# Type Inference, Principal Typings, and Let-Polymorphism for First-Class Mixin Modules *


Henning Makholm

Technical University of Denmark †

http://henning.makholm.net/

J. B. Wells

Heriot-Watt University

http://www.macs.hw.ac.uk/∼jbw/



## Abstract

A *mixin module* is a programming abstraction that simultaneously generalizes λ-abstractions, records, and mutually recursive definitions. Although various mixin module type systems have been developed, no one has investigated *principal typings* or developed *type inference* for first-class mixin modules, nor has anyone added Milner's *let-polymorphism* to such a system.

This paper proves that typability is NP-complete for the naive approach followed by previous mixin module type systems. Because a λ-calculus extended with *record concatenation* is a simple restriction of our mixin module calculus, we also prove the folk belief that typability is NP-complete for the naive early type systems for record concatenation.

To allow feasible type inference, we present Martini, a new system of *simple types* for mixin modules with *principal typings*. Martini is conceptually simple, with no subtyping and a clean and balanced separation between unification-based type inference with type and row variables and constraint solving for safety of linking and field extraction. We have implemented a type inference algorithm and we prove its complexity to be $O(n^2)$, or $O(n)$ given a fixed bound on the number of field labels.[1] To prove the complexity, we need to present an algorithm for *row unification* that may have been implemented by others, but which we could not find written down anywhere. Because Martini has principal typings, we successfully extend it with Milner's let-polymorphism.



***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures; modules, packages; polymorphism

***General Terms*** Design, theory, algorithms

***Keywords*** Type systems, mixin modules, record concatenation, row unification, polymorphism



* Supported by EC FP5/IST/FET grant IST-2001-33477 "DART", and Sun Microsystems equipment grant EDUD-7826-990410-US.


† Work done while at Heriot-Watt University.

[1] Technically, there is also a factor of $\alpha(n)$, but $\alpha(n) \leq 4$ for $n < 10^{10^{100}}$ (a "googolplex").




*ICFP'05* September 26–28, 2005, Tallinn, Estonia.




## 1. Introduction

The goal of this work is to produce a type system for first-class *mixin modules* that would have principal typings and thus support compositional type analysis.

As a programming tool, first-class mixin modules are aimed not only at programming-in-the-large issues such as generic modules and dynamic linking, but also at programming-in-the-small issues, because they combine the features of λ-abstractions (first-class functions), records, environments with mutually recursive definitions, and namespaces [31]. A mixin module consists of named components; some are *exports* that the module defines for other modules, some are *imports* to be supplied by other modules, and some are *locals*, i.e., private to the module. Once all imports are satisfied by *linking* modules, the exports can be *extracted*.

Linking is *symmetric*: when $A$ and $B$ are linked together, $A$'s exports can satisfy $B$'s imports and *vice versa*. For example, consider the following two modules, where $N(\mathtt{g},\mathtt{i})$ stands for some expression containing the identifiers g and i and so forth for $O$, $P$, $Q$:

$$\begin{aligned} A &= \{\!\!\{ \textbf{export}\,\mathtt{f} = N(\mathtt{g},\mathtt{i});\ \textbf{import}\,\mathtt{g},\mathtt{h};\ \textbf{local}\,\mathtt{i} = O(\mathtt{h}) \}\!\!\} \\ B &= \{\!\!\{ \textbf{import}\,\mathtt{f};\ \textbf{export}\,\mathtt{g} = P(\mathtt{f},\mathtt{i});\ \textbf{local}\,\mathtt{i} = Q \}\!\!\} \end{aligned}$$

Linking $A$ and $B$ produces this combined module:

$$\begin{aligned} A \oplus B = \{\!\!\{ &\textbf{export}\,\mathtt{f} = N(\mathtt{g},\mathtt{j}),\ \mathtt{g} = P(\mathtt{f},\mathtt{k});\ \textbf{import}\,\mathtt{h}; \\ &\textbf{local}\,\mathtt{j} = O(\mathtt{h}),\ \mathtt{k} = Q \}\!\!\} \end{aligned}$$

Because the local definitions of i in $A$ and $B$ are independent, they (or at least one) must be renamed in $A \oplus B$ to avoid conflicts. Also note that f and g in the linked module are mutually recursive, though no recursion is apparent in $A$ or $B$ alone.

This behavior is like compilation unit linking in C (and indeed, most languages) and is quite different from the asymmetric linking of the ML family's *structures* and *functors*. However, unlike C "modules", the mixin modules we investigate are *first-class*, i.e., they can be stored in data structures, passed as arguments, returned as results, nested using variables that are in scope, etc., and which modules are linked may depend on arbitrary run-time computations. In fact, modern programs in C and other languages do dynamically link modules at run-time (sometimes entire libraries that are also loaded at run-time). However, this is generally outside the language definitions and the type systems do not prevent linking failures due to missing or multiple definitions for a name. We focus instead on a *strongly typed* situation with better static guarantees.

Type analysis is *compositional* when each program fragment's analysis result does not depend on its lexical context. Compositionality simplifies type inference algorithms and helps with issues like separate compilation and accurate type error reporting. The main problem in compositional type inference for mixin modules is ensuring that the type system contains *principal typings* for expressions like $\lambda xy.x \oplus y$, which links two unknown modules. In this example, the type system must not allow modules given as $x$ and $y$ to both export the same name. But to be compositional, we must analyze $\lambda xy.x \oplus y$ without any knowledge about its eventual arguments.

The problem for mixin modules is similar to compositional analysis of calculi with *record concatenation*. Records are the special case of mixin modules with no imports and no internal recursion, and record concatenation is just a special case of linking mixin modules. The fundamental problem in analyzing record concatenation turns out to be similar to that for linking mixin modules, although mixin modules have more complications. Record concatenation has been intensively investigated as a potentially useful programming feature and also as one possible way to model object-oriented multiple inheritance. (Note that type inference for record *concatenation* is much harder than single-field record *extension*.)

Our approach to types for mixin modules is inspired in some respects by previous work on type inference for record concatenation. However, we could not build directly on a type system for record concatenation, because the most successful such systems use *subtyping polymorphism*. For technical reasons, we find this undesirable for mixin modules; for example, we would need different type rows for a module's imports and exports, which is problematic in our favored family of mixin module calculi where the externally visible names of imports and exports use the same namespace.

Thus, while our type system Martini for mixin modules has some features in common with record concatenation systems, it also exhibits interesting properties of its own which become apparent when it is *restricted* to work with a λ-calculus with records. The restriction, which we call Bowtie, does not type as many terms as some previous systems, but it has some other advantages, among which is a fast and conceptually simple type inference algorithm, which runs in almost linear time. In contrast, among record concatenation type inference algorithms with complexity analyses, the next best runs in cubic time [19].

Along the way to our main goal, we prove that the most obvious straightforward systems of simple types for mixin modules and record concatenation have NP-complete typability problems.

Martini (and its restriction Bowtie) has *principal typings* [29], which is a precondition for compositional analysis and is also needed for adding Milner's let-polymorphism. Principal typings should not be confused with the weaker notion of principal *types* which is usually all that remains *after* let-polymorphyism is added.

Martini is a *simple* type system that does not yet include *polymorphism*, which is needed for a serious strongly typed language. We present the simply typed version Martini first, because the machinery of polymorphism would obscure the novel features that handle mixin module linking. Sect. 7 shows how to extend Martini to Martini$^{\forall}$ that has Milner's let-polymorphism, as used in ML and other languages. This ought to be enough to support programming-in-the-small. For programming-in-the-large, further work is needed to add to Martini encapsulation and parameterization capabilities like those of the ML module language.

Our type inference implementation for Martini can be found at ⟨URL:`http://www.macs.hw.ac.uk/DART/software/Martini/`⟩ as a web application and as source code.

---

See Section 2 for some essential notation used here, such as $\boxed{\cdot}$, $\xrightarrow{\text{fin}}$, $\boxplus$, $\#$, *et cetera.*

---

| | |
|---|---|
| Variables: | $x ::= \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \cdots$ |
| Field labels: | $\ell ::= \texttt{f0} \mid \texttt{f1} \mid \texttt{f2} \mid \cdots$ |
| Label sets: | $\mathcal{L} \in \mathcal{P}_{\text{fin}}(\boxed{\ell})$ |
| External parts: | $E \in \boxed{\ell} \xrightarrow[\text{inj}]{\text{fin}} \boxed{x}$ |
| Internal parts: | $I \in \boxed{x} \xrightarrow{\text{fin}} \boxed{M}$ |
| Values: | $V ::= \{\!\!\{E\,;I\}\!\!\}$ |
| Terms: | $M,N ::= V \mid x \mid M \oplus N \mid M.\ell \mid M \mathbin{\backslash\!\!\backslash} \mathcal{L}$ |

To be closer to the notation in earlier module calculi, we allow writing $E$ functions as "$\ell_1 \triangleright x_1, \ldots, \ell_n \triangleright x_n$" instead of "$\{\ell_1 \mapsto x_1, \ldots, \ell_n \mapsto x_n\}$" and $I$ functions as "$y_1 = N_1, \ldots, y_k = N_k$" instead of "$\{y_1 \mapsto N_1, \ldots, y_k \mapsto N_k\}$".

---

$$\frac{M \hookrightarrow M'}{M \oplus N \hookrightarrow M' \oplus N}\ \mathsf{RC1} \qquad \frac{N \hookrightarrow N'}{M \oplus N \hookrightarrow M \oplus N'}\ \mathsf{RC2}$$

$$\frac{M \hookrightarrow N}{M.\ell \hookrightarrow N.\ell}\ \mathsf{RCdot} \qquad \frac{M \hookrightarrow N}{M \mathbin{\backslash\!\!\backslash} \mathcal{L} \hookrightarrow N \mathbin{\backslash\!\!\backslash} \mathcal{L}}\ \mathsf{RChide}$$

$$\frac{\begin{array}{c}\mathrm{FV}(\mathrm{Rng}\, I_1) \# \mathcal{X}_2 \qquad \mathcal{X}_1 \# \mathcal{X}_2 \qquad \mathrm{FV}(\mathrm{Rng}\, I_2) \# \mathcal{X}_1 \\ \text{where } \mathcal{X}_i = \mathrm{Rng}\, E_i \cup (\mathrm{Dom}\, I_i \setminus \mathrm{Rng}\, E) \quad \text{for } i = 1,2\end{array}}{\{\!\!\{E \boxplus E_1\,; I_1\}\!\!\} \oplus \{\!\!\{E \boxplus E_2\,; I_2\}\!\!\} \hookrightarrow \{\!\!\{E \boxplus E_1 \boxplus E_2\,; I_1 \boxplus I_2\}\!\!\}}\ \mathsf{RLink}$$

$$\frac{\begin{array}{c}\mathrm{Rng}\, E \subseteq \mathrm{Dom}\, I \\ \mathcal{S} = \{\, x \mapsto \{\!\!\{\texttt{f} \triangleright x; I\}\!\!\}.\texttt{f} \mid x \in \mathrm{Dom}\, I \,\}\end{array}}{\{\!\!\{E\,;I\}\!\!\}.\ell \hookrightarrow [\mathcal{S}](I(E(\ell)))}\ \mathsf{RExtract}$$

$$\frac{\begin{array}{c}\mathrm{Dom}\, E_1 \# \mathcal{L} \qquad \mathrm{Dom}\, E_2 \subseteq \mathcal{L} \\ \mathrm{Rng}\, E_2 \subseteq \mathrm{Dom}\, I\end{array}}{\{\!\!\{E_1 \boxplus E_2\,; I\}\!\!\} \mathbin{\backslash\!\!\backslash} \mathcal{L} \hookrightarrow \{\!\!\{E_1\,; I\}\!\!\}}\ \mathsf{RHide}$$

***Figure 1.*** *Syntax and semantics of the* $\overline{\text{m}}$*-calculus*

## 2. Notation

These notations are fairly standard: A *function* $f$ is a set of pairs, where we write each pair in the form "$a \mapsto b$", such that $\{(a \mapsto b), (a \mapsto c)\} \subseteq f$ implies $b = c$. In this section, let $f$ and $g$ range over functions and let $A$ and $B$ range over sets. The *domain* of $f$ is $\mathrm{Dom}\, f = \{\, x \mid (x \mapsto y) \in f \,\}$. The *range* of $f$ is $\mathrm{Rng}\, f = \{\, y \mid (x \mapsto y) \in f \,\}$. The *inverse* of $f$ is $f^{-1} = \{\, x \mapsto y \mid (y \mapsto x) \in f \,\}$. The expression $f(A)$ is $\{\, f(a) \mid a \in A \cap \mathrm{Dom}\, f \,\}$ if $A \notin \mathrm{Dom}\, f$. Composition is given by $(f \circ g)(x) = f(g(x))$. $\mathcal{P}(A)$ is the set of all subsets of $A$. The set difference $A \setminus B$ is $\{\, a \in A \mid a \notin B \,\}$. The disjoint union $A \uplus B$ is $A \cup B$ if $A \cap B = \varnothing$, and undefined otherwise.

These notations are less common: For any metavariable symbol $X$, $\boxed{X}$ is the set that $X$ ranges over. $\mathcal{P}_{\text{fin}}(A)$ is the set of all *finite* subsets of $A$. The statement $A \# B$ abbreviates $A \cap B = \varnothing$. The expression $A \xrightarrow{\text{fin}} B$ is the set of all finite functions $f \subseteq A \times B$, and $A \xrightarrow[\text{inj}]{\text{fin}} B$ is the set of all *injective* functions in $A \xrightarrow{\text{fin}} B$. The expression $f \boxplus g$ means $f \cup g$ if $\mathrm{Dom}\, f \# \mathrm{Dom}\, g$, and is undefined otherwise.

If $\boxed{X}$ is a set of syntactic entities that can contain (concrete) variables in the set $\boxed{x} \subset \boxed{X}$, then any (partial or total) function $f$ from $\boxed{x}$ to $\boxed{X}$ can be used as a *substitution*. The application of $f$ to an entier $X$ is defined in the usual way by recursive descent through $X$, replacing each variable $x$ by $f(x)$ whenever $x \in \mathrm{Dom}\, f$ and renaming bound variables in $X$ as needed to avoid name capture. We identify syntactic entities modulo renaming of bound variables.

