Rank 2 Intersection Types for Modules

Ferruccio Damiani[*]
Dipartimento di Informatica, Università di Torino
Corso Svizzera 185
10149 Torino, Italy
damiani@di.unito.it

ABSTRACT

We propose a rank 2 intersection type system for a language of modules built on a core ML-like language. The *principal typing property* of the rank 2 intersection type system for the core language plays a crucial role in the design of the type system for the module language. We first consider a "plain" notion of module, where a module is just a set of mutually recursive top-level definitions, and illustrate the notions of: *module intrachecking* (each module is typechecked in isolation and its interface, which is the set of typings of the defined identifiers, is inferred); *interface interchecking* (when linking modules, typechecking is done just by looking at the interfaces); *interface specialization* (interface intrachecking may require to specialize the typing listed in the interfaces); *principal interfaces* (the principal typing property for the type system of modules); and *separate typechecking* (looking at the code of the modules does not provide more type information than looking at their interfaces). Then we illustrate some limitations of the "plain" framework and extend the module language and the type system in order to overcome these limitations. The decidability of the system is shown by providing algorithms for the fundamental operations involved in module intrachecking and interface interchecking.

Categories and Subject Descriptors

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

General Terms

Algorithms, Languages, Theory

[*]Partially supported by IST-2001-33477 DART and MIUR Cofin'01 NAPOLI projects.

Keywords

Type inference, Principal typings, Separate compilation

1. INTRODUCTION

The Damas/Milner type system [9] is the core of the type systems of modern functional programming languages, like ML [16] and Haskell. The fact that this type system is somewhat inflexible[1] has motivated the search for more expressive, but still decidable, type systems. Extension based on *intersection types* [5; 4; 6; 2] are particularly interesting since they generally have the *principal typing property* (a.k.a. *principal pair property*),[2] whose advantages w.r.t. the *principal type property* [3] of the ML type system has been described, for instance, in [14]. The fact that there is no reasonable way to define the principal typing property such that ML has it has been suspected for a long time, see e.g. [8]. Recently, Wells [19] gave a precise definition of principal typing property[4] and proved that ML does not have it. The system of *rank 2 intersection types* [15, 18, 20, 14, 11] seems to deserve particular attention since it is able to type all ML programs, has the principal typing property, decidable type inference, and complexity of type inference which is of the same order as in ML. 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, [13]) 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 $\mathsf{int} \rightarrow \mathsf{int}$ and $\mathsf{bool} \rightarrow \mathsf{bool}$, so it has type $(\mathsf{int} \rightarrow \mathsf{int}) \wedge (\mathsf{bool} \rightarrow \mathsf{bool})$. Rank 2 intersection types are types that may contain inter-

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

[2]Roughly speaking, a type system with judgement $A \vdash e : v$ (where A is a set of type assumptions for identifiers and v is a type), has the *principal typing property* if, whenever a term e is typable, there exist a *typing* $\langle A_0, v_0 \rangle$ (such that $A_0 \vdash e : v_0$ holds) "representing all possible typings for e".

[3]Roughly speaking, a type system has the *principal type property* if, whenever a term e is typable with the type assumptions A_0, there exists a type v_0 (such that $A_0 \vdash e : v_0$ holds) "representing all possible types for e in A_0".

[4]According to [19]: The typing $\langle A_1, v_1 \rangle$ is *stronger* than the typing $\langle A_2, v_2 \rangle$ iff, for every term e, $A_1 \vdash e : v_1$ implies $A_2 \vdash e : v_2$. A typing $\langle A, v \rangle$ is *principal* for a term e iff $A \vdash e : v$ holds and if $A' \vdash e : v'$ holds then $\langle A, v \rangle$ is stronger than $\langle A', v' \rangle$. The system $\vdash$ has the *principal typing property* if any typable term has a principal typing.

sections only to the left of a single arrow. So, for instance, $((\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{bool} \to \mathsf{bool})) \to \mathsf{int} \to \mathsf{int}$ is a rank 2 intersection type,[5] while $(((\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{bool} \to \mathsf{bool})) \to \mathsf{int}) \to \mathsf{int}$ is not a rank 2 intersection type.

Separate compilation allows to divide a large program into smaller modules, which can be typechecked and compiled in isolation. The complete program is closed, but modules may contain free identifiers (since may refer to other modules). As pointed out in Section 4 of [14], the principal typing property provides, among others things, elegant support for separate compilation, including *incremental type inference* [1] and *smartest recompilation* [17].

In this paper we propose a rank 2 intersection type system for a language of modules built on the small ML-like language considered in [11]. Separate compilation is intended as separate typechecking and separate code generation for modules. Following [3] we do not consider issues of code generation and focus on typechecking, which (at least from the point of view of language design) is the hardest part of separate compilation.

In the following we illustrate the framework by using the simple language of modules considered in [14], where a module is just a set of mutually recursive top-level definitions $\{x_1 = e_1, \ldots, x_n = e_n\}$ (note that, although module syntax is the same as record syntax in Standard ML [16], modules are not first-class values). The principal typing property of the rank 2 intersection type system for the core language allows to perform type inference on program fragments with free identifiers. For instance, the core language expression *tolist* 3 has principal typing $\langle\{\mathit{tolist} : \mathsf{int} \to \alpha\}, \alpha\rangle$ and the core language expression *tolist* true has principal typing $\langle\{\mathit{tolist} : \mathsf{bool} \to \beta\}, \beta\rangle$. We define the *interface* of a module to be a set of typings for the identifiers defined in the module. So, the (open) module

$$pm_1 = \{x = \mathit{tolist}\, 3,\ y = \mathit{tolist}\, \mathsf{true}\}$$

is typable (or, according to [3], *intrachecks*) with principal interface

$$\begin{array}{rcl} I_1 & = & \{x : \forall \alpha \,.\, \langle\{\mathit{tolist} : \mathsf{int} \to \alpha\}, \alpha\rangle, \\ & & \ y : \forall \beta \,.\, \langle\{\mathit{tolist} : \mathsf{bool} \to \beta\}, \beta\rangle\} \end{array}$$

specifying that the module pm_1 provides the definition for the identifiers x and y and, in order to produce a complete program (that is a closed module), it must be linked with a module providing a definition for the identifier *tolist* having a type that can be "unified" both with the type $\mathsf{int} \to \alpha$ and with the type $\mathsf{bool} \to \beta$. The (closed) module

$$pm_2 = \{\mathit{tolist} = \lambda\, z.\mathsf{cons}\ z\ \mathsf{nil}\},$$

which defines the identifier *tolist*, intrachecks with principal interface

$$I_2 = \{\mathit{tolist} : \forall \gamma \,.\, \langle\emptyset, \gamma \to \gamma\, \mathsf{list}\rangle\}.$$

Since the interface I_1 can be *specialized* to the interface

$$\begin{array}{rcl} I'_1 & = & \{x : \forall \epsilon \,.\, \langle\{\mathit{tolist} : \mathsf{int} \to \mathsf{int\, list}\}, \mathsf{int\, list}\rangle, \\ & & \ y : \forall \epsilon \,.\, \langle\{\mathit{tolist} : \mathsf{bool} \to \mathsf{bool\, list}\}, \mathsf{bool\, list}\rangle\} \end{array}$$

and $\forall \gamma \,.\, \gamma \to \gamma\, \mathsf{list}$ (the type scheme for *tolist* specified by I_2) can be instantiated to both $\mathsf{int} \to \mathsf{int\, list}$ and $\mathsf{bool} \to \mathsf{bool\, list}$ (the types for *tolist* required by I'_1) we have that the interfaces I_1 and I_2 *intercheck*. Their *composition*,

$$\begin{array}{rcl} I_1 \oplus I_2 & = & \{x : \forall \epsilon \,.\, \langle\emptyset, \mathsf{int\, list}\rangle,\ y : \forall \epsilon \,.\, \langle\emptyset, \mathsf{bool\, list}\rangle, \\ & & \ \mathit{tolist} : \forall \gamma \,.\, \langle\emptyset, \gamma \to \gamma\, \mathsf{list}\rangle\}, \end{array}$$

turns out to be an interface for the (closed) module

$$pm_1 \cup pm_2 = \{x = \mathit{tolist}\, 3,\ y = \mathit{tolist}\, \mathsf{true},\ \mathit{tolist} = \lambda\, z.\mathsf{cons}\ z\ \mathsf{nil}\}.$$

The various rank 2 intersection type systems for modules presented in this paper satisfy the following properties:

Principal interface property. If a module pm intrachecks, then it has a principal interface, that is, an interface I such that

1. every interface I' for pm is a specialization of I, and

2. every specialization I' of I is an interface for pm.

Separate type inference property. If pm_1 intrachecks with interface I_1 and pm_2 intrachecks with interface I_2, then

1. the interfaces I_1 and I_2 intercheck iff the module $pm_1 \cup pm_2$ intrachecks with interface $I_1 \oplus I_2$, and

2. I_i principal interface for pm_i $(1 \leq i \leq 2)$ imply that, if $pm_1 \cup pm_2$ intrachecks with interface $I_1 \oplus I_2$, then $I_1 \oplus I_2$ is a principal interface for $pm_1 \cup pm_2$.

The "plain" module system outlined above (which corresponds essentially to the separate compilation framework proposed in [14]) is simple and elegant, but has some limitations. A first problem is due to the fact that a module $\{x_1 = e_1, \ldots, x_n = e_n\}$ is typed by assuming simple types for the uses of the identifiers $x_1, \ldots, x_n$ in $e_1, \ldots, e_n$. This strategy limits the possibility of dividing a complex function into smaller functions, which can be reused in different contexts. For instance, the module

$$pm_3 = \{\mathit{twice} = \lambda\, f.\lambda\, x.f\ (f\ x),\ g = \mathit{twice}\, (\lambda\, z.\mathsf{cons}\ z\ \mathsf{nil})\}$$

does not intracheck, since the system tries to type it by assigning a simple type to the occurrence of the identifier *twice* in the body of the function g, while to type pm_3 it is necessary to assign to that occurrence of *twice* a rank 2 type of the form $((u \to u\, \mathsf{list}) \wedge (u\, \mathsf{list} \to u\, \mathsf{list\, list})) \to u \to u\, \mathsf{list\, list}$.

A second problem is due to the fact that modules are typed by assuming simple types for the references to other modules. This strategy limits the possibility of decomposing a program in modules. For instance, the modules

$$\begin{array}{rcl} pm_4 & = & \{\mathit{twice} = \lambda\, f.\lambda\, x.f\ (f\ x)\}\ \text{ and} \\ pm_5 & = & \{g = \mathit{twice}\, (\lambda\, z.\mathsf{cons}\ z\ \mathsf{nil})\} \end{array}$$

intracheck with principal interfaces $I_4 = \{\mathit{twice} : \forall\, \alpha_1\, \alpha_2\, \alpha_3 \,.\, \langle\emptyset, ((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3\rangle\}$ and $I_5 = \{g : \forall\, \delta_1\, \delta_2 \,.\, \langle\{\mathit{twice} : (\delta_1 \to \delta_1\, \mathsf{list}) \to \delta_2\}, \delta_2\rangle\}$, respectively, but these interfaces do not intercheck (since the type inferred for *twice* in pm_4, $((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3$, cannot be "unified" with the type assumed for the use of *twice* in pm_5, $(\delta_1 \to \delta_1\, \mathsf{list}) \to \delta_2$).

