A subtyping algorithm for intersection and union types

Claude Stolze

IRIF Université de Paris

Journée CoGITARe November 7, 2019

Outline

- Polymorphism in the λ-calculus
- Subtyping as an effective semantic tool to increase expressivity on programming languages
- The power of intersection $\bigcap$ and union $\bigcup$ types assigned to pure lambda-calculus
- Description of the subtyping algorithm

Simply typed λ-calculus (Curry-style)

- Types: $\sigma ::= \varphi \mid \sigma_1 \to \sigma_2$ φ is an atomic type
- We note $\sigma \to \tau \to \rho$ for $\sigma \to (\tau \to \rho)$
- Typing rules:

$$\begin{array}{rcl} (x{:}\sigma) \in \Gamma & \Rightarrow & \Gamma \vdash x : \sigma \\ \Gamma \vdash M : \sigma \to \tau \text{ and } \Gamma \vdash N : \sigma & \Rightarrow & \Gamma \vdash M\ N : \tau \\ \Gamma, x{:}\sigma \vdash M : \tau & \Rightarrow & \Gamma \vdash \lambda x.M : \sigma \to \tau \end{array}$$

- Preferred notation for the typing rules:

$$\frac{(x{:}\sigma) \in \Gamma}{\Gamma \vdash x : \sigma}\ (Var) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ (\to E)$$

$$\frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (\to I)$$

Example

$$\dfrac{\dfrac{\dfrac{\overline{f{:}\sigma \to \tau, x{:}\sigma \vdash f : \sigma \to \tau} \quad \overline{f{:}\sigma \to \tau, x{:}\sigma \vdash x{:}\sigma}}{f{:}\sigma \to \tau, x{:}\sigma \vdash f\ x : \tau}}{f{:}\sigma \to \tau \vdash \lambda x.f\ x : \sigma \to \tau}}{\vdash \lambda f.\lambda x.f\ x : (\sigma \to \tau) \to \sigma \to \tau}$$

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f{:}\sigma \to \tau, x{:}\sigma \vdash f : \sigma \to \tau} \quad \dfrac{}{f{:}\sigma \to \tau, x{:}\sigma \vdash x{:}\sigma}}{f{:}\sigma \to \tau, x{:}\sigma \vdash f\ x : \tau}}{f{:}\sigma \to \tau \vdash \lambda x.f\ x : \sigma \to \tau}}{\vdash \lambda f.\lambda x.f\ x : (\sigma \to \tau) \to \sigma \to \tau}$$

Notice that, in this case, the type is not unique: there is an infinity of possible types:

- $\vdash \lambda f.\lambda x.f\ x : (\tau \to \sigma) \to \tau \to \sigma$
- $\vdash \lambda f.\lambda x.f\ x : (\sigma \to \sigma) \to \sigma \to \sigma$
- $\vdash \lambda f.\lambda x.f\ x : (\sigma \to \tau \to \rho) \to \sigma \to \tau \to \rho$
- ...

Girard's parametric polymorphism ($\lambda 2$)

- Idea: add the quantifier $\forall$ in types, on order to deal with polymorphism
- Rules: the same as in the simply-typed λ-calculus, and:

$$\frac{\Gamma \vdash M : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha.\sigma}\ (\forall I) \qquad \frac{\Gamma \vdash M : \forall \alpha.\sigma}{\Gamma \vdash M : \sigma[\alpha := \tau]}\ (\forall E)$$

- OCaml implements a subset of $\lambda 2$, thanks to the Damas-Milner type inference algorithm:

```
# (fun f -> fun x -> f x);;
- : ('a -> 'b) -> 'a -> 'b = <fun>
```

Barendregt-Coppo-Dezani's *ad hoc* polymorphism ($\lambda \cap$)

- No $\forall$ quantifier
- Instead, a $\cap$ operator indicating that a term can have several types:

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau} \ (\cap I) \qquad \frac{\Gamma \vdash M : \sigma_1 \cap \sigma_2}{\Gamma \vdash M : \sigma_i} \ (\cap E_i)$$

- Example : $\vdash \lambda f.\lambda x.f\ x : ((\sigma \to \tau) \to \sigma \to \tau) \cap ((\tau \to \sigma) \to \tau \to \sigma)$
- Example : $\vdash \lambda x.x\ x : ((\sigma \to \tau) \cap \sigma) \to \tau$

Barbanera-Dezani-de'Liguoro union types ($\lambda \cap \cup$)