We sometimes enclose $f$ in square brackets (like $[f]$) in order to emphasize that it is being used as a substitution. This happens in particular when $f$ is given by listing its elements, in which case we omit the set braces and write just $[x_1 \mapsto X_1, \ldots, x_n \mapsto X_n]$.

## 3. The $\overline{\text{m}}$-calculus of mixin modules

Our goal is to create a type system with compositional type inference for the simple mixin module calculus defined in Figure 1, called the $\overline{\text{m}}$-calculus. Its syntax is essentially isomorphic to that of the m-calculus of Wells and Vestergaard [31]. The key difference is a simplified call-by-name semantics, because the more sophisticated semantics (and equational theory) of the m-calculus are irrelevant for the typing issues this paper investigates. Hirschowitz et al. [13] give a similar calculus with a call-by-value semantics. The $\overline{\text{m}}$-calculus is in a family of mixin module calculi where imports and exports share a single namespace; an alternative family with separate import and export namespaces starts from CMS [4].

The basic construct is a **mixin module**, written $\{\!|E\,;I|\!\}$, where $E$ (the external part) maps field labels to variables and $I$ (the internal part) maps variables to terms. The module expression $\{\!|\ell_1 \triangleright x_1,\ldots,\ell_n \triangleright x_n\,;\, y_1 = N_1,\ldots,y_k = N_k|\!\}$ **binds** all of the variables $x_1$ to $x_n$ and $y_1$ to $y_k$ within the $N_i$'s. Thus the free variables of $\{\!|E\,;I|\!\}$ are $\mathrm{FV}(\{\!|E\,;I|\!\}) = \left(\bigcup_{N\in \mathrm{Rng}\, I} \mathrm{FV}(N)\right) \setminus (\mathrm{Rng}\, E \cup \mathrm{Dom}\, I)$ and all other cases of $\mathrm{FV}(M)$ just collect free variables componentwise. Each bound name $x$ falls in one of three classes:

- $(\ell \triangleright x) \in E$ is an **import** with **external name** $\ell$ and **internal name** $x$ iff there is no $(x = N) \in I$. When the module is linked with a module that has an export with name $\ell$, references to $x$ in the $N_i$'s become bound to the exported expression.
- $(\ell \triangleright x) \in E$ together with $(x = N) \in I$ is an **export**. The exported expression $N$ can be used from outside via the **field extraction** operation $M.\ell$, but only once all imports have been satisfied via linking. Then $N$ will be evaluated in the context of the other definitions in $I$. The exported term $N$ can satisfy imports of the name $\ell$ by other modules via linking. The internal name $x$ can be mentioned in all of the $N_i$'s, and the defined value can thereby be directly or indirectly recursive.
- $(x = N) \in I$ is a **local** iff there is no $(\ell \triangleright x) \in E$. Local definitions can be used in all of the $N_i$'s. They can be directly or indirectly recursive, but are not observable outside their containing module except by being referred to by an export.

Field labels (i.e., external names) are fixed, but bound variables (i.e., internal names) are subject to α-conversion (which must keep them distinct from other bound variables of the same module) and the actual names of the bound variables are not visible outside the module expression. We identify α-equivalent terms.

The fundamental mixin operation is **linking**, written $M_1 \oplus M_2$. Its reduction rule RLink is intuitively simple; linking two modules puts their internal parts side by side (choosing appropriate α-variants to avoid wrong name captures), and joins their external parts. The rule divides each of the two incoming external parts into a *common* part $E$ and a *separate* part $E_i$. The separate parts are untouched by the linking. The linking happens in the common part $E$, containing the labels mentioned by *both* operands; such labels cannot be in $E_1$ and $E_2$, because then $E_1 \boxplus E_2$ would be undefined on the right-hand side (RHS) of the rule RLink. A label occurring in $E$ maps to the *same* internal name in both operands; this is always possible by α-conversion. No label can be exported by both operands; otherwise $I_1 \boxplus I_2$ would be undefined on the RHS of RLink. A label exported by one module and imported by the other is forced by the common $E$ to have the same internal name (variable) in both operands, so importing variable references in one $I_i$ can become bound to definitions from the other after the linking.

The premises of RLink ensure that wrong name captures do not occur. $\mathcal{X}_i$ is the set of internal names in operand $i$ that do not participate in the link. The names in $\mathcal{X}_i$ will be bound on both sides post-linking, so they must be disjoint from the other side's internal names and free variables. For example, the linking $\{\!|\mathtt{f} \triangleright \mathtt{x}, \mathtt{g} \triangleright \mathtt{y}\,;\, \mathtt{y} = \mathtt{x}|\!\} \oplus \{\!|\mathtt{h} \triangleright \mathtt{z}\,;\, \mathtt{x} = 13, \mathtt{z} = \mathtt{x} - 6|\!\}$ (where the common $E$ is empty) does not proceed with the shown α-variants of the operands because of the premise $\mathcal{X}_1 \;\#\; \mathcal{X}_2$ where both sets contain x. Without this premise, the result would be a module that (wrongly) exported 13 as f. To avoid this, the rule forces us to first α-convert one or both of the x's. Another example needing α-conversion is $\{\!|\mathtt{f} \triangleright \mathtt{x}\,;\, \mathtt{x} = 42|\!\} \oplus \{\!|\mathtt{g} \triangleright \mathtt{y}\,;\, \mathtt{y} = \mathtt{x}|\!\}$ where the x on the left must be α-renamed to avoid wrongly capturing the (free) x on the right.

The **field extraction** rule RExtract extracts the export with the given label while unfolding into the field body the implicit letrec of the module $M$ to achieve call-by-name semantics. The **hiding** operator $M \setminus\!\!\setminus \mathcal{L}$ removes exported fields if they exist in the operand. A hidden export turns into a local because the internal part $I$ is kept unchanged. If none of the fields in $\mathcal{L}$ are exported or imported the hiding is simply a no-op.

It is a run-time error to try to link two modules that both export the same label $\ell$, as well as to try to extract a field from a module that does not export it or has imports, or to try to hide an import. Our task is designing a type system that prevents these errors while also allowing compositional type inference.

### 3.1 Syntactic sugar for mixin modules and records

The representation of a mixin module as separate $E$ and $I$ parts is formally convenient in that it allows our type and reduction rules to be stated relatively compactly. However, it is not very intuitive for actual programming, so our implementation also supports a more readable notation (which we have used already in the example in the introduction):

$$
\begin{array}{lrl}
\text{Values:} & V ::= & \cdots \mid \{\!|\gamma_1;\ldots;\gamma_k|\!\} \\
\text{Module groups:} & \gamma ::= & \textbf{import}\ \ell_1 \triangleright x_1,\ldots,\ell_k \triangleright x_k \\
& \mid & \textbf{export}\ \ell_1 \triangleright x_1 = M_1,\ldots,\ell_k \triangleright x_k = M_k \\
& \mid & \textbf{local}\ x_1 = M_1,\ldots,x_k = M_k
\end{array}
$$

The construct $\{\!|\gamma_1;\ldots;\gamma_k|\!\}$ is sugar for

$$\{\!|[\gamma_1]^{\mathbf{E}} \boxplus \cdots \boxplus [\gamma_k]^{\mathbf{E}}\,;\,[\gamma_1]^{\mathbf{I}} \boxplus \cdots \boxplus [\gamma_k]^{\mathbf{I}}|\!\},$$

where

$$
\begin{array}{rl}
[\textbf{import}\ \ell_1 \triangleright x_1,\ldots,\ell_k \triangleright x_k]^{\mathbf{E}} & = \{\, \ell_i \mapsto x_i \mid 1 \le i \le k \,\} \\
[\textbf{export}\ \ell_1 \triangleright x_1 = M_1,\ldots,\ell_k \triangleright x_k = M_k]^{\mathbf{E}} & = \{\, \ell_i \mapsto x_i \mid 1 \le i \le k \,\} \\
[\textbf{local}\ x_1 = M_1,\ldots,x_k = M_k]^{\mathbf{E}} & = \varnothing \\
[\textbf{import}\ \ell_1 \triangleright x_1,\ldots,\ell_k \triangleright x_k]^{\mathbf{I}} & = \varnothing \\
[\textbf{export}\ \ell_1 \triangleright x_1 = M_1,\ldots,\ell_k \triangleright x_k = M_k]^{\mathbf{I}} & = \{\, x_i \mapsto M_i \mid 1 \le i \le k \,\} \\
[\textbf{local}\ x_1 = M_1,\ldots,x_k = M_k]^{\mathbf{I}} & = \{\, x_i \mapsto M_i \mid 1 \le i \le k \,\}
\end{array}
$$

and where it is required that $\mathrm{Rng}\ [\gamma_i]^{\mathbf{E}} \;\#\; \mathrm{Dom}\ [\gamma_j]^{\mathbf{I}}$ for $i \neq j$. (Note that syntactic correctness of the result follows from implicit constraints imposed by the definition of $\boxplus$ and the fact that each mixin external part $E$ is a bijection.) For additional compactness, "$\ell \triangleright x$" in **import** and **export** groups can be written as just "$\ell$" if $x$ and $\ell$ are textually identical (even though they belong to different namespaces).

We further define a fourth form of module group (distinguished by its lack of keyword):

$$\gamma ::= \cdots \mid \ell_1 = M_1,\ldots,\ell_k = M_k,$$

which is syntactic sugar for "**export** $\ell_1 \triangleright y_1 = M_1,\ldots,\ell_k \triangleright y_k = M_k$", where the $y_i$'s are chosen fresh. This allows modules that have neither imports nor local definitions nor internal references to their own exports to be written with a record-like syntax.

For example, the record-like expression $\{\!|\mathtt{f} = 5,\ \mathtt{g} = \mathsf{true}|\!\}$ abbreviates $\{\!|\textbf{export}\ \mathtt{f} \triangleright \mathtt{x} = 5,\ \mathtt{g} \triangleright \mathtt{y} = \mathsf{true}|\!\}$, which in turn abbreviates $\{\!|\mathtt{f} \triangleright \mathtt{x}, \mathtt{g} \triangleright \mathtt{y}\,;\, \mathtt{x} = 5, \mathtt{y} = \mathsf{true}|\!\}$, which finally abbreviates $\{\!|\{\mathtt{f} \mapsto \mathtt{x}, \mathtt{g} \mapsto \mathtt{y}\}\,;\,\{\mathtt{x} \mapsto 5, \mathtt{y} \mapsto \mathsf{true}\}|\!\}$.

### 3.2 Encoding of the λ-calculus

The reader may wonder how interesting the $\overline{\text{m}}$-calculus is given that its *only* data constructor is the mixin module. It is natural to assume that one would need an additional language layer for manipulating mixin modules programatically. However, this turns out to be unneeded, because we can encode the λ-calculus using only mixin constructions. A function $\lambda x.M$ can be represented as

$$
\begin{array}{ll}
\text{Rows:} & \Sigma \in \boxed{\boxed{\ell}} \xrightarrow{\text{fin}} \boxed{\boxed{\tau}} \\
\text{Types:} & \tau ::= \texttt{'a} \mid \texttt{'b} \mid \texttt{'c} \mid \ldots \mid \{\!\!\{ \Sigma \,/\, \mathcal{L} \}\!\!\} \\
\text{Environments:} & \Gamma \in \boxed{\boxed{x}} \xrightarrow{\text{fin}} \boxed{\boxed{\tau}} \\
\text{Typings:} & T ::= \langle \Gamma \vdash \tau \rangle
\end{array}
$$

$$
\frac{}{x : \langle \Gamma \vdash \Gamma(x) \rangle}
\qquad
\frac{\begin{array}{c}\mathrm{Dom}\,\Gamma = \mathrm{Rng}\,E \cup \mathrm{Dom}\,I \\ I(x) : \langle \Gamma_0 \boxplus \Gamma \vdash \Gamma(x) \rangle \ \text{ for all } x \in \mathrm{Dom}\,I \\ \mathcal{L} = E^{-1}(\mathrm{Dom}\,I) \qquad (\Gamma \circ E) = \Sigma \end{array}}{\{\!\!\{ E\,; I \}\!\!\} : \langle \Gamma_0 \vdash \{\!\!\{ \Sigma \,/\, \mathcal{L} \}\!\!\} \rangle}
$$

$$
\frac{M : \langle \Gamma \vdash \{\!\!\{ \Sigma \,/\, \mathcal{L} \}\!\!\} \rangle \qquad \mathcal{L} = \mathrm{Dom}\,\Sigma}{M.\ell : \langle \Gamma \vdash \Sigma(\ell) \rangle}
$$

$$
\frac{\begin{array}{c}\mathcal{L}_1 \mathbin{\#} \mathcal{L}_2 \\ M : \langle \Gamma \vdash \{\!\!\{ \Sigma \boxplus \Sigma_1 \,/\, \mathcal{L}_1 \}\!\!\} \rangle \qquad N : \langle \Gamma \vdash \{\!\!\{ \Sigma \boxplus \Sigma_2 \,/\, \mathcal{L}_2 \}\!\!\} \rangle\end{array}}{M \oplus N : \langle \Gamma \vdash \{\!\!\{ \Sigma \boxplus \Sigma_1 \boxplus \Sigma_2 \,/\, \mathcal{L}_1 \cup \mathcal{L}_2 \}\!\!\} \rangle}
$$

$$
\frac{\begin{array}{c} M : \langle \Gamma \vdash \{\!\!\{ \Sigma_1 \boxplus \Sigma_2 \,/\, \mathcal{L} \}\!\!\} \rangle \\ \mathrm{Dom}\,\Sigma_1 \mathbin{\#} \mathcal{L}_0 \qquad \mathrm{Dom}\,\Sigma_2 \subseteq (\mathcal{L}_0 \cap \mathcal{L}) \end{array}}{M \mathbin{\backslash\!\!\backslash} \mathcal{L}_0 : \langle \Gamma \vdash \{\!\!\{ \Sigma_1 \,/\, (\mathcal{L} \setminus \mathcal{L}_0) \}\!\!\} \rangle}
$$

***Figure 2.*** Riviera*: Naive simple types for the $\overline{\mathrm{m}}$-calculus*

$\{\!\!\{$**import** $\texttt{arg} \triangleright x$; **export** $\texttt{res} \triangleright y = M\}\!\!\}$, and an application $N\,M$ as $(N \oplus \{\!\!\{$**export** $\texttt{arg} \triangleright y = M\}\!\!\})\texttt{.res}$, where $\texttt{arg}$ (argument) and $\texttt{res}$ (result) are globally fixed labels and in both cases we choose $y \notin \mathrm{FV}(M)$. It is easy to see that an application of RLink followed by a number of RExtract will simulate a β-reduction in the λ-calculus. This is not a new result [31, 4], but is essential for understanding how the $\overline{\mathrm{m}}$-calculus can be used. We will freely use this translation as syntactic sugar in examples and constructions.

