Programming Languages 2 (PL II)

Algebraic Subtyping

Summer Term 2023

David Binder
University of Tübingen

12.07.2023

Algebraic Subtyping

Algebraic Subtyping

Today's lecture is about a (somewhat) recent development in programming language theory: Algebraic Subtyping. Algebraic Subtyping combines three ideas you have seen in previous lectures:

- Parametric polymorphism: $\forall\overline{\alpha}.\tau$
- Subtyping polymorphism: $\sigma <: \tau$
- Complete type inference based on solving of constraints: $\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$

Algebraic Subtyping is novel ($\sim$2016), and accessible resources are still hard to find.

A Hard Problem

Everyone agrees that the combination of parametric polymorphism, subtyping and complete and principal type inference is a *hard problem*. Most languages therefore reject either subtyping or powerful type inference:

- Haskell/Idris/Agda/. . . : Don't support subtyping.
- Scala/Java/C#/. . . : Don't support complete type inference.

A lot of researchers thought that the problem is essentially insoluble!

The Combination Is Possible

Groundwork by F. Pottier [1] and breakthrough by S. Dolan [2] in their PhD's: The Hindley-Milner type inference algorithm can be modified to enable:

- Complete type inference: No type annotations are necessary.
- Principal type inference: The best, most general type is inferred.
- Type simplification: Simple and readable types are inferred.

Challenge: The inference algorithm has to work with inequality constraints $\sigma <: \tau$.

- [1] François Pottier(1998): Type Inference in the Presence of Subtyping: from Theory to Practice
- [2] Stephen Dolan(2016): Algebraic Subtyping

Today's Focus: Solving Type Inequations

The central problem is how to solve sets of inequations between types:

$$\{\ \sigma_1 <: \tau_1, \ldots, \sigma_n <: \tau_n\ \}$$

We have to answer three questions:

- What does an answer to such a problem look like?
- Why is this problem so difficult?
- How does an algorithm look like which solves such constraints?

Today we will mainly focus on these three questions.

Review Of Previous Lectures

Parametric Polymorphism

We can introduce polymorphism by replacing parts of types by type variables, and by then quantifying over them:

$$\lambda x.x : \forall \alpha.\alpha \to \alpha$$
$$\lambda x.\lambda y.x : \forall \alpha\beta.\alpha \to \beta \to \alpha$$
$$\lambda x.5 : \forall \alpha.\alpha \to \mathbb{N}$$

Using type variables α and universal quantification $\forall$ we can express polymorphism.

Instantiation of Type Variables

If we instantiate type variables in a type scheme by other types, we obtain an *instance*. I will write $<^{\forall}$ for this relation between two type schemes.

$$\mathbb{N} \to \mathbb{N} <^{\forall} \forall \alpha . \alpha \to \alpha$$

$$\mathbb{B} \to \mathbb{N} \to \mathbb{B} <^{\forall} \forall \alpha \beta . \alpha \to \beta \to \alpha$$

$$\forall \beta . \mathbb{B} \to \beta \to \mathbb{B} <^{\forall} \forall \alpha \beta . \alpha \to \beta \to \alpha$$

The relation $<^{\forall}$ *orders* type schemes by how general or specific they are.

Hindley-Damas-Milner Type Inference

The central part of Hindley-Damas-Milner type inference is *constraint solving*:

$$\{\sigma_1 = \tau_1, \dots, \sigma_n = \tau_n\}$$

A *solution* to such a set of equations has the form of a substitution, i.e. a mapping from variables to types.

- If we apply the solution to the original set of constraints, then all equations become trivial.
- We want the most general solution, i.e. all other solutions are more specific.

The unification algorithm computes the most general solution, if it exists.

Example of Unification

Consider the following set of equations:

$$\{\mathbb{N} = \alpha, \beta = \alpha \to \mathbb{B}, \delta = \gamma \to \textsf{List}[\beta]\}$$

This problem has the following most general solution:

$$\begin{aligned}
&\alpha \mapsto \mathbb{N}\\
&\beta \mapsto \mathbb{N} \to \mathbb{B}\\
&\gamma \mapsto \gamma\\
&\delta \mapsto \gamma \to \textsf{List}[\mathbb{N} \to \mathbb{B}]
\end{aligned}$$

If we map γ to $\mathbb{N}$, we still have a solution, but it is no longer the most general one.