The two problems are related, since they both concern the (im)possibility of using the expressive power provided by rank 2 intersection types. In this paper we extend the "plain" module system in order to overcome these limitations. Our aim is to show that the technology of rank 2

[5] As usual, the arrow type constructor is right associative, so $((\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{bool} \to \mathsf{bool})) \to \mathsf{int} \to \mathsf{int}$ means $((\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{bool} \to \mathsf{bool})) \to (\mathsf{int} \to \mathsf{int})$.

intersection types, which brings the advantages of the principal typing property (see, e.g., [14, 19]), is "sufficiently mature" to be used in (improved versions of) the type system of languages like Standard ML, O'Caml and Haskell.

1.1 Organization of the paper

Section 2 of this paper recalls syntax, type system, and principal typings for the small ML-like language considered in [11], which is our core language. Section 3 introduces the algorithms for scheme instantiation and specialization, that are the fundamental operations involved in module intrachecking and interface interchecking. In order to simplify the exposition, we introduce our module language and type system incrementally: In Section 4 we consider modules to be just sets of mutually recursive top-level definitions (this is essentially the separate compilation framework proposed in [14]). Then we illustrate some limitations of the "plain" framework and propose, in Sections 5 and 6, two orthogonal extensions that allow to overcome such limitations. A further refinement of the system of Section 6 is described in Section 7. A version of this paper with conclusions and appendices ([10]) is available at the author's web page.

2. SYSTEM $\vdash^{\mathrm{Core}}$: RANK 2 INTERSECTION TYPES FOR THE CORE LANGUAGE

In this section we introduce the rank 2 intersection type system for the core language. We first present the syntax of the language (Section 2.1) and the syntax of our rank 2 intersection types together with other basic definitions that will be used in the rest of the paper (Section 2.2). Then we present the type system for the core language (Section 2.3) and state its principal typing property (Section 2.4).

2.1 Syntax

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 cs^n range over n-ary constructors ($n \neq 1$). The syntax of constants (ranged over by c) is as follows

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

Sometimes we will use pair as short for tuple^2.

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

$$\begin{array}{lcl}
e & ::= & x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{let}\, x = e_0 \,\mathsf{in}\, e \\
& \mid & \mathsf{rec}_i \,\{x_1 = e_1, \ldots, x_n = e_n\} \mid \mathsf{if}\, e_0 \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 \\
& \mid & \mathsf{match}\, e_0 \,\mathsf{with}\, \{pt_1 \Rightarrow e_1 [\!] \cdots [\!] pt_n \Rightarrow e_n\} \\
pt & ::= & x \mid cs^0 \mid cs^2 pt_1 pt_2 \mid cs^3 pt_1 pt_2 pt_3 \mid \cdots
\end{array}$$

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 $\mathrm{FV}(e)$.

2.2 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.

2.2.1 *Types and schemes*

We will be defining several classes of types. The set of *simple types* ($\mathbf{T}_0$), ranged over by u, is defined by the pseudo-grammar:

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

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 ($n \geq 2$).

$$d^0 \;::=\; \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \qquad d^1 \;::=\; \mathsf{list} \qquad d^n \;::=\; \times^n \;(n \geq 2)$$

For sake of readibility, we will write $u_1 \times \cdots \times u_n$ instead of $u_1 \cdots u_n \times^n$.

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

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

$$\begin{array}{lcll}
ui & ::= & u_1 \wedge \cdots \wedge u_n & \text{(rank 1 types)} \\
v & ::= & u \mid ui \rightarrow v & \text{(rank 2 types)} \\
vs & ::= & \forall \overrightarrow{\alpha}.v & \text{(rank 2 schemes)}
\end{array}$$

where u ranges over the set of simple types $\mathbf{T}_0$, $n \geq 1$, and $\overrightarrow{\alpha}$ is a finite (possibly empty) sequence of type variables $\alpha_1 \cdots \alpha_m$ ($m \geq 0$). Note that $\mathbf{T}_0 = \mathbf{T}_1 \cap \mathbf{T}_2$. Let ϵ denote the empty sequence, $\forall\epsilon.v$ is a legal expression, syntactically different from v, so that $\mathbf{T}_2 \cap \mathbf{T}_{\forall 2} = \emptyset$. The constructor $\wedge$ binds more tightly than $\rightarrow$, e.g., $u_1 \wedge u_2 \rightarrow u_3$ means $(u_1 \wedge u_2) \rightarrow u_3$.

Free and *bound* type variables are defined as usual. For every type $t \in \mathbf{T}_1 \cup \mathbf{T}_2 \cup \mathbf{T}_{\forall 2}$ let $\mathrm{FTV}(t)$ denote the set of free type variables of t. For every scheme $\forall\overrightarrow{\alpha}.v$ it is assumed that $\{\overrightarrow{\alpha}\} \subseteq \mathrm{FTV}(v)$. Moreover, schemes are considered equal modulo renaming of bound type variables. We say that a scheme vs is *closed* if $\mathrm{FTV}(vs) = \emptyset$. Let $\mathbf{T}_{\forall c2}$ be the *set of close rank 2 type schemes*, ranged over by *vcs*.

We consider $\wedge$ to be associative, commutative, and idempotent. So any type in $\mathbf{T}_1$ can be considered as a set of types in $\mathbf{T}_0$.

We assume a countable set $\mathbf{Tv}$ 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 $\mathbf{FTVR}(\mathbf{s}) = \cup_{\alpha \in \mathbf{Dom}(\mathbf{s})}\mathrm{FTV}(\mathbf{s}(\alpha))$. Substitutions will be denoted by $[\alpha_1 := u_1, \ldots, \alpha_n := u_n]$ ($n \geq 0$); the empty substitution will be denoted by $[\,]$.

The *composition* of two substitutions $\mathbf{s}_1$ and $\mathbf{s}_2$ is the substitution, denoted by $\mathbf{s}_1 \circ \mathbf{s}_2$, such that $\mathbf{s}_1 \circ \mathbf{s}_2(\alpha) = \mathbf{s}_1(\mathbf{s}_2(\alpha))$, for all type variables α. 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 $\mathbf{T}_{\forall 2}$ are closed under substitution.

The following definitions are fairly standard. Note that we keep a clear distinction between *subtyping relations* (between two types) and *instantiation relations* (between a scheme and a type). A third kind of relation, *specialization* (between two schemes), will be introduced in Section 2.2.3. All these three kinds of relation capture the fact that the type/scheme on the left is *stronger* (i.e., according to [19], it can be assigned to *less* expressions) than the type/scheme on the right.

Definition 1. (Subtyping relations $\leq_1$ and $\leq_2$). The subtyping relations $\leq_1$ ($\subseteq \mathbf{T}_1 \times \mathbf{T}_1$) and $\leq_2$ ($\subseteq \mathbf{T}_2 \times \mathbf{T}_2$) are defined by the rules in Fig. 1.[6]

The relations $\leq_1$ and $\leq_2$ are reflexive and transitive.

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

- $u \in \mathbf{T}_0$, let $\forall \overrightarrow{\alpha}.v \leq_{\forall 2,0} u$ to mean that $u = \mathbf{s}(v)$, for some substitution $\mathbf{s}$;
- $u_1 \wedge \cdots \wedge u_n \in \mathbf{T}_1$, let $\forall \overrightarrow{\alpha}.v \leq_{\forall 2,1} u_1 \wedge \cdots \wedge u_n$ to mean that $\forall \overrightarrow{\alpha}.v \leq_{\forall 2,0} u_i$, for every $i \in \{1, \ldots, n\}$.

Example 1. For $vs = \forall \alpha_1\alpha_2\alpha_3.((\alpha_1 \to \alpha_3) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_3$, we have (remember that $\wedge$ is idempotent):

- $vs \leq_{\forall 2,0} (\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ (by using the substitution $\mathbf{s}_1 = [\alpha_1, \alpha_2, \alpha_3 := \mathsf{int}]$), and
- $vs \leq_{\forall 2,1} ((\mathsf{int} \to \mathsf{int}) \to \mathsf{int}) \wedge ((\mathsf{bool} \to \mathsf{bool}) \to \mathsf{bool})$ (by $\mathbf{s}_1$ as above, and $\mathbf{s}_2 = [\alpha_1, \alpha_2, \alpha_3 := \mathsf{bool}]$).

Subtyping and instantiation are closed by substitution.

2.2.2 Environments and pair schemes

An *environment* T is a set $\{x_1 : t_1, \ldots, x_n : t_n\}$ 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 $\mathrm{Dom}(T)$ denotes the *domain* of T, which is the set $\{x_1, \ldots, x_n\}$. We write

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

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

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

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

Definition 3. (Rank 0 and rank 1 environments).

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

A *pair scheme* ps is a formula $\forall \overrightarrow{\alpha}.\langle A, v\rangle$ where A is a rank 1 environment, v is a rank 2 type, and $\overrightarrow{\alpha} = \mathrm{FTV}(A) \cup \mathrm{FTV}(v)$.

Definition 4. (Scheme environments, closed rank 2 environments, and pair scheme environments).

1. A *scheme environment* D is an environment $\{x_1 : t_1, \ldots, x_n : t_n\}$ of closed rank 2 type schemes and pair scheme assumptions for identifiers (that is, each t_i is either a closed rank 2 type scheme or a pair scheme) such that
$$\mathrm{Dom}(D) \cap \mathrm{FVR}(D) = \emptyset,$$
where the expression $\mathrm{FVR}(D)$ denotes the *set of identifiers occurring in the range of D*, which is the set $\cup_{x:\forall \overrightarrow{\alpha}.\langle A,v\rangle \in \mathrm{Dom}(D)} \mathrm{Dom}(A)$.
2. A *closed rank 2 environment* G is a scheme environment $\{x_1 : t_1, \ldots, x_n : t_n\}$ where each t_i ($1 \leq i \leq n$) is a closed rank 2 scheme.
3. A *pair scheme environment* L is a scheme environment $\{x_1 : t_1, \ldots, x_n : t_n\}$ where each t_i ($1 \leq i \leq n$) is a pair scheme.

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

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

and write $A_1 \leq_1 A_2$ to mean that

- $\mathrm{Dom}(A_1) = \mathrm{Dom}(A_2)$,[7] and
- for every assumption $x : ui_2 \in A_2$ there is an assumption $x : ui_1 \in A_1$ such that $ui_1 \leq_1 ui_2$.

Definition 5. (Subtyping relation $\leq_\bullet$). We write $\langle A, v\rangle \leq_\bullet \langle A', v'\rangle$ to mean that $v \leq_2 v'$ and $A' \leq_1 A$.

Definition 6. (Instantiation relation $\leq_{\forall\bullet,\bullet}$). We write $\forall \overrightarrow{\alpha}.\langle A, v\rangle \leq_{\forall\bullet,\bullet} \langle A', v'\rangle$ (to be read "the pair $\langle A', v'\rangle$ is an instance of the pair scheme $\forall \overrightarrow{\alpha}.\langle A, v\rangle$") if there is a substitution $\mathbf{s}$ such that $\mathrm{Dom}(\mathbf{s}) = \overrightarrow{\alpha}$ and $\mathbf{s}(\langle A, v\rangle) \leq_\bullet \langle A', v'\rangle$.

