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)

Useful Logics, Types, Rewriting, and their Automation

What is this all about?

- The $\overline{\text{m}}$-calculus:

$$
\begin{array}{lrcl}
\text{Variables:} & x & ::= & \mathtt{x} \mid \mathtt{y} \mid \mathtt{z} \mid \cdots \\
\text{Field labels:} & \ell & ::= & \mathtt{f} \mid \mathtt{g} \mid \mathtt{h} \mid \cdots \\
\text{Terms:} & M, N & ::= & x \mid M.\ell \mid M \oplus N \mid \{\!|\gamma_1; \ldots; \gamma_k|\!\} \\
\text{Module items:} & \gamma & ::= & \mathbf{local}\ x = M \\
 & & \mid & \mathbf{export}\ \ell \triangleright x = M \\
 & & \mid & \mathbf{import}\ \ell \triangleright x
\end{array}
$$

- We recognize variables, field selection, some kind of fancy sum operator, but what on earth is "$\{\!|\gamma_1, \ldots, \gamma_k|\!\}$"?
  - Is it a *record*...?
  - Is it a *letrec*...?
  - Is it a *closure*...?
- It is a *mixin module*!

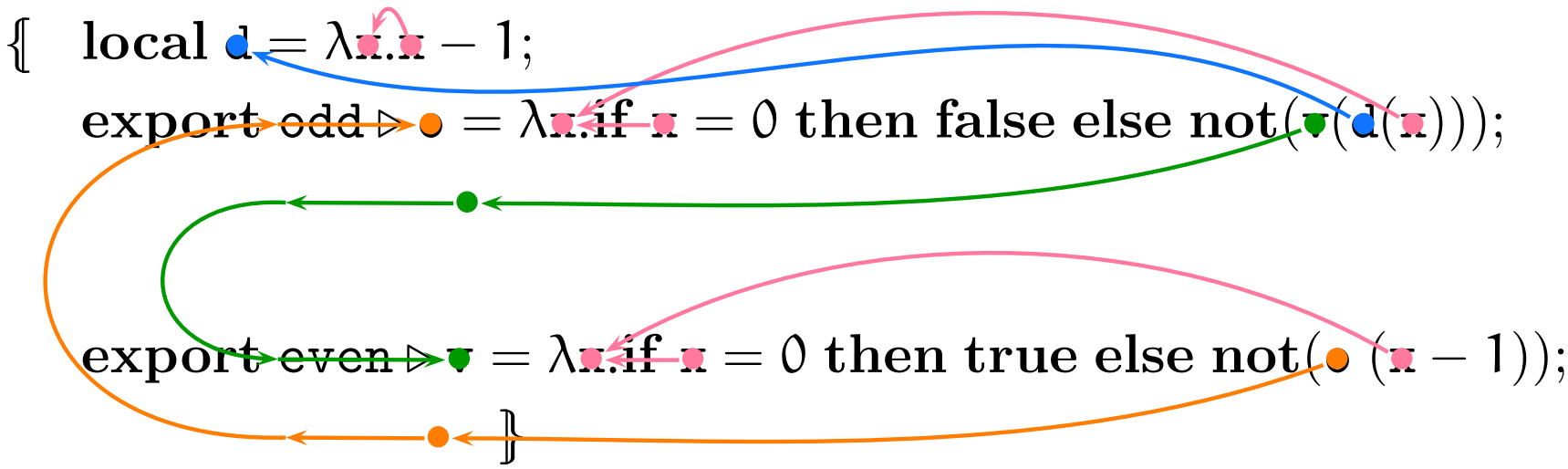
Anatomy of a mixin module

- The $\overline{\text{m}}$-calculus:

$$
\begin{array}{lrcl}
\text{Variables:} & \mathrm{x} & ::= & \mathtt{x} \mid \mathtt{y} \mid \mathtt{z} \mid \cdots \\
\text{Field labels:} & \ell & ::= & \mathtt{f} \mid \mathtt{g} \mid \mathtt{h} \mid \cdots \\
\text{Terms:} & \mathrm{M}, \mathrm{N} & ::= & \mathrm{x} \mid \mathrm{M}.\ell \mid \mathrm{M} \oplus \mathrm{N} \mid \{\!|\gamma_1; \ldots; \gamma_k|\!\} \\
\text{Module items:} & \gamma & ::= & \mathbf{local}\ \mathrm{x} = \mathrm{M} \\
 & & \mid & \mathbf{export}\ \ell \triangleright \mathrm{x} = \mathrm{M} \\
 & & \mid & \mathbf{import}\ \ell \triangleright \mathrm{x}
\end{array}
$$