Subtyping

Some types are subtypes or supertypes of other types:

$$\mathbb{N} <: \mathbb{Z} \qquad \text{(Subtyping between primitive types)}$$

$$\{x : \mathsf{String}, y : \mathbb{N}\} <: \{x : \mathsf{String}\} \qquad \text{(Record subtyping)}$$

$$\mathsf{List}[\mathbb{N}] <: \mathsf{List}[\mathbb{Z}] \qquad \text{(Covariant subtyping)}$$

$$\mathbb{Z} \to \mathbb{B} <: \mathbb{N} \to \mathbb{B} \qquad \text{(Contravariant subtyping)}$$

And we can make use of the subtyping relation with the subsumption rule:

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T}}{\Gamma \vdash \mathtt{t} : \mathtt{T}} \qquad (\text{T-}\textsc{Sub})$$

Partial Orders and Lattices

Partial Orders and Lattices

Lattice theory studies sets A equipped with a partial order $\sqsubseteq$. Algebraic subtyping deals with the following three orders:

- The subtyping relation between types: $\sigma <: \tau$.
- The instantiation relation $<^{\forall}$ between type schemes.
- The combination of the above two relations: $<:^{\forall}$.

We need some basics of lattice theory to understand the fundamental ideas of algebraic subtyping. In order to prove the correctness of the algorithms, we would need more substantial machinery.

The central role of lattice theory in *algebraic* subtyping motivates the name.

Partially Ordered Sets

A relation $\sqsubseteq$ is a *partial order*, if it satisfies the following three axioms.

$$x \sqsubseteq x \tag{Reflexivity}$$

$$x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z \tag{Transitivity}$$

$$x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y \tag{Antisymmetry}$$

A set A equipped with a partial order $\sqsubseteq$, which we abbreviate as $\langle A, \sqsubseteq \rangle$, is called a *partially ordered set* or *poset*.

Lattices

A partially ordered that is equipped with two binary operations $\sqcup$ (join) and $\sqcap$ (meet) is called a *lattice*, if the following axioms are satisfied:

$$
\begin{array}{rcccclr}
 & & x \sqsubseteq x \sqcup y & \wedge & y \sqsubseteq x \sqcup y & & (\sqcup \text{ is an upper bound}) \\
x \sqsubseteq c & \wedge & y \sqsubseteq c & \Rightarrow & x \sqcup y \sqsubseteq c & & (\sqcup \text{ is the least upper bound}) \\
 & & x \sqcap y \sqsubseteq x & \wedge & x \sqcap y \sqsubseteq y & & (\sqcap \text{ is a lower bound}) \\
c \sqsubseteq x & \wedge & c \sqsubseteq y & \Rightarrow & c \sqsubseteq x \sqcap y & & (\sqcap \text{ is the greatest lower bound})
\end{array}
$$

A poset equipped with binary meets and joins, which we write $\langle A, \sqsubseteq, \sqcap, \sqcup \rangle$ is called a *lattice*.

Why do we need joins?

Consider the following function:

$$\lambda x.\text{if } x \text{ then } 5 \text{ else True}$$

The type should be:

$$\mathbb{B} \to \alpha$$

From the two branches we know that:

$$\mathbb{N} <: \alpha \quad \mathbb{B} <: \alpha$$

And we can infer the type:

$$\mathbb{B} \to \mathbb{N} \sqcup \mathbb{B}$$

Joins are there for combining multiple possible outputs!

Why we need meets

Consider the following function:

$$\lambda x.\ \ldots(\mathsf{not}\ x)\ \ldots\ (x+1)\ \ldots$$

If α is the type of the variable x, then we know that:

$$\alpha <: \mathbb{B} \quad \alpha <: \mathbb{N}$$

From this we can infer the type:

$$\mathbb{B} \sqcap \mathbb{N} \to \ldots$$

Meets are for combining multiple requirements on inputs!

Bounded Lattices

Some lattices contain least and greatest elements, which we write $\top$ and $\bot$ and pronounce "top" and "bottom".

$$x \sqsubseteq \top \qquad (\top \text{ is the greatest element})$$
$$\bot \sqsubseteq x \qquad (\bot \text{ is the least element})$$

A lattice with top and bottom elements, which we write $\langle A, \sqsubseteq, \sqcap, \sqcup, \top, \bot \rangle$, is called a *bounded lattice*.