2.2.3 $\forall$-closure and scheme specialization

Given a type $v \in \mathbf{T}_2$ and a rank 1 environment A, we write

- $\mathrm{Close}(v)$ for the $\forall$-*closure of* v, i.e. for the scheme $\forall \overrightarrow{\alpha}.v$ where $\overrightarrow{\alpha} = \mathrm{FTV}(v)$, and
- $\mathrm{Close}(\langle A, v\rangle)$ for the $\forall$-*closure of* $\langle A, v\rangle$, i.e. for the scheme $\forall \overrightarrow{\alpha}.\langle A, v\rangle$ where $\overrightarrow{\alpha} = \mathrm{FTV}(A) \cup \mathrm{FTV}(v)$.

[7]The requirement $\mathrm{Dom}(A_1) = \mathrm{Dom}(A_2)$ 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 $\mathrm{Dom}(A_1) \supseteq \mathrm{Dom}(A_2)$. We have added such a requirement since it will simplify the presentation of the type inference system for the core language (in Section 2.3) and for modules (in Sections 4, 5, 6, and 7).

$$(\mathrm{Ref})\ \frac{u \in \mathbf{T}_0 \text{ and } u \text{ is not an arrow type}}{u \le_2 u} \qquad (\wedge)\ \frac{\{u_1,\ldots,u_n\} \supseteq \{u'_1,\ldots,u'_m\}}{u_1 \wedge\cdots\wedge u_n \le_1 u'_1 \wedge\cdots\wedge u'_m} \qquad (\rightarrow)\ \frac{ui' \le_1 ui \quad v \le_2 v'}{ui \rightarrow v \le_2 ui' \rightarrow v'}$$

Figure 1: Subtyping relations $\le_1$ and $\le_2$

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

Definition 7. (Scheme specialization relations $\le_{\forall\bullet}$ and $\le_{\forall c2}$).

1. We write $ps \le_{\forall\bullet} ps'$ (to be read "ps is a specialization of ps'") to mean that $ps' = \mathrm{Close}(\langle A', v'\rangle)$ and $ps \le_{\forall\bullet,\bullet} \langle A', v'\rangle$.
2. We write $vcs \le_{\forall c2} vcs'$ (to be read "vcs is a specialization of vcs'") to mean that $vcs = \mathrm{Close}(v)$, $vcs' = \mathrm{Close}(v')$, and $\mathrm{Close}(\langle\emptyset, v\rangle) \le_{\forall\bullet,\bullet} \langle\emptyset, v'\rangle$.

Example 2. We have

$$\forall\alpha_1\alpha_2\alpha_3.\langle\{f : (\alpha_1 \rightarrow \alpha_2) \wedge (\alpha_2 \rightarrow \alpha_3)\}, \alpha_1 \rightarrow \alpha_3\rangle \le_{\forall\bullet} \forall\beta.\langle\{f : (\beta \rightarrow \beta\,\mathsf{list}) \wedge (\beta\,\mathsf{list} \rightarrow \beta\,\mathsf{list}\,\mathsf{list})\}, \beta \rightarrow \beta\,\mathsf{list}\,\mathsf{list}\rangle$$

and

$$\forall\alpha_1\alpha_2\alpha_3.((\alpha_1 \rightarrow \alpha_2) \wedge (\alpha_2 \rightarrow \alpha_3)) \rightarrow \alpha_1 \rightarrow \alpha_3 \le_{\forall c2} \forall\beta.((\beta \rightarrow \beta\,\mathsf{list}) \wedge (\beta\,\mathsf{list} \rightarrow \beta\,\mathsf{list}\,\mathsf{list})) \rightarrow \beta \rightarrow \beta\,\mathsf{list}\,\mathsf{list}.$$

2.3 The type inference rules of $\vdash^{\mathrm{Core}}$

The type inference system $\vdash^{\mathrm{Core}}$ has judgements of the form

$$D; A \vdash^{\mathrm{Core}} e : v,$$

(to be read "e has $\vdash^{\mathrm{Core}}$-pair $\langle A, v\rangle$ w.r.t. D" or "$\langle A, v\rangle$ is a $\vdash^{\mathrm{Core}}$-pair for e w.r.t. D") where

- v is the rank 2 type inferred for e,
- D is a scheme environment specifying closed rank 2 schemes for the *globally defined* identifiers (i.e. the identifiers defined in the libraries available to the programmer) and pair schemes for the *locally defined* identifiers (i.e. the identifiers defined in the program by using the let construct), and
- A is a rank 1 environment containing the type assumptions for the free identifiers of e which are not in $\mathrm{Dom}(D)$.

In any valid judgement we have: $\mathrm{Dom}(D) \cap \mathrm{Dom}(A) = \emptyset$, $\mathrm{Dom}(A) = (\mathrm{FV}(e) - \mathrm{Dom}(D)) \cup \mathrm{FVR}(D|_{\mathrm{FV}(e)})$, and $\mathrm{FV}(e) = \mathrm{Dom}(D|_{\mathrm{FV}(e)}) \cup \mathrm{Dom}(A|_{\mathrm{FV}(e)})$.

System $\vdash^{\mathrm{Core}}$ is meant to be used to infer *pairs* (and not just *types*).[8] We call *undefined* the identifiers in $\mathrm{Dom}(A)$, since their definition is not available when typechecking the expression e.[9]

[8]A way to emphasize this aspect is to use typing judgements of the shape $D \vdash^{\mathrm{Core}} e : \langle A, v\rangle$. However, to make easier the comparison with other type systems, we adopt the more usual notation $D; A \vdash^{\mathrm{Core}} e : v$.

[9]The possibility of dealing with undefined identifiers will be a key ingredient of the type systems for modules presented in Sections 4, 5, 6, and 7.

In order to be able to understand the rest of this paper it is not necessary to know the detail of the typing rules of system $\vdash^{\mathrm{Core}}$. The rules are given in Appendix A of [10] (more detailed explanations, examples, and proofs can be found in [11]).

2.4 Principal typings for $\vdash^{\mathrm{Core}}$

Definition 8. (Principal pairs for $\vdash^{\mathrm{Core}}$). A $\vdash^{\mathrm{Core}}$-pair p for e is *principal* w.r.t. D if

- any pair for e w.r.t. D is an instance of $\mathrm{Close}(p)$, and
- any instance of $\mathrm{Close}(p)$ is a pair for e w.r.t. D.

It is worth mentioning that, when the environment D is fixed, *principal pairs* as defined above turns out to be *principal typings* in the sense of [19].[10]

If $\langle A, v\rangle$ is a principal pair for e w.r.t. D we say that $D; A \vdash^{\mathrm{Core}} e : v$ is a *principal typing judgement for* e *w.r.t.* D.

The type derivations of system $\vdash^{\mathrm{Core}}$ are closed by substitution (remember that $\mathrm{FTV}(D) = \emptyset$) and the system has the principal typing property.

Lemma 1. *(Substitutivity property for* $\vdash^{\mathrm{Core}}$*). If* $D; A \vdash^{\mathrm{Core}} e : v$*, then* $D; \mathbf{s}(A) \vdash^{\mathrm{Core}} e : \mathbf{s}(v)$*, for every substitution* $\mathbf{s}$.

Theorem 2.1. *(Principal pair property for* $\vdash^{\mathrm{Core}}$*). If* e *has a* $\vdash^{\mathrm{Core}}$*-pair w.r.t.* D*, then it has a principal pair w.r.t.* D.

3. ALGORITHMS FOR SCHEME INSTANTIATION AND SPECIALIZATION

The instantiation relation $\le_{\forall 2,1}$ (in Definition 2), the notion of $\le_{\forall 2,1}$-satisfaction problem (in Section 3.1), and the specialization relation $\le_{\forall c2}$ (in Definition 7) play a central role in the type systems for modules that will be presented in Sections 4, 5, 6, and 7. In this section we show that they are decidable.

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

3.1 Algorithms for the instantiation relation $\le_{\forall 2,1}$ and $\le_{\forall 2,1}$-satisfaction problems

In this section we give the notion of $\le_{\forall 2,1}$-satisfaction problem and state its decidability [14] (algorithms and proofs can be found, for instance, in [11]).

Following Jim [14] we say that a $\le_{\forall 2,1}$*-satisfaction problem* is a formula $\exists\overrightarrow{\alpha}.P$, where $\overrightarrow{\alpha}$ is a (possibly empty) sequence of type variables occurring free in P, and P is a set in which every element is

[10]Observe that the fact that $\mathrm{FTV}(D) = \emptyset$ makes Definition 8 substantially different from the notion of "A-principal typing" that can be found, for instance, in [12, 19].

- 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 $\mathbf{T}_{\forall 2}$ type and a $\mathbf{T}_1$ type.

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

- $\mathbf{s}(\alpha) = \mathbf{s}'(\alpha)$ for all $\alpha \notin \overrightarrow{\alpha}$,
- $\mathbf{s}'(vs) \leq_{\forall 2,1} \mathbf{s}'(ui)$ 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. (Most general solutions of a $\leq_{\forall 2,1}$-satisfaction problem).

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

THEOREM 3.1. *If a $\leq_{\forall 2,1}$-satisfaction problem is solvable, then there is a most general solution for it. In particular, there is an algorithm that decides, for any $\leq_{\forall 2,1}$-satisfaction problem, whether it is solvable, and, if so, returns a most general solution.*

3.2 Algorithms for the specialization relation $\leq_{\forall c2}$

The algorithms introduced in this section are a key component of both intracheking and interchecking algorithms for the systems $\vdash^{\mathrm{Dec}}$ (in Section 6) and $\vdash^{\mathrm{Bin}}$ (in Section 7). Note that they are not needed for the systems $\vdash^{\mathrm{Plain}}$ (in Section 4) and $\vdash^{\mathrm{Hid}}$ (in Section 5), since these systems do not use the $\leq_{\forall c2}$ relation.

For all i ($0 \leq i \leq 2$), the algorithm MS_i, defined in Fig. 2, takes a pair of types in $t, t' \in \mathbf{T}_i$ and returns a set of substitutions, $\mathrm{MS}_i(t, t')$, that we call the "set of *matching substitutions* on t against t'". The fundamental property of the algorithms MS_i is given by the following lemma, which can be proved by induction on the definition of MS_i.

LEMMA 2. *0.* $\mathrm{MS}_0(u, u') = \{\mathbf{s} \mid \mathrm{Dom}(\mathbf{s}) = \mathrm{FTV}(u) - \mathrm{FTV}(u')$ *and* $\mathbf{s}(u) = u'\}$.

1. $\mathrm{MS}_1(ui, ui') = \{\mathbf{s} \mid \mathrm{Dom}(\mathbf{s}) = \mathrm{FTV}(ui) - \mathrm{FTV}(ui')$ *and* $ui' \leq_1 \mathbf{s}(ui)\}$.

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

By observing that, for all rank 2 types v and v' such that $\mathrm{FTV}(v) \cap \mathrm{FTV}(v') = \emptyset$, $\mathrm{Close}(v) \leq_{\forall c2} \mathrm{Close}(v')$ if and only if $\mathrm{MS}_2(v, v') \neq \emptyset$, we have that Lemma 2 implies the following theorem.

THEOREM 3.2. *There is an algorithm that, given two closed rank 2 schemes vcs and vcs' decides whether the inequality $vcs \leq_{\forall c2} vcs'$ holds.*

The design of the algorithms MS_i has been influenced by the unification procedure proposed in [7].