- A *mixin module* consists of three kinds of *items*:
  - A $\mathbf{local}$ definition binds an *internal name* x to an *expression* M
  - An $\mathbf{export}$ binds an *external label* ℓ to an *expression* M (while also providing a local definition for x)
  - An $\mathbf{import}$ 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

$$\{\!|\ \mathbf{local}\ \mathtt{d} = \lambda \mathtt{x}.\mathtt{x} - 1;$$
$$\quad \mathbf{export}\ \mathtt{odd} \triangleright \mathtt{o} = \lambda \mathtt{x}.\mathbf{if}\ \mathtt{x} = 0\ \mathbf{then}\ \mathbf{false}\ \mathbf{else}\ \mathbf{not}(\mathtt{v}(\mathtt{d}(\mathtt{x})));$$
$$\quad \mathbf{export}\ \mathtt{even} \triangleright \mathtt{v} = \lambda \mathtt{x}.\mathbf{if}\ \mathtt{x} = 0\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{not}(\mathtt{o}(\mathtt{x} - 1));$$
$$|\!\}.\mathtt{even}(42) \hookrightarrow^* \mathbf{true}$$

- Field hiding makes exports into locals

$$\{\!|\ \mathbf{export}\ \mathtt{f} \triangleright \mathtt{x} = 5;$$
$$\quad \mathbf{export}\ \mathtt{g} \triangleright \mathtt{y} = \mathtt{x} + 17;\ \ |\!\} \setminus\!\setminus \{\mathtt{f}\}$$
$$\hookrightarrow\ \ \{\!|\ \mathbf{local}\ \mathtt{x} = 5;$$
$$\qquad \mathbf{export}\ \mathtt{g} \triangleright \mathtt{y} = \mathtt{x} + 17;\ \ |\!\}$$

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
  - $\{\!|\mathbf{local}\ \mathtt{x} = 5;\ \mathbf{export}\ \mathtt{f} \triangleright \_ = \{\!|\mathbf{export}\ \mathtt{g} \triangleright \_ = \mathtt{x} + 7|\!\}|\!\}$
- Constructs from lambda calculus can be simulated directly:
  - $(\lambda \mathrm{x}.\mathrm{M}) \Longrightarrow \{\!|\mathbf{import}\ \mathtt{in} \triangleright \mathrm{x};\ \mathbf{export}\ \mathtt{out} \triangleright \_ = \mathrm{M}|\!\}$
  - $\mathrm{M}\ \mathrm{N} \Longrightarrow (\mathrm{M} \oplus \{\!|\mathbf{export}\ \mathtt{in} \triangleright \_ = \mathrm{N}|\!\}).\mathtt{out}$
- Records are a restricted case of modules:
  - $\{\!|\mathtt{f} = 42;\ \mathtt{g} = \mathtt{"foo"}|\!\}$
    $\Longrightarrow \{\!|\mathbf{export}\ \mathtt{f} \triangleright \_ = 42; \mathbf{export}\ \mathtt{g} \triangleright \_ = \mathtt{"foo"}|\!\}$
  - 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 $\texttt{odd} \triangleright \_ = \lambda x.$**if** $x = 0$ **then false**
  else not$(e(d(x)));$
  import $\texttt{even} \triangleright e;$ $|\!\}$
- Some valid types for M are:
  - $\{\!|\texttt{even:int}\to\texttt{bool}, \texttt{odd:int}\to\texttt{bool} / \{\texttt{even}\} \Rightarrow \{\texttt{odd}\}\ |\!\}$
    Types of all imports and exports — Imports — Exports
  - $\{\!|\texttt{even:int}\to\texttt{bool}, \texttt{odd:int}\to\texttt{bool}, \texttt{k:string} / \{\texttt{even}\} \Rightarrow \{\texttt{odd}\}\ |\!\}$
    More types than necessary are OK
  - $\{\!|\texttt{even:int}\to\texttt{bool}, \texttt{odd:int}\to\texttt{bool}, \texttt{k:string} / \{\texttt{even},\texttt{k}\} \Rightarrow \{\texttt{odd}\}\ |\!\}$
    Overapproximation here OK
  - $\{\!|\texttt{even:int}\to\texttt{bool}, \texttt{odd:int}\to\texttt{bool}, \texttt{r0} / \{\texttt{even},\texttt{k}\} \Rightarrow \{\texttt{odd}\}\ |\!\}$
    Any row defining `even` and `odd`
  - $\{\!|\texttt{even:int}\to\texttt{bool}, \texttt{odd:int}\to\texttt{bool}, \texttt{r0} / \texttt{q0} \Rightarrow \{\texttt{odd}\}\ |\!\}$
    whenever the *constraint set* $\{\texttt{q0} \supseteq \{\texttt{even}\}\}$ is satisfied