- Another form of *ad hoc* polymorphism

$$\frac{\Gamma \vdash M : \sigma_i}{\Gamma \vdash M : \sigma_1 \cup \sigma_2}\ (\cup I_i) \qquad \frac{\begin{array}{c}\Gamma, x{:}\sigma_1 \vdash M : \sigma_3 \\ \Gamma, x{:}\sigma_2 \vdash M : \sigma_3 \quad \Gamma \vdash N : \sigma_1 \cup \sigma_2\end{array}}{\Gamma \vdash M[N/x] : \sigma_3}\ (\cup E)$$

- Example : $\vdash \lambda x.x : (\sigma \to \sigma) \cup \sigma$
- Example : $\vdash \lambda f.\lambda x.f\ x : ((\sigma \to \rho) \cap (\tau \to \rho)) \to (\sigma \cup \tau) \to \rho$

Subtyping in programming languages 1/3

- Subtyping, denoted by $\leqslant$ is a form of implicit polymorphism (aka implicit type conversion or type coercion)

Subtyping in programming languages 1/3

- Subtyping, denoted by $\leqslant$ is a form of implicit polymorphism (aka implicit type conversion or type coercion)

- Subtyping allows us to implicitly and safely promote some variables of some type into another type

```
int x = 3;                          x is an integer
float y = 4.0;                      y is a float
float z = x + y;                    x is is implicitly coerced into a float
                                    //  the result is 7.0
```

Subtyping in programming languages 1/3

- Subtyping, denoted by $\leqslant$ is a form of implicit polymorphism (aka implicit type conversion or type coercion)

- Subtyping allows us to implicitly and safely promote some variables of some type into another type

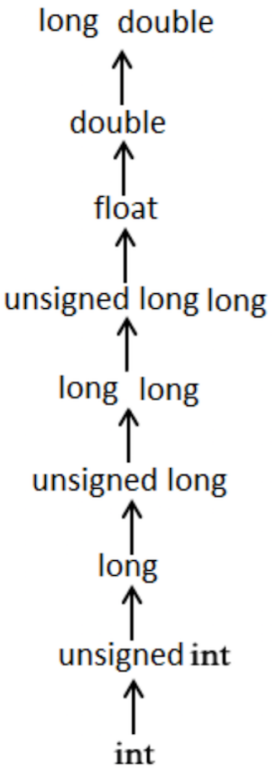
```
int x = 3;                          x is an integer
float y = 4.0;                      y is a float
float z = x + y;                    x is is implicitly coerced into a float
                                    //  the result is 7.0
```

- Subtyping is not an explicit type conversion (aka type casting)

```
float x = 3.3;                      x is an float
float y = 4.7;                      y is an float
int z =(int)x + (int)y;             x and y are casted into integers
                                    //  the result is 7
```

Subtyping in programming languages 2/3

Subtyping hierarchy in C



Subtyping rule

$$\frac{\Gamma \vdash M : \sigma \quad \sigma \leqslant \tau}{\Gamma \vdash M : \tau}\ (\leqslant)$$

Subtyping in OO programming languages 3/3

- Subtyping lurks also in object-oriented programming

  “An object of class T may be substituted with any object of a subclass S” *(Barbara Liskov substitution principle)*

- Inheritance as subtyping

Subtyping in OO programming languages 3/3

- Subtyping lurks also in object-oriented programming

  "An object of class T may be substituted with any object of a subclass S" *(Barbara Liskov substitution principle)*

- Inheritance as subtyping

- Subtyping hierarchy in Java

```
class Point { int x = 0; int y = 0; }
class ColPoint extends Point with { String col = "red"; }

   Point p = new Point();
   ColPoint q = new ColPoint();
```

Subtyping in OO programming languages 3/3

- Subtyping lurks also in object-oriented programming

  "An object of class T may be substituted with any object of a subclass S" *(Barbara Liskov substitution principle)*

- Inheritance as subtyping

- Subtyping hierarchy in Java

```
class Point { int x = 0; int y = 0; }
class ColPoint extends Point with { String col = "red"; }
```

```
Point p = new Point();
ColPoint q = new ColPoint();
```

~~`q = p;`~~ reject

`q = (ColPoint) p;` accept (explicit cast), but runtime error

`p = q;` accept

Ad hoc vs. *Parametric polymorphism*

- Intersection types $\bigcap$ [Barendregt-Coppo-Dezani 82] characterize the set of "strongly normalizable" λ-terms
- Ad hoc (C)

```
  int a, b;
float x, y;
printf("%d %f", a+b, x+y);
```