4. SYSTEM $\vdash^{\mathrm{Plain}}$: RANK 2 INTERSECTION FOR PLAIN MODULES

In this section we introduce a rank 2 intersection type system for a simple language of modules. We first define the module syntax (Section 4.1). Then we present the type system (Section 4.2), its principal typing property (in Section 4.3), and its separate type inference property (in Section 4.4). The framework considered in this section is essentially the one proposed in [14]. In Sections 5, 6, and 7 we will increase the flexibility of the framework by extending both the module language and the type system.

4.1 Syntax

Plain modules (ranged over by pm) have the following syntax

$$pm \quad ::= \quad \{x_1 = e_1, \ldots, x_n = e_n\}$$

where $x_1, \ldots, x_n$ are distinct identifiers and $e_1, \ldots, e_n$ are expressions of the core language (see Section 2.1). The module pm is just a set of mutually recursive top-level definitions. The expression $\mathrm{Def}(pm)$ denotes the set $\{x_1, \ldots, x_n\}$ of the *identifiers defined by pm* and the expression $\mathrm{FV}(pm)$ denotes the set $(\cup_{i \in \{1,\ldots,n\}} \mathrm{FV}(e_i)) - \mathrm{Def}(pm)$ of the *free identifiers* of pm.

Let G be closed rank 2 environment specifying types for the library identifiers. We say that a module pm is *complete w.r.t. G* if $\mathrm{FV}(pm) \subseteq \mathrm{Dom}(G)$.

4.2 The type inference rules of $\vdash^{\mathrm{Plain}}$

The type inference system $\vdash^{\mathrm{Plain}}$ has judgements of the form

$$G \vdash^{\mathrm{Plain}} pm : I$$

(to be read "*pm* $\vdash^{\mathrm{Plain}}$*-intrachecks* with *interface I* w.r.t. G") where

- I is a pair scheme environment containing the pair schemes inferred for the identifiers in $\mathrm{Def}(pm)$, and
- G is a closed rank 2 environment specifying types for the library identifiers.

In any valid judgement we have: $\mathrm{Dom}(I) = \mathrm{Def}(pm)$, $\mathrm{Dom}(I) \cap \mathrm{Dom}(G) = \emptyset$, and $\mathrm{FVR}(I) = \mathrm{FV}(pm) - \mathrm{Dom}(G)$.

The (unique) type inference rule of system $\vdash^{\mathrm{Plain}}$ (in Figure 3) is based on the rule (REC2) of System $\vdash^{\mathrm{Core}}$ (in Fig. 10 of Appendix A of [10]).

Example 3. The modules

$$\begin{array}{rcl} pm_1 & = & \{x = \mathit{tolist}\, 3,\ y = \mathit{tolist}\, \mathsf{true}\} \text{ and} \\ pm_2 & = & \{\mathit{tolist} = \lambda\, z.\mathsf{cons}\ z\ \mathsf{nil}\}, \end{array}$$

introduced in Section 1, have (principal) typing judgements $\emptyset \vdash^{\mathrm{Plain}} pm_1 : I_1$ and $\emptyset \vdash^{\mathrm{Plain}} pm_2 : I_2$, where

$$\begin{array}{rcl} I_1 & = & \{x : \forall\, \alpha\, .\, \langle\{\mathit{tolist} : \mathsf{int} \rightarrow \alpha\}, \alpha\rangle, \\ & & \ y : \forall\, \beta\, .\, \langle\{\mathit{tolist} : \mathsf{bool} \rightarrow \beta\}, \beta\rangle\} \text{ and} \\ I_2 & = & \{\mathit{tolist} : \forall\, \gamma\, .\, \langle\emptyset, \gamma \rightarrow \gamma\ \mathsf{list}\rangle\}. \end{array}$$

4.3 Principal typings for $\vdash^{\mathrm{Plain}}$

Definition 10. (Interface specialization and principal interfaces for $\vdash^{\mathrm{Plain}}$).

$$
\begin{array}{rcl}
(\text{For all } i \in \{0,1,2\}) \;\; \mathrm{MS}_i(t,t') & = & \mathrm{MS}'_i(t,t',\mathrm{FTV}(t')), \;\; \text{where} \\[1em]
\mathrm{MS}'_0(u,u,W) & = & \{[\,]\} \\
\mathrm{MS}'_0(\alpha,u,W) & = & \{[\alpha := u]\}, \text{ if } \alpha \notin W \\
\mathrm{MS}'_0(u_1 \to u_2, u'_1 \to u'_2, W) & = & \{\mathbf{s}_2 \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in \mathrm{MS}'_0(u_1,u'_1,W) \;\text{ and }\; \mathbf{s}_2 \in \mathrm{MS}'_0(\mathbf{s}_1(u_2),u'_2,W)\} \\
\mathrm{MS}'_0(u_1\cdots u_n d^n, u'_1 \cdots u'_n d^n, W) & = & \{\mathbf{s}_n \circ \cdots \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in \mathrm{MS}'_0(u_1,u'_1,W), \mathbf{s}_2 \in \mathrm{MS}'_0(\mathbf{s}_1(u_2),u'_2,W), \ldots, \\
 & & \;\mathbf{s}_n \in \mathrm{MS}'_0(\mathbf{s}_{n-1} \circ \cdots \circ \mathbf{s}_1(u_n), u'_n, W)\} \\
\mathrm{MS}'_0(u,u',W) & = & \emptyset, \text{ otherwise} \\[1em]
\mathrm{MS}'_1(u_1 \wedge \cdots \wedge u_m, u'_1 \wedge \cdots \wedge u'_n, W) & = & \{\mathbf{s}_n \circ \cdots \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in \mathrm{MS}'_0(u_1,u'_{i_1},W), \mathbf{s}_2 \in \mathrm{MS}'_0(\mathbf{s}_1(u_2),u'_{i_2},W), \ldots, \\
 & & \;\mathbf{s}_n \in \mathrm{MS}'_0(\mathbf{s}_{n-1} \circ \cdots \circ \mathbf{s}_1(u_m), u'_{i_m}, W) \text{ and } \; i_1,\ldots,i_m \in \{1,\ldots,n\}\} \\[1em]
\mathrm{MS}'_2(u,u',W) & = & \mathrm{MS}'_0(u,u',W) \\
\mathrm{MS}'_2(\alpha, ui \to v, W) & = & \{\mathbf{s}_2 \circ \mathbf{s}_1 \circ [\alpha := \alpha_1 \to \alpha_2] \mid \mathbf{s}_1 \in \mathrm{MS}'_1(\alpha_1, ui, W) \;\text{ and }\; \mathbf{s}_2 \in \mathrm{MS}'_2(\alpha_2, v, W)\}, \\
 & & \;\text{if } \alpha \notin W, \text{ for some fresh } \alpha_1 \text{ and } \alpha_2 \\
\mathrm{MS}'_2(ui_1 \to v_2, ui'_1 \to v'_2, W) & = & \{\mathbf{s}_2 \circ \mathbf{s}_1 \mid \mathbf{s}_1 \in \mathrm{MS}'_1(ui_1, ui'_1, W) \;\text{ and }\; \mathbf{s}_2 \in \mathrm{MS}'_2(\mathbf{s}_1(v_2), v'_2, W)\} \\
\mathrm{MS}'_2(v,v',W) & = & \emptyset, \text{ otherwise}
\end{array}
$$

Figure 2: Algorithms MS_0**,** MS_1**, and** MS_2

$$
(\textsc{ModPlain}) \;\; \frac{(\text{for all } i \in \{1,\ldots,n\}) \;\; G; A_i \vdash^{\mathrm{Core}} e_i : v_i}{G \vdash^{\mathrm{Plain}} \{x_1 = e_1, \cdots, x_n = e_n\} : \{x_1 : ps_1, \ldots, x_n : ps_n\}}
$$
$$
\begin{array}{ll}
\text{provided} & A_1 + \cdots + A_n = ((A_1 + \cdots + A_n)|_{\mathrm{Dom}(A_1 + \cdots + A_n) - \{x_1,\ldots,x_n\}}), \, x_{j_1} : ui_{j_1}, \ldots, x_{j_m} : ui_{j_m}, \\
 & (\text{for all } j \in \{j_1,\ldots,j_m\}) \;\; \mathrm{Gen}(A_1 + \cdots + A_n, \, v_j) \leq_{\forall 2,1} ui_j, \quad \text{and} \\
 & (\text{for all } i \in \{1,\ldots,n\}) \;\; \mathrm{Close}(\langle A_i|_{\mathrm{Dom}(A_i) - \{x_1,\ldots,x_n\}}, v_i \rangle) \leq_{\forall \bullet} ps_i
\end{array}
$$

Figure 3: Typing rules for plain modules (system $\vdash^{\mathrm{Plain}}$**)**

1. An interface I' is a *specialization* of an interface I if $I' = \{x_i : ps'_i \mid i \in \{1,\ldots,n\}\}$, $I = \{x_i : ps_i \mid i \in \{1,\ldots,n\}\}$, and for all $i \in \{1,\ldots,n\}$ $ps_i \leq_{\forall\bullet} ps'_i$.

2. An interface I for pm is *principal* w.r.t. G if

   - any interface I' for pm w.r.t. G is a specialization of I, and
   - any specialization I' of I is an interface for pm w.r.t. G.

It is worth mentioning that, when the environment G is fixed, *principal interfaces* as defined above turns out to be *principal typings* in the sense of [19].

If I is a principal interface for pm w.r.t. G we say that $G \vdash^{\mathrm{Plain}} pm : I$ is a *principal typing judgement for pm w.r.t. G*.

The type system $\vdash^{\mathrm{Plain}}$ has the interface specialization property and the principal interface property.

Lemma 3. *(Interface specialization property for* $\vdash^{\mathrm{Plain}}$*). If* $G \vdash^{\mathrm{Plain}} pm : I$*, then* $G \vdash^{\mathrm{Plain}} pm : I'$*, for every specialization* I' *of* I*.*

Theorem 4.1. *(Principal interface property for* $\vdash^{\mathrm{Plain}}$*). If pm* $\vdash^{\mathrm{Plain}}$*-intrachecks w.r.t.* G*, then it has a principal interface w.r.t.* G*.*

4.4 Separate type inference for $\vdash^{\mathrm{Plain}}$

Definition 11. (Interface interchecking and composition for $\vdash^{\mathrm{Plain}}$).

1. The interfaces $I_1 = \{x_i : \forall\overrightarrow{\alpha^{(i)}}.\langle A_i, v_i\rangle \mid i \in \{1,\ldots,h\}\}$ and $I_2 = \{x_i : \forall\overrightarrow{\alpha^{(i)}}.\langle A_i, v_i\rangle \mid i \in \{h+1,\ldots,h+k\}\}$ $\vdash^{\mathrm{Plain}}$*-intercheck* if $\mathrm{Dom}(I_1) \cap \mathrm{Dom}(I_2) = \emptyset$ and the $\leq_{\forall 2,1}$-satisfaction problem $\pi = \exists\epsilon.\{\mathrm{Gen}(A_j, v_j) \leq ui_j \mid j \in J\}$, where

   - J is such that $A_1 + \cdots + A_{h+k} = ((A_1 + \cdots + A_{h+k})|_{\mathrm{Dom}(A_1 + \cdots + A_{h+k}) - \{x_1,\ldots,x_{h+k}\}}), \{x_j : ui_j \mid j \in J\}$, and
   - $\overrightarrow{\alpha^{(1)}}, \ldots, \overrightarrow{\alpha^{(h+k)}}$ are pairwise disjoint (note that this condition can always be satisfied since these type variables are bound),

   is solvable, and

2. if I_1 and I_2 $\vdash^{\mathrm{Plain}}$-intercheck, then the *composition* of I_1 and I_2 is the pair environment $I_1 \oplus I_2 = \{x_i : \mathrm{Close}(\langle \mathbf{s}(A_i|_{\mathrm{Dom}(A_i) - \{x_1,\ldots,x_{h+k}\}}), \mathbf{s}(v_i)\rangle) \mid i \in (J \cup L)\} \cup \{x_i : \forall\overrightarrow{\alpha^{(i)}}.\langle A_i, v_i\rangle \mid i \in (\{1,\ldots,h+k\} - (J \cup L))\}$, where $L = \{l \mid l \in \{1,\ldots,h+k\}$ and $\mathrm{Dom}(A_l) \cap \{x_1,\ldots,x_{h+k}\} \neq \emptyset\}$ and $\mathbf{s} \in \mathbf{MGS}(\pi)$.