Let $\boldsymbol{\lambda}^{\oplus}$ denote the fragment of $\overline{\mathrm{m}}$-calculus that can be written using only the constructions from the following grammar, some of which use the syntactic sugar for λ-calculus and records:

$$M ::= \lambda x.M \mid M\,N \mid \{\!\!\{ \ell_1 = M_1, \ldots, \ell_k = M_k \}\!\!\} \mid M_1 \oplus M_2 \mid M.\ell$$

We will use $\boldsymbol{\lambda}^{\oplus}$ when comparing Martini with earlier record concatenation type systems.

## 4. Riviera: Naive simple types for mixin modules

A natural first attempt at defining a (Curry-style) simple type system for the $\overline{\mathrm{m}}$-calculus is the naive type system called Riviera shown in Figure 2. Following [29], we write typing judgements as "$M : \langle \Gamma \vdash \tau \rangle$" rather than using the older convention of "$\Gamma \vdash M : \tau$".

A Riviera type has the shape $\{\!\!\{ \Sigma \,/\, \mathcal{L} \}\!\!\}$ where $\mathcal{L} \subseteq \mathrm{Dom}\,\Sigma$, and denotes a module that exports every label in $\mathcal{L}$ and imports every label in $(\mathrm{Dom}\,\Sigma) \setminus \mathcal{L}$, with the types of the exports and imports given by the **row** $\Sigma$. (In the context of Riviera, a row is just a finite map from labels to types. Later in our type system Martini in Section 5 we will consider partially unknown rows, which will use different metavariables.) Of course, we could equally well have written $\{\!\!\{ \Sigma_1 \Rightarrow \Sigma_2 \}\!\!\}$ where $\Sigma_1$ and $\Sigma_2$ partition the field types into imports and exports, but the $\{\!\!\{ \Sigma \,/\, \mathcal{L} \}\!\!\}$ syntax makes the typing rules more compact (and we do not intend to actually use Riviera as we will prove it has unfeasible type inference).

An expression written with the syntactic sugar $\lambda x.M$ defined in Section 3.2 will always have a type that can be written $\{\!\!\{ \{\texttt{arg} \mapsto \tau_1, \texttt{res} \mapsto \tau_2\} \,/\, \{\texttt{res}\} \}\!\!\}$. We will abbreviate such a type $\tau_1 \to \tau_2$.

### 4.1 Typability in Riviera is NP-hard

Riviera can be proven *sound* (i.e., programs it accepts do not "go wrong"), but that about exhausts its nice formal properties. In particular, even though it is only at the level of simple types, *type inference* is provably hard.

**Theorem 4.1.** *Typability*[2] *for* Riviera *is NP-complete.* □

We define the construction that proves this in several stages.

Let $M \triangleright N$ abbreviate $(\lambda x.N)\,M$, and let $M \sim N$ abbreviate $(\lambda x.(x\,M) \triangleright (x\,N))\,(\lambda y.y)$, where for both we choose $x \notin \mathrm{FV}(M\,N)$.

**Lemma 4.2.** *$M \triangleright N : \langle \Gamma \vdash \tau \rangle$ iff there exists a type $\tau'$ such that both $M : \langle \Gamma \vdash \tau' \rangle$ and $N : \langle \Gamma \vdash \tau \rangle$.* □

**Lemma 4.3.** *$M \sim N : T$ if and only if $M : T$ and $N : T$.* □

Call **canonical** the following two fixed types: $\mathbf{Off} = \{\!\!\{ \varnothing \,/\, \varnothing \}\!\!\}$ and $\mathbf{On} = \{\!\!\{ \{\texttt{a} \mapsto \{\!\!\{ \varnothing \,/\, \varnothing \}\!\!\}\} \,/\, \{\texttt{a}\} \}\!\!\}$. Let $\mathbf{Nand}(x_1, \ldots, x_n)$ abbreviate $\lambda y_1 \ldots \lambda y_n.\big((y_1 \oplus x_1)\texttt{.a} \oplus \cdots \oplus (y_n \oplus x_n)\texttt{.a}\big)\texttt{.b}$.

**Lemma 4.4.** *In an environment that maps each $x_i$ to a canonical type, $\mathbf{Nand}(x_1, \ldots, x_n)$ has a type iff the type of some $x_i$ is* **Off**. □

Let $\mathbf{P}(x, y) = \mathbf{Nand}(x, y) \triangleright (x \oplus y) \sim (y \oplus x) \sim \{\!\!\{ \texttt{a} = \{\!\!\{ \varnothing \}\!\!\} \}\!\!\}$.

**Lemma 4.5.** *$\mathbf{P}(x, y)$ has a type exactly in environments where one of $x$ and $y$ has type* **Off** *and the other has type* **On**. □

We will be reducing from Boolean formula satisfiability, so let a countable set of Boolean variables $\boxed{\boxed{P}}$ be given, and select a fixed correspondence that assigns a unique $\overline{\mathrm{m}}$-calculus variable $\texttt{v}_P$ to each Boolean variable. An environment naturally corresponds to a Boolean valuation for those $P$ such that it assigns $\texttt{v}_P$ either type **On** or **Off**. The corresponding valuation makes $P$ true if $\texttt{v}_P$ has type **On** and false if $\texttt{v}_P$ has type **Off**.

Now define a translation from formulae to terms by

$$
\begin{aligned}
[P] &= \lambda \texttt{t}.\lambda \texttt{f}.\mathbf{P}(\texttt{t}, \texttt{f}) \triangleright \texttt{v}_P \sim \texttt{t} \\
[\neg \mathcal{A}] &= \lambda \texttt{t}.\lambda \texttt{f}.[\mathcal{A}]\,\texttt{f}\,\texttt{t} \\
[\mathcal{A} \vee \mathcal{B}] &= \lambda \texttt{t}.\lambda \texttt{f}.\lambda \texttt{t}_1.\lambda \texttt{f}_1.\lambda \texttt{t}_2.\lambda \texttt{f}_2. \\
&\qquad \mathbf{P}(\texttt{t}, \texttt{f}) \triangleright [\mathcal{A}]\,\texttt{t}_1\,\texttt{f}_1 \triangleright [\mathcal{B}]\,\texttt{t}_2\,\texttt{f}_2 \triangleright \\
&\qquad \mathbf{Nand}(\texttt{t}_1, \texttt{f}) \triangleright \mathbf{Nand}(\texttt{t}_2, \texttt{f}) \triangleright \mathbf{Nand}(\texttt{t}, \texttt{f}_1, \texttt{f}_2) \\
[\mathcal{A} \wedge \mathcal{B}] &= [\neg(\neg \mathcal{A} \vee \neg \mathcal{B})]
\end{aligned}
$$

**Lemma 4.6.** *$[\mathcal{A}]$ has a type exactly in environments that correspond to valuations for $\mathcal{A}$'s variables. The types of $[\mathcal{A}]$ in such an environment have the shape* $\mathbf{On} \to \mathbf{Off} \to \tau$ *if $\mathcal{A}$ is true under the corresponding valuation and* $\mathbf{Off} \to \mathbf{On} \to \tau$ *if $\mathcal{A}$ is false. (There may be one or more $\tau$'s that can appear in a valid type for $[\mathcal{A}]$.)* □

*Proof.* By a straightforward induction over the structure of $\mathcal{A}$. □

**Proof of Theorem 4.1**. It is clear that typability is in NP; given an oracle which tells for each variable which fields its type imports and exports, type inference is like for the simply typed λ-calculus.

For NP-hardness, we reduce from satisfiability of arbitrary Boolean formulae. A formula $\mathcal{A}$ is satisfiable exactly if the term $M_{\mathcal{A}} = [\mathcal{A}]\,\texttt{t}\,\texttt{f} \triangleright \mathbf{Nand}(\texttt{f})$ is Riviera-typable. It follows from Lemma 4.6 that any typing for $M_{\mathcal{A}}$ implies a satisfying truth assignment for $\mathcal{A}$. Conversely, any satisfying truth assignment corresponds to an environment $\Gamma$ such that $[\mathcal{A}] : \langle \Gamma \vdash \mathbf{On} \to \mathbf{Off} \to \tau \rangle$ for some $\tau$. From there one easily gets $M_{\mathcal{A}} : \langle \Gamma, \texttt{t} : \mathbf{On}, \texttt{f} : \mathbf{Off} \vdash \tau \rangle$. □

### 4.2 Asymmetric variants

In a calculus with **asymmetric** linking, $M \oplus N$ succeeds even when $M$ and $N$ both define some label $\ell$, in which case the result will use the definition from $N$. One can construct a Riviera-like type system $\text{Riviera}_{\mathsf{A}}$ for such a calculus by replacing the $\oplus$ typing rule with:

[2] Typability means deciding for a given term $M$ whether there exists a typing $T$ such that $M : T$. Note that specifying the $\Gamma$ part of $T$ would not make the problem easier: one can just abstract over $M$'s free variables and set $\Gamma = \varnothing$.

$$\frac{\begin{array}{c}\mathrm{Dom}\,\Sigma_{11} = \mathcal{L}_1 \cap \mathcal{L}_2 \\ M : \langle \Gamma \vdash \{\!\!\{ \Sigma \boxplus \Sigma_1 \boxplus \Sigma_{11} \;/\; \mathcal{L}_1 \}\!\!\} \rangle \qquad N : \langle \Gamma \vdash \{\!\!\{ \Sigma \boxplus \Sigma_2 \;/\; \mathcal{L}_2 \}\!\!\} \rangle\end{array}}{M \oplus N : \langle \Gamma \vdash \{\!\!\{ \Sigma \boxplus \Sigma_1 \boxplus \Sigma_2 \;/\; \mathcal{L}_1 \cup \mathcal{L}_2 \}\!\!\} \rangle}$$

Another possible variant of linking allows asymmetric linking, but requires that a field that is defined in both linked modules must have the same *type* even though at run-time one of them is discarded. This gives rise to a Riviera variant with this linking rule:

$$\frac{M : \langle \Gamma \vdash \{\!\!\{ \Sigma \boxplus \Sigma_1 \;/\; \mathcal{L}_1 \}\!\!\} \rangle \qquad N : \langle \Gamma \vdash \{\!\!\{ \Sigma \boxplus \Sigma_2 \;/\; \mathcal{L}_2 \}\!\!\} \rangle}{M \oplus N : \langle \Gamma \vdash \{\!\!\{ \Sigma \boxplus \Sigma_1 \boxplus \Sigma_2 \;/\; \mathcal{L}_1 \cup \mathcal{L}_2 \}\!\!\} \rangle}$$

We call this variant **join-like**, and name it Riviera$_\mathsf{J}$, because the type behavior of this rule is identical to the type-level behavior of the *join* operator $\bowtie$ in relational algebras [15, 25]. It is not common for modules or records (an exception is the Church-style calculus of [32]), but we mention it to highlight the similarities between type problems in relational calculi and in calculi with concatenation and linking. Further, the simplest variation of our system Martini (introduced later in Section 5) which allows overwriting linking would have a join-like typing rule.

**Theorem 4.7.** *Typability for* Riviera$_\mathsf{A}$ *and* Riviera$_\mathsf{J}$ *is NP-complete.* □

The proof for Theorem 4.1 has been constructed to work *verbatim* for Riviera$_\mathsf{A}$ and Riviera$_\mathsf{J}$. (This does make it more complex, however. Simpler constructions for Riviera and Riviera$_\mathsf{A}$ can be found in the long version of this paper [14]).

### 4.3 Other calculi

Riviera is very similar to Typed CMS as defined by Ancona and Zucca [4], except that Typed CMS is a Church-style system with mandatory type annotations in terms. There are also minor differences, such that the fact the CMS has different namespaces for import and export labels, and uses a special "freeze" operator to connect imports with exports. However, the proof of Theorem 4.1 does not use imports at all, except in the syntactic sugar for translating $\lambda$ abstractions. If one uses the $\lambda$ translation for CMS (defined in [4]), our proof directly yields NP-completeness of typability in the Curry-style variant of Typed CMS (i.e., Typed CMS with type annotations removed), a previously unknown result.

We believe our proof of NP-completeness of typability can also be adapted to the implicitly typed systems CMS$_v$ [12], MM [13] and Mix [11], although we have not checked this formally.

If one restricts Riviera$_\mathsf{A}$ to (asymmetric) $\boldsymbol{\lambda}^{\oplus}$, one gets a type system equivalent to Wand's type system for record concatenation [28]. Because the proof of Theorem 4.1 uses only the $\boldsymbol{\lambda}^{\oplus}$ fragment of the $\overline{\mathrm{m}}$-calculus, we get as a bonus result a direct proof of the folk belief that typability in Wand's system is NP-hard.

### 4.4 Discussion

NP-completeness may not sound bad when one compares it to the familiar result that typability for the Hindley/Milner type system is DEXPTIME-complete. However, it should be kept in mind that Riviera is a *simply* typed system where polymorphism has yet to be added. Thus the proper comparison would be the simply typed $\lambda$-calculus, where typability is almost linear (i.e., $O(n\,\alpha(n))$).

Our result suggests that type inference for mixin modules is hard independently of the details of the calculus. A feasible type system for mixin modules (or record concatenation) must introduce complications that are not aimed solely at strengthening the type system itself (in the sense of enlarging the set of typable programs) but serve to make type inference a tractable problem.

A related result was achieved by Palsberg and Zhao [16], who prove NP-completeness of typability for a typed object calculus with symmetric record concatenation *and* subtyping. It is not clear to us whether this proof can be easily adapted to record-concatenation type systems *without* subtyping. Vansummeren [25] proved NP-completenes of typability in various fragments of a naively typed relational algebra. Ohori and Buneman [15] proved NP-completeness of typability in a lambda calculus with primitive sets of records and join operator. Their system is defined in terms of constraints, but appears to be equivalent in expressive power to a Riviera$_\mathsf{J}$-like type system.