Why we need top

Some functions don't use their input:

$$\lambda x.5$$

The inferred type should not require anything from the argument:

$$\top \to \mathbb{N}$$

Top is for inputs which we don't use.

Why we need bottom

Some functions never return a value:

$$\lambda s.\, \texttt{crash\_with\_msg}(s)$$

The type we infer for this function is:

$$\mathsf{String} \to \bot$$

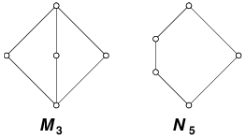
Bottom is for functions which don't return.

Distributive Lattices

A lattice which satisfies the following two additional distributive laws is called a *distributive lattice.*

$$x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$$
$$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$$

Not all lattices are distributive!



Difficulties

Difficulties

Before we look at the solution, we should first look at why researches considered the combination of subtyping, polymorphism and type inference a difficult problem.

$$\lambda x.2 : \begin{cases} \forall\alpha.\alpha \to \mathbb{N} \\ \top \to \mathbb{N} \end{cases}$$

$$\lambda x.\lambda y.x : \begin{cases} \forall\alpha\beta.\alpha \to \beta \to \alpha \\ \forall\alpha.\alpha \to \top \to \alpha \end{cases}$$

In these cases it is not obvious that either type is the more general one.

Solving Highschool Arithmetical Equations

Consider the following set of *equations*:

$$\{y = 3 + x,\ x = 2,\ y = z\ \}$$

A solution to a system of equations consists in a assignment of *values* to variables:

$$x := 2$$
$$y := 5$$
$$z := 5$$

When we solve equations, we can substitute values for variables!

Solving Highschool Arithmetical Inequations

Consider the following set of *inequations*:

$$\{x \leq 2,\ x \leq y,\ y \leq 1,\ -2 \leq x,\ 0 \leq y\}$$

A solution to a system of inequations consists in an assignment of *bounds* to variables:

$$-2 \leq x \leq 1$$
$$0 \leq y \leq 1$$

When we solve inequations, we have to compute bounds of variables!

A (Bi-)Unification Algorithm for Subtyping Constraints

A Biunification Algorithm for Subtyping Constraints

We will now see the algorithm at the core of algebraic subtyping: Biunification.

- The input is a set of subtyping constraints $\{ \sigma_1 <: \tau_1, \ldots, \sigma_n <: \tau_n \}$
- The algorithm fails if the input contains unsatisfiable bounds.
- If the algorithm succeeds, then it assigns a list of lower and upper bounds to every unification variable contained in the input.
- Most cases are very similar to the unification algorithm.

Simple cases

A lot of cases are very similar between unification and biunification.

Unification		*Biunification*	
$\sigma = \sigma$	(solve directly)	$\sigma <: \sigma$	(solve directly)
$\mathbb{N} = \mathbb{B}$	(fail directly)	$\mathbb{N} <: \mathbb{B}$	(fail directly)
$\text{List}[\sigma] = \text{List}[\tau]$	(Generate $\sigma = \tau$)	$\text{List}[\sigma] <: \text{List}[\tau]$	(Generate $\sigma <: \tau$)
$\sigma \to \tau = \xi \to \rho$	(Gen. $\sigma = \xi, \tau = \rho$)	$\sigma \to \tau <: \xi \to \rho$	(Gen. $\xi <: \sigma, \tau <: \rho$)

Watch out for variance when decomposing constraints!

Simple Lattice Cases

Some cases involving lattice operations are simple.

Unification	*Biunification*	
$\emptyset$	$\tau <: \top$	(solve directly)
$\emptyset$	$\bot <: \tau$	(solve directly)
$\emptyset$	$\sigma_1 \sqcup \sigma_2 <: \tau$	(Generate $\sigma_1 <: \tau, \sigma_2 <: \tau$)
$\emptyset$	$\tau <: \sigma_1 \sqcap \sigma_2$	(Generate $\tau <: \sigma_1, \tau <: \sigma_2$)

Recall the equations for joins and meets:

$$\begin{array}{ccccc} x \sqsubseteq c & \wedge & y \sqsubseteq c & \Rightarrow & x \sqcup y \sqsubseteq c \\ c \sqsubseteq x & \wedge & c \sqsubseteq y & \Rightarrow & c \sqsubseteq x \sqcap y \end{array}$$

Hard Lattice Cases

Some cases involving lattice cases are very hard.

