i[x_1 := x'_1] \dots [x_n := x'_n])$ are fresh, $\{x'_{j_1}, \dots, x'_{j_m}\} = \{x'_1, \dots, x'_n\} \cap \text{Dom}(A_1 + \dots + A_n)$, $A_1 + \dots + A_n = A$, $x'_{j_1} : ui_{j_1}, \dots, x'_{j_m} : ui_{j_m}$, and $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{\text{Gen}(A_1 + \dots + A_n, v_j) \leq ui_j \mid j \in \{j_1, \dots, j_m\}\})$, then

$$\langle \mathbf{s}(A), \mathbf{s}(v_{i_0}) \rangle \in \mathbf{PP}(D, \text{rec}_{i_0} \{x_1 = e_1, \dots, x_n = e_n\})$$
- If $e = \text{if } e_0 \text{ then } e_1 \text{ else } e_2$, $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$, $\mathbf{s}_0 \in \mathbf{MGS}(\exists \epsilon. \{v_0 \leq \text{bool}\})$, $\langle A_1, v_1 \rangle \in \mathbf{PP}(D, e_1)$ and $\langle A_2, v_2 \rangle \in \mathbf{PP}(D, e_2)$ are fresh, $j = \mathbf{Max}(\mathbf{Ind}^\rightarrow(v_1), \mathbf{Ind}^\rightarrow(v_2))$, and $\mathbf{s} \in \mathbf{MGS}(\langle j, \{v_1, v_2\} \rangle)$, then

$$\langle \mathbf{s}_0(A_0) + \mathbf{s}(A_1) + \mathbf{s}(A_2), \mathbf{Lub}_{\leq_2}(\mathbf{s}(v_1), \mathbf{s}(v_2)) \rangle \in \mathbf{PP}(D, \text{if } e_0 \text{ then } e_1 \text{ else } e_2)$$
- If $e = \text{match } e_0 \text{ with } \{p_1 \Rightarrow e_1, \dots, p_n \Rightarrow e_n\}$, $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$, for all $l \in \{1, \dots, n\}$
 - $\{x_{l,1}, \dots, x_{l,m_l}\} = \mathbf{FV}(p_l)$,
 - $x'_{l,1}, \dots, x'_{l,m_l}$ are fresh,
 - $p'_l = p_l[x_{l,1} := x'_{l,1}] \dots [x_{l,m_l} := x'_{l,m_l}]$,
 - $e'_l = e_l[x_{l,1} := x'_{l,1}] \dots [x_{l,m_l} := x'_{l,m_l}]$,
 - $\langle U_l, u_l \rangle \in \mathbf{PAT}(p'_l)$ are fresh,
 - $\langle A_l, v_l \rangle \in \mathbf{PP}(D, e'_l)$ are fresh,
 - $\{y_{l,1}, \dots, y_{l,h_l}\} = \text{Dom}(U_l) \cap \text{Dom}(A_l)$,³⁹

³⁹Observe that $\{y_{l,1}, \dots, y_{l,h_l}\} \subseteq \{x'_{l,1}, \dots, x'_{l,m_l}\}$, since $\text{Dom}(U_l) = \mathbf{FV}(p'_l)$.

$-U_l = U'_l, \{y_{l,1} : u_{l,1}, \dots, y_{l,h_l} : u_{l,h_l}\},$
 $-A_l = A'_l, \{y_{l,1} : ui_{l,1}, \dots, y_{l,h_l} : ui_{l,h_l}\},$
 α is a fresh type variable,

$\mathbf{s} \in \mathbf{MGS}(\exists \alpha. \{v_0 \leq \alpha\} \cup (\bigcup_{1 \leq l \leq n} \{u_l \leq \alpha, ui_{l,1} \leq u_{l,1}, \dots, ui_{l,h_l} \leq u_{l,h_l}\})),$ and
 $j = \mathbf{Max}(\mathbf{Ind}^{\rightarrow}(\mathbf{s}(v_1)), \dots, \mathbf{Ind}^{\rightarrow}(\mathbf{s}(v_n))),$ and $\mathbf{s}' \in \mathbf{MGS}(\langle j, \{\mathbf{s}(v_1), \dots, \mathbf{s}(v_n)\} \rangle),$ then

$$\langle \mathbf{s}'(\mathbf{s}(A_0 + A'_1 + \dots + A'_n)), \mathbf{Lub}_{\leq_2}^n(\mathbf{s}'(\mathbf{s}(v_1)), \dots, \mathbf{s}'(\mathbf{s}(v_n))) \rangle \in \mathbf{PP}(D, \text{match } e_0 \text{ with } \{p_1 \Rightarrow e_1, \dots, p_n \Rightarrow e_n\})$$

For every expression e and environment D , the set $\mathbf{PP}(D, e)$ is an equivalence class of pairs modulo renaming of the type variables in a pair.

LEMMA 9.12. *For every expression e and environment D , if $\langle A, v \rangle \in \mathbf{PP}(D, e)$, then*

- (1) $\text{Dom}(A) = (\text{FV}(e) - \text{Dom}(D)) \cup \text{FVR}(D|_{\text{FV}(e)}),$ and
- (2) $\langle A', v' \rangle \in \mathbf{PP}(D, e)$ if and only if there is a bijection $\mathbf{s} : \mathbf{Tv} \rightarrow \mathbf{Tv}$ such that $\mathbf{s}(A) = A'$ and $\mathbf{s}(v) = v'$.

PROOF. By induction on Definition 9.11. \square

Indeed Definition 9.11 specifies an inference algorithm: to perform type inference on an expression e with respect to the environment D simply follow the definition of $\mathbf{PP}(D, e)$, choosing fresh type variables and using the \leq_2 -satisfaction, \mathbf{Lub}_{\leq_2} -satisfaction, and **PAT** algorithms as necessary.

THEOREM 9.13. *(Soundness and completeness of **PP** for $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$). For every expression e and environment D :*

(Soundness). If $\langle A, v \rangle \in \mathbf{PP}(D, e)$, then $D; A \vdash_{\wedge_2}^{\text{Loc, Rec, Con}} e : v$.

(Completeness). If $D; A' \vdash_{\wedge_2}^{\text{Loc, Rec, Con}} e : v'$, then there are a pair $\langle A, v \rangle \in \mathbf{PP}(D, e)$ and a substitution \mathbf{s} such that $A' \leq_1 \mathbf{s}(A)$ and $\mathbf{s}(v) \leq_2 v'$.

PROOF. See Appendix A.3. \square

10. RELATED WORK AND CONCLUSION

The literature related to the present work has been partially quoted through the paper. We add here a brief survey on inference algorithms for systems involving intersection types.

The system of intersection types [Coppo and Dezani-Ciancaglini 1980; Coppo 1980; Coppo et al. 1981; Barendregt et al. 1983] is undecidable. A first approach for designing a (semi-)algorithm for finding a principal pair is that of splitting the problem in two parts: first find a normal form for the term being typed (this problem is undecidable) and then build a principal pair for the normal form (this problem is decidable). To show that the approach is sound it is necessary to prove that both the normal form and the original term have the same typings. This approach has been followed, for instance, by Coppo et al. [1980], Ronchi della Rocca and Venneri [1984], and van Bakel [1993].

The first unification-based approach to intersection type inference was proposed by Ronchi della Rocca [1988] where, in particular, a decidable restriction which bounds the height of types is presented. More recently, Kfoury and Wells [1999] presented a simpler unification-based approach using a novel form of unification, called β -unification.

In spite of the fact that System F [Girard 1972; Reynolds 1974] and its finite rank restrictions above 3 have neither principal typings nor decidable type inference [Wells 1994; Kfoury and Wells 1994], combining intersection types and universal quantification results in a system with principal typings [Margaria and Zacchi 1995]. Moreover, decidable restrictions of the combined system have been developed (e.g. [Damiani and Giannini 1994; Coppo and Giannini 1995; Jim 2000]).

Due to the expressive power of the corresponding type systems, the inference algorithms mentioned above are inherently quite expensive. More efficient inference algorithms can be developed by considering the rank 2 restriction of intersection types [Leivant 1983; van Bakel 1993; Yokouchi 1995; Jim 1996]. Following this line of research, in the present paper we have proposed a sound and complete principal pair inference algorithm for a rank 2 intersection type system with new typing rules for assigning rank 2 types to:

- (1) local definitions, by using the principal pair property of the system to associate a different instance of the principal typing to each occurrence of the locally defined identifier;
- (2) mutually recursive definitions, by allowing a restricted form of polymorphic recursion (as explained in Section 6, this rule is a slight improvement of a rule proposed by Jim [1996, 1995]);
- (3) conditional expressions, by restricting the use of intersection in the types assigned to the branches in order to preserve the principal pair property.

Both the techniques for local definitions and recursive definitions do not depend on the particulars of rank 2 intersection typing, but apply to any system with principal pairs. The technique for conditional expressions, instead, is tailored to rank 2 intersection types. However, the strategy of introducing metrics on types to limit the use of the intersection type constructor in the types assigned to the branches of the conditionals expressions might also be useful for other type systems.