## 5. Martini: A better type system for mixin modules

Figure 3 show our type system Martini for the $\overline{\mathrm{m}}$-calculus. It is designed to simultaneously reach two goals in addition to the usual basic safety. (1) It should have feasible type inference. (2) It should have *principal typings* [29], which are needed to express intermediate results in a compositional inference algorithm using Martini's own type language. Also, principal typings allow adding Milner's let-polymorphism (as we do in Section 7).

A mixin module type in Martini has the shape $\{\!\!\{ R \;/\; Q \Rightarrow S \}\!\!\}$, where $Q$ represents the set of *import* labels and $S$ represents the set of *export* labels. The row $R$ gives the type of both inputs and outputs, like $\Sigma$ in Riviera; however a Martini row may define types for fields that the module neither imports nor exports.

For example, given obvious extensions with integers and booleans, the term $M = \{\!\!\{ \mathtt{f} \triangleright \mathtt{x}, \mathtt{g} \triangleright \mathtt{y};\, \mathtt{x} = \mathtt{y} > 5 \}\!\!\}$ has the typing $\langle \varnothing \vdash \{\!\!\{ \mathtt{f}{:}\mathsf{bool}, \mathtt{g}{:}\mathsf{int}, \mathtt{r} \;/\; \{\mathtt{g}\} \Rightarrow \{\mathtt{f}\} \}\!\!\} \mid \varnothing \rangle$ ($\mathtt{r}$ is a *row variable* whose role will be described shortly). This is also a typing for $N = \{\!\!\{ \mathtt{f} \triangleright \mathtt{x};\, \mathtt{x} = \mathsf{true} \}\!\!\}$, even though $N$ does not import $\mathtt{g}$ — it is allowed for a $Q$ to overapproximate the true set of imports. This allows constructions such as $\mathsf{if} \cdots \mathsf{then}\ M\ \mathsf{else}\ N$ to be typed without requiring a dummy import of $\mathtt{g}$ in $N$.

In Riviera it was essentially an arbitrary choice to use a single row for imports as well as exports. In contrast, the same decision in Martini is essential for our type inference strategy. In order to facilitate type inference with principal typings, the three parts $R$, $Q$ and $S$ of a mixin type $\{\!\!\{ R \;/\; Q \Rightarrow S \}\!\!\}$ have a more elaborate internal structure than the monolithic $\Sigma$ and $\mathcal{L}$ of Riviera — in particular, each part can be a *variable*.

Wand [26] introduced row variables for manipulating partial knowledge about mappings from field labels to types. Using the same row variable at the end of different row expressions can express that the two rows agree at some but not all labels, as happens in the rule THide. Adding more fields to the row part of a module type does not change the type's meaning if the set expression part stays the same. Therefore Martini needs no syntax for an "empty row"; a row variable can always be used for this.

Typings now contain *constraint* sets, which assert relations between label sets that are not known yet. As a concrete example, the term $\{\!\!\{ \mathbf{import}\ \mathtt{f} \triangleright \mathtt{x}, \mathtt{g} \triangleright \mathtt{y};\, \mathtt{h} = \mathtt{x} \oplus \mathtt{y} \}\!\!\}$ has the Martini typing

$$\left\langle \vdash \{\!\!\{ \begin{pmatrix} \mathtt{f}{:}\{\!\!\{ \mathtt{r} \;/\; \mathtt{q1} \Rightarrow \mathtt{s1} \}\!\!\}, \\ \mathtt{g}{:}\{\!\!\{ \mathtt{r} \;/\; \mathtt{q2} \Rightarrow \mathtt{s2} \}\!\!\}, \\ \mathtt{h}{:}\{\!\!\{ \mathtt{r} \;/\; \mathtt{q3} \Rightarrow \mathtt{s3} \}\!\!\}, \mathtt{r0} \end{pmatrix} \;/\; \{\mathtt{f},\mathtt{g}\} \Rightarrow \{\mathtt{h}\} \}\!\!\} \;\middle|\; \begin{array}{l} \mathtt{s3} = \mathtt{s1} \uplus \mathtt{s2}, \\ \mathtt{q3} \supseteq \mathtt{q1} \setminus \mathtt{s2}, \\ \mathtt{q3} \supseteq \mathtt{q2} \setminus \mathtt{s1} \end{array} \right\rangle.$$

The recurrence of $\mathtt{r}$ in all three field types means that *if* one of the imports contains a field $\ell$ at all, its type must be the same as that expected for $\ell$ in the result. The constraint $\mathtt{s3} = \mathtt{s1} \uplus \mathtt{s2}$ says that every field defined in the result must be defined in exactly one of the arguments. This particular typing is not principal; Martini also allows the term to have the type $\{\!\!\{ \cdots \;/\; \{\mathtt{f},\mathtt{g},\mathtt{k}\} \Rightarrow \{\mathtt{h}\} \}\!\!\}$ even though it does not import $\mathtt{k}$. A principal typing for the term would have $\{\!\!\{ \cdots \;/\; \mathtt{q} \Rightarrow \{\mathtt{h}\} \}\!\!\}$ as its type part and an extra constraint $\mathtt{q} \supseteq \{\mathtt{f},\mathtt{g}\}$.

Martini's strategy for escaping the NP-hardness of Riviera is to consider the constraint $\mathtt{s3} = \mathtt{s1} \uplus \mathtt{s2}$ "good" *for as long as we have no information about* $\mathtt{s1}$ *or* $\mathtt{s2}$, regardless of whether information

| | | | |
|---|---|---|---|
| Type variables: | $\alpha ::=$ `'a` $\mid$ `'b` $\mid$ `'c` $\mid \ldots$ | Types: | $\tau ::= \alpha \mid \{\!\!\{ R \,/\, Q \Rightarrow S \}\!\!\}$ |
| Row variables: | $r ::=$ `r0` $\mid$ `r1` $\mid$ `r2` $\mid \cdots$ | Environments: | $\Gamma \in \boxed{x} \xrightarrow{\text{fin}} \boxed{\tau}$ |
| Import set variables: | $q ::=$ `q0` $\mid$ `q1` $\mid$ `q2` $\mid \cdots$ | Constraints: | $c ::= S = S_1 \uplus S_2 \mid S = S_1 \setminus \mathcal{L}$ |
| Export set variables: | $s ::=$ `s0` $\mid$ `s1` $\mid$ `s2` $\mid \cdots$ | | $\mid\ Q \supseteq Q_1 \setminus S \mid Q \# \mathcal{L}$ |
| Type rows: | $R ::= r \mid \ell{:}\tau, R$ | Const. sets: | $C \in \mathcal{P}_{\text{fin}}(\boxed{c})$ |
| Import set expressions: | $Q ::= q \mid \mathcal{L}$ | Typings: | $T ::= \langle \Gamma \vdash \tau \mid C \rangle$ |
| Export set expressions: | $S ::= s \mid \mathcal{L} \mid \bot$ | | |

We forbid as ill-formed rows that define any label $\ell$ more than once and we consider types (etc.) modulo this **row structure** equation:

$$(\ell_1{:}\tau_1, \ell_2{:}\tau_2, R) = (\ell_2{:}\tau_2, \ell_1{:}\tau_1, R) \qquad \text{when } \ell_1 \neq \ell_2$$

For $\Sigma = \{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\} \in \boxed{\ell} \xrightarrow{\text{fin}} \boxed{\tau}$, let $\Sigma \bullet R$ abbreviate $\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n, R$.
Let $S_1 = S_2$, $Q_1 \supseteq Q_2$, and $Q \subseteq S$ abbreviate the constraints $S_1 = S_2 \uplus \varnothing$, $Q_1 \supseteq Q_2 \setminus \varnothing$, and $\varnothing \supseteq Q \setminus S$, respectively.
We allow omitting the set braces around concrete environments and constraint sets in typings.

Let $\mathcal{T}$ range over **type substitutions**, which are functions that map $\boxed{q}$ to $\boxed{Q}$, $\boxed{s}$ to $\boxed{S}$, $\boxed{r}$ to $\boxed{R}$, and $\boxed{\alpha}$ to $\boxed{\tau}$, such that only finitely many variables do not map to themselves. Type substitutions are extended componentwise to map each of the classes $\boxed{Q}$, $\boxed{S}$, $\boxed{R}$, $\boxed{\tau}$, $\boxed{\Gamma}$, $\boxed{c}$, $\boxed{C}$, and $\boxed{T}$ to itself.

A constraint $c$ is **solved**, written $\Vdash c$, iff *either* $c$ is a true statement of set theory ($\mathcal{L} = \mathcal{L}_1 \uplus \mathcal{L}_2$ is true iff $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$ and $\mathcal{L}_1 \# \mathcal{L}_2$)
*or* $c$ contains $\bot$ to the *right* of the relation sign (that is, to the right of $=$, $\in$, or $\supseteq$).

Let $\Vdash C$ abbreviate $\forall c \in C : \Vdash c$, and let $C_1 \Vdash C_2$ abbreviate $\forall \mathcal{T} : \Vdash \mathcal{T}(C_1) \Rightarrow \Vdash \mathcal{T}(C_2)$.
Write $\tau_1 \approx \tau_2$ iff $\tau_1$ and $\tau_2$ are identical except for import and export set expressions.
Let $C \Vdash \tau_1 = \tau_2$ abbreviate $\tau_1 \approx \tau_2 \wedge \forall \mathcal{T} : \Vdash \mathcal{T}(C) \Rightarrow \mathcal{T}(\tau_1) = \mathcal{T}(\tau_2)$ .
Let $C \Vdash \Gamma_1 = \Gamma_2$ mean that $\text{Dom}\,\Gamma_1 = \text{Dom}\,\Gamma_2$ and $C \Vdash \Gamma_1(x) = \Gamma_2(x)$ for $x \in \text{Dom}\,\Gamma_1$.

$$\frac{C \Vdash \Gamma(x) = \tau}{x : \langle \Gamma \vdash \tau \mid C \rangle}\ \mathsf{TVar}$$

$$\frac{\begin{array}{c}\text{Dom}\,\Gamma = \text{Rng}\,E \cup \text{Dom}\,I \qquad I(x) : \langle \Gamma_0 \boxplus \Gamma \vdash \Gamma(x) \mid C \rangle \ \text{ for all } x \in \text{Dom}\,I \\ \mathcal{L} = E^{-1}(\text{Dom}\,I) \qquad \mathcal{L}' = \text{Dom}\,E \setminus \mathcal{L} \qquad C \Vdash \{Q \supseteq \mathcal{L}',\ S = \mathcal{L}\}\end{array}}{\{\!\!\{ E\,;I \}\!\!\} : \langle \Gamma_0 \vdash \{\!\!\{ (\Gamma \circ E) \bullet R \,/\, Q \Rightarrow S \}\!\!\} \mid C \rangle}\ \mathsf{TMixin}$$

$$\frac{\begin{array}{c}M : \langle \Gamma \vdash \{\!\!\{ R \,/\, Q_1 \Rightarrow S_1 \}\!\!\} \mid C \rangle \qquad N : \langle \Gamma \vdash \{\!\!\{ R \,/\, Q_2 \Rightarrow S_2 \}\!\!\} \mid C \rangle \\ C \Vdash \{Q \supseteq Q_1 \setminus S_2,\ S = S_1 \uplus S_2,\ Q \supseteq Q_2 \setminus S_1\}\end{array}}{M \oplus N : \langle \Gamma \vdash \{\!\!\{ R \,/\, Q \Rightarrow S \}\!\!\} \mid C \rangle}\ \mathsf{TLink}
\qquad
\frac{M : \langle \Gamma \vdash \{\!\!\{ \ell{:}\tau, R \,/\, \varnothing \Rightarrow S \}\!\!\} \mid C \rangle \qquad C \Vdash \{\{\ell\} \subseteq S\}}{M.\ell : \langle \Gamma \vdash \tau \mid C \rangle}\ \mathsf{TExtract}$$

$$\frac{\text{Dom}\,\Sigma_1 = \text{Dom}\,\Sigma_2 = \mathcal{L} \qquad M : \langle \Gamma \vdash \{\!\!\{ \Sigma_1 \bullet R \,/\, Q \Rightarrow S_1 \}\!\!\} \mid C \rangle \qquad C \Vdash \{Q \# \mathcal{L},\ S_2 = S_1 \setminus \mathcal{L}\}}{M \mathbin{\backslash\!\!\backslash} \mathcal{L} : \langle \Gamma \vdash \{\!\!\{ \Sigma_2 \bullet R \,/\, Q \Rightarrow S_2 \}\!\!\} \mid C \rangle}\ \mathsf{THide}$$

***Figure 3.*** *Our type system* Martini *for the* $\overline{m}$*-calculus*

about `s3` shows up later. Thus, e.g., $\lambda$`x.(x`$\oplus$`x).f` is typable in Martini even though there is *no* value that it can safely be applied to. This is still sound, as Martini will reject attempts to actually call it. Thus, Martini does not reject some nonsensical (but dynamically safe) programs as type errors, but also escapes NP-hardness of typability. Theorem 4.1 depends on being able to ask the type system: "Can *any* possible call to this function be proved error-free?". Martini refuses to answer until we refine the question to "Will *this* particular call be error-free?".

The term $\lambda$`x.(x`$\oplus$`x).f` has a principal typing with the shape $\langle \varnothing \vdash \{\!\!\{ \texttt{arg:}\{\!\!\{ \texttt{f:'a, r} \,/\, \texttt{q} \Rightarrow \texttt{s} \}\!\!\}, \texttt{res:'a, r0} \,/\, \texttt{q0} \Rightarrow \{\texttt{res}\} \}\!\!\} \mid \texttt{s1} = \texttt{s} \uplus \texttt{s}, \{\texttt{f}\} \subseteq \texttt{s1}, \ldots \rangle$. This typing is "good" because we can solve its constraint set by substituting the special set expression $\bot$ for each of `s` and `s1`. This is the role of $\bot$; it allows a constraint to be solved as long as we have no evidence that a use of the linking or field extraction operation that the constraint corresponds to will actually go wrong at run-time. Principal typings never need to contain $\bot$. Effectively, $\{\!\!\{ R \,/\, Q \Rightarrow \bot \}\!\!\}$ is a type that describes no values at all. An expression with such a type must be either **dead** (its result will never be used, perhaps because its evaluation diverges) or **sleeping** (its result will not be used unless the program is put into a larger context). Martini accepts mistakes in dead or sleeping code that are hard to check for until a concrete calling context is provided, but it still rejects mistakes that are *easy* to find in dead or sleeping code.

