Polymorphism, Subtyping, and Type Inference in MLsub

Stephen Dolan and Alan Mycroft

Computer Laboratory
University of Cambridge

November 8, 2016

The `select` function

$$\texttt{select}\ p\ v\ d = \texttt{if}\ (p\ v)\ \texttt{then}\ v\ \texttt{else}\ d$$

In ML, `select` has type scheme

$$\forall\alpha.\,(\alpha \to \texttt{bool}) \to \alpha \to \alpha \to \alpha$$

Data flow in `select`

$$\texttt{select}\ p\ v\ d = \texttt{if}\ (p\ v)\ \texttt{then}\ v\ \texttt{else}\ d$$

$$\begin{tikzcd}
 & v \arrow[dl] \arrow[dr] & & d \arrow[dl] \\
\text{argument to } p & & \text{result} &
\end{tikzcd}$$

Data flow in `select`

$$\texttt{select}\ p\ v\ d = \texttt{if}\ (p\ v)\ \texttt{then}\ v\ \texttt{else}\ d$$

$$\begin{tikzcd}[column sep=small]
& v \arrow[dl] \arrow[dr] & & d \arrow[dl] \\
\text{argument to } p & & \text{result} &
\end{tikzcd}$$

In MLsub, `select` has this type scheme:

$$\forall\alpha,\beta.\,(\alpha \to \texttt{bool}) \to \alpha \to \beta \to (\alpha \sqcup \beta)$$

$$\Gamma \vdash e : \tau$$

$$\Gamma \vdash e : \tau$$

Expressions of MLsub

We have functions

$$x \qquad \lambda x.e \qquad e_1\, e_2$$

... and records

$$\{\ell_1 = e_1, \ldots, \ell_n = e_n\} \qquad e.\ell$$

... and booleans

$$\text{true} \qquad \text{false} \qquad \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

... and `let`

$$\hat{\mathbf{x}} \qquad \text{let } \hat{\mathbf{x}} = e_1 \text{ in } e_2$$

$$\Gamma \vdash e : \tau$$

Typing rules of MLsub

MLsub is

ML $+$

$$(\textsc{Sub}) \quad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2} \; \tau_1 \leq \tau_2$$

$$\Gamma \vdash e : \tau$$

Constructing Types

The standard definition of types looks like:

$$\tau ::= \bot \mid \tau \to \tau \mid \top$$

(ignoring records and booleans for now)

Constructing Types

The standard definition of types looks like:

$$\tau ::= \bot \mid \tau \to \tau \mid \top$$

(ignoring records and booleans for now)
with a subtyping relation like:

$$\frac{}{\bot \leq \tau} \qquad \frac{}{\tau \leq \top} \qquad \frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

Lattices

These types form a lattice:

- least upper bounds $\tau_1 \sqcup \tau_2$
- greatest lower bounds $\tau_1 \sqcap \tau_2$

Lattices

These types form a lattice:

- least upper bounds $\tau_1 \sqcup \tau_2$
- greatest lower bounds $\tau_1 \sqcap \tau_2$

$$\frac{e_1 : \tau_1 \qquad e_2 : \tau_2}{\text{if } \texttt{rand ()} \text{ then } e_1 \text{ else } e_2 : \tau_1 \sqcup \tau_2}$$

Bizzarely difficult questions

Is this true, for all α?

$$\alpha \to \alpha \leq \bot \to \top$$

Bizzarely difficult questions

Is this true, for all α?

$$\alpha \to \alpha \leq \bot \to \top$$

How about this?

$$(\bot \to \top) \to \bot \leq (\alpha \to \bot) \sqcup \alpha$$

Bizzarely difficult questions

Is this true, for all α?

$$\alpha \to \alpha \leq \bot \to \top$$

How about this?

$$(\bot \to \top) \to \bot \leq (\alpha \to \bot) \sqcup \alpha$$

Yes, it turns out, by case analysis on α.

Bizzarely difficult questions

Is this true, for all α?

$$\alpha \to \alpha \leq \bot \to \top$$

How about this?

$$(\bot \to \top) \to \bot \leq (\alpha \to \bot) \sqcup \alpha$$

Yes, it turns out, by case analysis on α.

And *only* by case analysis.

Extensibility

Let's add a new type of function $\tau_1 \overset{\circ}{\to} \tau_2$.

Extensibility

Let's add a new type of function $\tau_1 \overset{\circ}{\rightarrow} \tau_2$.