Useful Logics, Types, Rewriting, and their Automation

Another example (principal) typing

- Let $\mathtt{N} = \lambda \mathtt{x}.\lambda \mathtt{y}.(\mathtt{x} \oplus \mathtt{y}).\mathtt{f} + 42$

  Giving this a principal typing is not trivial:

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

- Row overapproximation to the rescue!

  $\mathtt{N}$ has type $\{\!|\, \mathtt{f}\!:\!\mathsf{int},\ \mathtt{r} \mathbin{/} \mathtt{q1} \Rightarrow \mathtt{s1} \,|\!\} \to \{\!|\, \mathtt{f}\!:\!\mathsf{int},\ \mathtt{r} \mathbin{/} \mathtt{q2} \Rightarrow \mathtt{s2} \,|\!\} \to \mathsf{int}$
  with *constraint set* $\{\mathtt{s} = \mathtt{s1} \uplus \mathtt{s2},\ \{\mathtt{f}\} \subseteq \mathtt{s},\ \mathtt{s2} \supseteq \mathtt{q1},\ \mathtt{s1} \supseteq \mathtt{q2}\}$

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

Grammar for Martini types

Label sets:	$\mathcal{L} ::= \{\ell_1, \ldots, \ell_k\}$	$(k \geq 0)$
Type rows:	$R ::= \texttt{r0} \mid \texttt{r1} \mid \texttt{r2} \mid \cdots \mid \ell{:}\tau,\ R$	
Import set expressions:	$Q ::= \texttt{q0} \mid \texttt{q1} \mid \texttt{q2} \mid \cdots \mid \mathcal{L}$	
Export set expressions:	$S ::= \texttt{s0} \mid \texttt{s1} \mid \texttt{s2} \mid \cdots \mid \mathcal{L} \mid \bot$	
Types:	$\tau ::= \texttt{'a} \mid \texttt{'b} \mid \texttt{'c} \mid \ldots \mid \{\!\!\{ R \,/\, Q \Rightarrow S \}\!\!\}$	

- 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 $\bot$; the type $\{\!\!\{ R \,/\, Q \Rightarrow \bot \}\!\!\}$ 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

$$
\begin{array}{lrcl}
\text{S constraints:} & c^{S} & ::= & S = S_1 \uplus S_2 \mid S = S_1 \setminus \mathcal{L} \\
\text{Q constraints:} & c^{Q} & ::= & Q \supseteq Q_1 \setminus S \mid Q \# \mathcal{L} \\
\text{Constraints:} & c & ::= & c^{S} \mid c^{Q} \\
\text{Constraint sets:} & C & ::= & \{c_1, \ldots, c_k\} \qquad (k \geq 0)
\end{array}
$$

- 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 $\bot$ 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 $\boxed{M : \langle \Gamma \vdash \tau \mid C \rangle}$
- An example rule:

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

  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 $\mathcal{C} \Vdash \{\cdots\}$
  - Then inference can be done by standard techniques using *first-order unification in the presence of rows*
  - During this phase $\mathcal{Q}$'s and $\mathcal{S}$'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 $\mathcal{C}$ be union of the right-hand-side of all the ignored $\mathcal{C} \Vdash \{\cdots\}$ premises
  - This $\mathcal{C}$ 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 $\mathcal{C}$
- In practice most of the three phases will be *interleaved*
  (This gives more readable principal typings)

Formal properties

- Subject reduction holds
- If $M : \langle \varnothing \vdash \tau \mid C \rangle$ for C *solvable*, then M does not go wrong
- C can be checked for solvability in time
  $O\big((\text{number of constraints}) \cdot (\text{number of different field labels})\big)$
  Solutions sometimes contain $\bot$
- Type inference and constraint solving takes time
  $O\big((\text{length of program}) \cdot (\text{number of different field labels}) \cdot \alpha(\cdots)\big)$
- 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 $\bot$-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*

Useful Logics, Types, Rewriting, and their Automation

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 stratifed type inference could also be useful.

The end

Thank you

Questions?