Now return to the typing for $\{\!\!\{ \textbf{import}\ \texttt{f} \triangleright \texttt{x}, \texttt{g} \triangleright \texttt{y};\ \texttt{h} = \texttt{x} \oplus \texttt{y} \}\!\!\}$ and consider its triple occurrence of `r`. Because `r` appears in the arguments as well as in the result, Martini can begin to resolve their connection before enough information to solve the constraint arrives. This lets some internal errors in sleeping code be caught early and also gives more readable principal typings by expressing more relations between types without constraints. However, some strength is sacrificed: Each module type must contain the field types of any module it may be linked with. (The programmer need not write them as they will be inferred.) Thus Martini will reject, for example, $\lambda\texttt{x}.\{\!\!\{ \texttt{a} = \texttt{x} \oplus \{\!\!\{ \texttt{f} = 5 \}\!\!\}, \texttt{b} = \texttt{x} \oplus \{\!\!\{ \texttt{f} = \texttt{true} \}\!\!\} \}\!\!\}$, because the type of `x` must predict a single type for `f` in all of its descendants. One way to mitigate this **row pollution** problem is to use let-polymorphism (Section 7) instead of $\lambda$ to bind `x`, to allow polymorphism in the type of `f`. Another is to insert dummy hiding operators: $\lambda\texttt{x}.\{\!\!\{ \texttt{a} = \texttt{x} \mathbin{\backslash\!\!\backslash} \{\texttt{f}\} \oplus \{\!\!\{ \texttt{f} = 5 \}\!\!\}, \texttt{b} = \texttt{x} \mathbin{\backslash\!\!\backslash} \{\texttt{f}\} \oplus \{\!\!\{ \texttt{f} = \texttt{true} \}\!\!\} \}\!\!\}$ is typable. Since this solution depends on guessing which fields will be defined by the other operand to $\oplus$, it cannot be applied fully automatically; it can be considered a programmer-supplied typing hint.

### 5.1 Soundness

Proving type soundness for Martini proceeds as usual, except for the details of handling constraint sets.

**Lemma 5.1 (weakening).** *Assume $M : \langle \Gamma \vdash \tau \mid C \rangle$. For any $C' \Vdash C$ and $\Gamma' \supseteq \Gamma$ it holds that $M : \langle \Gamma' \vdash \tau \mid C' \rangle$.* $\square$

A constraint set $C$ is **solvable** iff $\Vdash \mathcal{T}(C)$ for some substitution $\mathcal{T}$. A typing is called solvable iff its constraint set component is.

**Lemma 5.2.** *Assume $C \Vdash \tau_1 = \tau_2$ and $C \Vdash \Gamma_1 = \Gamma_2$. Then $M : \langle \Gamma_1 \vdash \tau_1 \mid C \rangle$ implies $M : \langle \Gamma_2 \vdash \tau_2 \mid C \rangle$.* $\square$

**Lemma 5.3 (substitution).** *If $M : T$, then $M : \mathcal{T}(T)$.* $\square$

**Lemma 5.4 (term substitution).** *Assume $M : \langle \Gamma \boxplus \{x \mapsto \tau'\} \vdash \tau \mid C \rangle$ and $N : \langle \varnothing \vdash \tau' \mid C \rangle$. Then $[x \mapsto N]M : \langle \Gamma \vdash \tau \mid C \rangle$.* $\square$

**Theorem 5.5 (subject reduction).** *If $M \hookrightarrow N$ and $M : \langle \varnothing \vdash \tau \mid C \rangle$ for solvable $C$, then $N : \langle \varnothing \vdash \tau \mid C \rangle$.* $\square$

**Theorem 5.6 (progress).** *If $M : \langle \varnothing \vdash \tau \mid C \rangle$ for solvable $C$, then either $M$ is a value or $M \hookrightarrow N$ for some $N$.* $\square$

**Theorem 5.7 (type soundness).** *Programs (closed terms) with solvable typings do not get stuck.* $\square$

### 5.2 Testing constraint set solvability

To make Thm. 5.7 useful, this section develops an algorithm to identify solvable constraint sets.

Define the relation $C \mathrel{☞} \mathcal{T}$ by:

1. $\forall C, s, \mathcal{L}_1, \mathcal{L}_2 :\ C \cup \{s = \mathcal{L}_1 \uplus \mathcal{L}_2\} \mathrel{☞} [s \mapsto (\mathcal{L}_1 \cup \mathcal{L}_2)]$ if $\mathcal{L}_1 \# \mathcal{L}_2$,
2. $\forall C, s, \mathcal{L}_1, \mathcal{L}_2 :\ C \cup \{s = \mathcal{L}_1 \setminus \mathcal{L}_2\} \mathrel{☞} [s \mapsto (\mathcal{L}_1 \setminus \mathcal{L}_2)]$.

**Lemma 5.8.** *If $C \mathrel{☞} \mathcal{T}$, then $C$ is solvable if and only if $\mathcal{T}(C)$ is.* $\square$

**Lemma 5.9.** *Assume that there is no $\mathcal{T}$ such that $C \mathrel{☞} \mathcal{T}$, and let $\mathcal{T}_0$ map every export set variable in $C$ to $\bot$ (and every other variable to itself). Then $C$ is solvable if and only if $\mathcal{T}_0(C)$ is.* $\square$

*Proof.* The "if" direction is obvious. For "only if", assume that $C$ is solved by some $\mathcal{T}_1$. We claim that $\mathcal{T}_1$ also solves $\mathcal{T}_0(C)$. For otherwise there would be a $c \in C$ such that $\mathcal{T}_1(c)$ is solved but $\mathcal{T}_1(\mathcal{T}_0(c))$ is not. Their only difference is that $\mathcal{T}_1(\mathcal{T}_0(c))$ may contain $\bot$ in a place where $\mathcal{T}_1(c)$ has another (input) set expression. The only place $\bot$ can appear without solving $\mathcal{T}_1(\mathcal{T}_0(c))$ is to the left of "=", so $c$ must have the form $s = \mathcal{L}_1 \setminus \mathcal{L}_2$ or $s = \mathcal{L}_1 \uplus \mathcal{L}_2$ (where in the latter case $\mathcal{L}_1$ and $\mathcal{L}_2$ must be disjoint, because $\mathcal{T}_1(c)$ is solved). But in either of these two cases we could find a $\mathcal{T}$ such that $C \mathrel{☞} \mathcal{T}$, contradicting the assumption. $\square$

For any constraint set $C$ and any $\ell$ that occurs in $C$, let $\mathbf{Q}^C_\ell$ be the least subset of $\boxed{q}$ such that

1. When $(q \supseteq \mathcal{L}_1 \setminus \mathcal{L}_2) \in C$ with $\ell \in (\mathcal{L}_1 \setminus \mathcal{L}_2)$, then $q \in \mathbf{Q}^C_\ell$, and
2. When $(q_2 \supseteq q_1 \setminus \mathcal{L}) \in C$ with $\ell \notin \mathcal{L}$, then $q_1 \in \mathbf{Q}^C_\ell \Rightarrow q_2 \in \mathbf{Q}^C_\ell$.

Computing $\mathbf{Q}^C_\ell$ for a given $C$ and $\ell$ is a simple $O(n)$ graph reachability problem. The intended intuition is that $\mathbf{Q}^C_\ell$ is "the set of $q$'s that $\ell$ can reach according to $C$" and $\{\, \ell \mid q \in \mathbf{Q}^C_\ell \,\}$ is a lower bound on the possible values of $q$ in all ground solutions of $C$. Let $C \mathrel{☚} \mathcal{T}$ hold when $\mathcal{T}(q) = \{\, \ell \mid q \in \mathbf{Q}^C_\ell \,\}$ for all $q$ mentioned in $C$ and $\mathcal{T}$ maps all other variables to themselves.

**Lemma 5.10.** *If $C \mathrel{☚} \mathcal{T}$ and $C$ is solvable and free of export set variables, then $\mathcal{T}(C)$ is solved.* $\square$

*Proof.* Because $C$ is assumed solvable, any constraint in $C$ that does not contain at least one $q$ variable must be solved already, and can therefore be ignored. Constraints of the form $Q_2 \supseteq Q_1 \setminus \bot$ are also solved, so the only constraints we need to consider are those of the forms $Q_2 \supseteq Q_1 \setminus \mathcal{L}$ and $Q \# \mathcal{L}$.

Given an arbitrary $\mathcal{T}'$ that solves $C$, whenever $q \in \mathbf{Q}^C_\ell$ it must hold that $\ell \in \mathcal{T}'(q)$; this follows directly from the inductive construction of $\mathbf{Q}^C_\ell$. Also, if we let $\mathcal{T}'_\ell(q)$ be $\mathcal{T}'(q) \setminus \{\ell\}$ for $q \notin \mathbf{Q}^C_\ell$ and $\mathcal{T}'(q)$ otherwise, then $\mathcal{T}'_\ell$ still solves $C$ — it would contradict the construction of $\mathbf{Q}^C_\ell$ if this change caused an unsolved constraint to appear. Now, if any solution to $C$ exists at all, then $\mathcal{T}$ is also a solution, because an arbitrary solution can be transformed into $\mathcal{T}$ label for label by the preceding remarks. $\square$

**Theorem 5.11.** *Constraint sets can be tested for solvability in time $O(nm)$, where $n$ is the number of constraints and $m$ is the number of distinct labels in the constraints.* $\square$

*Proof.* The testing procedure consists of first rewriting as much as possible by Lemma 5.8, and then eliminating the remaining export set variables by Lemma 5.9. The constraint set now contains only $q$ variables, and solvability can therefore be decided by Lemma 5.10.

Complexity bound: In the Lemma 5.8 phase, each constraint is processed at most four times: Once to see if it can be rewritten immediately, up to twice when set variables on its left-hand side are instantiated, and once to check whether $\mathcal{T}(c)$ is solved. If $\mathcal{L}$'s are represented as bit vectors, each visit of the constraint takes $O(m)$ time. Lemma 5.9 can obviously be completed in time $O(n)$. For Lemma 5.10, the $\mathbf{Q}^C_\ell$'s and $\mathcal{T}$ can be straightforwardly constructed in time $O(mn)$; checking $\Vdash \mathcal{T}(C)$ takes $O(mn)$ time. $\square$

### 5.3 Type inference

Define the relation $\sqsubseteq$ between typings by: $\langle \Gamma_1 \vdash \tau_1 \mid C_1 \rangle \sqsubseteq \langle \Gamma_2 \vdash \tau_2 \mid C_2 \rangle$ iff there exists a type substitution $\mathcal{T}$ and $\Gamma'_2 \subseteq \Gamma_2$ such that $C_2 \Vdash \Gamma'_2 = \mathcal{T}(\Gamma_1)$ and $C_2 \Vdash \tau_2 = \mathcal{T}(\tau_1)$ and $C_2 \Vdash \mathcal{T}(C_1)$.

**Lemma 5.12.** *If $M : T_1$ and $T_1 \sqsubseteq T_2$, then $M : T_2$.* $\square$

Let $T \leq T'$ ($T$ is "at least as precise as" $T'$) mean $M : T \Rightarrow M : T'$ for all $M$. A typing $T$ is **principal** [29] for $M$ iff $M : T$ and $M : T' \Rightarrow T \leq T'$ for all $T'$. A typing $T$ is **syntactically principal** for $M$ iff $M : T$ and $M : T' \Rightarrow T \sqsubseteq T'$ for all $T'$. Because $T \sqsubseteq T'$ implies $T \leq T'$, a syntactically principal typing for $M$ is principal.

The definitions directly imply that if $T_1 \sqsubseteq T_2$ and $T_2$ is solvable, then $T_1$ is also solvable. Therefore a syntactically principal typing for a term $M$ is solvable unless $M$ has no solvable typings at all.

**Theorem 5.13.** *The algorithm* TYPEINF *defined in Figure 4 computes a syntactically principal typing for every typable term. The algorithm runs in time $O(nm\,\alpha(n))$ where $n$ is the size of the analyzed term and $m$ is the number of distinct field labels in it.*

*As a corollary,* Martini *has principal typings.* $\square$

Before we prove the theorem, here are some high-level remarks about the algorithm. In our description it consists of two phases. The COLLECT phase processes the input term to collect type equations and constraints; the RUNIFY phase solves the type equations by a unification algorithm extended to deal with rows. The collected constraints are not touched (except for the side effects of the unification step); they appear directly in the principal typing. Implementations will usually perform the two phases in parallel as co-processes, and check constraint solvability using Thm. 5.11 afterward; we leave such refinements to the reader's imagination.

The recursive syntax-directed COLLECT phase takes two inputs: a term $M$ and a type environment $\Gamma$ that maps all of $M$'s free variables to distinct type variables. It produces a triple $\mathrm{COLLECT}(M, \Gamma) = (W, \tau, C)$, where $W$ is a set of type equations

COLLECT$(x, \mathit{\Gamma}) = (\varnothing, \mathit{\Gamma}(x), \varnothing)$

COLLECT$(\{\!\{E\,; I\}\!\}, \mathit{\Gamma}) =$
let $\mathcal{X} = \mathrm{Rng}\, E \cup \mathrm{Dom}\, I$ ;
let $\mathcal{L} = E^{-1}(\mathrm{Dom}\, I)$ and $\mathcal{L}' = \mathrm{Dom}\, E \setminus \mathcal{L}$;
let $r$, $q$, $s$, and $\alpha_x$ for each $x \in \mathcal{X}$ be fresh;
let $\mathit{\Gamma}'(x)$ be $\alpha_x$ for $x \in \mathcal{X}$, and $\mathit{\Gamma}(x)$ otherwise;
let $(W_x, \tau_x, C_x) =$ COLLECT$(I(x), \mathit{\Gamma}')$ for all $x \in \mathrm{Dom}\, I$;
let $W = \bigcup_{x \in \mathrm{Dom}\, I} \big(W_x \cup \{\tau_x = \alpha_x\}\big)$;
let $\Sigma = \{\, \ell \mapsto \mathit{\Gamma}'(E(\ell)) \mid \ell \in \mathrm{Dom}\, E \,\}$
in $(W, \{\!\{\, \Sigma \bullet r \mathbin{/} q \Rightarrow s \}\!\}, \bigcup_{x \in \mathrm{Dom}\, I} C_x \cup \{q \supseteq \mathcal{L}',\ s = \mathcal{L}\})$.