APPENDIX

A.1 An Extension of Jim's Subtype Satisfaction Algorithm (Theorem 9.2)

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

A *unification problem* is a $\leq_{\forall 2,1}$ -satisfaction problem that involves only equalities.⁴⁰

⁴⁰A unification problem can be solved by Robinson's algorithm (see, for instance, Chapter 3 of Hindley [1997]), which can decide whether it is solvable, and, if so, return a most general solution.

The following lemma implies Theorem 9.2.

LEMMA A.1. *Every $\leq_{v2,1}$ -satisfaction problem is equivalent to a unification problem. In particular, there is an algorithm that, given a $\leq_{v2,1}$ -satisfaction problem, either proves that it cannot be satisfied or transforms it into an equivalent unification problem.*

PROOF. Following Jim [1996, 1995], we prove the lemma by providing the transformation algorithm. We first introduce a set of transformation rules of the form

$$t \leq ui \Rightarrow \exists \vec{\alpha}.P \quad (\text{where } t \in \mathbf{T}_2 \cup \mathbf{T}_{v2}).$$

Each rule transforms an inequality in a $\leq_{v2,1}$ -satisfaction problem. The transformation rules are in Figure 13. Note that, for each transformation rule $t \leq ui \Rightarrow \exists \vec{\alpha}.P$, we have that

$$\text{FTV}(t) \cup \text{FTV}(ui) = \text{FTV}(\exists \vec{\alpha}.P). \quad (4)$$

In particular, the type variables $\vec{\alpha}$ are either fresh type variables introduced by the transformation rule (i.e., they do not appear in the left-hand side) or are the bound variables of t (when $t \in \mathbf{T}_{v2}$).

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

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

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

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

- (1) every rewriting step (corresponding to the application of one of the rules in Figure 13) transforms a $\leq_{v2,1}$ -satisfaction problem into another $\leq_{v2,1}$ -satisfaction problem,
- (2) every rewriting step preserves the set of solutions (note that the condition (4) above is necessary to ensure this preservation),
- (3) every inequality matches the left-hand side of at most one rule, and
- (4) repeated application of these rules must terminate.⁴¹ This is proved by defining a metric $|\cdot|$ on problems for which each rewriting step is strictly decreasing. The size of inequalities will be defined by structural induction. By point (3), it is sufficient to consider the cases given by the left-hand sides of the rules in Figure 13. For the base case we set $|u_0 \leq u| = 1$. Every other case is simply defined so that the size of the left-hand side of a rule is greater than the sum of the of the inequalities appearing on the corresponding right-hand side. For example, define

$$\begin{aligned} -|(ui \rightarrow v) \leq \alpha| &= |\alpha_1 \leq ui| + |v \leq \alpha_2| + 1, \\ -|(ui \rightarrow v) \leq (u_1 \rightarrow u_2)| &= |u_1 \leq ui| + |v \leq u_2| + 1, \end{aligned}$$

⁴¹In the original termination proof given in Jim [1996, 1995] there is an error. Here we are following the termination proof given in Jim [2000].

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

Fig. 13. Transformation rules for $\leq_{\forall 2, 1}$ -satisfaction.

$$\begin{aligned}
& - |t \leq (u_1 \wedge \dots \wedge u_n)| = |t \leq u_1| + \dots + |t \leq u_n| + 1, \text{ and} \\
& - |(\forall \vec{\alpha}. v) \leq u| = |v \leq u| + 1.
\end{aligned}$$

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

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

Remark A.2. (Comparison with Jim's subtype satisfaction algorithm). Lemma A.1 and Theorem 9.2 are a slight extension of results presented by Jim [1996, 1995]. The main differences are in the transformation rules. The transformation rules in Figure 13 either produce a unification problem or a satisfaction problem which is clearly not satisfiable. The transformation rules in Jim [1996, 1995], instead, always produce a unification problem. This difference is due to the richer set of types handled by our rules⁴²: if we restrict ourselves to the set of types considered in Jim [1996, 1995] the rules in Figure 13 always produce a unification problem.

A.2 A Lub Satisfaction Algorithm (Theorem 9.7)

We say that a $\mathbf{Lub}_{\leq 2}$ -satisfaction problem and a $\leq_{\forall 2, 1}$ -satisfaction problem are *equivalent* if they have the same solutions.

A $\mathbf{Lub}_{\leq 2}$ -satisfaction problem $\langle j, \{v_1, \dots, v_n\} \rangle$ is *binary* if $n = 2$. The following lemma implies Theorem 9.7 restricted to binary $\mathbf{Lub}_{\leq 2}$ -satisfaction problems.

LEMMA A.3. *Every binary $\mathbf{Lub}_{\leq 2}$ -satisfaction problem is equivalent to a $\leq_{\forall 2, 1}$ -satisfaction problem. In particular, there is an algorithm that, given a $\mathbf{Lub}_{\leq 2}$ -satisfaction problem transforms it into an equivalent $\leq_{\forall 2, 1}$ -problem.*

⁴²The rules in Jim [1996, 1995] consider only type variables, arrow types, and top-level conjunctions. We also consider parametric datatypes.

$$\begin{aligned} \mathbf{TA}(0, v_1, v_2) &= \mathbf{MGS}(\exists\alpha.\{v_1 \leq \alpha, v_2 \leq \alpha\}), \\ &\quad \text{where } \alpha \text{ is a fresh type variable} \\ \mathbf{TA}(j+1, ui_1 \rightarrow v_1, ui_2 \rightarrow v_2) &= \mathbf{TA}(j, v_1, v_2) \\ \mathbf{TA}(j+1, ui \rightarrow v, \alpha) &= \\ \mathbf{TA}(j+1, \alpha, ui \rightarrow v) &= \exists\alpha_1\alpha_2 \uplus \vec{\beta}.(\{\alpha = \alpha_1 \rightarrow \alpha_2\} \cup P), \text{ where} \\ &\quad \alpha_1, \alpha_2 \text{ are fresh type variables, and} \\ &\quad \exists\vec{\beta}.P = \mathbf{TA}(j, [\alpha := \alpha_1 \rightarrow \alpha_2]v, \alpha_2), \\ &\quad \text{if } v = ui_1 \rightarrow \dots \rightarrow ui_m \rightarrow u \text{ (} m \geq 0 \text{) for some } u \in \mathbf{T}_0 \text{ such that} \\ &\quad u \text{ is not an arrow type and } \alpha \notin \mathbf{FTV}(u) \\ \mathbf{TA}(j+1, v_1, v_2) &= \exists\epsilon.\{\text{int} = \text{bool}\}, \text{ otherwise} \end{aligned}$$

Fig. 14. Transformation algorithm for binary $\mathbf{Lub}_{\leq 2}$ -satisfaction.

PROOF. We prove the lemma by providing the transformation algorithm. The transformation algorithm is specified as a recursive function \mathbf{TA} defined by the clauses in Figure 14.

To show that the recursive function \mathbf{TA} describes an algorithm that proves the lemma, for every binary $\mathbf{Lub}_{\leq 2}$ -satisfaction problem $\langle j, \{v_1, v_2\} \rangle$ we will prove that:

- (1) $\mathbf{TA}(j, v_1, v_2)$ is terminating and returns a $\leq_{v_2,1}$ -satisfaction problem,
- (2) $\mathbf{FTV}(\mathbf{TA}(j, v_1, v_2)) \subseteq \mathbf{FTV}(\{v_1, v_2\})$, and
- (3) the transformation preserves the set of solutions (i.e. $\langle j, \{v_1, v_2\} \rangle$ and $\mathbf{TA}(j, v_1, v_2)$ are equivalent).

Points (1) and (2) are straightforward, so we consider only point (3). The proof is by induction on $j \in \{0, \dots, \mathbf{Max}\{\mathbf{Ind}^\rightarrow(v_1), \mathbf{Ind}^\rightarrow(v_2)\}\}$.

$j = 0$. From the definition of \mathbf{TA} (see Figure 14) we have $\mathbf{TA}(0, v_1, v_2) = \exists\alpha.P$, where α is fresh and $P = \{v_1 \leq \alpha, v_2 \leq \alpha\}$.

—If \mathbf{s} is a solution of $\langle 0, \{v_1, v_2\} \rangle$ then $\mathbf{Ind}^\wedge(\mathbf{Lub}_{\leq 2}(\mathbf{s}(v_1), \mathbf{s}(v_2))) = 0$, which implies $\mathbf{s}(v_1) = \mathbf{s}(v_2) \in \mathbf{T}_0$. So \mathbf{s} is a solution of $\exists\alpha.P$.

—If \mathbf{s} is a solution of $\exists\alpha.P$ then $\mathbf{s}(v_1) = \mathbf{s}(v_2) \in \mathbf{T}_0$, which implies $\mathbf{Ind}^\wedge(\mathbf{Lub}_{\leq 2}(\mathbf{s}(v_1), \mathbf{s}(v_2))) = 0$. So \mathbf{s} is a solution of $\langle 0, \{v_1, v_2\} \rangle$.

$j > 0$. By definition of $\mathbf{Lub}_{\leq 2}$ -satisfaction problem we have $j \leq \mathbf{Max}\{\mathbf{Ind}^\rightarrow(v_1), \mathbf{Ind}^\rightarrow(v_2)\}$, so at least one of v_1 and v_2 is an arrow type.

