



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

Henning Makholm and Joe Wells

Technical University of Denmark and Heriot-Watt University

ICFP'05 – Tallinn, Estonia – September 27, 2005

Work supported by EU/IST/FET grant 2001-33477 (DART)



What is this all about?

• The \bar{m} -calculus:

Variables: $x ::= x \mid y \mid z \mid \dots$

Field labels: $\ell ::= f \mid g \mid h \mid \dots$

Terms: $M, N ::= x \mid M.\ell \mid M \oplus N \mid \{\gamma_1; \dots; \gamma_k\}$

Module items: $\gamma ::= \text{local } x = M$
 | $\text{export } \ell \triangleright x = M$
 | $\text{import } \ell \triangleright x$

• We recognize variables, field selection, some kind of fancy sum operator, but what on earth is “ $\{\gamma_1, \dots, \gamma_k\}$ ”?

• Is it a *record*...?

• Is it a *letrec*...?

• Is it a *closure*...?

• It is a *mixin module*!



Anatomy of a mixin module

• The \bar{m} -calculus:

Variables: $x ::= x \mid y \mid z \mid \dots$

Field labels: $\ell ::= f \mid g \mid h \mid \dots$

Terms: $M, N ::= x \mid M.\ell \mid M \oplus N \mid \{\gamma_1; \dots; \gamma_k\}$

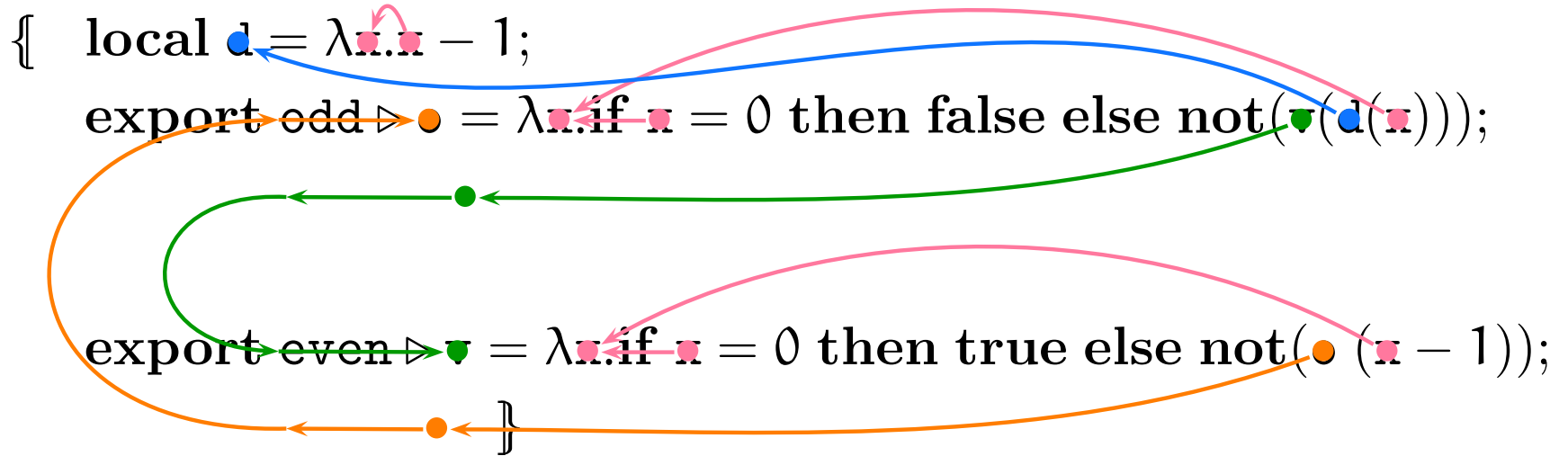
Module items: $\gamma ::=$
 $\text{local } x = M$
 \mid $\text{export } \ell \triangleright x = M$
 \mid $\text{import } \ell \triangleright x$

• A *mixin module* consists of three kinds of *items*:

- A local definition binds an *internal name* x to an *expression* M
- An export binds an *external label* ℓ to an *expression* M (while also providing a local definition for x)
- An import binds an *internal name* x to a *linking promise* represented by the *external label* ℓ .