COLLECT$(M_1 \oplus M_2, \mathit{\Gamma}) =$
let $r$, $q$, $q_1$, $q_2$, $s$, $s_1$, and $s_2$ be fresh;
let $(W_i, \tau_i, C_i) =$ COLLECT$(M_i, \mathit{\Gamma})$ for $i \in \{1,2\}$;
let $C = C_1 \cup C_2 \cup \{q \supseteq q_1 \setminus s_2,\ s = s_1 \uplus s_2,\ q \supseteq q_2 \setminus s_1\}$;
let $W = W_1 \cup W_2 \cup \{\tau_1 = \{\!\{ r \mathbin{/} q_1 \Rightarrow s_1 \}\!\},\ \tau_2 = \{\!\{ r \mathbin{/} q_2 \Rightarrow s_2 \}\!\}\}$
in $(W, \{\!\{ r \mathbin{/} q \Rightarrow s \}\!\}, C)$.

COLLECT$(M.\ell, \mathit{\Gamma}) =$
let $\alpha$, $r$, $s$ be fresh;
let $(W, \tau, C) =$ COLLECT$(M, \mathit{\Gamma})$
in $(W \cup \{\tau = \{\!\{ \ell{:}\alpha,\ r \mathbin{/} \varnothing \Rightarrow s \}\!\}\}, \alpha, C \cup \{\{\ell\} \subseteq s\})$.

COLLECT$(M \mathbin{\setminus\!\!\setminus} \mathcal{L}, \mathit{\Gamma}) =$
let $r$, $q$, $s_0$, $s$, and $\alpha_\ell$ and $\alpha'_\ell$ for each $\ell \in \mathcal{L}$ be fresh;
let $\Sigma_0 = \{\, \ell \mapsto \alpha'_\ell \mid \ell \in \mathcal{L} \,\}$ and $\Sigma = \{\, \ell \mapsto \alpha_\ell \mid \ell \in \mathcal{L} \,\}$;
let $(W, \tau, C) =$ COLLECT$(N, \mathit{\Gamma})$;
let $W' = W \cup \{\tau = \{\!\{\, \Sigma_0 \bullet r \mathbin{/} q \Rightarrow s_0 \}\!\}\}$
in $(W', \{\!\{\, \Sigma \bullet r \mathbin{/} q \Rightarrow s \}\!\}, C \cup \{q \# \mathcal{L},\ s = s_0 \setminus \mathcal{L}\})$.

TYPEINF$(M) =$
let $\mathit{\Gamma}$ map each free variable of $M$ to a fresh type variable;
let $(W, \tau, C) =$ COLLECT$(M, \mathit{\Gamma})$;
let $\mathcal{T} =$ RUNIFY$(W)$
in $\langle \mathcal{T}(\mathit{\Gamma}) \vdash \mathcal{T}(\tau) \mid \mathcal{T}(C) \rangle$.

***Figure 4.*** *The definition of the type inference algorithm. The sub-algorithm* RUNIFY$(W)$ *will be defined in Section 6.*

(of the form $\tau_1 = \tau_2$), $\tau$ is a type expression, and $C$ is a constraint set. Note that the input $\mathit{\Gamma}$ is only used to map term variables to type variables (not arbitrary types) and it is never changed by COLLECT, although the equations in the produced $W$ may give rise to substitutions for the type variables in Dom $\mathit{\Gamma}$ later.

REMARK 5.14. In a strictly *compositional* inference algorithm, the type environment $\mathit{\Gamma}$ should be an output instead of an input. However, computing the principal environment for any subexpression bottom-up will incur possibly large reconciliation costs when the principal environments for two sibling expressions meet each other, and would therefore destroy the advertised $O(nm\,\alpha(n))$ complexity. The best type inference solution that we could find that produced $\mathit{\Gamma}$ only as output had complexity $O(n \log n + nm\,\alpha(n))$ (which is worse when $m$ is small), but even that depended (as our current algorithm does) on a mutable union-find data structure shared by all the subcomputations.[3] One possible solution could be to start by α-renaming all variables away from each other and then deriving the type variable name $\mathit{\Gamma}(x)$ from the name of $x$. However, this does not work in practice, where type variables are represented by heap-allocated union-find elements to facilitate unification. □

A **unifier** for a type equation set $W$ is a type substitution $\mathcal{T}$ such that every equation in $\mathcal{T}(W)$ is an identity (remembering that types are already identified modulo the row structure equation).

**Lemma 5.15.** *Assume that* $(W, \tau, C) =$ COLLECT$(M, \mathit{\Gamma})$ *where* $\mathit{\Gamma}$ *maps each free variable in* $M$ *to a unique type variable. Let* $T = \langle \mathit{\Gamma} \vdash \tau \mid C \rangle$*. For each unifier* $\mathcal{T}$ *for* $W$*, it holds that* $M : \mathcal{T}(T)$*. Conversely, whenever* $M : T'$*, it holds that there is a unifier* $\mathcal{T}$ *for* $W$ *such that* $\mathcal{T}(T) \sqsubseteq T'$. □

**Lemma 5.16.** COLLECT$(M, \mathit{\Gamma})$ *can be computed in time linear in the size of* $M$. □

*Proof.* The only parts of the computation for which this is not obviously the case are lookups and additions in $\mathit{\Gamma}$. If we represent $\mathit{\Gamma}$ as a *trie*, the operations are all linear in the length of the variable name, so the total time spent here is linear in the size of $M$.[4] □

A **most general unifier** (MGU) for $W$ is a unifier $\mathcal{T}$ for $W$ such that every other unifier for $W$ can be written as $\mathcal{T}' \circ \mathcal{T}$ for some $\mathcal{T}$. The second phase of type inference computes an MGU of the final equation set $W$. Section 6 will construct a function RUNIFY such that RUNIFY$(W)$ computes one of the MGUs of $W$, if it has any, in time $O(nm\,\alpha(n))$, where $n$ is the size of $W$ and $m$ is the number of distinct field labels mentioned in $W$.

**Proof of Thm. 5.13**. The correctness of the result of TYPINF follows from Lemma 5.15 and the property that RUNIFY computes most general unifiers.

According to Lemma 5.16, the COLLECT phase completes in time $O(n)$, so the size of $W$ is $O(n)$ too. Therefore, the computation (and, implicitly, application) of MGU$(W)$ can be completed in time $O(nm\,\alpha(n))$ — a detailed argument for this will be given in Section 6. Thus the entire computation has complexity $O(nm\,\alpha(n))$. □

### 5.4 Incremental constraint simplification

Superficially, it seems that we have solved type analysis for Martini. To analyze a term, one computes its principal typing (Thm. 5.13) and then checks whether its constraint part is solvable (Thm. 5.11). Each step is efficient and relatively simple. So are we happy?

Not entirely. Postponing constraint solving until the end has practical and theoretical disadvantages. One problem is that the constraint parts of the inferred typings can grow quite large, because they can track each module operation in the term, *even those that are irrelevant for observable behavior*. For example, the term $(\{\!\{\mathtt{f} = 5\}\!\} \oplus \{\!\{\}\!\}).\mathtt{f}$ gets the inferred typing $\langle \varnothing \vdash \mathtt{int} \mid \mathtt{s} = \{\mathtt{f}\}, \{\mathtt{f}\} \subseteq \mathtt{s} \rangle$ rather than the more concise $\langle \varnothing \vdash \mathtt{int} \mid \varnothing \rangle$ — which, incidentally, is also syntactically principal for it.

Instead, we want to *interleave* the constraint solving (Section 5.2) with other type inference steps. But we must be a little careful; if we blindly applied the entire constraint solving procedure from

[3] Incidentally, this suggests that a consistently compositional type inference algorithm cannot achieve the folklore $O(n\,\alpha(n))$ behavior on programs where the number of free variables in a subexpression may be large. However, that is not a real problem; the main point of compositional inference is that the analysis *can* be broken at any point in the expression tree, but there is no obligation to actually break the analysis at every point.

[4] A reader of early drafts of this explanation complained that this assumes that variable names are represented with a fixed finite alphabet in the input, whereas the rest of our complexity analysis assumes a constant-cost RAM with a word length that expands with the input size such that all parts of the input can be addressed. The reader complained that it was not fair not to allow the representation of variable names to take advantage of this increasing word length, and that the complexity bounds ought to include a logarithmic factor to take account of this. We disagree with this objection; we believe it is a common convention in complexity theory to measure the "size of the input" in *bits* even when one is working with a cost model where *internal* operations can work with larger pieces of data at once. This means we are measuring our algorithm the same way other algorithms in the literature are measured, so comparisons will be meaningful.

Section 5.2, a simple term such as $\mathtt{x.f}$ would get its original principal typing $\langle \mathtt{x} : \{\!\!\{\mathtt{f}{:}\mathtt{'a}, \mathtt{r} \mathbin{/} \mathtt{q}{\Rightarrow}\mathtt{s}\}\!\!\} \vdash \mathtt{'a} \mid \{\mathtt{f}\} \subseteq \mathtt{s}, \varnothing \supseteq \mathtt{q}\rangle$ rewritten to $\langle \mathtt{x} : \{\!\!\{\mathtt{f}{:}\mathtt{'a}, \mathtt{r} \mathbin{/} \varnothing{\Rightarrow}\bot\}\!\!\} \vdash \mathtt{'a} \mid \varnothing\rangle$ which is far from principal. To preserve principality we must make sure that whenever we rewrite $T$ to $T'$ it holds that $T \sqsubseteq T'$ *and* $T' \sqsubseteq T$, in which case we say it is **safe** to rewrite $T$ to $T'$. In this section we identify some rewritings that are safe.

First, constraints that are already solved can be dropped:

**Lemma 5.17.** *Let* $T = \langle \Gamma \vdash \tau \mid C\rangle$ *and* $T' = \langle \Gamma \vdash \tau \mid \{c \in C \mid \neg \Vdash c\}\rangle$. *Then it is safe to rewrite* $T$ *to* $T'$. □

Second, it is safe to perform the $\bot$-less solving of $s$ variables described by Lemma 5.8:

**Lemma 5.18.** *Let* $T = \langle \Gamma \vdash \tau \mid C\rangle$ *and assume* $C \mathrel{\text{☞}} \mathcal{T}$. *Then it is safe to rewrite* $T$ *to* $\mathcal{T}(T)$. □

The construction in Lemma 5.9 is not safe. The one in Lemma 5.10 is only partially safe; $C \mathrel{\text{☛}} \mathcal{T}$ sets all $q$'s to a *lower* bound on the possible values of $q$. We should only *use* this lower bound when that entails no restriction on set variables that are visible in the type and environment parts of the typing. We should also exclude variables where the lower bound cannot yet be computed because the $S$ part of a $Q \supseteq Q \setminus S$ constraint is still a variable.

**Lemma 5.19.** *Let* $T = \langle \Gamma \vdash \tau \mid C\rangle$, *and let* $\mathbf{Q}$ *be the least subset of* $\boxed{\boxed{q}}$ *such that*

1. *If $q$ appears anywhere in $\Gamma$ or $\tau$, then $q \in \mathbf{Q}$.*
2. *If $(q \supseteq Q \setminus s) \in C$, then $q \in \mathbf{Q}$.*
3. *If $(q \supseteq q' \setminus \mathcal{L}) \in C$ then $q' \in \mathbf{Q} \Rightarrow q \in \mathbf{Q}$.*

*Let* $C \mathrel{\text{☛}} \mathcal{T}$, *and let* $\mathcal{T}'(q)$ *be* $\mathcal{T}(q)$ *when* $q \notin \mathbf{Q}$ *and* $q$ *otherwise. Then it is safe to rewrite* $T$ *to* $\mathcal{T}'(T)$. □

As a simple example of incremental constraint simplification, consider the term $(\mathtt{x} \oplus \{\!\!\{\mathtt{f} \triangleright \mathtt{y}, \mathtt{h} \triangleright \mathtt{z}\,;\, \mathtt{y} = \mathtt{z}\}\!\!\}).\mathtt{g}$. Its inferred but not simplified typing is $\langle \mathtt{x} : \{\!\!\{\mathtt{f}{:}\mathtt{'a}, \mathtt{g}{:}\mathtt{'b}, \mathtt{h}{:}\mathtt{'a}, \mathtt{r} \mathbin{/} \mathtt{q}{\Rightarrow}\mathtt{s}\}\!\!\} \vdash \mathtt{'b} \mid C\rangle$ where

$$C = \left\{ \begin{array}{l} \mathtt{q1} \supseteq \{\mathtt{h}\},\ \mathtt{s1} = \{\mathtt{f}\},\ \varnothing \supseteq \mathtt{q1} \setminus \mathtt{s}, \\ \mathtt{s2} = \mathtt{s} \uplus \mathtt{s1},\ \varnothing \supseteq \mathtt{q} \setminus \mathtt{s1},\ \{\mathtt{g}\} \subseteq \mathtt{s2} \end{array} \right\}$$

We first reduce for $s$ variables by Lemma 5.8. The only reduction in this case is $C \mathrel{\text{☞}} [\mathtt{s1} \mapsto \{\mathtt{f}\}]$; after this we have the same typing but with constraint set

$$C' = \left\{ \begin{array}{l} \mathtt{q1} \supseteq \{\mathtt{h}\},\ \{\mathtt{f}\} = \{\mathtt{f}\},\ \varnothing \supseteq \mathtt{q1} \setminus \mathtt{s}, \\ \mathtt{s2} = \mathtt{s} \uplus \{\mathtt{f}\},\ \varnothing \supseteq \mathtt{q} \setminus \{\mathtt{f}\},\ \{\mathtt{g}\} \subseteq \mathtt{s2} \end{array} \right\}$$

We now compute the various $\mathbf{Q}_\ell^{C'}$'s and get $C' \mathrel{\text{☛}} [\mathtt{q} \mapsto \varnothing, \mathtt{q1} \mapsto \{\mathtt{h}\}]$. In Lemma 5.19, $\mathbf{Q}$ is $\{\mathtt{q}\}$ (as $\mathtt{q}$ appears in the type assumption for $\mathtt{x}$), so we apply just the substitution $[\mathtt{q1} \mapsto \{\mathtt{h}\}]$ to get

$$C'' = \left\{ \begin{array}{l} \{\mathtt{h}\} \supseteq \{\mathtt{h}\},\ \{\mathtt{f}\} = \{\mathtt{f}\},\ \varnothing \supseteq \{\mathtt{h}\} \setminus \mathtt{s}, \\ \mathtt{s2} = \mathtt{s} \uplus \{\mathtt{f}\},\ \varnothing \supseteq \mathtt{q} \setminus \{\mathtt{f}\},\ \{\mathtt{g}\} \subseteq \mathtt{s2} \end{array} \right\}$$

After dropping solved constraints and using abbreviations for the unsolved ones, we get the simplified typing

$$\left\langle \mathtt{x} : \{\!\!\{ \begin{pmatrix} \mathtt{f}{:}\mathtt{'a}, \mathtt{g}{:}\mathtt{'b}, \\ \mathtt{h}{:}\mathtt{'a}, \mathtt{r} \end{pmatrix} \mathbin{/} \mathtt{q}{\Rightarrow}\mathtt{s} \}\!\!\} \vdash \mathtt{'b} \;\middle|\; \begin{array}{l} \{\mathtt{h}\} \subseteq \mathtt{s},\ \mathtt{s2} = \mathtt{s} \uplus \{\mathtt{f}\}, \\ \mathtt{q} \subseteq \{\mathtt{f}\},\ \{\mathtt{g}\} \subseteq \mathtt{s2} \end{array} \right\rangle$$

### 5.5 Discussion

Martini allows compositional type inference for mixin modules without type annotations. Previous typed mixin module calculi [7, 4, 1, 12, 13] have not considered type inference, and several expect full type annotations for difficult operations. Because of the row pollution problem, Martini sometimes fails to type some terms unless the programmer switches to let-polymorphism (not always possible) or inserts dummy hiding operations. In contrast, previous mixin module systems tend to need full type annotations for these problem terms. Also, as Section 4.4 points out, the naive approach of previous type systems leads to unfeasible NP-complete typability in the absence of full type annotations.