Note that the composition operation for $\vdash^{\mathrm{Plain}}$-intercheckable interfaces ($\oplus$) is both commutative and associative.

Theorem 4.2. *(Separate type inference property for* $\vdash^{\mathrm{Plain}}$*). If* $G \vdash^{\mathrm{Plain}} pm_i : I_i$ $(1 \leq i \leq n)$*, then*

1. $I_1, \ldots, I_n$ $\vdash^{\mathrm{Plain}}$*-intercheck if and only if* $G \vdash^{\mathrm{Plain}} \cup_{i \in \{1,\ldots,n\}} pm_i : \oplus_{i \in \{1,\ldots,n\}} I_i$*, and*

2. *$G \vdash^{\text{Plain}} pm_i : I_i$ $(1 \leq i \leq n)$ principal w.r.t. G imply that if $G \vdash^{\text{Plain}} \cup_{i\in\{1,\ldots,n\}} pm_i : \oplus_{i\in\{1,\ldots,n\}} I_i$ holds then it is principal w.r.t. G.*

Example 4. Consider the principal typing judgements $\emptyset \vdash^{\text{Plain}} pm_1 : I_1$ and $\emptyset \vdash^{\text{Plain}} pm_2 : I_2$ of Example 3. We have $\emptyset \vdash^{\text{Plain}} pm_1 \cup pm_2 : I_1 \oplus I_2$, where

$$\begin{aligned} I_1 \oplus I_2 \quad = \quad & \{x : \forall\,\epsilon\,.\,\langle\emptyset, \mathsf{int\,list}\rangle,\ y : \forall\,\epsilon\,.\,\langle\emptyset, \mathsf{bool\,list}\rangle, \\ & \ \textit{tolist} : \forall\,\gamma\,.\,\langle\emptyset, \gamma \to \gamma\,\mathsf{list}\rangle\} \end{aligned}$$

Remark 1. (About automatic type inference). Replacing the inequality $\text{Close}(\langle A'_i, v_i\rangle) \leq_{\forall\bullet} ps_i$, in the last row of Figure 3, with the equality $\text{Close}(\langle A'_i, v_i\rangle) = ps_i$ does not reduce the expressive power of system $\vdash^{\text{Plain}}$. In fact the resulting system, $\vdash^{\text{Plain}'}$, is such that: if $G \vdash^{\text{Plain}} pm : I$ is principal w.r.t. G then $G \vdash^{\text{Plain}'} pm : I$. So, in order to be able to compute principal interfaces, we do not need an algorithm for the relation $\leq_{\forall\bullet}$. The same holds for the systems $\vdash^{\text{Hid}}$, $\vdash^{\text{Dec}}$, and $\vdash^{\text{Bin}}$ of Sections 5, 6, and 7.

5. SYSTEM $\vdash^{\text{Hid}}$: RANK 2 INTERSECTION FOR MODULES WITH HIDDEN DEFINITIONS

A plain module $\{x_1 = e_1, \ldots, x_n = e_n\}$ is just a set of mutually recursive definitions. This very simple structure has some drawbacks, as illustrated by the following example.

Example 5. The module

$$pm_3 = \{g = \textit{twice}\,(\lambda\, z.\mathsf{cons}\ z\ \mathsf{nil}),\ \textit{twice} = \lambda\, f.\lambda\, x.f\ (f\ x)\},$$

introduced in Section 1, cannot be typed by using $\vdash^{\text{Plain}}$. This is due to fact that system $\vdash^{\text{Plain}}$ forces to assign a rank 0 type to the occurrence of the identifier *twice* in the body of the function g, while to type pm_3 it is necessary to assign to that occurrence of *twice* a rank 2 type of the form $((u \to u\,\mathsf{list}) \wedge (u\,\mathsf{list} \to u\,\mathsf{list\,list})) \to u \to u\,\mathsf{list\,list}$.

In this section we extend the language of modules and the type system in order to overcome this problem. Indeed, a "brute force" strategy to solve the problem is already available: we can perform some "inlining" and replace pm_3 by the module

$$\begin{aligned} pm'_3 \quad = \quad & \{\textit{twice} = \lambda\, f.\lambda\, x.f\ (f\ x), \\ & \ g = (\lambda\, f.\lambda\, x.f\ (f\ x))\ (\lambda\, z.\mathsf{cons}\ z\ \mathsf{nil})\} \end{aligned}$$

or by the module

$$\begin{aligned} pm''_3 \quad = \quad & \{\textit{twice} = \lambda\, f.\lambda\, x.f\ (f\ x), \\ & \ g = \mathsf{let}\ h = \lambda\, f.\lambda\, x.f\ (f\ x)\ \mathsf{in}\ h\ (\lambda\, z.\mathsf{cons}\ z\ \mathsf{nil})\} \end{aligned}$$

having principal typing judgements $\emptyset \vdash^{\text{Plain}} pm'_3 : I_3$ and $\emptyset \vdash^{\text{Plain}} pm''_3 : I_3$, where $I_3 =$

$$\begin{aligned} & \{\textit{twice} : \forall\alpha_1\alpha_2\alpha_3.\langle\emptyset, ((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3\rangle, \\ & \ g : \forall\gamma.\langle\emptyset, \gamma \to \gamma\,\mathsf{list\,list}\rangle\}. \end{aligned}$$

We have qualified "brute force" the above strategy since, both in pm'_3 and in pm''_3, we have payed the price of duplicating the code of the function *twice* in the code of the function g. The extension proposed in this section avoids such duplication of code. For instance, it will allow to replace the module pm'_3 (or pm''_3) by the module

$$\begin{aligned} hm_3 \quad = \quad & \mathsf{hide}\ h = \lambda\, f.\lambda\, x.f\ (f\ x) \\ & \mathsf{define}\ \{\textit{twice} = h,\ g = h\ (\lambda\, z.\mathsf{cons}\ z\ \mathsf{nil})\}. \end{aligned}$$

We first extend the module syntax with top-level hidden (or private) definitions (Section 5.1). Then we present the new type system (Section 5.2) and discuss principal typings and separate type inference (in Section 5.3).

5.1 Syntax

Modules with hidden definitions (ranged over by hm) have the following syntax

$$hm \quad ::= \quad \mathsf{hide}\ x_1 = e_1, \ldots, x_q = e_q\ \mathsf{define}\ pm$$

where

- $x_1, \ldots, x_q$ are distinct identifiers,
- pm, called the *body* of hm, is a plain module such that $\text{Def}(pm) \cap \{x_1, \ldots, x_q\} = \emptyset$, and
- $e_1, \ldots, e_q$ are expressions of the core language such that, for all $j \in \{1, \ldots, q\}$, $(\text{FV}(e_1) \cup \cdots \cup \text{FV}(e_j)) \cap (\{x_j, \ldots, x_q\} \cup \text{Def}(pm)) = \emptyset$.

The module hm defines the identifiers in $\text{Def}(pm)$ by using the locally defined identifiers $x_1, \ldots, x_q$ that are visible only inside the module. According to this semantics we define $\text{Def}(hm) = \text{Def}(pm)$ to be the *set the identifiers defined by hm* and $\text{FV}(hm) = ((\cup_{i\in\{1,\ldots,q\}}\text{FV}(e_i)) \cup \text{FV}(pm)) - \{x_1, \ldots, x_q\}$ be the *set of free identifiers of hm.*

Let G be a closed rank 2 environment specifying types for the library identifiers. We say that a module hm is *complete w.r.t. G* if $\text{FV}(hm) \subseteq \text{Dom}(G)$.

5.2 The type inference rules of $\vdash^{\text{Hid}}$

The type inference system $\vdash^{\text{Hid}}$ has judgements of the form

$$G \vdash^{\text{Hid}} hm : I$$

(to be read "hm $\vdash^{\text{Hid}}$-intrachecks with interface I w.r.t. G") where I and G obey to the same restrictions listed for $\vdash^{\text{Plain}}$ (see Section 4.2). The (unique) type inference rule of system $\vdash^{\text{Hid}}$ (in Figure 4) is based on rules (LetPs) and (Rec2) of System $\vdash^{\text{Core}}$ (in Fig. 10 of Appendix A of [10]).

Example 6. The module hm_3 introduced at the end of Example 5 has (principal) typing judgement $\emptyset \vdash^{\text{Hid}} hm_3 : I_3$, where I_3 is the interface introduced in Example 5.

5.3 Principal typings and separate type inference for $\vdash^{\text{Hid}}$

The principal typing and the separate type inference property for $\vdash^{\text{Hid}}$ follows straightforwardly from the corresponding properties of $\vdash^{\text{Plain}}$ by the following lemma.

Lemma 4. *Let $hm = \mathsf{hide}\ x_1 = e_1, \ldots, x_q = e_q\ \mathsf{define}\ pm$ be a module with hidden definitions. Then $G \vdash^{\text{Hid}} hm : I$ if and only if $G \vdash^{\text{Plain}} pm[x_q := e_q] \cdots [x_1 := e_1] : I$.*

6. SYSTEM $\vdash^{\text{Dec}}$: RANK 2 INTERSECTION FOR MODULES WITH EXPLICIT DECLARATIONS

The language of modules considered in Sections 4 and 5 forces to assign a rank 0 type to each use of an external identifier. This strategy limits the possibility of dividing a program into modules, as illustrated by the following example.

$$\text{(ModHid)}\ \frac{\begin{array}{l} G; A'_1 \vdash^{\text{Core}} e'_1 : v'_1 \quad \cdots \quad G, y_1 : \text{Close}(\langle A'_1, v'_1\rangle), \ldots, y_{q-1} : \text{Close}(\langle A'_{q-1}, v'_{q-1}\rangle); A'_q \vdash^{\text{Core}} e'_q : v'_q \\ (\text{for all } i \in \{1,\ldots,n\})\ \ G, y_1 : \text{Close}(\langle A'_1, v'_1\rangle), \ldots, y_q : \text{Close}(\langle A'_q, v'_q\rangle); A_i \vdash^{\text{Core}} e_i : v_i \end{array}}{G \ \ \vdash^{\text{Hid}} \ \mathsf{hide}\ y_1 = e'_1, \cdots, y_q = e'_q\ \mathsf{define}\ \{x_1 = e_1, \cdots, x_n = e_n\} : \{x_1 : ps_1, \ldots, x_n : ps_n\}}$$

$$\begin{array}{ll} \text{provided} & A_1 + \cdots + A_n = ((A_1 + \cdots + A_n)|_{\text{Dom}(A_1+\cdots+A_n)-\{x_1,\ldots,x_n\}}), x_{j_1} : ui_{j_1}, \ldots, x_{j_m} : ui_{j_m}, \\ & (\text{for all } j \in \{j_1,\ldots,j_m\})\ \ \text{Gen}(A_1 + \cdots + A_n,\ v_j) \leq_{\forall 2,1} ui_j, \quad \text{and} \\ & (\text{for all } i \in \{1,\ldots,n\})\ \ \text{Close}(\langle A_i|_{\text{Dom}(A)-\{x_1,\ldots,x_n\}}, v_i\rangle) \leq_{\forall\bullet} ps_i \end{array}$$