An example



- Linking uses one operand's *exports* to bind the other one's *imports*
- ... and vice versa: Linking is *symmetric*
 - *Recursion* may arise “spontaneously”
 - Well-known examples of symmetric linking include C compilation units, but *not* ML modules



Other module operations

● Field extraction

```
{ local d = λx.x - 1;  
  export odd ▷ o = λx.if x = 0 then false else not(v(d(x)));  
  export even ▷ v = λx.if x = 0 then true else not(o(x - 1));  
}.even(42) ↪* true
```

● Field hiding makes exports into locals

```
{ export f ▷ x = 5;  
  export g ▷ y = x + 17; } ∥ {f}
```

```
↪ { local x = 5;  
    export g ▷ y = x + 17; }
```



Modules as first-class citizens

- Modules can be bound to variables and can be imported/exported from other modules
- Expressions in module constructions can mention variables that are bound in an enclosing scope
 - $\{\text{local } x = 5; \text{ export } f \triangleright _ = \{\text{export } g \triangleright _ = x + 7\}\}$
- Constructs from **lambda calculus** can be simulated directly:
 - $(\lambda x.M) \implies \{\text{import in } \triangleright x; \text{ export out } \triangleright _ = M\}$
 - $M N \implies (M \oplus \{\text{export in } \triangleright _ = N\}).\text{out}$
- **Records** are a restricted case of modules:
 - $\{f = 42; g = \text{"foo"}\}$
 $\implies \{\text{export } f \triangleright _ = 42; \text{ export } g \triangleright _ = \text{"foo"}\}$
 - In this case, *linking* becomes *record concatenation*.



The Martini type system: design goals

- *Type safety* by progress & subject reduction
 - No **extraction** of a field that *is not exported*
 - No **extraction** from a module with *unsatisfied imports*
 - No **linking** of two modules that *export the same name*
 - No **hiding** of names that the module *imports*
- Tractable automatic *type inference*. (First for mixin modules!)
 - Not trivial: The most straightforward typing rules for linking lead to an *NP-complete* typing problem
- *Principal typings* for compositional type inference
 - Principal *types*: Given M and Γ , find “best” τ s.t. $\Gamma \vdash M : \tau$
 - Principal *typings*: Given M , find “best” (Γ, τ) s.t. $\Gamma \vdash M : \tau$
 - ML types have principal types, but not principal typings

To emphasize principal typings, we write $M : \langle \Gamma \vdash \tau \rangle$

Martini types explained by example

- Let $M = \{$ `local d = $\lambda x.x - 1$;`
`export odd \triangleright _ = $\lambda x.$ if x = 0 then false`
`else not(e(d(x)));`
`import even \triangleright e; }`
- Some valid types for M are:
 - $\{ \underbrace{\text{even:int} \rightarrow \text{bool}, \text{odd:int} \rightarrow \text{bool}}_{\text{Types of all imports and exports}} / \underbrace{\{\text{even}\}}_{\text{Imports}} \Rightarrow \underbrace{\{\text{odd}\}}_{\text{Exports}} \}$
 - $\{ \underbrace{\text{even:int} \rightarrow \text{bool}, \text{odd:int} \rightarrow \text{bool}, \text{k:string}}_{\text{More types than necessary are OK}} / \{\text{even}\} \Rightarrow \{\text{odd}\} \}$
 - $\{ \text{even:int} \rightarrow \text{bool}, \text{odd:int} \rightarrow \text{bool}, \text{k:string} / \underbrace{\{\text{even}, \text{k}\}}_{\text{Overapproximation here OK}} \Rightarrow \{\text{odd}\} \}$
 - $\{ \underbrace{\text{even:int} \rightarrow \text{bool}, \text{odd:int} \rightarrow \text{bool}, \text{r0}}_{\text{Any row defining even and odd}} / \{\text{even}, \text{k}\} \Rightarrow \{\text{odd}\} \}$
 - $\{ \text{even:int} \rightarrow \text{bool}, \text{odd:int} \rightarrow \text{bool}, \text{r0} / \text{q0} \Rightarrow \{\text{odd}\} \}$
whenever the *constraint set* $\{\text{q0} \supseteq \{\text{even}\}\}$ is satisfied

Another example (principal) typing

Let $N = \lambda x. \lambda y. (x \oplus y).f + 42$

Giving this a principal typing is not trivial:

- The input modules x and y may need to exchange values before one of them can calculate a value for f .
- We don't even know which of x and y will export f . Exactly one of them must.

Row overapproximation to the rescue!

N has type $\{f:\text{int}, r / q1 \Rightarrow s1\} \rightarrow \{f:\text{int}, r / q2 \Rightarrow s2\} \rightarrow \text{int}$
with *constraint set* $\{s = s1 \uplus s2, \{f\} \subseteq s, s2 \supseteq q1, s1 \supseteq q2\}$

- Both rows contain “ $f:\text{int}$ ”
... meaning *if* the module exports f , *then* it has type int
- Constraints ensure that exactly one of $s1$ and $s2$ contain f
- More constraint to ensure that x 's imports are satisfied by y 's exports, and vice versa



Grammar for Martini types

Label sets: $\mathcal{L} ::= \{\ell_1, \dots, \ell_k\}$ ($k \geq 0$)
Type rows: $R ::= r_0 \mid r_1 \mid r_2 \mid \dots \mid \ell:\tau, R$
Import set expressions: $Q ::= q_0 \mid q_1 \mid q_2 \mid \dots \mid \mathcal{L}$
Export set expressions: $S ::= s_0 \mid s_1 \mid s_2 \mid \dots \mid \mathcal{L} \mid \perp$
Types: $\tau ::= 'a \mid 'b \mid 'c \mid \dots \mid \{R / Q \Rightarrow S\}$