—If they are both arrow types, say $v_1 = ui_1 \rightarrow v'_1$ and $v_2 = ui_2 \rightarrow v'_2$, then $\mathbf{TA}(j, ui_1 \rightarrow v'_1, ui_2 \rightarrow v'_2) = \mathbf{TA}(j-1, v'_1, v'_2)$. By definition of $\mathbf{Lub}_{\leq 2}$ we have that, for every substitution \mathbf{s} , $\mathbf{Lub}_{\leq 2}(\mathbf{s}(v_1), \mathbf{s}(v_2)) = \mathbf{s}(ui_1) \wedge \mathbf{s}(ui_2) \rightarrow \mathbf{Lub}_{\leq 2}(\mathbf{s}(v'_1), \mathbf{s}(v'_2))$. So $\langle j, \{v_1, v_2\} \rangle$ and $\langle j-1, \{v'_1, v'_2\} \rangle$ are equivalent, and the result follows by induction.

—If v_1 is a type variable α and $v_2 = ui \rightarrow v$, for some $v = ui_1 \rightarrow \dots \rightarrow ui_m \rightarrow u$ ($m \geq 0$) with $u \in \mathbf{T}_0$ such that u is not an arrow type and $\alpha \notin \mathbf{FTV}(u)$ (the symmetric case is similar), then $\mathbf{TA}(j, \alpha, ui \rightarrow v) = \exists\alpha_1\alpha_2 \uplus \vec{\beta}.(\{\alpha = \alpha_1 \rightarrow$

$$\begin{aligned}
\mathbf{TA}^1(j, v_1) &= \exists \epsilon. \emptyset \\
\mathbf{TA}^n(j, v_1, \dots, v_n) &= \exists \vec{\alpha} \uplus \vec{\beta} \uplus \vec{\gamma}. (P \cup P'), \text{ where} \\
&\quad \exists \vec{\alpha}. P = \mathbf{TA}^{n-1}(j, v_1, \dots, v_{n-1}), \\
&\quad \mathbf{s} \in \mathbf{MGS}(\exists \vec{\alpha}. P), \\
&\quad \mathbf{FTV}(v_n) \cap \mathbf{FTV}(\mathbf{FTVR}(\mathbf{s})) = \emptyset, \\
&\quad \exists \vec{\beta}. P' = \mathbf{TA}(j, \mathbf{Lub}_{\leq 2}^{n-1}(\mathbf{s}(v_1), \dots, \mathbf{s}(v_{n-1})), \mathbf{s}(v_n)), \text{ and} \\
&\quad \vec{\gamma} = \mathbf{FTVR}(\mathbf{s}) - (\vec{\alpha} \uplus \vec{\beta}), \\
&\quad \text{if } n > 1 \text{ and } \mathbf{TA}^{n-1}(j, v_1, \dots, v_{n-1}) \text{ has solutions} \\
\mathbf{TA}^n(j, v_1, \dots, v_n) &= \exists \epsilon. \{\text{int} = \text{bool}\} \\
&\quad \text{if } n > 1 \text{ and } \mathbf{TA}^{n-1}(j, v_1, \dots, v_{n-1}) \text{ has no solutions}
\end{aligned}$$

Fig. 15. Transformation algorithm for n -ary $\mathbf{Lub}_{\leq 2}$ -satisfaction.

$\alpha_2\} \cup P)$, where α_1, α_2 are fresh type variables, and $\exists \vec{\beta}. P = \mathbf{TA}(j-1, [\alpha := \alpha_1 \rightarrow \alpha_2]v, \alpha_2)$. We now prove that $\langle j, \{\alpha, ui \rightarrow v\} \rangle$ and $\exists \alpha_1 \alpha_2 \uplus \vec{\beta}. (\{\alpha = \alpha_1 \rightarrow \alpha_2\} \cup P)$ are equivalent.

- By definition of $\mathbf{Lub}_{\leq 2}$ we have that every solution \mathbf{s} of $\langle j, \{\alpha, ui \rightarrow v\} \rangle$ is such that $\mathbf{s}(\alpha) = u_1 \rightarrow u_2$ and $\mathbf{Ind}^\wedge(\mathbf{Lub}_{\leq 2}(\mathbf{s}(\alpha), \mathbf{s}(ui \rightarrow v))) \leq j$. So $\mathbf{s} = \mathbf{s}' \circ [\alpha := \alpha_1 \rightarrow \alpha_2]$ for some fresh type variables α_1, α_2 and substitution \mathbf{s}' such that $\mathbf{s}'(\gamma) = \mathbf{s}(\gamma)$ for every $\gamma \notin \{\alpha_1, \alpha_2\}$ and \mathbf{s}' is a solution of $\langle j-1, \{[\alpha := \alpha_1 \rightarrow \alpha_2]v, \alpha_2\} \rangle$. By induction \mathbf{s}' is a solution to $\exists \vec{\beta}. P = \mathbf{TA}(j-1, [\alpha := \alpha_1 \rightarrow \alpha_2]v, \alpha_2)$, so \mathbf{s} is a solution to $\exists \alpha_1 \alpha_2 \uplus \vec{\beta}. (\{\alpha = \alpha_1 \rightarrow \alpha_2\} \cup P)$.
- If \mathbf{s} is a solution to $\exists \alpha_1 \alpha_2 \uplus \vec{\beta}. (\{\alpha = \alpha_1 \rightarrow \alpha_2\} \cup P)$, then $\mathbf{s} = \mathbf{s}' \circ [\alpha := \alpha_1 \rightarrow \alpha_2]$, for some substitution \mathbf{s}' such that $\mathbf{s}'(\gamma) = \mathbf{s}(\gamma)$ for every $\gamma \notin \{\alpha_1, \alpha_2\}$ and \mathbf{s}' is a solution of $\exists \vec{\beta}. P = \mathbf{TA}(j-1, [\alpha := \alpha_1 \rightarrow \alpha_2]v, \alpha_2)$. By induction \mathbf{s}' is a solution to $\langle j-1, \{[\alpha := \alpha_1 \rightarrow \alpha_2]v, \alpha_2\} \rangle$, so \mathbf{s} is a solution to $\langle j, \{\alpha, ui \rightarrow v\} \rangle$.
- Otherwise $\langle j, \{v_1, v_2\} \rangle$ has no solutions and is, therefore, equivalent to $\exists \epsilon. \{\text{int} = \text{bool}\}$. \square

The following lemma generalizes LemmaA.3 to n -ary $\mathbf{Lub}_{\leq 2}$ -satisfaction problems, and so implies Theorem 9.7.

LEMMA A.4. *Every $\mathbf{Lub}_{\leq 2}$ -satisfaction problem is equivalent to a $\leq_{v_2, 1}$ -satisfaction problem. In particular, there is an algorithm that, given a $\mathbf{Lub}_{\leq 2}$ -satisfaction problem transforms it into an equivalent $\leq_{v_2, 1}$ -problem.*

PROOF. We prove the lemma by providing the transformation algorithm. The transformation algorithm is specified as a recursive function \mathbf{TA}^n defined by the clauses in Figure 15, that generalizes the function \mathbf{TA} in Figure 14 to an arbitrary number of arguments.

To show that the recursive function \mathbf{TA}^n describes an algorithm that proves the lemma, for every n -ary ($n \geq 1$) $\mathbf{Lub}_{\leq 2}$ -satisfaction problem $\langle j, \{v_1, \dots, v_n\} \rangle$

we will prove that, if $j \leq \mathbf{Ind}^\rightarrow(v_1)$ then⁴³

- (1) $\mathbf{TA}^n(j, v_1, \dots, v_n)$ is terminating and returns a $\leq_{v_2,1}$ -satisfaction problem,
- (2) $\mathbf{FTV}(\mathbf{TA}^n(j, v_1, \dots, v_n)) \subseteq \mathbf{FTV}(\{v_1, \dots, v_n\})$, and
- (3) the transformation preserves the set of solutions (i.e. $\langle j, \{v_1, \dots, v_n\} \rangle$ and $\mathbf{TA}^n(j, v_1, \dots, v_n)$ are equivalent).

Points (1) and (2) are straightforward, so we consider only point (3). The proof is by induction on n .

$n = 1$. Every substitution \mathbf{s} is a solution to $\langle j, \{v_1\} \rangle$. Therefore, $\langle j, \{v_1\} \rangle$ and $\exists \epsilon. \emptyset$ are equivalent.