It's a supertype of $\tau_1 \rightarrow \tau_2$

"function that may have side effects"

Extensibility

Let's add a new type of function $\tau_1 \overset{\circ}{\to} \tau_2$.
It's a supertype of $\tau_1 \to \tau_2$

"function that may have side effects"

Now we have a counterexample:

$$\alpha = (\top \overset{\circ}{\to} \bot) \overset{\circ}{\to} \bot$$

Extensible type systems

Two techniques give us an extensible system:

- Add explicit type variables as indeterminates

  gets rid of case analysis

Extensible type systems

Two techniques give us an extensible system:

- Add explicit type variables as indeterminates

  gets rid of case analysis
- Require a distributive lattice

  gets rid of vacuous reasoning

Combining types

How to combine different types into a single system?

$$\tau \ ::= \ \mathsf{bool} \mid \tau_1 \to \tau_2 \mid \{\ell_1 : \tau_1, \ \ldots, \ \ell_n : \tau_n\}$$

Combining types

How to combine different types into a single system?

$$\tau \ ::= \ \mathsf{bool} \mid \tau_1 \to \tau_2 \mid \{\ell_1 : \tau_1, \ \ldots, \ \ell_n : \tau_n\}$$

We should read '|' as **coproduct**

Concrete syntax

Build an actual syntax for types, by writing down all the operations on types:

$$\begin{aligned} \tau \;::= \;& \mathsf{bool} \mid \tau_1 \rightarrow \tau_2 \mid \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \mid \\ & \alpha \mid \top \mid \bot \mid \tau \sqcup \tau \mid \tau \sqcap \tau \end{aligned}$$

Concrete syntax

Build an actual syntax for types, by writing down all the operations on types:

$$\begin{aligned} \tau \ ::= {} & \mathsf{bool} \mid \tau_1 \to \tau_2 \mid \{\ell_1 : \tau_1, \ \ldots, \ \ell_n : \tau_n\} \mid \\ & \alpha \mid \top \mid \bot \mid \tau \sqcup \tau \mid \tau \sqcap \tau \end{aligned}$$

then quotient by the equations of distributive lattices, and the subtyping order.

Resulting types

We end up with all the standard types

Resulting types

We end up with all the standard types
... with the same subtyping order

Resulting types

We end up with all the standard types
... with the same subtyping order
... but we identify fewer of the weird types

$$\{\texttt{foo} : \mathsf{bool}\} \sqcap (\top \to \top) \not\leq \mathsf{bool}$$

$$\Gamma \vdash e : \tau$$

Principality in ML

Intuitively,
For any e typeable under Γ, there's a best type τ

Principality in ML

Intuitively,
For any e typeable under Γ, there's a best type τ

but it's a bit more complicated than that:
For any e typeable under Γ, there's a τ **and a substitution** σ *such that every possible typing of e under Γ is a substitution instance of $\sigma\Gamma, \tau$.*

Reformulating the typing rules

The complexity arises because Γ is part question, part answer.

Reformulating the typing rules

The complexity arises because Γ is part question, part answer.
Instead, split Γ:

- Δ maps λ-bound x to a type τ
- Π maps let-bound $\hat{\mathbf{x}}$ to a *typing schemes* $[\Delta]\tau$

$$\Pi \Vdash e : [\Delta]\tau$$

$$\overbrace{\Pi \Vdash e}^{\text{question}} : \overbrace{[\Delta]\tau}^{\text{answer}}$$

Subsumption

Define $\leq^{\forall}$ as the least relation closed under:

- *Instatiation*, replacing type variables with types
- *Subtyping*, replacing types with supertypes

Principality in MLsub

A *principal typing scheme* for e under Π is a $[\Delta]\tau$ that subsumes any other.

The choose function

`choose` takes two values and returns one of them:

$$\texttt{choose} : \forall\alpha.\alpha^{1} \to \alpha^{2} \to \alpha^{3}$$

The choose function

`choose` takes two values and returns one of them:

$$\texttt{choose} : \forall\alpha.\alpha^1 \to \alpha^2 \to \alpha^3$$

In ML, $\alpha^1 = \alpha^2 = \alpha^3$.
With subtyping, $\alpha^1 \leq \alpha^3$, $\alpha^2 \leq \alpha^3$,
but α^1 and α^2 may be incomparable.