- The type of the operator + is

$$\texttt{+} : (\texttt{int} \times \texttt{int} \to \texttt{int}) \cap (\texttt{float} \times \texttt{float} \to \texttt{float})$$

- Parametric (caml)

```
> fun x -> x : 'a -> 'a
```

or $\forall\alpha.\alpha \to \alpha$

- Well known: Girard's parametric polymorphism (System F) is equivalent to *ad hoc* polymorphism

$$\forall\alpha.\sigma \approx \bigcap_{i=1\ldots\infty} \sigma_i$$

Union types as a dual of intersection types

- Union types $\bigcup$ [McQueen-Plotkin-Sethi 85] are considered as a dual of intersection types

- Union corresponds “roughly” to OCaml `match` construct

```
type 'a or = In1 of 'a | In2 of 'a ;;        'a is a type variable
let f x = match x with                  case analysis on the shape of x
| In1 y -> "case 1"                                          first case
| In2 y -> "case 2"                                         second case
;;
```

- The big difference between sum types and union types is that, for union types, both cases should have the same structure

Ex: Type assignment judgments with $\cap$ and $\cup$

- The Forsythe code [by Pierce 91]

$$\begin{array}{rcl} \text{Test} & \triangleq & \text{if } b \text{ then } 1 \text{ else } -1 : \mathit{Pos} \cup \mathit{Neg} \\ \text{Is\_0} & : & (\mathit{Neg} \to F) \cap (\mathit{Zero} \to T) \cap (\mathit{Pos} \to F) \\ (\text{Is\_0 Test}) & : & F \end{array}$$

Without union types the best information we can get for (Is_0 Test) is a Boolean type

Subtyping rules 1/4

A subtyping relation is a preorder, ie. a reflexive and transitive order. $\mathtt{U}$ is a universal type, corresponding to the $\top$ constant in the lattice of types (with $\cup$ as $\sqcup$ and $\cap$ as $\sqcap$)

$\sigma \leqslant \sigma$	Reflexivity
$\sigma \leqslant \tau$ and $\tau \leqslant \rho \Rightarrow \sigma \leqslant \rho$	Transitivity
$\sigma \leqslant \mathtt{U}$	Universal type
$\mathtt{U} \leqslant \mathtt{U} \to \mathtt{U}$	Universal type is also a function

Subtyping rules 2/4

Main rules for intersection:

$\sigma \leqslant \sigma \cap \sigma$

$\sigma \cap \tau \leqslant \sigma$

$\sigma \cap \tau \leqslant \tau$

$\sigma_1 \leqslant \sigma_2$ and $\tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \cap \tau_1 \leqslant \sigma_2 \cap \tau_2$ Intersection compositionality

Subtyping rules 2/4

Main rules for intersection:

$\sigma \leqslant \sigma \cap \sigma$

$\sigma \cap \tau \leqslant \sigma$

$\sigma \cap \tau \leqslant \tau$

$\sigma_1 \leqslant \sigma_2$ and $\tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \cap \tau_1 \leqslant \sigma_2 \cap \tau_2$ Intersection compositionality

Main rules for union:

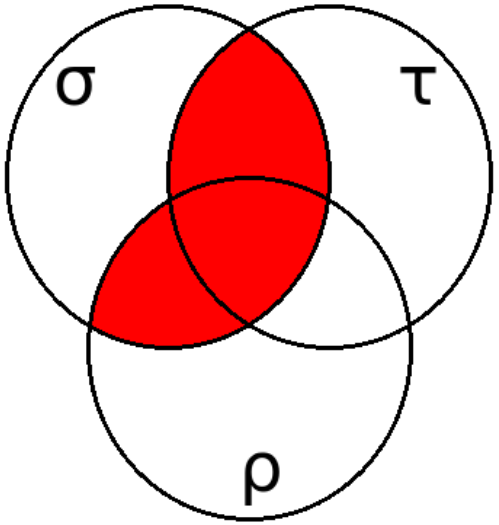
$\sigma \cup \sigma \leqslant \sigma$

$\sigma \leqslant \sigma \cup \tau$

$\tau \leqslant \sigma \cup \tau$

$\sigma_1 \leqslant \sigma_2$ and $\tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \cup \tau_1 \leqslant \sigma_2 \cup \tau_2$ Union compositionality