Unification	*Biunification*	
$\emptyset$	$\top <: \tau$	???
$\emptyset$	$\tau <: \bot$	???
$\emptyset$	$\sigma_1 \sqcap \sigma_2 <: \tau$	???
$\emptyset$	$\tau <: \sigma_1 \sqcup \sigma_2$	???

Luckily, we don't have to solve these cases, due to a deep theorem by Dolan:

Every principal type has an equivalent polar type.

Unification Variables

What about the base case?

Unification	*Biunification*
$\alpha = \tau$ (Substitute τ for α everywhere)	$\alpha <: \tau$ (Add τ to upper bounds of α)

There are some complications:

- When we add an upper bound to α, we have to ensure that it is consistent with all existing lower bounds.
- We have to deal with recursive constraints. E.g. $\alpha = \text{List}[\alpha]$ and $\alpha <: \text{List}[\alpha]$.
- We can either implement an occurs-check, or allow recursive types $\mu\alpha.\text{List}[\alpha]$ as solutions. Hindley-Milner usually implements the occurs-check, MLsub implements recursive types.

Keeping track of lower and upper bounds

The constraint solver has to keep track of all collected lower and upper bounds for every unification variable.We do this in a data structure called the *variable state*:

$$\{\sigma_1, \dots, \sigma_n\} <: \alpha <: \{\tau_1, \dots, \tau_m\}$$

When we solve the constraint $\alpha <: \xi$, we have to add ξ to the upper bounds of α:

$$\{\sigma_1, \dots, \sigma_n\} <: \alpha <: \{\tau_1, \dots, \tau_m, \xi\}$$

In order to make sure that this upper bound is consistent with the lower bounds, we have to generate new constraints:

$$\{\ \sigma_1 <: \xi, \dots, \sigma_n <: \xi\ \}$$

We can use a cache of solved constraints to guarantee performance and termination.

Type Coalescing

Type Coalescing

After constraint solving is finished, we end up with a final state which contains all the computed lower and upper bounds for each unification variable:

$$\{\sigma_1, \ldots, \sigma_n\} <: \alpha <: \{\tau_1, \ldots, \tau_m\}$$

From these datastructures we compute a *bisubstitution* for the unification variables.

$$\{\ \alpha^+ \mapsto \rho^+, \alpha^- \mapsto \rho^-, \ldots\ \}$$

Intuitively, ρ^+ corresponds to the upper bound of α, and ρ^- to the lower bound.

$$\rho^+ \approx \alpha \sqcup \tau_1 \sqcup \ldots \sqcup \tau_m$$
$$\rho^- \approx \alpha \sqcap \sigma_1 \sqcap \ldots \sqcap \sigma_n$$

Applying Bisubstitutions

Suppose we have inferred the following type for the program:

$$(\alpha \to \beta) \to \gamma$$

We first mark for all unification variables if they occur in positive or negative position:

$$(\alpha^+ \to \beta^-) \to \gamma^+$$

We can then apply the bisubstitution described in the previous slide.

What else is there?

What else is there?

Algebraic subtyping is a big topic. We didn't have time to talk about everything:

- How to generate subtyping constraints for a program.
- How to simplify types: $\forall\alpha.\alpha \sqcap \mathbb{N} \to \alpha \sqcup \mathbb{N}$ is the same type as $\mathbb{N} \to \mathbb{N}$.
- How to prove that the algorithm is correct.
- How to define polymorphic subsumption $\sigma <:^{\forall} \tau$ which combines subtyping $\sigma <: \tau$ with instantiation $\sigma <^{\forall} \tau$.

But you already know all the central ideas of the algebraic subtyping approach!

Literature

Literature

Here are some further references:

- Stephen Dolan: Algebraic Subtyping
- Stephen Dolan, Alan Mycroft: Polymorphism, subtyping, and type inference in MLsub
- Lionel Parreaux: The simple essence of algebraic subtyping
- Lionel Parreaux, Chun Yin Chau: MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types
- David Binder, Ingo Skupin, David Läwen, Klaus Ostermann: Structural Refinement Types