$n > 1$. By induction $\langle j, \{v_1, \dots, v_{n-1}\} \rangle$ and $\mathbf{TA}^{n-1}(j, v_1, \dots, v_{n-1})$ are equivalent. Therefore, if $\mathbf{TA}^{n-1}(j, v_1, \dots, v_{n-1})$ has no solutions, then $\langle j, \{v_1, \dots, v_n\} \rangle$ and $\exists \epsilon. \{\text{int} = \text{bool}\}$ are equivalent (they both have no solutions). Assume that $\mathbf{TA}^{n-1}(j, v_1, \dots, v_{n-1}) = \exists \vec{\alpha}. P$ has solutions. For all $\mathbf{s} \in \mathbf{MGS}(\exists \vec{\alpha}. P)$ we have that, by Lemma A.3,

$\langle j, \mathbf{Lub}_{\leq_2}^{n-1}(\mathbf{s}(v_1), \dots, \mathbf{s}(v_{n-1})), \mathbf{s}(v_n) \rangle$ and $\mathbf{TA}(j, \mathbf{Lub}_{\leq_2}^{n-1}(\mathbf{s}(v_1), \dots, \mathbf{s}(v_{n-1})), \mathbf{s}(v_n))$ are equivalent. Let $\mathbf{TA}(j, \mathbf{Lub}_{\leq_2}^{n-1}(\mathbf{s}(v_1), \dots, \mathbf{s}(v_{n-1})), \mathbf{s}(v_n)) = \exists \vec{\beta}. P'$ and $\vec{\gamma} = \mathbf{FTVR}(\mathbf{s}) - (\vec{\alpha} \uplus \vec{\beta})$. We now prove that $\langle j, \{v_1, \dots, v_n\} \rangle$ and $\exists \vec{\alpha} \uplus \vec{\beta} \uplus \vec{\gamma}. (P \cup P')$ are equivalent.

—If \mathbf{s}' is a solution to $\langle j, \{v_1, \dots, v_n\} \rangle$ then it is a solution of $\langle j, \{v_1, \dots, v_{n-1}\} \rangle$. So there is $\mathbf{s} \in \mathbf{MGS}(\exists \vec{\alpha}. P)$ such that $\mathbf{s} \leq \mathbf{s}'$ and $\mathbf{FTV}(v_n) \cap \mathbf{FTVR}(\mathbf{s}) = \emptyset$. Since \mathbf{s} is idempotent, \mathbf{s}' is a solution to $\mathbf{TA}(j, \mathbf{Lub}_{\leq_2}^{n-1}(\mathbf{s}(v_1), \dots, \mathbf{s}(v_{n-1})), \mathbf{s}(v_n)) = \exists \vec{\beta}. P'$. Therefore, \mathbf{s}' is also a solution to $\exists \vec{\alpha} \uplus \vec{\beta} \uplus \vec{\gamma}. (P \cup P')$.

—If \mathbf{s}' is a solution to $\exists \vec{\alpha} \uplus \vec{\beta} \uplus \vec{\gamma}. (P \cup P')$, then \mathbf{s}' is a solution to both $\langle j, \{v_1, \dots, v_{n-1}\} \rangle$ and $\langle j, \{\mathbf{Lub}_{\leq_2}^{n-1}(\mathbf{s}(v_1), \dots, \mathbf{s}(v_{n-1})), \mathbf{s}(v_n)\} \rangle$, for some $\mathbf{s} \in \mathbf{MGS}(\exists \vec{\alpha}. P)$ such that $\mathbf{FTV}(v_n) \cap \mathbf{FTVR}(\mathbf{s}) = \emptyset$. Therefore, since $\mathbf{s} \leq \mathbf{s}'$ and \mathbf{s} is idempotent, $\mathbf{Ind}^\wedge(\mathbf{Lub}_{\leq_2}^n(\mathbf{s}'(v_1), \dots, \mathbf{s}'(v_n))) \leq j$. \mathbf{s}' is a solution to $\langle j, \{v_1, \dots, v_n\} \rangle$. \square

A.3 Soundness and Completeness of the Inference Algorithm

for $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$ (Theorem 9.13)

Let $\vdash_{\wedge_2}^{\text{sd}}$ be the syntax directed restriction of $\vdash_{\wedge_2}^{\text{Loc, Rec, Con}}$ obtained by removing rule (SUB) and by replacing the rules (IFI) and (MATCHI) by the following rules:

$$\begin{array}{c}
 D; A \vdash e_0 : \text{bool} \quad D; A_1 \vdash e_1 : v_1 \quad D; A_2 \vdash e_2 : v_2 \\
 v = \mathbf{Lub}_{\leq_2}(v_1, v_2) \\
 \mathbf{Ind}^\wedge(v) \leq \mathbf{Max}\{ \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v'_1) \mid D; A'_1 \vdash e_1 : v'_1\}, \\
 \mathbf{Min}\{\mathbf{Ind}^\rightarrow(v'_2) \mid D; A'_2 \vdash e_2 : v'_2\} \} \\
 \text{(IFISD)} \frac{}{D; A + A_1 + A_2 \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : v}
 \end{array}$$

⁴³By definition of \mathbf{Lub}_{\leq_2} -satisfaction problem we have $0 \leq j \leq \mathbf{Max}\{\mathbf{Ind}^\rightarrow(v_1), \dots, \mathbf{Ind}^\rightarrow(v_n)\}$. The condition $j \leq \mathbf{Ind}^\rightarrow(v_1)$ is necessary to ensure that, for every $k \in \{1, \dots, n\}$, $\langle j, \{v_1, \dots, v_k\} \rangle$ is a \mathbf{Lub}_{\leq_2} -satisfaction problem (i.e. $0 \leq j \leq \mathbf{Max}\{\mathbf{Ind}^\rightarrow(v_1), \dots, \mathbf{Ind}^\rightarrow(v_k)\}$).

$e \equiv x$.

(*Soundness*). We have to consider three cases.

- If $x : \forall \vec{\alpha}. \langle A_0, v_0 \rangle \in D$, then $\langle \mathbf{s}(A_0), \mathbf{s}(v_0) \rangle \in \mathbf{PP}(D, x)$, for some fresh renaming \mathbf{s} of $\vec{\alpha}$. We have $D; \mathbf{s}(A_0) \vdash_{\wedge_2}^{\text{sd}} x : \mathbf{s}(v_0)$ by rule (ID_{local}).
- If $x : \forall \vec{\alpha}. v \in D$, use rule (ID_{global}).
- If $x \notin \text{Dom}(D)$, use rule (ID_{undefined}).

(*Completeness*). Again, we have to consider three cases.

- If $x : \forall \vec{\alpha}. \langle A_0, v_0 \rangle \in D$, then the last (and only) rule applied must be (ID_{local}).
- If $x : \forall \vec{\alpha}. v \in D$, the last (and only) rule applied must be (ID_{global}).
- If $x \notin \text{Dom}(D)$, the last (and only) rule applied must be (ID_{undefined}).

In all the three cases the proof is immediate.

$e \equiv c$.

(*Soundness*). Use rule (CONST).

(*Completeness*). Immediate, since the last (and only) rule applied must be (CONST).

$e = \lambda x. e_0$.

(*Soundness*). We have to consider two cases.

- If $x \in \text{FV}(e_0)$, then $\langle A, x' : ui, v_0 \rangle \in \mathbf{PP}(D, e_0[x := x'])$ and $\langle A, ui \rightarrow v_0 \rangle \in \mathbf{PP}(D, \lambda x. e_0)$. By induction, $D; A, x' : ui \vdash_{\wedge_2}^{\text{sd}} e_0[x := x'] : v_0$, and by rule (ABS) we get:

$$D; A \vdash_{\wedge_2}^{\text{sd}} \lambda x. e_0 : ui \rightarrow v_0.$$

- If $x \notin \text{FV}(e_0)$, then $\langle A, v_0 \rangle \in \mathbf{PP}(D, e_0)$ and $\langle A, \alpha \rightarrow v_0 \rangle \in \mathbf{PP}(D, \lambda x. e_0)$ for some fresh type variable α . By induction, $D; A \vdash_{\wedge_2}^{\text{sd}} e_0 : v_0$, and by rule (ABSVAC) we get:

$$D; A \vdash_{\wedge_2}^{\text{sd}} \lambda x. e_0 : \alpha \rightarrow v_0.$$

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

- If $x \in \text{FV}(e_0)$, then the last rule applied must be (ABS). We have $v' = ui' \rightarrow v'_0$ and $D; A', x' : ui' \vdash_{\wedge_2}^{\text{sd}} e_0[x := x'] : v'_0$. By induction there exist a fresh pair $\langle (A, x' : ui), v_0 \rangle \in \mathbf{PP}(D, e_0[x := x'])$ and a substitution \mathbf{s} such that

$$A', x' : ui' = \mathbf{s}(A, x' : ui) \text{ and } \mathbf{s}(v_0) = v'_0.$$

So $\langle A, ui \rightarrow v_0 \rangle \in \mathbf{PP}(D, \lambda x. e_0)$ and \mathbf{s} are the desired pair and substitution.

- If $x \notin \text{FV}(e_0)$, then the last rule applied must be (ABSVAC). We have $v' = u \rightarrow v'_0$ and $D; A' \vdash_{\wedge_2}^{\text{sd}} e_0 : v'_0$. By induction there exist a fresh pair $\langle A, v_0 \rangle \in \mathbf{PP}(D, e_0)$ and a substitution \mathbf{s} such that

$$A' = \mathbf{s}(A) \text{ and } \mathbf{s}(v_0) = v'_0.$$

So $\langle A, \alpha \rightarrow v_0 \rangle \in \mathbf{PP}(D, \lambda x. e_0)$, where α is a fresh type variable, and $[\alpha := u] \circ \mathbf{s}$ are the desired pair and substitution.