The choose function

`choose` takes two values and returns one of them:

$$\texttt{choose} \ : \ \forall\alpha.\alpha^{1} \to \alpha^{2} \to \alpha^{3}$$

In ML, $\alpha^{1} = \alpha^{2} = \alpha^{3}$.
With subtyping, $\alpha^{1} \leq \alpha^{3}$, $\alpha^{2} \leq \alpha^{3}$,
but α^{1} and α^{2} may be incomparable.

$$\texttt{choose} \ : \ \forall\alpha\beta.\alpha \to \beta \to \alpha \sqcup \beta$$

The choose function

`choose` takes two values and returns one of them:

$$\texttt{choose} : \forall \alpha . \alpha^{1} \rightarrow \alpha^{2} \rightarrow \alpha^{3}$$

In ML, $\alpha^{1} = \alpha^{2} = \alpha^{3}$.
With subtyping, $\alpha^{1} \leq \alpha^{3}$, $\alpha^{2} \leq \alpha^{3}$,
but α^{1} and α^{2} may be incomparable.

$$\texttt{choose} : \forall \alpha \beta . \alpha \rightarrow \beta \rightarrow \alpha \sqcup \beta$$

These are *equivalent* ($\equiv^{\forall}$): subsume each other

Input and output types

$\tau \sqcup \tau'$: produces a value which is a τ or a τ'
$\tau \sqcap \tau'$: requires a value which is a τ and a τ'

$\sqcup$ is for outputs, and $\sqcap$ is for inputs.

Input and output types

$\tau \sqcup \tau'$: produces a value which is a τ or a τ'
$\tau \sqcap \tau'$: requires a value which is a τ and a τ'

$\sqcup$ is for outputs, and $\sqcap$ is for inputs.

Divide types into

- *output types* τ^+
- *input types* τ^-

Polar types

$$\begin{aligned}
\tau^{+} \;&::=\; \mathsf{bool} \mid \tau_1^{-} \rightarrow \tau_2^{+} \mid \{\ell_1 : \tau_1^{+}, \ldots, \ell_n : \tau_n^{+}\} \mid \\
&\;\; \alpha \mid \tau_1^{+} \sqcup \tau_2^{+} \mid \bot \mid \mu\alpha.\tau^{+} \\
\tau^{-} \;&::=\; \mathsf{bool} \mid \tau_1^{+} \rightarrow \tau_2^{-} \mid \{\ell_1 : \tau_1^{-}, \ldots, \ell_n : \tau_n^{-}\} \mid \\
&\;\; \alpha \mid \tau_1^{-} \sqcap \tau_2^{-} \mid \top \mid \mu\alpha.\tau^{-}
\end{aligned}$$

Cases of unification

In HM inference, unification happens in three situations:

- Unifying two input types
- Unifying two output types
- Using the output of one expression as input to another

Cases of unification

In HM inference, unification happens in three situations:

- Unifying two input types

  Introduce $\sqcup$
- Unifying two output types

  Introduce $\sqcap$
- Using the output of one expression as input to another

  $\tau^+ \leq \tau^-$ *constraint*

Eliminating variables, ML-style

Suppose we have an identity function

$$\alpha \to \alpha$$

Eliminating variables, ML-style

Suppose we have an identity function, which uses its argument as a τ

$$\alpha \to \alpha \mid \alpha = \tau$$

Eliminating variables, ML-style

Suppose we have an identity function, which uses its argument as a τ

$$\begin{aligned} & \alpha \to \alpha \mid \alpha = \tau \\ \equiv^{\forall}\ & \tau \to \tau \end{aligned}$$

Eliminating variables, ML-style

Suppose we have an identity function, which uses its argument as a τ

$$
\begin{aligned}
& \alpha \to \alpha \mid \alpha = \tau \\
\equiv^{\forall}\ & \tau \to \tau
\end{aligned}
$$

The substitution $[\tau/\alpha]$ **solves** the constraint $\alpha = \tau$

“solves?”

What does it mean to **solve** a constraint?

1. $[\tau/\alpha]$ trivialises the constraint $\alpha = \tau$
   (it is a *unifier*),
   and all other unifiers are an instance of it
   (it is a *most general unifier*)

“solves?”

What does it mean to **solve** a constraint?

1. $[\tau/\alpha]$ trivialises the constraint $\alpha = \tau$
   (it is a *unifier*),
   and all other unifiers are an instance of it
   (it is a *most general unifier*)

2. For any type τ', the following sets agree:
   the instances of τ', subject to $\alpha = \tau$
   the instances of $[\tau/\alpha]\tau'$

Definition 2, now with subtyping

Suppose we have an identity function, which uses its argument as a τ^-.

$$\alpha \to \alpha \mid \alpha \leq \tau^-$$

Definition 2, now with subtyping

Suppose we have an identity function, which uses its argument as a τ^-.

$$\alpha \to \alpha \mid \alpha \leq \tau^-$$
$$\equiv^\forall (\alpha \sqcap \tau^-) \to (\alpha \sqcap \tau^-)$$

Definition 2, now with subtyping

Suppose we have an identity function, which uses its argument as a τ^-.