Subtyping rules 3/4



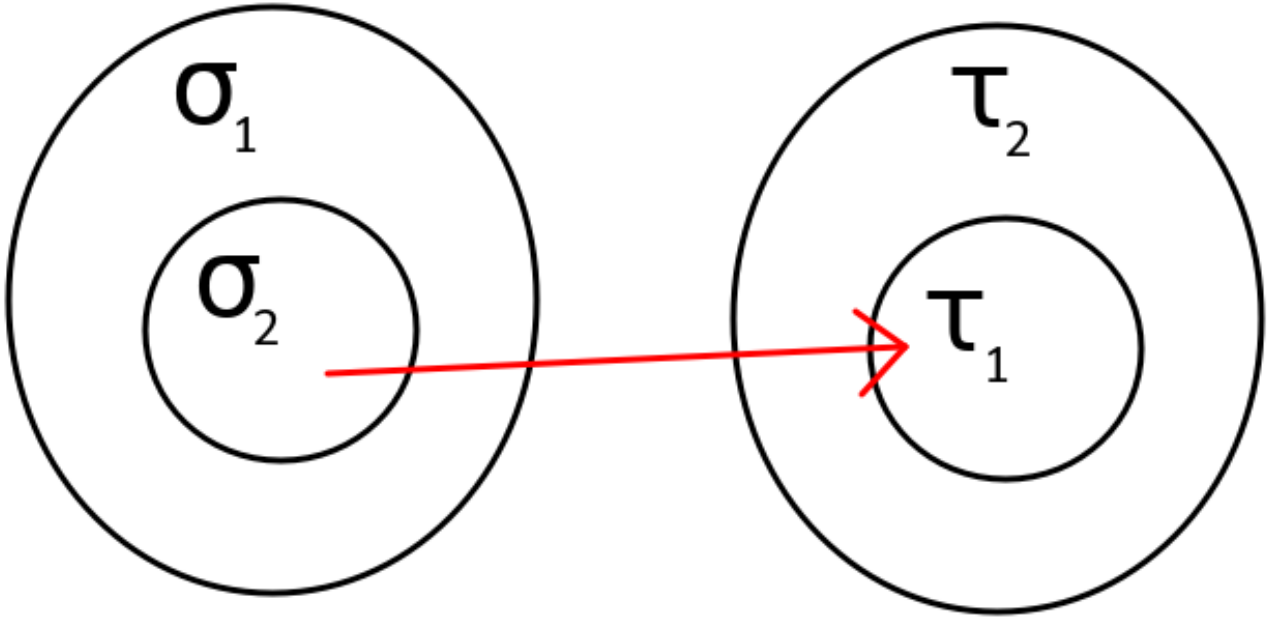
$\sigma \cap (\tau \cup \rho) \leqslant (\sigma \cap \tau) \cup (\sigma \cap \rho)$ Distributivity of intersection over union

$(\sigma \to \tau) \cap (\sigma \to \rho) \leqslant \sigma \to (\tau \cap \rho)$ Codomain factorization

$(\sigma \to \rho) \cap (\tau \to \rho) \leqslant (\sigma \cup \tau) \to \rho$ Domain factorization

Distributivity of union over intersection can be inferred, so there is no need for another distributivity axiom

Subtyping rules 4/4



Domain contravariance & codomain variance

$$\sigma_2 \leqslant \sigma_1 \quad \text{and} \quad \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \to \tau_1 \leqslant \sigma_2 \to \tau_2$$

The subtyping algorithm $\mathcal{A}$ in a nutshell

Fully detailed in [Stolze Liquori TTCS'17]

We note $\sigma \sim \tau$ if $\sigma \leqslant \tau$ and $\tau \leqslant \sigma$

Idea:

- First step: we rewrite the types into an equivalent form that is easier to process, using four subroutines $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4$
- Second step: we apply $\mathcal{A}$ on these normal forms
  - $\mathcal{A}$ proceeds by case analysis:
  - we decompose the unions and intersections
  - and we proceed by structural recursion

Rewriting subroutine $\mathcal{R}_1$ 1/3

We can show that:

$$\begin{aligned} \mathtt{U} \cap \sigma &\sim \sigma \\ \mathtt{U} \cup \sigma &\sim \mathtt{U} \\ \sigma \to \mathtt{U} &\sim \mathtt{U} \end{aligned}$$

This subroutine simplifies all the subterms containing $\mathtt{U}$

- $\mathtt{U} \cap \sigma$ and $\sigma \cap \mathtt{U}$ rewrite to σ
- $\mathtt{U} \cup \sigma$ and $\sigma \cup \mathtt{U}$ rewrite to $\mathtt{U}$
- $\sigma \to \mathtt{U}$ rewrites to $\mathtt{U}$

Rewriting subroutines $\mathcal{R}_2, \mathcal{R}_3$ 2/3

We can show that:

$$\sigma \cup (\tau \cap \rho) \sim (\sigma \cup \tau) \cap (\sigma \cup \rho)$$
$$\sigma \cap (\tau \cup \rho) \sim (\sigma \cap \tau) \cup (\sigma \cap \rho)$$

These subroutines rewrite types into conjunctive normal form (CNF) and disjunctive normal form (DNF).