$e \equiv e_0 e_1$.

(*Soundness*). We have $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$ and, by induction, $D; A_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : v_0$.

—If v_0 is a type variable α , then we must have a fresh pair $\langle A_1, v_1 \rangle \in \mathbf{PP}(D, e_1)$, $A = \mathbf{s}(A_0 + A_1)$ and $v = \mathbf{s}(\alpha_2)$, where $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{v_1 \leq \alpha_1, \alpha = \alpha_1 \rightarrow \alpha_2\})$ for fresh type variables α_1 and α_2 . By induction, $D; A_1 \vdash_{\wedge_2}^{\text{sd}} e_1 : v_1$. By substitutivity

$D; \mathbf{s}(A_0) \vdash_{\wedge_2}^{\text{sd}} e_0 : \mathbf{s}(\alpha)$, and

$D; \mathbf{s}(A_1) \vdash_{\wedge_2}^{\text{sd}} e_1 : \mathbf{s}(v_1)$.

Then, by rule (APP), we have:

$$D; \mathbf{s}(A_0 + A_1) \vdash_{\wedge_2}^{\text{sd}} e_0 e_1 : \mathbf{s}(\alpha_2).$$

—If $v_0 = u_1 \wedge \dots \wedge u_n \rightarrow v'$, then we must have fresh pairs $\langle A_i, v_i \rangle \in \mathbf{PP}(D, e_1)$ (for all $i \in \{1, \dots, n\}$), $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{v_i \leq u_i \mid i \in \{1, \dots, n\}\})$, $A = \mathbf{s}(A_0 + A_1 + \dots + A_n)$, and $v = \mathbf{s}(v')$. By induction and substitutivity

$D; \mathbf{s}(A_0) \vdash_{\wedge_2}^{\text{sd}} e_0 : \mathbf{s}(u_1 \wedge \dots \wedge u_n \rightarrow v')$, and

$D; \mathbf{s}(A_1) \vdash_{\wedge_2}^{\text{sd}} e_i : \mathbf{s}(v_i)$ (for all $i \in \{1, \dots, n\}$).

Then, by rule (APP), we have:

$$D; \mathbf{s}(A_0 + A_1 + \dots + A_n) \vdash_{\wedge_2}^{\text{sd}} e_0 e_1 : \mathbf{s}(v').$$

(*Completeness*). The last rule applied must be (APP). We have $D; A'_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : u'_1 \wedge \dots \wedge u'_n \rightarrow v'$, $D; A'_i \vdash_{\wedge_2}^{\text{sd}} e_1 : u'_i$ ($\forall i \in \{1, \dots, n\}$), and $A' = A'_0 + A'_1 + \dots + A'_n$. By induction both $\mathbf{PP}(D, e_0)$ and $\mathbf{PP}(D, e_1)$ are not empty and, by Lemma 9.12.(2), it is enough to consider the following two cases on the structure of the pairs $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$.

— $v_0 = \alpha$ (a type variable). By induction there exists a substitution \mathbf{s}_0 such that

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

By definition of substitution, $\mathbf{s}_0(\alpha) \in \mathbf{T}_0$, so we have that $n = 1$ and $v' = u$ for some $u \in \mathbf{T}_0$.

By induction and Lemma 9.12.(2) there is a fresh pair $\langle A_1, v_1 \rangle \in \mathbf{PP}(D, e_1)$ and a substitution \mathbf{s}_1 such that

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

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

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

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

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

$$\langle \mathbf{s}(A_0 + A_1), \mathbf{s}(\alpha_2) \rangle \in \mathbf{PP}(D, e_0 e_1)$$

and $\mathbf{s} \leq \mathbf{s}'$.

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

$$A'_0 = \mathbf{s}_0(A_0) \text{ and } \mathbf{s}_0(v) = u'_1 \wedge \dots \wedge u'_m \rightarrow v'.$$

By induction and Lemma 9.12.(2), for all $j \in \{1, \dots, m\}$, there are fresh pairs $\langle A_j, v_j \rangle \in \mathbf{PP}(D, e_1)$ and substitutions \mathbf{s}_j such that

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

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

$$\langle \mathbf{s}(A_0 + A_1 + \dots + A_m), \mathbf{s}(v) \rangle \in \mathbf{PP}(D, e_0 e_1)$$

and $\mathbf{s} \leq \mathbf{s}'$.

$e = \text{let } x = e_0 \text{ in } e_1$.

(*Soundness*). We have fresh pairs $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$ and $\langle A_1, v \rangle \in \mathbf{PP}(D, x' : \forall \vec{\alpha}. \langle A_0, v_0 \rangle), e_1[x := x']$, where $\vec{\alpha} = \text{FTV}(A_0) \cup \text{FTV}(v_0)$. By induction, $D; A_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : v_0$ and $D, x' : \forall \vec{\alpha}. \langle A_0, v_0 \rangle; A_1 \vdash_{\wedge_2}^{\text{sd}} e_1[x := x'] : v_1$.

—If $x \in \text{FV}(e_1)$, then $A = A_1$. So by rule (LETPSI) we have:⁴⁵

$$D; A \vdash_{\wedge_2}^{\text{sd}} \text{let } x = e_0 \text{ in } e_1 : v.$$

—If $x \notin \text{FV}(e_1)$, then $A = A_1 + A_0$. So by rule (LETPSI) we have:

$$D; A \vdash_{\wedge_2}^{\text{sd}} \text{let } x = e_0 \text{ in } e_1 : v.$$

(*Completeness*). The last rule applied must be (LETPSI). We have $D; A'_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : v'_0$, $\mathbf{Ind}^{\rightarrow}(v'_0) = \mathbf{Min}\{\mathbf{Ind}^{\rightarrow}(v''_0) \mid D; A'_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : v''_0\}$, $D, x' : \forall \vec{\alpha}. \langle A'_0, v'_0 \rangle; A'_1 \vdash_{\wedge_2}^{\text{sd}} e_1[x := x'] : v'$, and

$$A' = \begin{cases} A'_1, & \text{if } x \in \text{FV}(e_1) \\ A'_1 + A'_0, & \text{otherwise} \end{cases}$$

Let $x \in \text{FV}(e_1)$ (the other case is similar). By induction there are fresh pairs $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$, $\langle A_1, v_1 \rangle \in \mathbf{PP}(D, x' : \forall \vec{\alpha}. \langle A'_0, v'_0 \rangle), e_1[x := x']$, and substitutions $\mathbf{s}_0, \mathbf{s}_1$, such that:

0. $A'_0 = \mathbf{s}_0(A_0)$ and $\mathbf{s}_0(v_0) = v'_0$,
1. $A'_1 = \mathbf{s}_1(A_1)$ and $\mathbf{s}_1(v_1) = v'$.

By point (0), there are a fresh pair $\langle A_2, v \rangle \in \mathbf{PP}(D, x' : \forall \vec{\alpha}. \langle A_0, v_0 \rangle), e_1[x := x']$ and a substitution \mathbf{s}_2 such that:

2. $A'_1 = \mathbf{s}_2(A_2)$ and $\mathbf{s}_2(v) = v'$.

So $\langle A_2, v \rangle \in \mathbf{PP}(D, \text{let } x = e_0 \text{ in } e_1)$ and \mathbf{s}_2 are the desired pair and substitution.

$e = \text{rec}_{i_0} \{x_1 = e_1, \dots, x_n = e_n\} (i_0 \in \{1, \dots, n\})$.

(*Soundness*). We have fresh pairs $\langle A_i, v_i \rangle \in \mathbf{PP}(D, e_i[x_1 := x'_1] \dots [x_n := x'_n])$ (for all $i \in \{1, \dots, n\}$), and $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. \{\text{Gen}(A_1 + \dots + A_n, v_j) \leq u_i \mid j \in \{j_1, \dots, j_m\}\})$, with $\{x'_{j_1}, \dots, x'_{j_m}\} = \{x'_1, \dots, x'_n\} \cap (\text{Dom}(A_1 + \dots + A_n))$ and

⁴⁵The rule can be applied, since:

$$\mathbf{Ind}^{\rightarrow}(v_0) = \mathbf{Min}\{\mathbf{Ind}^{\rightarrow}(v'_0) \mid D; A'_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : v'_0\}$$

(by induction and Proposition 7.9).