Figure 4: Typing rules for modules with hidden definitions (system $\vdash^{\text{Hid}}$)

Example 7. System $\vdash^{\text{Plain}}$ does not allow to decompose the module pm'_3 or the module pm''_3 of Example 5 into the modules

$$\begin{array}{rcl} pm_4 & = & \{twice = \lambda f.\lambda x.f\ (f\ x)\} \ \text{ and} \\ pm_5 & = & \{g = twice\ (\lambda z.\mathsf{cons}\ z\ \mathsf{nil})\} \end{array}$$

introduced in Section 1 (the same holds for system $\vdash^{\text{Hid}}$ and the module hm_3 of Example 5). In fact, we have the principal typing judgements $\emptyset \vdash^{\text{Plain}} pm_4 : I_4$ and $\emptyset \vdash^{\text{Plain}} pm_5 : I_5$, where the interfaces $I_4 = \{twice : \forall\, \alpha_1\, \alpha_2\, \alpha_3 \,.\, \langle\emptyset, ((\alpha_1 \to \alpha_2)\wedge(\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3\rangle\}$ and $I_5 = \{g : \forall\, \delta_1\, \delta_2 \,.\, \langle\{twice : (\delta_1 \to \delta_1\ \mathsf{list}) \to \delta_2\}, \delta_2\rangle\}$ do not $\vdash^{\text{Plain}}$-intercheck.

The extension proposed in this section allows to declare closed rank 2 schemes for external identifiers. For instance, it will be possible to replace the modules pm_4 and pm_5 with the modules

$$\begin{array}{rcl} dm_4 & = & \mathsf{declare}\ \{\,\}\ \mathsf{define}\ \{twice = \lambda f.\lambda x.f\ (f\ x)\} \ \text{ and} \\ dm_5 & = & \mathsf{declare}\ E_5\ \mathsf{define}\ \{g = twice\ (\lambda z.\mathsf{cons}\ z\ \mathsf{nil})\}, \end{array}$$

where the environment

$$\begin{array}{rcl} E_5 & = & \{twice : \forall\, \alpha_1\, \alpha_2\, \alpha_3 \,. \\ & & \quad ((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3\} \end{array}$$

contains the type declaration for the identifier *twice*.

We first extend the module syntax (Section 6.1). Then we present type system (Section 6.2), principal typings (Section 6.3), and separate type inference (Section 6.4). To simplify the presentation we do not consider hidden definitions (introduced in Section 5), but they can be added without problems.

6.1 Syntax

Modules with explicit assumptions (ranged over by dm) have the following syntax

$$dm \ ::= \ \mathsf{declare}\ E\ \mathsf{define}\ pm$$

where

- E, called the *declaration part* of dm, is a closed rank 2 environment declaring the type of some of the identifiers used in pm, and
- pm, called the *body* of dm, is a plain module such that $\text{Dom}(E) \subseteq \text{Def}(pm) \cup \text{FV}(pm)$.

The plain module $pm = \{x_1 = e_1, \ldots, x_n = e_n\}$ is typed by using the closed rank 2 schemes in E to type the occurrences in $e_1, \ldots, e_n$ of the identifiers in $\text{Dom}(E)$. According to this semantics we define $\text{Def}(dm) = \text{Def}(pm)$ to be the *set the identifiers defined by* dm and $\text{FV}(dm) = \text{FV}(pm) - \text{Dom}(E)$ to be the *set of free identifiers of* dm.

Let G be a closed rank 2 environment specifying types for the library identifiers. We say that a module $dm = \mathsf{declare}\ E$ $\mathsf{define}\ pm$ is *complete w.r.t.* G if $\text{FV}(dm) \subseteq \text{Dom}(G)$ and $\text{Dom}(E) \subseteq \text{Def}(pm)$.

6.2 The type inference rules of $\vdash^{\text{Dec}}$

The type inference system $\vdash^{\text{Dec}}$ has judgements of the form

$$G \vdash^{\text{Dec}} dm : \langle E, I\rangle$$

(to be read "dm $\vdash^{\text{Dec}}$-intrachecks with *declaration interface* E and *definition interface* I w.r.t. G") where

- E is the declaration part of dm (that is, the closed rank 2 environment occurring between the keywords $\mathsf{declare}$ and define),
- I is a pair scheme environment containing the pair schemes inferred for the identifiers in $\text{Def}(dm)$, and
- G is a closed rank 2 environment specifying types for the library identifiers.

In any valid judgement we have: $\text{Dom}(I) = \text{Def}(dm)$, $(\text{Dom}(I) \cup \text{Dom}(E)) \cap \text{Dom}(G) = \emptyset$, and $\text{FVR}(I) = \text{FV}(dm) - \text{Dom}(G)$.

The (unique) type inference rule of system $\vdash^{\text{Dec}}$ is given in Figure 5.

Example 8. The modules dm_4 and dm_5 introduced at the end of Example 7 have (principal) typing judgements $\emptyset \vdash^{\text{Dec}} dm_4 : \langle\emptyset, I_4\rangle$ and $\emptyset \vdash^{\text{Dec}} dm_5 : \langle E_5, I'_5\rangle$, where

I_4	is	the interface introduced in Example 7,
E_5	is	the environment introduced in Example 7, and
I'_5	$=$	$\{g : \forall\,\gamma \,.\, \langle\emptyset, \gamma \to \gamma\ \mathsf{list}\ \mathsf{list}\rangle\}$.

6.3 Principal typings for $\vdash^{\text{Dec}}$

The type system $\vdash^{\text{Dec}}$ has the definition interface specialization property and the principal definition interface property.

Lemma 5. *(Definition interface specialization property for $\vdash^{\text{Dec}}$). If $G \vdash^{\text{Dec}} dm : \langle E, I\rangle$, then $G \vdash^{\text{Dec}} dm : \langle E, I'\rangle$, for every specialization I' of I.*

Theorem 6.1. *(Principal definition interface property for $\vdash^{\text{Dec}}$). If dm $\vdash^{\text{Dec}}$-intrachecks w.r.t. G, then it has a principal definition interface w.r.t. G.*

6.4 Separate type inference for $\vdash^{\text{Dec}}$

Definition 12. (Interface interchecking and composition for $\vdash^{\text{Dec}}$).

1. The interfaces $\langle E_1, I_1\rangle$ and $\langle E_2, I_2\rangle$, where

$$\begin{array}{rcl} E_1 & = & \{y_j : vcs_j \mid j \in \{1, \ldots, k_1\}\}, \\ I_1 & = & \{x_i : \forall\overrightarrow{\alpha^{(i)}}.\langle A_i, v_i\rangle \mid i \in \{1, \ldots, h_1\}\}, \\ E_2 & = & \{y_j : vcs_j \mid j \in \{k_1 + 1, \ldots, k_1 + k_2\}\}, \ \text{ and} \\ I_2 & = & \{x_i : \forall\overrightarrow{\alpha^{(i)}}.\langle A_i, v_i\rangle \mid i \in \{h_1 + 1, \ldots, h_1 + h_2\}\}, \end{array}$$

$$(\textsc{ModDec})\ \frac{(\text{for all } i\in\{1,\ldots,n\})\ \ G,E;A_i \vdash^{\mathrm{Core}} e_i : v_i}{G\vdash^{\mathrm{Dec}} \mathsf{declare}\ E\ \mathsf{define}\ \{x_1=e_1,\cdots,x_n=e_n\} : \langle E,\{x_1:ps_1,\ldots,x_n:ps_n\}\rangle}$$

provided $A_1+\cdots+A_n = ((A_1+\cdots+A_n)|_{\mathrm{Dom}(A_1+\cdots+A_n)-\{x_1,\ldots,x_n\}}),\ x_{j_1}:ui_{j_1},\ldots,x_{j_m}:ui_{j_m}$,
(for all $j\in\{j_1,\ldots,j_m\}$) $\mathrm{Gen}(A_1+\cdots+A_n,\,v_j)\le_{\forall 2,1} ui_j$,
(for all $i\in\{1,\ldots,n\}$) $\mathrm{Close}(\langle A_i|_{\mathrm{Dom}(A_i)-\{x_1,\ldots,x_n\}},v_i\rangle)\le_{\forall\bullet} ps_i$, and
(for all $x_l:\forall\overrightarrow{\alpha^{(l)}}.\langle\emptyset,v_l\rangle\in\{x_1:ps_1,\ldots,x_n:ps_n\}$) $x_l:vcs_l\in E$ implies $\forall\overrightarrow{\alpha^{(l)}}.v_l\le_{\forall c2} vcs_l$

Figure 5: Typing rules for modules with explicit declarations (system $\vdash^{\mathrm{Dec}}$)

$\vdash^{\mathrm{Dec}}$-*intercheck* if

(a) $y:vcs'\in E_1$ and $y:vcs''\in E_2$ imply $vcs'=vcs''$,
(b) $(\mathrm{Dom}(I_1)\cup\mathrm{FVR}(I_1))\cap(\mathrm{Dom}(E_2)-\mathrm{Dom}(E_1))=\emptyset$ and $(\mathrm{Dom}(I_2)\cup\mathrm{FVR}(I_2))\cap(\mathrm{Dom}(E_1)-\mathrm{Dom}(E_2))=\emptyset$,
(c) I_1 and I_2 $\vdash^{\mathrm{Plain}}$-intercheck, and
(d) for all $z:\forall\overrightarrow{\gamma}.\langle\emptyset,v\rangle\in I_1\oplus I_2$, if $z:vcs\in E_1\cup E_2$ then $\forall\overrightarrow{\gamma}.v\le_{\forall c2} vcs$.

2. The *composition* of two interfaces $\langle E_1,I_1\rangle$ and $\langle E_2,I_2\rangle$ that $\vdash^{\mathrm{Dec}}$-intercheck is the interface $\langle E_1,I_1\rangle\oplus\langle E_2,I_2\rangle=\langle E_1\cup E_2,I_1\oplus I_2\rangle$.

Note that the composition operation for $\vdash^{\mathrm{Dec}}$-intercheckable interfaces ($\oplus$) is both commutative and associative.

THEOREM 6.2. *(Separate type inference property for $\vdash^{\mathrm{Dec}}$). If $G\vdash^{\mathrm{Dec}}$ declare E_i define $pm_i:\langle E_i,I_i\rangle$ $(1\le i\le n)$, then*

1. $\langle E_1,I_1\rangle,\ldots,\langle E_n,I_n\rangle$ $\vdash^{\mathrm{Dec}}$-intercheck if and only if $G\vdash^{\mathrm{Dec}}$ declare $\cup_{i\in\{1,\ldots,n\}}E_i$ define $\cup_{i\in\{1,\ldots,n\}}pm_i : \oplus_{i\in\{1,\ldots,n\}}\langle E_i,I_i\rangle$, and

2. $G\vdash^{\mathrm{Dec}}$ declare E_i define $pm_i:\langle E_i,I_i\rangle$ $(1\le i\le n)$ principal w.r.t. G imply that if $G\vdash^{\mathrm{Dec}}$ declare $\cup_{i\in\{1,\ldots,n\}}E_i$ define $\cup_{i\in\{1,\ldots,n\}}pm_i:\oplus_{i\in\{1,\ldots,n\}}\langle E_i,I_i\rangle$ holds then it is principal w.r.t. G.