Some previous mixin calculi have features not supported by the $\overline{\mathrm{m}}$-calculus or Martini. Duggan and Sourelis [5] allow different cases of a function that operates by pattern matching to come from different modules. Ancona et al. [2] allow extracting an export from a mixin module with imports, if the types prove that the export does not depend on the imports.

If we restrict Martini to the $\boldsymbol{\lambda}^{\oplus}$ fragment of the $\overline{\mathrm{m}}$-calculus that we defined in Section 3.2, we can derive a type system for the $\lambda$-calculus with record concatenation. We call this system Bowtie; it arises by adding a primitive type constructor $\rightarrow$ for function types and then fixing the $Q$ part of all remaining record types to $\varnothing$.

Our entire development for Martini, including the type inference algorithm and its complexity analysis, carries over to Bowtie without change (except that of course constraint solving for $q$ variables can be omitted).

Bowtie is not the first type system for a $\lambda$-calculus with record concatenation that supports type inference, but to the best of our knowledge there are no previous systems with a published type-inference complexity bound as low as $O(nm\alpha(n))$. One system by Rémy [21] can probably be implemented within this bound; unfortunately the types in this system are excessively inflexible and it can not reasonably be used for programming. A series of systems have been defined by Pottier [17, 18, 19], but the only one of these that has a complexity analysis [19] has complexity $O(n^3 m \log m)$, which is significantly more than Bowtie's $O(nm\alpha(n))$.

Previous type systems for record concatenation include work by Wand [27, 28], Harper and Pierce [8, 9], Rémy [21, 23], Zwanenburg [32], Pottier [17, 18, 19], and Palsberg and Zhao [16]. Rémy's system in [23] is the earliest to tolerate type "errors" in dead and sleeping code. The Bot concept in Pottier's system in [17] is very similar to the "$\bot$" in Bowtie (and Martini).

## 6. Row unification

In this section we describe an efficient algorithm for computing most general unifiers modulo the row structure equation $(\ell_1{:}\tau_1, \ell_2{:}\tau_2, R) = (\ell_2{:}\tau_2, \ell_1{:}\tau_1, R)$. This is used in our type inference procedure, but we also believe it has some general interest. This **row unification** problem has a standard algorithm (see for example [20]), but a direct implementation of this is too slow to fit within the $O(nm\alpha(n))$ complexity we need. The more efficient procedure we define here may have been independently developed by many implementers, but appears never to have been written down explicitly. We think it deserves to be recorded.

The key idea is to handle an entire row equation $(\ell_1{:}\tau_1, \ldots, \ell_k{:}\tau_k, r) = (\ell'_1{:}\tau'_1, \ldots, \ell'_{k'}{:}\tau'_{k'}, r')$ in a single step. Whenever $\ell_i = \ell'_j$, a unification of $\tau_i = \tau'_j$ must be scheduled; labels that appear only on one side must have corresponding entries added to the end of the list of the other side. The standard algorithm uses up to $\Omega(m^2)$ label commutation steps to bring the elements on either side of the equation into the same order. This corresponds to linear searches through the two row expressions; with a monolithic solution we are free to use more efficient data structures such a search trees or arrays for this purpose.

The only point where our algorithm differs from an implementation of the standard algorithm for row unification is in steps (b) and (c) of the RECUNIF operation in Figure 6. We describe the algorithm at a more concrete level than most unification descriptions in the literature. It is usual for such description to allow wholesale manipulation of type terms and substitutions; but working at that level would reduce our complexity analysis to pure handwaving.

### 6.1 Data representation

During row unification (and type inference in general) labels will be represented as small integers between 1 and $m$, the number of distinct labels in the program. This allows representing label-indexed maps as arrays. In the original input labels are usually strings that must be interned to small integers before type inference; this can be done in a linear-time pass over the program using a trie recording indexes for already-seen labels.

In the algorithm type expressions are represented as a directed acyclic graph of pointer-linked memory blocks. We use the metavariable $X$ for pointers to nodes in the type graph. Each node has a **sort** from the set $\{\mathsf{TYPE}, \mathsf{ROW}, \mathsf{QSET}, \mathsf{SSET}\}$, corresponding to the syntactic categories $\boxed{\tau}$, $\boxed{R}$, $\boxed{Q}$ and $\boxed{S}$, respectively. The sorts need not be explicitly present in the graph at run-time, but it is useful to imagine that they are. Each node also has a **contents** which is either $\mathsf{VAR}$ or $\xi(X_1,\dots,X_n)$ where $\xi$ is one of the following type constructors:[5]

- $\{\!\{\cdot\,/\,\cdot\Rightarrow\cdot\}\!\}$ with kind $\mathsf{ROW}\times\mathsf{QSET}\times\mathsf{SSET}\to\mathsf{TYPE}$.
- $\ell{:}\cdot,\cdot$, for any $\ell$, with kind $\mathsf{TYPE}\times\mathsf{ROW}\to\mathsf{ROW}$.
- $\bot$ with kind $\mathsf{SSET}$.
- $\mathcal{L}$, for any $\mathcal{L}$, with kind $\mathsf{SSET}$ or $\mathsf{QSET}$.

When the contents of a node $X$ is $\xi(X_1,\dots,X_n)$, the number and sorts of nodes $X_1$ through $X_n$, as well as the sort of $X$ itself, must be as specified by the kind of $\xi$.

A node with contents $\mathsf{VAR}$ represents a type (or row or set) variable of the appropriate sort. Each such variable is represented by exactly one node; we can thus identify the variable with the node and do not need explicit names for type variables until and unless we want to output the inferred type textually.

The nodes in the type graph are elements of a union-find data structure, which provides the following four primitives: EQUAL$(X_1,X_2)$ tests whether $X_1$ and $X_2$ refer to the same node. READ$(X)$ produces the contents of node $X$. FUSE$(X_1,X_2,\zeta)$ creates a single node with contents $\zeta$ and destructively redirects all existing references to either $X_1$ or $X_2$ such that they now refer to the new node. (The unification algorithm only performs this operation when $X_1$ and $X_2$ have the same sort, so the sort of the new node is unambiguous.) Finally, one may CREATE a new node with a given sort and contents. It is well known that $k$ operations in the union-find structure can be done in total time at most $O(k\,\alpha(k))$, where $\alpha$ is an extremely slow-growing function.[6]

### 6.2 Consistent graphs

The **exposed nodes** of the type graph are all nodes of sort $\mathsf{ROW}$ that are mentioned in the contents of a node whose sort is *not* $\mathsf{ROW}$. That is, the only $\mathsf{ROW}$ nodes that are *not* exposed are those that are linked to either not at all or only from $\mathsf{ROW}$ nodes. The state of the algorithm is **consistent** iff there exists a function $\psi\in\boxed{X}\to\mathcal{P}_{\text{fin}}(\boxed{\ell})$ such that $\psi(X)=\varnothing$ for all exposed nodes $X$, and whenever READ$(X)=\ell{:}X', X''$ it holds that $\psi(X'')=\psi(X)\uplus\{\ell\}$ (and $\psi(X)$ can be undefined if $X$ is not of sort $\mathsf{ROW}$). Consistency means that each row variable always appears after the same set of labels (although they may appear in different orders).

It is an invariant of the algorithm that its state is consistent. This ensures that we will not have to worry about creating ill-formed rows that define the same label twice; the map $\psi$ in the definition corresponds to the kinding discipline of [20, pp. 647ff]. In particular, the initial state produced by the caller must be consistent or the algorithm will not work. Fortunately, it is easy see that the equation sets produced by the COLLECT procedure of Section 5.3 will naturly be represented by consistent graphs. (Each recursive invocation of COLLECT constructs an isolated component of the graph that is initially connected to other components solely through the unification queue $W$, which has no influence on consistency.)

[5] For other type systems than $\mathsf{Martini}$ it is easy to add more type constructors, for example "$\to$" of kind $\mathsf{TYPE}\to\mathsf{TYPE}\to\mathsf{TYPE}$. The only fixed part of the signature is the constructors for $\mathsf{ROW}$ which must be exactly "$\ell{:}\cdot,\cdot$" and nothing else.

[6] A possible definition is: $\alpha(k)$ is the least $i\geq 1$ such that $A(i,4)>\log k$, where $A$ is a variant of Ackermann's function [24].

1. For each type, row or set variable $\beta$ in $W$, CREATE a node with the appropriate sort and contents $\mathsf{VAR}$. Let $\theta$ be the map from each $\beta$ to its corresponding node.
2. Replace each $(\tau,\tau')$ in $W$ with the pair $(\mathrm{BUILD}_\theta(\tau), \mathrm{BUILD}_\theta(\tau'))$.
3. Do steps (4)-(10) for as long as $W$ is not empty. Go to step (11) when $W$ becomes empty.
4. Select an equation $(X,X')$ from $W$ and remove it from $W$.
5. If EQUAL$(X,X')$, then discard the pair and start over from step (3).
6. If READ$(X)=\mathsf{VAR}$, then execute FUSE$(X,X',$ READ$(X'))$, and start over from step (3).
   If READ$(X')=\mathsf{VAR}$, then execute FUSE$(X,X',$ READ$(X))$, and start over from step (3).
7. Set $\xi(X_1,\dots,X_k)=\mathrm{READ}(X)$ and set $\xi'(X'_1,\dots,X'_{k'})=\mathrm{READ}(X')$. (This step will not be reached unless the contents of the two nodes have this form.)
8. If $\xi\neq\xi'$, then the unification problem is intrinsically unsolvable. Report failure and terminate the algorithm. ($\mathsf{Martini}$ has only one constructor of sort $\mathsf{TYPE}$, but this can happen for $\mathsf{SSET}$ and $\mathsf{QSET}$.)
9. Execute FUSE$(X,X',\xi(X_1,\dots,X_k))$.
10. Execute RECUNIF$(X_i,X'_i)$, defined below, for $1\leq i\leq k$, and start over from step (3).
11. (This step is reached when $W$ becomes empty.) Search for cycles in the type graph, for example by a depth-first traversal looking for back edges. If any are found, then stop and report failure.
12. If this step is reached, then the unification has succeeded. Return the substitution $\{\,\mathrm{READOUT}(\theta(\beta))\mid\beta\in\mathrm{Dom}\,\theta\,\}$.

***Figure 5.*** *Definition of the unification algorithm* RUNIFY$(W)$

### 6.3 The algorithm

The unification algorithm is defined in Figures 5 and 6. The algorithm keeps a queue $W$ of waiting equalities, represented as pairs $(X_1,X_2)$. It is an invariant that when $(X_1,X_2)\in W$, then $X_1$ and $X_2$ must have the same sort, which *must not be* $\mathsf{ROW}$. In step (4), the strategy for picking an equation to remove is not important; the usual *recursive* unification algorithm corresponds to a LIFO queue that coincides with the implementation language's call stack.

Step (11) implements the *occurs check*, which is usually described as being part of step (6), but the latter is executed too often to afford doing it there if we want almost-linear complexity. If (contrary to most real-world type checkers that want to report errors to the user in a readable way) we are not interested in knowing *why* the unification failed, it suffices to check the type graph for cycles once after the unification queue becomes empty.

Notice that part (b) and (c) of RECUNIF is the only place in the algorithm that specifically concerns rows. All other steps appear unchanged in the well-known first order unification algorithm.

**Theorem 6.1.** *The* RUNIFY *algorithm produces a most general unifier whenever its input has any unifier. If the input does not have a unifier it will terminate with a failure report.* □

This is the RECUNIF$(X,X')$ procedure used in step (10) of RUNIFY:

a. If the sort of $X$ and $X'$ is *not* ROW, just add $(X,X')$ to $W$ and return.

b. (If this step is reached, both of $X$ and $X'$ are exposed ROW nodes.) Let $(\varphi,X_0) = \text{GETROW}(X)$ and $(\varphi',X'_0) = \text{GETROW}(X')$. For each $\ell \in (\text{Dom}\,\varphi \cup \text{Dom}\,\varphi')$ (in some arbitrary order), do

   1. If $\ell \in (\text{Dom}\,\varphi \cap \text{Dom}\,\varphi')$, add $(\varphi(\ell),\varphi'(\ell))$ to $W$.
   2. Otherwise, if $\ell \in \text{Dom}\,\varphi$: CREATE a new node $X'_1$ with sort ROW and contents VAR. Then execute $\text{FUSE}(X'_0,X'_0,(\ell{:}\varphi(\ell),X'_1))$, and set $X'_0 := X'_1$.
   3. Otherwise, if $\ell \in \text{Dom}\,\varphi'$: CREATE a new node $X_1$ with sort ROW and contents VAR. Then execute $\text{FUSE}(X_0,X_0,(\ell{:}\varphi'(\ell),X_1))$, and set $X_0 := X_1$.

c. Execute $\text{FUSE}(\mathrm{X_0},\mathrm{X'_0},\text{VAR})$ and return.