$A_1 + \dots + A_n = A$, $x'_{j_1} : ui_{j_1}, \dots, x'_{j_m} : ui_{j_m}$. By induction and substitutivity $D; \mathbf{s}(A_i) \vdash_{\wedge_2}^{\text{sd}} e_i[x_1 := x'_1] \dots [x_n := x'_n] : \mathbf{s}(v_i)$ (for all $i \in \{1, \dots, n\}$). By Definition 9.1, for all $j \in \{j_1, \dots, j_m\}$, $\mathbf{s}(\text{Gen}(A_1 + \dots + A_n, v_j)) \leq_{\forall 2,1} \mathbf{s}(ui_j)$ and, by Lemma 3.8, $\text{Gen}(\mathbf{s}(A_1 + \dots + A_n), \mathbf{s}(v_j)) \leq_{\forall 2,1} \mathbf{s}(ui_j)$. Then, by rule (REC2), we have:

$$D; \mathbf{s}(A) \vdash_{\wedge_2}^{\text{sd}} \text{rec}_{i_0} \{x_1 = e_1, \dots, x_n = e_n\} : \mathbf{s}(v_{i_0}).$$

(Completeness). The last rule applied must be (REC2). We have $D; A'_i \vdash_{\wedge_2}^{\text{sd}} e_i[x_1 := x'_1] \dots [x_n := x'_n] : v'_i$ (for all $i \in \{1, \dots, n\}$), $\text{Gen}(A'_1 + \dots + A'_n, v'_j) \leq_{\forall 2,1} ui'_j$ (for all $j \in \{j_1, \dots, j_m\}$), with $\{x'_{j_1}, \dots, x'_{j_m}\} = \{x'_1, \dots, x'_n\} \cap \text{Dom}(A'_1 + \dots + A'_n)$ and $A'_1 + \dots + A'_n = A'$, $x'_{j_1} : ui'_{j_1}, \dots, x'_{j_m} : ui'_{j_m}$. By induction, for all $i \in \{1, \dots, n\}$, there are fresh pairs $\langle A_i, v_i \rangle \in \mathbf{PP}(D, e_i[x_1 := x'_1] \dots [x_n := x'_n])$ and substitutions \mathbf{s}_i such that

$$A'_i = \mathbf{s}_i(A_i) \text{ and } \mathbf{s}_i(v_i) = v'_i.$$

Let $A_1 + \dots + A_n = A$, $x'_{j_1} : ui_{j_1}, \dots, x'_{j_m} : ui_{j_m}$ and $P = \{\text{Gen}(A_1 + \dots + A_n, v_j) \leq ui_j \mid j \in \{j_1, \dots, j_m\}\}$. The substitution $\mathbf{s}' = \mathbf{s}_1 \circ \dots \circ \mathbf{s}_n$ is such that

$$A' = \mathbf{s}'(A) \text{ and } \mathbf{s}'(v_{i_0}) = v'_{i_0}$$

and is a solution of the satisfaction problem $\exists \epsilon. P$. So there is $\mathbf{s} \in \mathbf{MGS}(\exists \epsilon. P)$ such that

$$\langle \mathbf{s}(A), \mathbf{s}(v_{i_0}) \rangle \in \mathbf{PP}(D, \text{rec}_{i_0} \{x_1 = e_1, \dots, x_n = e_n\})$$

and $\mathbf{s} \leq \mathbf{s}'$.

$e = \text{if } e_0 \text{ then } e_1 \text{ else } e_2$.

(Soundness). We have fresh pairs $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$, $\langle A_1, v_1 \rangle \in \mathbf{PP}(D, e_1)$, $\langle A_2, v_2 \rangle \in \mathbf{PP}(D, e_2)$, and substitutions $\mathbf{s}_0 \in \mathbf{MGS}(\exists \epsilon. \{v_0 \leq \text{bool}\})$, $\mathbf{s} \in \mathbf{MGS}(\langle j, \{v_1, v_2\} \rangle)$, where $j = \mathbf{Max}(\mathbf{Ind}^{\rightarrow}(v_1), \mathbf{Ind}^{\rightarrow}(v_2))$. By induction and substitutivity

$$D; \mathbf{s}_0(A_0) \vdash_{\wedge_2}^{\text{sd}} e_0 : \mathbf{s}_0(v_0), \text{ and}$$

$$D; \mathbf{s}(A_i) \vdash_{\wedge_2}^{\text{sd}} e_i : \mathbf{s}(v_i) \text{ (for all } i \in \{1, 2\}).$$

Then by rule (IFISD) we have⁴⁶

$$D; \mathbf{s}_0(A_0) + \mathbf{s}(A_1) + \mathbf{s}(A_2) \vdash_{\wedge_2}^{\text{sd}} \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \mathbf{Lub}_{\leq 2}(\mathbf{s}(v_1), \mathbf{s}(v_2)).$$

(Completeness). The last rule applied must be (IFISD). We have $D; A'_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : \text{bool}$, $D; A'_1 \vdash_{\wedge_2}^{\text{sd}} e_1 : v'_1$, $D; A'_2 \vdash_{\wedge_2}^{\text{sd}} e_2 : v'_2$, $A' = A_0 + A_1 + A_2$, $v' = \mathbf{Lub}_{\leq 2}(v'_1, v'_2)$, and

$$\mathbf{Ind}^{\wedge}(v') \leq \mathbf{Max}\{ \mathbf{Min}\{\mathbf{Ind}^{\rightarrow}(v'_1) \mid D; A'_1 \vdash_{\wedge_2}^{\text{sd}} e_1 : v'_1\}, \mathbf{Min}\{\mathbf{Ind}^{\rightarrow}(v'_2) \mid D; A'_2 \vdash_{\wedge_2}^{\text{sd}} e_2 : v'_2\} \}$$

⁴⁶The rule can be applied, since

$$\mathbf{Ind}^{\wedge}(\mathbf{Lub}_{\leq 2}(\mathbf{s}(v_1), \mathbf{s}(v_2))) \leq j$$

(since $\mathbf{s} \in \mathbf{MGS}(\langle j, \{v_1, v_2\} \rangle)$), and

$$\begin{aligned} j &= \mathbf{Max}(\mathbf{Ind}^{\rightarrow}(v_1), \mathbf{Ind}^{\rightarrow}(v_2)) \\ &= \mathbf{Max}\{ \mathbf{Min}\{\mathbf{Ind}^{\rightarrow}(v'_1) \mid D; A'_1 \vdash_{\wedge_2}^{\text{sd}} e_1 : v'_1\}, \mathbf{Min}\{\mathbf{Ind}^{\rightarrow}(v_2) \mid D; A'_2 \vdash_{\wedge_2}^{\text{sd}} e_2 : v'_2\} \} \end{aligned}$$

(by induction and Proposition 7.9).

By induction there are fresh pairs $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$, $\langle A_1, v_1 \rangle \in \mathbf{PP}(D, e_1)$, $\langle A_2, v_2 \rangle \in \mathbf{PP}(D, e_2)$, and substitutions $\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2$, such that:

0. $A'_0 = \mathbf{s}_0(A_0)$ and $\mathbf{s}_0(v_0) = \text{bool}$,
1. $A'_1 = \mathbf{s}_1(A_1)$ and $\mathbf{s}_1(v_1) = v'_1$,
2. $A'_2 = \mathbf{s}_2(A_2)$ and $\mathbf{s}_2(v_2) = v'_2$.

Let $P = \{v_0 \leq \text{bool}\}$. The substitution $\mathbf{s}'' = \mathbf{s}_0 \circ \mathbf{s}_1 \circ \mathbf{s}_2$
— is such that

$$A'_0 + A'_1 + A'_2 = \mathbf{s}''(A_0 + A_1 + A_2), \quad \text{and} \quad \mathbf{Lub}_{\leq_2}(\mathbf{s}''(v_1), \mathbf{s}''(v_2)) = v',$$

— is a solution of the $\leq_{v_2, 1}$ -satisfaction problem $\exists \epsilon. P$, and

— is a solution to the \mathbf{Lub}_{\leq_2} -satisfaction problem $\langle j, \{v_1, v_2\} \rangle$, where $j = \mathbf{Max}(\mathbf{Ind}^{\rightarrow}(v_1), \mathbf{Ind}^{\rightarrow}(v_2))$.⁴⁷

So there are

— $\mathbf{s}_3 \in \mathbf{MGS}(\exists \epsilon. P)$, and

— $\mathbf{s} \in \mathbf{MGS}(\langle j, \{v_1, v_2\} \rangle)$,

such that

$$\langle \mathbf{s}_3(A_0) + \mathbf{s}(A_1) + \mathbf{s}(A_2), \mathbf{Lub}_{\leq_2}(\mathbf{s}(v_1), \mathbf{s}(v_2)) \rangle \in \mathbf{PP}(D, \text{if } e_0 \text{ then } e_1 \text{ else } e_2),$$

$\mathbf{s}'' = \mathbf{s}'_3 \circ \mathbf{s}_3$ and $\mathbf{s}'' = \mathbf{s}''' \circ \mathbf{s}$, for some substitutions $\mathbf{s}'_3, \mathbf{s}'''$ such that $\mathbf{Dom}(\mathbf{s}'_3) \cap \mathbf{Dom}(\mathbf{s}''') = \emptyset$, $\mathbf{FTVR}(\mathbf{s}'_3) \cap \mathbf{Dom}(\mathbf{s}''') = \emptyset$, $\mathbf{Dom}(\mathbf{s}''') \cap \mathbf{FTV}(\mathbf{s}'_3(\mathbf{s}_3(A_0))) = \emptyset$, and $\mathbf{Dom}(\mathbf{s}'_3) \cap (\mathbf{FTV}(\mathbf{s}(A_1)) \cup \mathbf{FTV}(\mathbf{s}(A_2)) \cup \mathbf{FTV}(\mathbf{s}(v_1)) \cup \mathbf{s}(\mathbf{FTV}(v_2))) = \emptyset$. Therefore, $\mathbf{s}' = \mathbf{s}''' \circ \mathbf{s}'_3$ is such that

$$A'_0 + A'_1 + A'_2 = \mathbf{s}'(\mathbf{s}_3(A_0) + \mathbf{s}(A_1) + \mathbf{s}(A_2)), \quad \text{and} \quad \mathbf{s}'(\mathbf{Lub}_{\leq_2}(\mathbf{s}(v_1), \mathbf{s}(v_2))) = v'.$$

$e = \text{match } e_0 \text{ with } \{p_1 \Rightarrow e_1, \dots, p_n \Rightarrow e_n\}$.