Example 9. Explicit declarations allow to overcome the limitation illustrated at the beginning of the section: the module pm_3' (or pm_3'') introduced in Example 5 can be divided into the modules dm_4 and dm_5 (introduced at the end of Example 7) that have principal typing judgements $\emptyset\vdash^{\mathrm{Dec}} dm_4:\langle\emptyset,I_4\rangle$ and $\emptyset\vdash^{\mathrm{Dec}} dm_5:\langle E_5,I_5'\rangle$ (see Example 8) such that the principal typing judgement $\emptyset\vdash^{\mathrm{Dec}} dm_{4,5}:\langle E_5,I_4\oplus I_5'\rangle$, where

$$\begin{array}{lcl} E_5 & \text{is} & \text{the environment introduced in Example 7,}\\ dm_{4,5} & = & \mathsf{declare}\ E_5\\ & & \mathsf{define}\ \{twice=\lambda f.\lambda x.f\ (f\ x),\\ & & \qquad g=twice\,(\lambda z.\mathsf{cons}\ z\ \mathsf{nil})\},\ \text{and}\\ I_4\oplus I_5' & = & I_3\ \text{(the interface introduced in Example 5),}\end{array}$$

holds.

The declaration interface of a module is not required to specify the principal closed rank 2 schemes for the declared identifiers. For instance, the system $\vdash^{\mathrm{Dec}}$ allows to decompose pm_3' (or pm_3'') into the modules dm_4 (as above) and

$$dm_5' \;=\; \mathsf{declare}\ E_5'\ \mathsf{define}\ \{g=twice\,(\lambda z.\mathsf{cons}\ z\ \mathsf{nil})\}$$

that has principal typing judgement $\emptyset\vdash^{\mathrm{Dec}} dm_5':\langle E_5',I_5'\rangle$, where $E_5'=$ $\{twice:\forall\delta.((\delta\to\delta\,\mathsf{list})\wedge(\delta\,\mathsf{list}\to\delta\,\mathsf{list}\,\mathsf{list}))\to\delta\to\delta\,\mathsf{list}\,\mathsf{list}\}$ and I_5' is the principal definition interface of dm_5 (see Example 8).

7. SYSTEM $\vdash^{\mathrm{Bin}}$: RANK 2 INTERSECTION FOR MODULES WITH BOUNDED DECLARATIONS

The module system of Section 6 forces to use the same declaration (possibly no declaration) for all the uses of an identifier in different modules. This strategy is indeed unnecessarily restrictive, as shown by the following example.

Example 10. Consider the principal typing judgements $\emptyset\vdash^{\mathrm{Dec}} dm_4:\langle\emptyset,I_4\rangle$ and $\emptyset\vdash^{\mathrm{Dec}} dm_5:\langle E_5,I_5'\rangle$ of Example 8 and the module

$$dm_6 \;=\; \mathsf{declare}\ \{\,\}\ \mathsf{define}\ \{h=twice\,(\lambda w.w)\}$$

that has principal typing judgement $\emptyset\vdash^{\mathrm{Dec}} dm_6:\langle\emptyset,I_6\rangle$, where

$$I_6=\{h:\forall\alpha_1\,\alpha_2\,.\langle\{twice:(\alpha_1\to\alpha_1)\to\alpha_2\},\alpha_2\rangle\}.$$

We have that $\langle\emptyset,I_4\rangle$ and $\langle E_5,I_5'\rangle$ $\vdash^{\mathrm{Dec}}$-intercheck (see Example 9), $\langle\emptyset,I_4\rangle$ and $\langle\emptyset,I_6\rangle$ $\vdash^{\mathrm{Dec}}$-intercheck (the check is straightforward), and $\langle E_5,I_5'\rangle$ and $\langle\emptyset,I_6\rangle$ do not $\vdash^{\mathrm{Dec}}$-intercheck (since the requirement (b) in Definition 12.1 is not satisfied), so the three modules cannot be combined.

Similarly, if we replace the module dm_6 with the module

$$dm_7 \;=\; \mathsf{declare}\ E_7\ \mathsf{define}\ \{r=twice\,(\lambda z.\mathsf{pair}\ z\ 3)\}$$

which has principal typing judgement $\emptyset\vdash^{\mathrm{Dec}} dm_7:\langle E_7,I_7\rangle$, where

$$\begin{array}{lcl} E_7 & = & \{twice:\forall\gamma.\\ & & \quad((\gamma\to(\gamma\times\mathsf{int}))\wedge((\gamma\times\mathsf{int})\to((\gamma\times\mathsf{int})\times\mathsf{int})))\\ & & \quad\to\gamma\to((\gamma\times\mathsf{int})\times\mathsf{int}))\}\quad\text{and}\\ I_7 & = & \{r:\forall\beta\,.\langle\emptyset,\beta\to((\beta\times\mathsf{int})\times\mathsf{int})\rangle\},\end{array}$$

we have that the requirement (a) in Definition 12.1 is not satisfied.

In this section, in order to be able to combine the modules considered in the above example, we introduce a "lower level" language of modules, the language of *modules with bounded declarations*. Indeed, a module with explicit declarations

$$\begin{array}{l}\mathsf{declare}\ \{y_1:vcs_1,\ldots,y_m:vcs_m\}\\ \mathsf{define}\ \{x_1=e_1,\cdots,x_n=e_n\}\end{array}$$

should be consider just as a "syntactic sugared" version for the module with bounded declarations

$$\begin{array}{l}\mathsf{bind}\ \{y_1\ :\ \{y_1':vcs_1\},\ldots,y_m\ :\ \{y_m':vcs_m\}\}\\ \mathsf{define}\ \{x_1=e_1,\cdots,x_n=e_n\}[y_1:=y_1',\ldots,y_1:=y_m']\end{array}$$

where the (bounded) identifiers $y_1',\ldots,y_m'$ are fresh.

7.1 Syntax

A *binding environment* B is a set $\{x_1 : G_1, \ldots, x_q : G_q\}$ where

- $x_1, \ldots, x_q$ are distinct identifiers, and
- $G_1, \ldots, G_q$ are non empty closed rank 2 environments such that $\mathrm{Dom}(G_1), \ldots, \mathrm{Dom}(G_q)$, and $\{x_1, \ldots, x_q\}$ are pairwise disjoint.

The binding environment B declares that, for all $i \in \{1, \ldots, q\}$, the (bounded) identifiers in $\mathrm{Dom}(G_i)$ are "aliases" for the identifier x_i. The expression $\mathrm{Dom}(B)$ denotes the *domain of* B, which is the set $\{x_1, \ldots, x_q\}$, and the expression $\mathrm{Env}(B)$ denotes the closed rank 2 environment $\cup_{i\in\{1,\ldots,q\}} G_i$.

Modules with bounded declarations (ranged over by bm) have the following syntax

$$bm \quad ::= \quad \mathsf{bind}\ B\ \mathsf{define}\ pm$$

where

- B, called the *binding part* of bm, is a binding environment such that the closed rank 2 environment $\mathrm{Env}(B)$ declares the type of some of the identifiers used in pm, and
- pm, called the *body* of bm, is a plain module such that $\mathrm{Dom}(B) \cap \mathrm{FV}(pm) = \emptyset$ and $\mathrm{Dom}(\mathrm{Env}(B)) \subseteq \mathrm{Def}(pm) \cup \mathrm{FV}(pm)$.

The keyword bind binds the identifiers of $\mathrm{Dom}(\mathrm{Env}(B))$ in pm. The plain module $pm = \{x_1 = e_1, \ldots, x_n = e_n\}$ is typed by using the closed rank 2 schemes in $\mathrm{Env}(B)$ to type the occurrences in $e_1, \ldots, e_n$ of the identifiers in $\mathrm{Dom}(\mathrm{Env}(B))$. According to this semantics we define $\mathrm{Def}(bm) = \mathrm{Def}(pm)$ to be the *set the identifiers defined by bm* and $\mathrm{FV}(bm) = \mathrm{FV}(pm) - \mathrm{Dom}(\mathrm{Env}(B))$ to be the *set of free identifiers of bm*.

Let G be a closed rank 2 environment specifying assumption for the library identifiers. We say that a module $bm = \mathsf{bind}\ B\ \mathsf{define}\ pm$ is *complete w.r.t. G* if $\mathrm{FV}(bm) \subseteq \mathrm{FV}(G)$ and $\mathrm{Dom}(\mathrm{Env}(B)) \subseteq \mathrm{Def}(pm)$.

7.2 The type inference rules of $\vdash^{\mathrm{Bin}}$

The type inference system $\vdash^{\mathrm{Bin}}$ has judgements of the form

$$G \vdash^{\mathrm{Bin}} bm : \langle B_0, I\rangle$$

(to be read "bm $\vdash^{\mathrm{Bin}}$-intrachecks with binding interface B_0 and definition interface I w.r.t. G") where

- B_0 is the subset of the binding part of bm (that is, the binding environment occurring between the keywords bind and define) containing only the declarations for the identifiers with an incomplete definition (i.e, either identifiers defined in other modules or identifiers defined in bm with a definition containing identifiers in $\mathrm{FV}(bm)$),
- I is a pair scheme environment containing the pair schemes inferred for the identifiers in $\mathrm{Def}(bm)$, and
- G is a closed rank 2 environment specifying types for the library identifiers.

In any valid judgement $G \vdash^{\mathrm{Bin}} \mathsf{bind}\ B\ \mathsf{define}\ pm : \langle B_0, I\rangle$ we have: $\mathrm{Dom}(I) = \mathrm{Def}(bm)$, $B_0 = B - \{z : G \in B \mid z : \forall\overrightarrow{\alpha}.\langle\emptyset, v\rangle \in I$ for some $\overrightarrow{\alpha}, v\}$, $(\mathrm{Dom}(I) \cup \mathrm{Dom}(B_0)) \cap \mathrm{Dom}(G) = \emptyset$, and $\mathrm{FVR}(I) = \mathrm{FV}(bm) - \mathrm{Dom}(G)$.

The (unique) type inference rule of system $\vdash^{\mathrm{Bin}}$ (in Figure 6) is based on the rule for System $\vdash^{\mathrm{Dec}}$ (in Figure 5).