- Each of the classes R , Q , S , τ has an infinity of **variables**
- The concrete form of Q and S is just a **set of field names**.
- An S can also be \perp ; the type $\{R / Q \Rightarrow \perp\}$ is always **void**
- A row R is a series of **field–type bindings**, always ending with a **row variable**
 - For all $\ell_1 \neq \ell_2$ we impose the equation $(\ell_1:\tau_1, \ell_2:\tau_2, R) = (\ell_2:\tau_2, \ell_1:\tau_1, R)$
 - There is no syntax for an empty row
 - a row variable can always be used instead



Constraint language

S constraints: $c^S ::= S = S_1 \uplus S_2 \mid S = S_1 \setminus \mathcal{L}$

Q constraints: $c^Q ::= Q \supseteq Q_1 \setminus S \mid Q \# \mathcal{L}$

Constraints: $c ::= c^S \mid c^Q$

Constraint sets: $C ::= \{c_1, \dots, c_k\}$ ($k \geq 0$)

- **Central principle:** Constraints speak about *label sets* only. (In particular, about possible instantiations of *set variables*).
The *structure of rows and types* is *independent* of constraints.
- Sets on a constraint **RHS** come from **operand** types. The set on the **LHS** is part of the **result** type.
- In S constraints the **operands** exactly determine the **result** set. Q constraints allow the **result** to overapproximate.
- A constraint with \perp on its **RHS** comes from an expression whose operand can never exist at run-time. Such a constraint is *by definition* solved.



Judgements and rules

• Typing judgements have the form $M : \langle \Gamma \vdash \tau \mid C \rangle$

• An example rule:

$$\frac{M : \langle \Gamma \vdash \{R / Q_1 \Rightarrow S_1\} \mid C \rangle \quad N : \langle \Gamma \vdash \{R / Q_2 \Rightarrow S_2\} \mid C \rangle \quad C \Vdash \{Q \supseteq Q_1 \setminus S_2, S = S_1 \uplus S_2, Q \supseteq Q_2 \setminus S_1\}}{M \oplus N : \langle \Gamma \vdash \{R / Q \Rightarrow S\} \mid C \rangle}$$

where $C_1 \Vdash C_2$ if all substitutions that solve C_1 also solve C_2 .

• The same C is used in the entire derivation.

• The concrete form of Q 's and S 's matters only in constraints.



Type inference for Martini

- **Phase 1a:** Ignore all rule premises of the form $C \Vdash \{\dots\}$
 - Then inference can be done by standard techniques using *first-order unification in the presence of rows*
 - During this phase Q's and S's are always **variables**
Set variables may get unified but get no more structure
 - Fixes the *type/row* structure of the principal typing
- **Phase 1b:** Let C be union of the right-hand-side of all the ignored $C \Vdash \{\dots\}$ premises
 - This C is the *most liberal* constraint set that entails all of the right-hand-sides
 - **Theorem:** The typing after Phase 1b is *principal*
- **Phase 2:** *Solve* the collected C
- In practice most of the three phases will be *interleaved*
(This gives **more readable** principal typings)



Formal properties

- Subject reduction holds
- If $M : \langle \emptyset \vdash \tau \mid C \rangle$ for C *solvable*, then M does not go wrong
- C can be checked for solvability in time
 $O(\text{(number of constraints)} \cdot \text{(number of different field labels)})$
Solutions sometimes contain \perp
- Type inference and constraint solving takes time
 $O(\text{(length of program)} \cdot \text{(number of different field labels)} \cdot \alpha(\dots))$
- **Oops!** $C_1 \Vdash C_2$ is hard to decide. But luckily we don't have to.



Reasons to read our paper

(beyond wanting to see the formalism behind the previous slides)

- How to *solve constraints* in our system and how to restrict the solving such as not to lose principality
- Extension of Martini with *let-polymorphism* by the Hindley–Milner procedure
- A *robust proof* that \perp -less typing rules for linking/concatenation operators make typability *NP-hard*. (Even just concatenable records in a first-order language)
- First written down presentation of an efficient algorithm for *row unification*
- URL for our *online type inference implementation*



Shortcomings and future work

- **Programming in the large:** Not nearly strong enough to subsume ML modules (yet!)
 - Needs polymorphism of `local` bindings in modules
 - Needs type components of modules
 - Needs type abstractions for modules
- **Row pollution:** When a module is constructed, the row part of its type will list types of all fields the module may ever be linked with.
 - Not that bad: The list will be *inferred*
 - Can be mitigated by *polymorphism* or by *programmer annotations*.
Perhaps stratified type inference could also be useful.



The end

Thank you

Questions?