(Soundness). We have fresh pairs $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$, $\langle U_l, u_l \rangle \in \mathbf{PAT}(p_l)$ and $\langle A_l, v_l \rangle \in \mathbf{PP}(D, e_l)$ (for all $l \in \{1, \dots, n\}$), and substitutions

$$\mathbf{s} \in \mathbf{MGS}(\exists \alpha. \{v_0 \leq \alpha\} \cup (\bigcup_{1 \leq l \leq n} \{u_l \leq \alpha, ui_{l,1} \leq u_{l,1}, \dots, ui_{l,h_l} \leq u_{l,h_l}\})),$$

$\mathbf{s}' \in \mathbf{MGS}(\langle j, \{\mathbf{s}(v_1), \dots, \mathbf{s}(v_n)\} \rangle)$, with $\{y_{l,1}, \dots, y_{l,h_l}\} = \mathbf{Dom}(U_l) \cap \mathbf{Dom}(A_l)$, $U_l = U'_l, \{y_{l,1} : u_{l,1}, \dots, y_{l,h_l} : u_{l,h_l}\}, A_l = A'_l, \{y_{l,1} : ui_{l,1}, \dots, y_{l,h_l} : ui_{l,h_l}\}$, α fresh type variable, and $j = \mathbf{Max}(\mathbf{Ind}^{\rightarrow}(\mathbf{s}(v_1)), \dots, \mathbf{Ind}^{\rightarrow}(\mathbf{s}(v_n)))$. By induction and substitutivity

$$D; \mathbf{s}'(\mathbf{s}(A_0)) \vdash_{\wedge_2}^{\text{sd}} e_0 : \mathbf{s}'(\mathbf{s}(v_0)),$$

$$\mathbf{s}'(\mathbf{s}(U_l)) \triangleright p'_l : \mathbf{s}'(\mathbf{s}(u_l)) \text{ and } D; \mathbf{s}'(\mathbf{s}(A_l)) \vdash_{\wedge_2}^{\text{sd}} e'_l : \mathbf{s}'(\mathbf{s}(v_l)) \text{ (for all } l \in \{1, \dots, n\}).$$

Then by rule (MATCHISD) we have⁴⁸

$$D; \mathbf{s}'(\mathbf{s}(A_0 + A'_1 + \dots + A'_n)) \vdash_{\wedge_2}^{\text{sd}} \text{match } e_0 \text{ with } \{p_1 \Rightarrow e_1, \dots, p_n \Rightarrow e_n\} : \mathbf{Lub}_{\leq_2}^n(\mathbf{s}'(\mathbf{s}(v_1)), \dots, \mathbf{s}'(\mathbf{s}(v_n))).$$

⁴⁷In fact

$$\begin{aligned} & \mathbf{Ind}^{\wedge}(\mathbf{Lub}_{\leq_2}(\mathbf{s}''(v_1), \mathbf{s}''(v_2))) = \mathbf{Ind}^{\wedge}(v') = \mathbf{Ind}^{\wedge}(\mathbf{Lub}_{\leq_2}(v'_1, v'_2)) \\ & \leq \mathbf{Max}\{\mathbf{Min}\{\mathbf{Ind}^{\rightarrow}(v'_1) \mid D; A'_1 \vdash_{\wedge_2}^{\text{sd}} e_1 : v'_1\}, \mathbf{Min}\{\mathbf{Ind}^{\rightarrow}(v'_2) \mid D; A'_2 \vdash_{\wedge_2}^{\text{sd}} e_2 : v'_2\}\} \\ & = \mathbf{Max}(\mathbf{Ind}^{\rightarrow}(v_1), \mathbf{Ind}^{\rightarrow}(v_2)) \\ & = j \end{aligned}$$

(by induction and Proposition 7.9).

⁴⁸The rule can be applied, since

$$\mathbf{Ind}^{\wedge}(\mathbf{Lub}_{\leq_2}^n(\mathbf{s}'(\mathbf{s}(v_1)), \dots, \mathbf{s}'(\mathbf{s}(v_n)))) \leq j$$

(Completeness). The last rule applied must be (MATCHISD). We have $D; A'_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : u'_0$ and, $\forall i \in \{1, \dots, n\}$, $U'_i \triangleright p'_i : u'_0$, $D; A'_i, \{y : u' \in U'_i \mid y \in \text{FV}(e'_i)\} \vdash_{\wedge_2}^{\text{sd}} e'_i : v'_i$, $A' = A'_0 + A'_1 + \dots + A'_n$, $v' = \mathbf{Lub}_{\leq_2}^n(v'_1, \dots, v'_n)$, and

$$\mathbf{Ind}^\wedge(v') \leq \mathbf{Max}(\cup_{1 \leq i \leq n} \mathbf{Min}\{ \mathbf{Ind}^\rightarrow(v'_i) \mid \begin{array}{l} D; A'_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : u'_0, \\ U'_i \triangleright p'_i : u'_0, \text{ and} \\ D; A'_i, \{y : u \in U'_i \mid y \in \text{FV}(e'_i)\} \vdash_{\wedge_2}^{\text{sd}} e'_i : v'_i \} \}).$$

By induction and Lemma 9.10 there are fresh pairs $\langle A_0, v_0 \rangle \in \mathbf{PP}(D, e_0)$ and, $\forall l \in \{1, \dots, n\}$, $\langle U_l, \{y_{l,1} : u_{l,1}, \dots, y_{l,h_l} : u_{l,h_l}\}, u_l \rangle \in \mathbf{PAT}(p'_l)$, $\langle A_l, \{y_{l,1} : u_{l,1}, \dots, y_{l,h_l} : u_{l,h_l}\}, v_l \rangle \in \mathbf{PP}(D, e'_l)$ (where $\{y_{l,1}, \dots, y_{l,h_l}\} = \text{Dom}(U_l) \cap \text{FV}(e'_l)$), and substitutions $\mathbf{s}_0, \mathbf{s}'_l, \mathbf{s}_l$ such that:

0. $A'_0 = \mathbf{s}_0(A_0)$ and $\mathbf{s}_0(v_0) = u'_0$,
1. $U'_l = \mathbf{s}'_l(U_l, \{y_{l,1} : u_{l,1}, \dots, y_{l,h_l} : u_{l,h_l}\})$ and $\mathbf{s}'_l(u_l) = u'_l$,
2. $A'_l, \{y : u' \in U'_l \mid y \in \text{FV}(e'_l)\} = \mathbf{s}_l(A_l, \{y_{l,1} : u_{l,1}, \dots, y_{l,h_l} : u_{l,h_l}\})$ and $\mathbf{s}_l(v_l) = v'_l$.

Let $P = \{v_0 \leq \alpha\} \cup \cup_{1 \leq l \leq n} \{u_l \leq \alpha, u_{l,1} \leq u_{l,1}, \dots, u_{l,h_l} \leq u_{l,h_l}\}$, where α is a fresh type variable. The substitution $\mathbf{s}'' = \mathbf{s}_0 \circ \mathbf{s}'_1 \circ \dots \circ \mathbf{s}'_l \circ \mathbf{s}_1 \circ \dots \circ \mathbf{s}_l$

—is such that

$$\begin{aligned} A'_0 + A'_1 + \dots + A'_n &= \mathbf{s}''(A_0 + A_1 + \dots + A_n) \text{ and} \\ \mathbf{Lub}_{\leq_2}^n(\mathbf{s}''(v_1), \dots, \mathbf{s}''(v_n)) &= v', \end{aligned}$$

—is a solution of the $\leq_{v_2,1}$ -satisfaction problem $\exists \alpha. P$, and

—is a solution to the \mathbf{Lub}_{\leq_2} -satisfaction problem $\langle j, \{\mathbf{s}(v_1), \dots, \mathbf{s}(v_n)\} \rangle$, where $\mathbf{s} \in \mathbf{MGS}(\exists \alpha. P)$ is such that $\mathbf{s} \leq \mathbf{s}''$ and $j = \mathbf{Max}(\mathbf{Ind}^\rightarrow(\mathbf{s}(v_1), \dots, \mathbf{Ind}^\rightarrow(\mathbf{s}(v_n))))$.⁴⁹

