Polymorphism, subtyping and type inference in MLsub

Stephen Dolan and Alan Mycroft

September 3, 2015

Computer Laboratory
University of Cambridge

Avoiding the hard problems

Type inference with subtyping is traditionally considered difficult.

We have two tricks for getting around the difficulties:

- Define types properly
- Only use half of them

Avoiding the hard problems

Type inference with subtyping is traditionally considered difficult.

We have two tricks for getting around the difficulties:

- Define types properly
- Only use half of them

Standard definition of types and subtyping

Types are either $\bot$, $\top$ or functions:

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

Standard definition of types and subtyping

Types are either $\bot$, $\top$ or functions:

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

$\bot$ is the least, $\top$ is the greatest, and functions have contravariant arguments and covariant results:

$$\frac{}{\bot \leq \tau} \qquad\qquad \frac{}{\tau \leq \top}$$

$$\frac{\tau_1' \;\leq\; \tau_1 \qquad \tau_2 \;\leq\; \tau_2'}{\tau_1 \to \tau_2 \;\leq\; \tau_1' \to \tau_2'}$$

Lattice structure

Any two types have a greatest common subtype ($\sqcap$) and a least common supertype ($\sqcup$).

For instance:

$$(\tau_1 \to \tau_2) \sqcup (\tau_3 \to \tau_4) = (\tau_1 \sqcap \tau_3) \to (\tau_2 \sqcup \tau_4)$$

Quite a well-behaved structure!

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

If a program typechecks,
it should still typecheck when more types are added

This is a useful property:

- New versions of the language
- User-defined types
- Seperate compilation

Do we have extensibility?

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

Let's add a new type of functions $\tau_1 \rightsquigarrow \tau_2$, so that $\to$-types are subtypes of $\rightsquigarrow$-types.

Now, $\alpha = (\top \rightsquigarrow \bot) \rightsquigarrow \bot$ is a counterexample!

The previous reasoning relied on the non-existence of $\rightsquigarrow$.

Do we have extensibility?

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

Let's add a new type of functions $\tau_1 \rightsquigarrow \tau_2$, so that $\to$-types are subtypes of $\rightsquigarrow$-types.

Now, $\alpha = (\top \rightsquigarrow \bot) \rightsquigarrow \bot$ is a counterexample!

The previous reasoning relied on the non-existence of $\rightsquigarrow$.

We need an open-world definition:

$$\tau = \alpha \mid \tau \sqcup \tau \mid \tau \sqcap \tau \mid \bot \mid \top \mid \tau \to \tau$$

Avoiding the hard problems

Type inference with subtyping is traditionally considered difficult.

We have two tricks for getting around the difficulties:

- Define types properly
- Only use half of them

Avoiding the hard problems

Type inference with subtyping is traditionally considered difficult.

We have two tricks for getting around the difficulties:

- Define types properly
- Only use half of them

Input and output types

$\tau \sqcup \tau'$: this code produces a value which is a τ or a τ'

$\tau \sqcap \tau'$: this code requires a value which is a τ and a τ'

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

Input and output types

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

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

Divide types into *output types* τ^+ and *input types* τ^-:

$$\begin{aligned} \tau^+ &::= \alpha \mid \tau^+ \sqcup \tau^+ \mid \bot \mid \tau^- \to \tau^+ \\ \tau^- &::= \alpha \mid \tau^- \sqcap \tau^- \mid \top \mid \tau^+ \to \tau^- \end{aligned}$$

Polymorphism

Here's a function that takes two values and returns one of them:

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

Polymorphism

Here's a function that takes two values and returns one of them:

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

In ML, you must have $\alpha^1 = \alpha^2 = \alpha^3$.

With subtyping, you must have $\alpha^1 \leq \alpha^3$, $\alpha^2 \leq \alpha^3$.
α^1 and α^2 may be incomparable.

Polymorphism

Here's a function that takes two values and returns one of them:

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

In ML, you must have $\alpha^1 = \alpha^2 = \alpha^3$.

With subtyping, you must have $\alpha^1 \leq \alpha^3, \ \ \alpha^2 \leq \alpha^3$.
α^1 and α^2 may be incomparable.

$$\texttt{choose} \ : \ \tau_1^- \to \tau_2^- \to \tau_3^+$$

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

We handle these as:

- Introducing $\sqcap$
- Introducing $\sqcup$
- Decomposing a $\tau^+ \leq \tau^-$ constraint

Decomposing constraints

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

$$\tau_1 \sqcup \tau_2 \leq \tau_3 \quad \equiv \quad \tau_1 \leq \tau_3,\ \tau_2 \leq \tau_3$$
$$\tau_1 \leq \tau_2 \sqcap \tau_3 \quad \equiv \quad \tau_1 \leq \tau_2,\ \tau_1 \leq \tau_3$$

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.

Typechecking OCaml's List module

Most of the OCaml List module typechecks, with the same types as OCaml (modulo syntax annoyances, and unimplemented features).

Questions?

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

stephen.dolan@cl.cam.ac.uk

Defining types extensibly

Include variables and lattice operations in the definition of types:

$$\tau = \alpha \mid \tau \sqcup \tau \mid \tau \sqcap \tau \mid \bot \mid \top \mid \tau \to \tau$$

Define some rules about when types are equal:

$$(\tau_1 \to \tau_2) \sqcup (\tau_3 \to \tau_4) = (\tau_1 \sqcap \tau_3) \to (\tau_2 \sqcup \tau_4)$$

$$\dots$$

This is the standard construction of a *free algebra*, and variables aren't presumed to be divided into cases.

$$(\bot \to \top) \to \bot \not\leq (\alpha \to \bot) \sqcup \alpha$$

Mutable references

References are generally considered "invariant".

Instead, consider `ref` a two-argument constructor

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

with operations:

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