---

$\text{BUILD}_\theta(t)$ is a side-effecting function from type terms $t$ to node names, parameterized by a map $\theta$ from variables to $\boxed{X}$:

$\text{BUILD}_\theta(\beta) = \theta(\beta)$.
$\text{BUILD}_\theta(\xi(t_1,\ldots,t_n)) =$
  let $X_i = \text{BUILD}_\theta(t_i)$ for $1 \le i \le n$
  in CREATE a node with contents $\xi(\mathrm{X}_1,\ldots,\mathrm{X}_\mathrm{m})$
    and return its name.

---

GETROW$(X)$ is defined when $X$ is a node of sort ROW and the unification graph is *consistent*:

$$\begin{array}{l}\text{GETROW}(X) = \\ \quad \text{case READ}(X) \text{ of } \ell{:}X', X'' \Rightarrow \text{let } (\varphi,X''') = \text{GETROW}(X'') \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{in } (\{\ell \mapsto X'\} \boxplus \varphi, X''') \\ \qquad\qquad\qquad\quad \mid \text{VAR} \quad \Rightarrow (\varnothing, X)\end{array}$$

---

READOUT maps a node name in an *acyclic* graph to a type term. It depends on a fixed injective mapping $\kappa$ from node names to variables of appropriate sorts.

$$\begin{array}{l}\text{READOUT}(X) = \\ \quad \text{case READ}(X) \text{ of} \\ \quad \xi(X_1,\ldots,X_n) \Rightarrow \xi(\text{READOUT}(X_1),\ldots,\text{READOUT}(X_n)) \\ \mid \text{VAR} \qquad\quad\ \Rightarrow \kappa(X)\end{array}$$

***Figure 6.*** *Helper definitions for* RUNIFY

**Theorem 6.2.** *The* RUNIFY *algorithm, except for the final* READ-OUT *operations, runs in time $O(nm\alpha(n))$, where n is the total size of the input and m is the number of different field labels mentioned in it.* □

*Proof.* Let $N$ be the largest number of operands of any type constructor; this is assumed to be a constant. When $\varphi$'s are represented as arrays, each GETROW operation uses at most $O(m)$ time plus $m$ union-find operations. Each RECUNIF operation uses at most $O(m)$ time plus $4m+1$ union-find operations. It may add up to $m$ pairs to $W$. The total number of RECUNIF operations is at most $Nn$, because step (10) is always done together with (9), which decreases by one the number of non-ROW nodes (which are never created after step (2)). Therefore, at most $Nnm$ pairs are added to the $O(n)$ initial pairs in $W$ during the algorithm. Steps (1)–(10) of the main algorithm are executed at most $O(Nnm) + O(n) = O(nm)$ times. Each execution spends constant time plus a constant number of union-find operations outside RECUNIF. Step (11) uses a constant number of union-find operations for each graph edge. The number of edges is bounded by the number of operations already performed (each edge must have been added at some time), so because step (11) happens only once, it can at most increase the number of operations performed by a constant factor.

In total, $O(nm) + O(Nn(4m+1)) = O(nm)$ union-find operations are executed. The work except for these is also $O(nm)$, so the total time complexity of the unification algorithm is $O(nm) + O(O(nm)\alpha(O(nm))) = O(nm\alpha(n))$, as required. □

| | | |
|---|---|---|
| Terms: | $M,N ::= \cdots \mid \mathbf{let}\ x = N\ \mathbf{in}\ M$ | |
| Generalized t-vars: | $\beta ::= s \mid q \mid r \mid \alpha$ | |
| Variable sets: | $B \in \mathcal{P}_{\text{fin}}(\boxed{\beta})$ | |
| Type schemes: | $\sigma ::= \forall B.\langle \tau \mid C\rangle$ | |
| Environments: | $\Gamma \in \boxed{x} \xrightarrow{\text{fin}} \boxed{\tau} \cup \boxed{\sigma}$ | |

$$\frac{}{\mathbf{let}\ x = N\ \mathbf{in}\ M \hookrightarrow [x \mapsto N]M}\ \textsf{RLet}$$

$$\frac{\Gamma(x) = \forall B.\langle \tau \mid C\rangle \qquad \forall \beta \notin B : \mathcal{T}(\beta) = \beta}{x : \langle \Gamma \vdash \mathcal{T}(\tau) \mid \mathcal{T}(C) \cup C'\rangle}\ \textsf{TPoly}$$

$$\frac{\begin{array}{c}N : \langle \Gamma \vdash \tau \mid C\rangle \qquad B = \text{FTV}(\tau,C) \setminus \text{FTV}(\Gamma) \\ M : \langle \Gamma \boxplus \{x \mapsto \forall B.\langle \tau \mid C\rangle\} \vdash \tau' \mid C'\rangle\end{array}}{\mathbf{let}\ x = N\ \mathbf{in}\ M : \langle \Gamma \vdash \tau' \mid C'\rangle}\ \textsf{TLet}$$

***Figure 7.*** *Adding Hindley/Milner polymorphism to* Martini *to make* Martini$^\forall$. *Things not defined here are in Figure 3.*

## 7. Let-polymorphism

Because Martini has principal typings, we can add to it Milner's let-polymorphism to get a type system Martini$^\forall$ that is to Martini what the Hindley/Milner (HM) type system (used by languages like ML) is to the simply typed λ-calculus. Figure 7 shows the completely conventional additions that do this. Note that type schemes have a constraint component, as is often done in HM extensions involving constraints.

We know that the additional rules in Figure 7 correctly implement let-polymorphism, because Martini$^\forall$ types the same programs as would be typed by Martini extended with the single rule

$$\frac{N : \langle \Gamma \vdash \tau \mid C\rangle \qquad [x \mapsto N]M : \langle \Gamma \vdash \tau' \mid C'\rangle}{\mathbf{let}\ x = N\ \mathbf{in}\ M : \langle \Gamma \vdash \tau' \mid C'\rangle}$$

which deliberately ignores $\tau$ and $C$ and is well known to correctly characterize the power of Milner's let-polymorphism. Principal types relative to a given $\Gamma$ in Martini$^\forall$ can be computed by the obvious extension of the standard algorithm $\mathcal{W}$ or one of its variants. Interleaved constraint simplification yields ground principal types. Note that, as usual for systems extended with Milner's let-polymorphism, Martini$^\forall$ has only the weaker principal *types*, not principal *typings*.

Note that let-polymorphism is not enough for ML-style modules. Entire mixin modules can be polymorphic, but not individual module components, unlike ML structures and functors, which also have features addressing type abstraction and the *diamond import* problem such as type components in structures and type sharing specifications. More work is needed to add such features to Martini.

# 8. Conclusion

## 8.1 Summary of contributions

This paper makes these novel contributions:

1. Section 4 proves that typability is NP-complete for Riviera, the straightforward system of simple types for the $\overline{\text{m}}$-calculus, a calculus of first-class mixin modules with symmetric linking. Riviera roughly corresponds to previous mixin module type systems. We point out that the expense comes from the type system checking constraints from *dead* or *sleeping* code.
2. Because our NP-completeness proof (1) works for the restriction of Riviera to the $\boldsymbol{\lambda}^{\oplus}$ subset of the $\overline{\text{m}}$-calculus and (2) is insensitive to whether linking is symmetric or asymmetric (overriding), we have also proven that type inference is NP-complete for Wand's type system for the $\lambda$-calculus with record concatenation [28]. The only similar previous NP-completeness result is by Palsberg and Zhao [16] for a more complicated system with subtyping.
3. Section 5 develops Martini, a system of simple types for the $\overline{\text{m}}$-calculus. Martini is conceptually simple, with no subtyping and a clean and balanced separation between (1) traditional simple types with type and row variables for determining field types and (2) constraints for safety of linking and field extraction.
4. Section 5 also develops type inference for Martini, and proves that Martini has principal typings [29]. Martini is the first type system for first-class mixin modules with a type inference algorithm. Its time complexity is $O(nm\,\alpha(n))$, where the input has size $n$ and $m$ distinct field labels, and $\alpha(n)$ is negligible.
5. By restricting Martini to the $\boldsymbol{\lambda}^{\oplus}$ subset of the $\overline{\text{m}}$-calculus, we achieve type inference for a $\lambda$-calculus with symmetric record concatenation with the same complexity, better than the previously best published complexity for any type inference algorithm for record concatenation. (Some previous type inference algorithms for record concatenation without published complexity analyses may have comparable complexities.)
6. Section 6 presents an efficient implementation of *row unification* with a rigorous complexity analysis.
7. We have implemented inference of principal typings for Martini; our implementation can be downloaded or used on-line at ⟨URL:`http://www.macs.hw.ac.uk/DART/software/Martini/`⟩.
8. Section 7 shows how to extend Martini with Milner's *let-polymorphism* to make Martini$^{\forall}$, which we believe is the first polymorphic type system for first-class mixin modules.

## 8.2 Acknowledgements

We are grateful to Davide Ancona, Sonia Fagorzi, and Elena Zucca for discussions during a visit by us to Genova which led directly to this work. We thank François Pottier and James Cheney for enlightening electronic discussions of the complexity of row unification, and Elena Zucca, Tom Hirschowitz, Gérard Boudol, and anonymous referees for helpful comments.

# References

[1] D. Ancona, S. Fagorzi, E. Moggi, E. Zucca. Mixin modules and computational effects. In *Proc. 30th Int'l Coll. Automata, Languages, and Programming*, vol. 2719 of *LNCS*. Springer-Verlag, 2003.

[2] D. Ancona, S. Fagorzi, E. Zucca. A calculus with lazy module operators. In *IFIP TC1 3rd Int'l Conf. Theoret. Comput. Sci. (TCS '04)*. Kluwer Academic Publishers, 2004.

[3] D. Ancona, E. Zucca. A primitive calculus for module systems. In *Proc. Int'l Conf. Principles & Practice Declarative Programming*, vol. 1702 of *LNCS*. Springer-Verlag, 1999.

[4] D. Ancona, E. Zucca. A calculus of module systems. *J. Funct. Programming*, 12(2), 2002. Extended version of [3].

[5] D. Duggan, C. Sourelis. Mixin modules. In *Proc. 1996 Int'l Conf. Functional Programming*. ACM Press, 1996.

[6] *Programming Languages & Systems, 9th European Symp. Programming*, vol. 1782 of *LNCS*. Springer-Verlag, 2000.

[7] M. Flatt, M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN '98 Conf. Prog. Lang. Design & Impl.*, 1998.

[8] R. Harper, B. C. Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157R, Carnegie Mellon Univ., 1991.

[9] R. Harper, B. C. Pierce. A record calculus based on symmetric concatenation. In *Conf. Rec. 18th Ann. ACM Symp. Princ. of Prog. Langs.*, 1991.

[10] T. Hirschowitz. *Mixin Modules, Modules, and Extended Recursion in a Call-by-Value Setting*. PhD thesis, Université Paris 7, 2003.

[11] T. Hirschowitz. Rigid mixin modules. In *Seventh International Symposium on Functional and Logic Programming (FLOPS 2004)*, 2004.

[12] T. Hirschowitz, X. Leroy. Mixin modules in a call-by-value setting. In *Programming Languages & Systems, 11th European Symp. Programming*, vol. 2305 of *LNCS*. Springer-Verlag, 2002.

[13] T. Hirschowitz, X. Leroy, J. B. Wells. Call-by-value mixin modules: Reduction semantics, side effects, types. In *Programming Languages & Systems, 13th European Symp. Programming*, vol. 2986 of *LNCS*. Springer-Verlag, 2004. More details can be found in [10].

[14] H. Makholm, J. B. Wells. Type inference, principal typings, and let-polymorphism for first-class mixin modules. Technical report, Heriot-Watt Univ., School of Math. & Comput. Sci., 2005.

[15] A. Ohori, P. Buneman. Type inference in a database programming language. In *Proc. 1988 ACM Conf. LISP Funct. Program.*, Snowbird, Utah, U.S.A., 1988.

[16] J. Palsberg, T. Zhao. Type inference for record concatenation and subtyping. *Inform. & Comput.*, 189, 2004.

[17] F. Pottier. A 3-part type inference engine. In ESOP '00 [6].

[18] F. Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4), 2000.

[19] F. Pottier. A constraint-based presentation and generalization of rows. In *Proc. 18th Ann. IEEE Symp. Logic in Comput. Sci.*, 2003.

[20] F. Pottier, D. Rémy. The essence of ML type inference. In B. C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, chapter 10. MIT Press, Cambridge, Massachusetts, 2005.

[21] D. Rémy. Typing record concatenation for free. In *Conf. Rec. 19th Ann. ACM Symp. Princ. of Prog. Langs.*, 1992. A later version is [22].

[22] D. Rémy. Typing record concatenation for free. In C. A. Gunter, J. C. Mitchell, eds., *Theoretical Aspects Of Object-Oriented Programming: Types, Semantics and Language Design*. MIT Press, 1993.

[23] D. Rémy. A case study of typechecking with constrained types: Typing record concatenation. Presented at the workshop on Advances in Types for Computer Science at the Newton Institute, Cambridge, UK, 1995.

[24] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2), 1975.

[25] S. Vansummeren. On the complexity of deciding typability in the relational algebra. *Acta Informatica*, 200X. To appear.

[26] M. Wand. Complete type inference for simple objects. In *Proc. 2nd Ann. Symp. Logic in Comput. Sci.*, 1987. A corrigendum appeared at LICS 1988.

[27] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. 4th Ann. Symp. Logic in Comput. Sci.*, Pacific Grove, CA, U.S.A., 1989. IEEE Comput. Soc. Press.

[28] M. Wand. Type inference for record concatenation and multiple inheritance. *Inform. & Comput.*, 93, 1991.

[29] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, vol. 2380 of *LNCS*. Springer-Verlag, 2002.

[30] J. B. Wells, R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules (long version). A short version is [31]. Full paper, 3 appendices of proofs, 1999.

[31] J. B. Wells, R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In ESOP '00 [6]. A long version is [30].

[32] J. Zwanenburg. A type system for record concatenation and subtyping. In K. Bruce, G. Longo, eds., *Third Workshop on Foundations of Object Oriented Languages (FOOL 3)*, Rutgers Univ., NJ, USA, 1996.