Let \mathbf{s}''' such that $\mathbf{s}'' = \mathbf{s}''' \circ \mathbf{s}$, since \mathbf{s}''' is a solution to $\langle j, \{\mathbf{s}(v_1), \dots, \mathbf{s}(v_n)\} \rangle$, there is

$$\mathbf{s}' \in \mathbf{MGS}(\langle j, \{\mathbf{s}(v_1), \dots, \mathbf{s}(v_n)\} \rangle)$$

such that

$$\begin{aligned} (\mathbf{s}'(\mathbf{s}(A_0 + A_1 + \dots + A_n)), \mathbf{Lub}_{\leq_2}^n(\mathbf{s}'(\mathbf{s}(v_1)), \dots, \mathbf{s}'(\mathbf{s}(v_n)))) &\in \\ \mathbf{PP}(D, \text{match } e_0 \text{ with } \{p_1 \Rightarrow e_1, \dots, p_n \Rightarrow e_n\}) & \end{aligned}$$

and $\mathbf{s}' \circ \mathbf{s} \leq \mathbf{s}''$. \square

(since $\mathbf{s}' \in \mathbf{MGS}(\langle j, \{\mathbf{s}(v_1), \dots, \mathbf{s}(v_n)\} \rangle)$), and

$$\begin{aligned} j &= \mathbf{Max}(\mathbf{Ind}^\rightarrow(\mathbf{s}(v_1)), \dots, \mathbf{Ind}^\rightarrow(\mathbf{s}(v_n))) \\ &= \mathbf{Max}(\cup_{1 \leq l \leq n} \mathbf{Min}\{ \mathbf{Ind}^\rightarrow(v'_l) \mid \begin{array}{l} D; A'_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : u'_0, \\ U'_l \triangleright p'_l : u'_0, \text{ and} \\ D; A'_l, \{y : u \in U'_l \mid y \in \text{FV}(e'_l)\} \vdash_{\wedge_2}^{\text{sd}} e'_l : v'_l \} \} \end{array} \end{aligned}$$

(by induction and Proposition 7.9).

⁴⁹In fact

$$\begin{aligned} \mathbf{Ind}^\wedge(\mathbf{Lub}_{\leq_2}^n(\mathbf{s}''(v_1), \dots, \mathbf{s}''(v_n))) &= \mathbf{Ind}^\wedge(v') = \mathbf{Ind}^\wedge(\mathbf{Lub}_{\leq_2}^n(v'_1, \dots, v'_n)) \\ &\leq \mathbf{Max}(\cup_{1 \leq i \leq n} \mathbf{Min}\{ \mathbf{Ind}^\rightarrow(v'_i) \mid \begin{array}{l} D; A'_0 \vdash_{\wedge_2}^{\text{sd}} e_0 : u'_0, \\ U'_i \triangleright p'_i : u'_0, \text{ and} \\ D; A'_i, \{y : u \in U'_i \mid y \in \text{FV}(e'_i)\} \vdash_{\wedge_2}^{\text{sd}} e'_i : v'_i \} \} \\ &= \mathbf{Max}(\mathbf{Ind}^\rightarrow(\mathbf{s}(v_1)), \dots, \mathbf{Ind}^\rightarrow(\mathbf{s}(v_n))) \\ &= j \end{aligned}$$

(by induction and Proposition 7.9).

ACKNOWLEDGMENTS

I thank Mario Coppo, Mariangiola Dezani, Paola Giannini, Ines Margaria, Maddalena Zacchi, and the referees of earlier versions of this paper, for valuable comments and suggestions. I also thank Emiliano Leporati for much useful feedback during the preparation of his Master thesis [Leporati 2000]. I'm particularly grateful to the TOPLAS referees for insightful and constructive comments, which greatly improved the submitted version.

REFERENCES

- ADITYA, S. AND NIKHIL, R. 1991. Incremental polymorphism. In *Functional Programming Languages and Computer Architecture*. LNCS 523. Springer, 379–405.
- BARENDREGT, H. P., COPPO, M., AND DEZANI-CIANCAGLINI, M. 1983. A filter lambda model and the completeness of type assignment. *J. Symb. Logic* 48, 931–940.
- COPPO, M. 1980. An extended polymorphic type system. In *Mathematical Foundations of Computer Science*. LNCS 88. Springer, 194–204.
- COPPO, M. AND DEZANI-CIANCAGLINI, M. 1980. An extension of basic functional theory for lambda-calculus. *Notre Dame J. Formal Logic* 21, 4, 685–693.
- COPPO, M., DEZANI-CIANCAGLINI, M., AND VENNERI, B. 1980. Principal Type Schemes and Lambda-calculus Semantics. In *To H. B. Curry. Essays on Combinatory Logic, Lambda-calculus and Formalism*, R. Hindley and J. Seldin, Eds. Academic Press, London, 480–490.
- COPPO, M., DEZANI-CIANCAGLINI, M., AND VENNERI, B. 1981. Functional Characters of Solvable Terms. *Zeith. Math. Logik Und Grund. Math.* 27, 45–58.
- COPPO, M. AND GIANNINI, P. 1995. Principal Types and Unification for Simple Intersection Types Systems. *Information and Computation* 122, 1, 70–96.
- DAMAS, L. M. M. 1984. Type Assignment in Programming Languages. Ph.D. thesis, University of Edinburgh.
- DAMAS, L. M. M. AND MILNER, R. 1982. Principal type schemas for functional programs. In *POPL'82*. ACM, 207–212.
- DAMIANI, F. 2000. Typing local definitions and conditional expressions with rank 2 intersection. In *FOSSACS'00 (part of ETAPS'00)*. LNCS 1784. Springer, 82–97.
- DAMIANI, F. AND GIANNINI, P. 1994. A Decidable Intersection Type System based on Relevance. In *TACS'94*. LNCS 789. Springer, 707–725.
- GIRARD, J. Y. 1972. Interpretation fonctionnelle et elimination des coupures dans l'arithmetique d'ordre superieur. Ph.D. thesis, Université Paris VII.
- HENGLEIN, F. 1993. Type inference with polymorphic recursion. *ACM Trans. Prog. Lang. Syst.* 15, 2, 253–289.
- HINDLEY, R. 1997. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, London.
- JIM, T. 1995. Rank 2 type systems and recursive definitions. Tech. Rep. MIT/LCS/TM-531, LCS, Massachusetts Institute of Technology.
- JIM, T. 1996. What are principal typings and what are they good for? In *POPL'96*. ACM, 42–53.
- JIM, T. 2000. A polar type system. In *ICALP Workshops*. Proceedings in Informatics, vol. 8. Carleton-Scientific, 323–338.
- KFOURY, A. J., TIURYN, J., AND URZYCZYN, P. 1993. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Prog. Lang. Syst.* 15, 2, 290–311.
- KFOURY, A. J. AND WELLS, J. B. 1994. A direct algorithm for type inference in the rank-2 fragment of the second-order lambda -calculus. In *LISP and Functional Programming '94*. ACM.
- KFOURY, A. J. AND WELLS, J. B. 1999. Principality and Decidable Type Inference for Finite-Rank Intersection Types. In *POPL'99*. ACM, 161–174.
- LEVANT, D. 1983. Polymorphic Type Inference. In *POPL'83*. ACM, 88–98.
- LEPORATI, E. 2000. Intersezione al rank 2 per MiniOcaml. M.S. thesis, Dipartimento di Informatica, Università di Torino.
- MARGARIA, I. AND ZACCHI, M. 1995. Principal typing in a $\forall\wedge$ discipline. *J. Logic Comp.* 5, 367–381.

- MEERTENS, L. 1983. Incremental polymorphic type checking in B. In *POPL'83*. ACM, 265–275.
- MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML - Revised*. MIT Press.
- MYCROFT, A. 1984. Polymorphic Type Schemes and Recursive Definitions. In *International Symposium on Programming*. LNCS 167. Springer, 217–228.
- O'CAML. The O'Caml Language Home Page, <http://www.ocaml.org>.
- REYNOLDS, J. C. 1974. Towards a Theory of Type Structure. In *Colloque sur la Programmation*. LNCS 19. Springer.
- RONCHI DELLA ROCCA, S. 1988. Principal Type Scheme and Unification for Intersection Type Discipline. *Theoret. Comput. Sci.* 59, 181–209.
- RONCHI DELLA ROCCA, S. AND VENNARI, B. 1984. Principal Types Schemes for an extended type theory. *Theoret. Comput. Sci.* 28, 151–169.
- SHAO, Z. AND APPEL, A. W. 1993. Smartest recompilation. In *POPL'93*. ACM, 439–450.
- URZYCZYN, P. 1997. Type reconstruction in F_{ω} . *Mathematical Structures in Computer Science* 7, 4, 329–358.
- VAN BAKEL, S. 1993. Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems. Ph.D. thesis, Katholieke Universiteit Nijmegen.
- VAN BAKEL, S., BARBANERA, F., AND FERNÁNDEZ, M. 2000. Polymorphic Intersection Type Assignment for Rewrite Systems with Intersection and beta-rule (Extended Abstract). In *TYPES'99*. LNCS 1956. Springer, 41–60.
- WELLS, J. 1994. Typability and type-checking in the second-order lambda-calculus are equivalent and undecidable. In *Logic in Computer Science*. IEEE, 176–185.
- YOKOUCHI, H. 1995. Embedding a Second-Order Type System into an Intersection Type System. *Inform. Comp.* 117, 206–220.

Received December 2001; revised October 2002; accepted January 2003