$$
\begin{aligned}
& \alpha \to \alpha \mid \alpha \leq \tau^- \\
\equiv^\forall\ & (\alpha \sqcap \tau^-) \to (\alpha \sqcap \tau^-) \\
\equiv^\forall\ & (\alpha \sqcap \tau^-) \to \alpha
\end{aligned}
$$

Definition 2, now with subtyping

Suppose we have an identity function, which uses its argument as a τ^-.

$$\begin{aligned} &\alpha \to \alpha \mid \alpha \leq \tau^- \\ \equiv^\forall\ &(\alpha \sqcap \tau^-) \to (\alpha \sqcap \tau^-) \\ \equiv^\forall\ &(\alpha \sqcap \tau^-) \to \alpha \end{aligned}$$

The *bisubstitution* $[\alpha \sqcap \tau^- / \alpha^-]$ solves $\alpha \leq \tau^-$

Decomposing constraints

We only need to decompose constraints of the form $\tau^+ \leq \tau^-$.

$$
\begin{array}{rcl}
\tau_1 \sqcup \tau_2 \leq \tau_3 & \equiv & \tau_1 \leq \tau_3,\ \tau_2 \leq \tau_3 \\
\tau_1 \leq \tau_2 \sqcap \tau_3 & \equiv & \tau_1 \leq \tau_2,\ \tau_1 \leq \tau_3
\end{array}
$$

Thanks to the input/output type distinction, the hard cases of $\tau_1 \sqcap \tau_2 \leq \tau_3$ and $\tau_1 \leq \tau_2 \sqcup \tau_3$ can never come up.

Combining solutions

We solve a system of multiple constraints C_1, C_2 by:

- Solving C_1, giving a bisubstitution ξ
- Applying that to C_2
- Solving ξC_2, giving a bisubstitution ζ

Then $\xi \circ \zeta$ solves the system C_1, C_2.

Putting it all together

biunify(C) takes a set of constraints C, and produces a bisubstitution solving them.

$$
\begin{aligned}
\text{biunify}(\emptyset) &= [] \\
\text{biunify}(\alpha \leq \alpha, C) &= \text{biunify}(C) \\
\text{biunify}(\alpha \leq \tau, C) &= \text{biunify}(\theta_{\alpha \leq \tau} H;\ \theta_{\alpha \leq \tau}\, C) \circ \theta_{\alpha \leq \tau} \\
\text{biunify}(\tau \leq \alpha, C) &= \text{biunify}(\theta_{\tau \leq \alpha} H;\ \theta_{\tau \leq \alpha}\, C) \circ \theta_{\tau \leq \alpha} \\
\text{biunify}(c, C) &= \text{biunify}(\text{decompose}(c), C)
\end{aligned}
$$

Putting it all together

biunify(C) takes a set of constraints C, and produces a bisubstitution solving them.

$$\begin{aligned}
\mathrm{biunify}(\emptyset) &= [] \\
\mathrm{biunify}(\alpha \leq \alpha, C) &= \mathrm{biunify}(C) \\
\mathrm{biunify}(\alpha \leq \tau, C) &= \mathrm{biunify}(\theta_{\alpha\leq\tau} H;\ \theta_{\alpha\leq\tau}\, C) \circ \theta_{\alpha\leq\tau} \\
\mathrm{biunify}(\tau \leq \alpha, C) &= \mathrm{biunify}(\theta_{\tau\leq\alpha} H;\ \theta_{\tau\leq\alpha}\, C) \circ \theta_{\tau\leq\alpha} \\
\mathrm{biunify}(c, C) &= \mathrm{biunify}(\mathrm{decompose}(c), C)
\end{aligned}$$

Replace the $\leq$ with $=$ and we have Martelli and Montanari's unification algorithm.

Summary

MLsub infers types by walking the syntax of the program, but must deal with subtyping constraints rather than just equalities. Thanks to:

- algebraically well-behaved types
- polar types, restricting occurrences of $\sqcup$ and $\sqcap$
- a careful definition of “solves”

the biunify algorithm can always handle these constraints, producing a principal type.

Questions?

http://www.cl.cam.ac.uk/~sd601/mlsub

stephen.dolan@cl.cam.ac.uk

Mutable references

References are generally considered "invariant".

Instead, consider `ref` a two-argument constructor

$$(\alpha, \beta)\ \texttt{ref}$$

with operations:

$$\begin{aligned} \texttt{make} &: (\alpha, \alpha)\ \texttt{ref} \\ \texttt{get} &: (\bot, \beta)\ \texttt{ref} \to \beta \\ \texttt{set} &: (\alpha, \top)\ \texttt{ref} \to \alpha \to \texttt{unit} \end{aligned}$$