Example 11. The "de-sugared versions" of the modules dm_4, dm_5, dm_6, and dm_7 of Example 10:

$$\begin{array}{lcl} bm_4 & = & \mathsf{bind}\ \{\,\}\ \mathsf{define}\ \{twice = \lambda f.\lambda x.f\ (f\ x)\},\\ bm_5 & = & \mathsf{bind}\ B_5\ \mathsf{define}\ \{g = twice_5\ (\lambda z.\mathsf{cons}\ z\ \mathsf{nil})\},\\ bm_6 & = & \mathsf{declare}\ \{\,\}\ \mathsf{define}\ \{h = twice\ (\lambda w.w)\},\ \text{ and}\\ bm_7 & = & \mathsf{declare}\ B_7\ \mathsf{define}\ \{r = twice_7\ (\lambda z.\mathsf{pair}\ z\ 3)\}, \end{array}$$

have (principal) typing judgements $\emptyset \vdash^{\mathrm{Bin}} bm_4 : \langle\emptyset, I_4\rangle$, $\emptyset \vdash^{\mathrm{Bin}} bm_5 : \langle B_5, I'_5\rangle$, $\emptyset \vdash^{\mathrm{Bin}} bm_6 : \langle\emptyset, I_6\rangle$, and $\emptyset \vdash^{\mathrm{Bin}} bm_7 : \langle B_7, I_7\rangle$, where

$$\begin{array}{lcl} I_4 & \text{is} & \text{the interface introduced in Example 7,}\\ B_5 & = & \{twice : \{twice_5 : \forall\alpha_1\,\alpha_2\,\alpha_3\,.\\ & & \quad ((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3\}\},\\ I'_5 & \text{is} & \text{the interface introduced in Example 8,}\\ I_6 & \text{is} & \text{the interface introduced in Example 10,}\\ B_7 & = & \{twice : \{twice_7 : \forall\gamma.\\ & & \quad ((\gamma \to (\gamma \times \mathsf{int})) \wedge ((\gamma \times \mathsf{int}) \to ((\gamma \times \mathsf{int}) \times \mathsf{int})))\\ & & \quad \to \gamma \to ((\gamma \times \mathsf{int}) \times \mathsf{int}))\}\},\ \text{ and}\\ I_7 & \text{is} & \text{the interface introduced in Example 10.} \end{array}$$

7.3 Principal typings for $\vdash^{\mathrm{Bin}}$

The type system $\vdash^{\mathrm{Bin}}$ has the definition interface specialization property and the principal definition interface property.

LEMMA 6. *(Definition interface specialization property for $\vdash^{\mathrm{Bin}}$). If $G \vdash^{\mathrm{Bin}} bm : \langle B, I\rangle$, then $G \vdash^{\mathrm{Bin}} bm : \langle B, I'\rangle$, for every specialization I' of I.*

THEOREM 7.1. *(Principal definition interface property for $\vdash^{\mathrm{Bin}}$). If bm $\vdash^{\mathrm{Bin}}$-intrachecks w.r.t. G, then it has a principal definition interface w.r.t. G.*

7.4 Separate type inference for $\vdash^{\mathrm{Bin}}$

Definition 13. (Interface compatibility and interface composition for $\vdash^{\mathrm{Bin}}$).

1. The *composition of two binding environments* B_1 *and* B_2 is the binding environment $B_1 \Cup B_2 =$

$$\begin{array}{l} \{x : (E_1 \cup E_2) \mid x : E_1 \in B_1 \ \text{ and } \ x : E_2 \in B_2\}\\ \cup\{x : E_1 \mid x : E_1 \in B_1 \ \text{ and } \ x \notin \mathrm{Dom}(B_2)\}\\ \cup\{x : E_2 \mid x : E_2 \in B_2 \ \text{ and } \ x \notin \mathrm{Dom}(B_1)\} \end{array}$$

   where it is assumed that $\mathrm{Dom}(\mathrm{Env}(B_1)) \cap \mathrm{Dom}(\mathrm{Env}(B_2)) = \emptyset$ (note that this condition can always be satisfied since the identifiers in $\mathrm{Dom}(\mathrm{Env}(B_1))$ and $\mathrm{Dom}(\mathrm{Env}(B_2))$ are bound).

2. The *composition of two binding environments* B_1 *and* B_2 *w.r.t. the definition interface* I is the binding environment $B_1 \overset{I}{\Cup} B_2 = (B_1 \Cup B_2) - \{z : G \in B_1 \Cup B_2 \mid z : \forall\overrightarrow{\gamma}.\langle\emptyset, v\rangle \in I$ for some $\overrightarrow{\gamma}, v\}$.

3. The interfaces $\langle B_1, I_1\rangle$ and $\langle B_2, I_2\rangle$ $\vdash^{\mathrm{Bin}}$-*intercheck* if

   - I_1 and I_2 $\vdash^{\mathrm{Plain}}$-intercheck, and

$$(\textsc{ModBin})\ \frac{G \vdash^{\mathrm{Dec}} \mathsf{declare}\ \mathrm{Env}(B)\ \mathsf{define}\ \{x_1 = e_1, \cdots, x_n = e_n\} : \langle \mathrm{Env}(B), I\rangle}{G \vdash^{\mathrm{Bin}} \mathsf{bind}\ B\ \mathsf{define}\ \{x_1 = e_1, \cdots, x_n = e_n\} : \langle B_0, I\rangle}$$
$$\text{where } B_0 = B - \{z : G \in B \mid z : \forall \overrightarrow{\alpha}.\langle \emptyset, v\rangle \in I \text{ for some } \overrightarrow{\alpha}, v\}$$

Figure 6: Typing rules for modules with bounded declarations (system $\vdash^{\mathrm{Bin}}$)

- for all $z : \forall \overrightarrow{\gamma}.\langle \emptyset, v\rangle \in I_1 \oplus I_2$, if $z : G \in B_1 \Cup B_2$ and $z' : vcs \in G$ then $\forall \overrightarrow{\gamma}.v \leq_{\forall c2} vcs$.

4. The *composition* of two interfaces $\langle B_1, I_1\rangle$ and $\langle B_2, I_2\rangle$ that $\vdash^{\mathrm{Bin}}$-intercheck is the interface $\langle B_1, I_1\rangle \oplus \langle B_2, I_2\rangle = \langle B_1 \overset{I_1 \oplus I_2}{\Cup} B_2, I_1 \oplus I_2\rangle$.

Theorem 7.2. *(Separate type inference property for $\vdash^{\mathrm{Bin}}$). If $G \vdash^{\mathrm{Bin}}$ bind B_i define $pm_i : \langle B_i, I_i\rangle$ $(1 \leq i \leq n)$, then*

1. *$\langle B_1, I_1\rangle, \ldots, \langle B_n, I_n\rangle \vdash^{\mathrm{Bin}}$-intercheck if and only if $G \vdash^{\mathrm{Bin}}$ bind B define $pm : \langle B_0, I\rangle$, and*
2. *$G \vdash^{\mathrm{Bin}}$ bind B_i define $pm_i : \langle B_i, I_i\rangle$ $(1 \leq i \leq n)$ principal w.r.t. G imply that if $G \vdash^{\mathrm{Bin}}$ bind B define $pm : \langle B_0, I\rangle$ holds then it is principal w.r.t. G,*

where $B = \Cup_{i \in \{1,\ldots,n\}} B_i$, $pm = \cup_{i \in \{1,\ldots,n\}} pm_i$, $I = \oplus_{i \in \{1,\ldots,n\}} I_i$, and $B_0 = \overset{I}{\Cup}_{i \in \{1,\ldots,n\}} B_i$.

Example 12. Bounded declarations allow to overcome the unnecessary restriction illustrated at the beginning of the section. Consider, for instance, the principal typing judgements $\emptyset \vdash^{\mathrm{Bin}} bm_4 : \langle \emptyset, I_4\rangle$, $\emptyset \vdash^{\mathrm{Bin}} bm_5 : \langle B_5, I'_5\rangle$, $\emptyset \vdash^{\mathrm{Bin}} bm_6 : \langle \emptyset, I_6\rangle$, and $\emptyset \vdash^{\mathrm{Bin}} bm_7 : \langle B_7, I_7\rangle$ of Example 11. We have the principal typing judgement

$$\emptyset \vdash^{\mathrm{Bin}} \mathsf{bind}\ B_5 \Cup B_7\ \mathsf{define}\ pm_4 \cup pm_5 \cup pm_6 \cup pm_7 : \langle \emptyset, I_4 \oplus I'_5 \oplus I_6 \oplus I_7\rangle,$$

where $B_5 \Cup B_7 =$

$$\begin{array}{l}\{twice : \\ \quad \{twice_5 : \forall\, \alpha_1\, \alpha_2\, \alpha_3\, .((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3, \\ \quad\ twice_7 : \forall \gamma.((\gamma \to (\gamma \times \mathsf{int})) \wedge ((\gamma \times \mathsf{int}) \to ((\gamma \times \mathsf{int}) \times \mathsf{int}))) \\ \qquad\qquad \to \gamma \to ((\gamma \times \mathsf{int}) \times \mathsf{int}))\}\}\end{array}$$

and $I_4 \oplus I'_5 \oplus I_6 \oplus I_7 =$

$$\begin{array}{l}\{twice : \forall\, \alpha_1\, \alpha_2\, \alpha_3\, .\, \langle \emptyset, ((\alpha_1 \to \alpha_2) \wedge (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3\rangle, \\ \ g : \forall\, \gamma\, .\, \langle \emptyset, \gamma \to \gamma\ \mathsf{list}\ \mathsf{list}\rangle, \\ \ h : \forall\, \delta\, .\, \langle \emptyset, \delta \to \delta\rangle, \\ \ r : \forall\, \beta\, .\, \langle \emptyset, \beta \to ((\beta \times \mathsf{int}) \times \mathsf{int})\rangle\ \}.\end{array}$$

8. ACKNOWLEDGMENTS

I thank Mario Coppo, Joe Wells and the PPDP'03 referees for valuable comments and suggestions. I'm also grateful to Davide Ancona and Elena Zucca for useful discussions on modules and separate compilation.

9. REFERENCES

[1] S. Aditya and R. Nikhil. Incremental polymorphism. In *FPLCA'91*, LNCS 523, pages 379–405. Springer, 1991.

[2] H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48:931–940, 1983.

[3] L. Cardelli. Program fragments, linking, and modularization. In *POPL'97*, pages 266–277. ACM, 1997.

[4] M. Coppo. An extended polymorphic type system. In *MFCS'80*, LNCS 88, pages 194–204. Springer, 1980.

[5] M. Coppo and M. Dezani-Ciancaglini. An extension of basic functional theory for lambda-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[6] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional Characters of Solvable Terms. *Zeith. Math. Logik Und Grund. Math.*, 27:45–58, 1981.

[7] M. Coppo and P. Giannini. Principal Types and Unification for Simple Intersection Types Systems. *Information and Computation*, 122(1):70–96, 1995.

[8] L. M. M. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984.

[9] L. M. M. Damas and R. Milner. Principal type schemas for functional programs. In *POPL'82*, pages 207–212. ACM, 1982.

[10] F. Damiani. Rank 2 Intersection Types for Modules. Version with conclusions and appendices - available at http://www.di.unito.it/~damiani/papers/ppdp03.html.

[11] F. Damiani. Rank 2 Intersection Types for Local Definitions and Conditional Expressions. *ACM TOPLAS*, 25(4):401–451, 2003. To appear - available at http://www.di.unito.it/~damiani/papers/tr02.html.

[12] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM TOPLAS*, 15(2):211–252, 1993.

[13] R. Hindley. *Basic Simple Type Theory*. Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, London, 1997.

[14] T. Jim. What are principal typings and what are they good for? In *POPL'96*, pages 42–53. ACM, 1996.

[15] D. Leivant. Polymorphic Type Inference. In *POPL'83*, pages 88–98. ACM, 1983.

[16] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[17] Z. Shao and A. W. Appel. Smartest recompilation. In *POPL'93*, pages 439–450. ACM, 1993.

[18] S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[19] J. Wells. The essence of principal typings. In *ICALP'02*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.

[20] H. Yokouchi. Embedding a Second-Order Type System into an Intersection Type System. *Information and Computation*, 117:206–220, 1995.