- $\mathcal{R}_2$: $\sigma \cup (\tau \cap \rho)$ rewrites to $(\sigma \cup \tau) \cap (\sigma \cup \rho)$
- $\mathcal{R}_3$: $\sigma \cap (\tau \cup \rho)$ rewrites to $(\sigma \cap \tau) \cup (\sigma \cap \rho)$

Well known: DNF and CNF can grow exponentially in size

Rewriting subroutine $\mathcal{R}_4$ 3/3

We can show that:

$$\sigma \to (\tau \cap \rho) \quad \sim \quad (\sigma \to \tau) \cap (\sigma \to \rho)$$

$$(\sigma \cup \tau) \to \rho \quad \sim \quad (\sigma \to \rho) \cap (\tau \to \rho)$$

Mixing these equivalences with CNF and DNF, we get two mutually defined new normal forms CANF and DANF :

- $\mathcal{R}_2 \circ \mathcal{R}_4 \circ \mathcal{R}_1$: Conjunctive arrow normal form (CANF):
  - for any subterm $\sigma \to \tau$, rewrite σ in DANF and τ in CANF,
  - rewrite $\sigma \to (\tau \cap \rho)$ into $(\sigma \to \tau) \cap (\sigma \to \rho)$
  - rewrite $(\sigma \cup \tau) \to \rho$ into $(\sigma \to \rho) \cap (\tau \to \rho)$

  Then rewrite the resulting type in CNF
- $\mathcal{R}_3 \circ \mathcal{R}_4 \circ \mathcal{R}_1$: Disjunctive arrow normal form (DANF): . . . as above (but rewrite the resulting type in DNF)

Subtyping algorithm $\mathcal{A}$ 1/2

Idea:

- Lemma

$$\sigma \cup \tau \leqslant \rho \quad \text{iff} \quad \sigma \leqslant \rho \quad \text{and} \quad \tau \leqslant \rho$$

- Lemma

$$\sigma \leqslant \tau \cap \rho \quad \text{iff} \quad \sigma \leqslant \tau \quad \text{and} \quad \sigma \leqslant \rho$$

- Theorem

  If $\sigma \stackrel{\triangle}{=} \cup_i (\cap_j \sigma_{i,j})$ is in DANF

  and $\tau \stackrel{\triangle}{=} \cap_h (\cup_k \tau_{h,k})$ is in CANF

  then $\sigma \leqslant \tau$ iff $\forall i, h, \exists j, k, \sigma_{i,j} \leqslant \tau_{h,k}$

Subtyping algorithm $\mathcal{A}$ 2/2

We start by asking $\mathcal{A}$ if $\sigma \leqslant \tau$, with σ in DANF and τ in CANF

- Case $\cup_i(\cap_j \sigma_{i,j}) \leqslant \cap_h(\cup_k \tau_{h,k})$:
  for every i, h, find j, k, such that that $\sigma_{i,j} \leqslant \tau_{h,k}$;
- Case $\sigma \leqslant \mathtt{U}$: accept;
- Case $\mathtt{U} \leqslant \phi$: reject;
- Case $\mathtt{U} \leqslant \sigma \to \tau$: reject;
- Case $\phi \leqslant \phi'$: accept if $\phi \equiv \phi'$, else reject;
- Case $\phi \leqslant \sigma \to \tau$: reject;
- Case $\sigma \to \tau \leqslant \phi$: reject;
- Case $\sigma \to \tau \leqslant \sigma' \to \tau'$: accept if $\sigma' \leqslant \sigma$ and $\tau \leqslant \tau'$, else reject.

Previous Pierce's example 1/4

Six atomic types: $Pos, Zero, Neg, Err, T, F$.

$$x \quad : \quad Pos \cup Neg \cup Err$$

$$is\_0 \quad : \quad (Pos \to F) \cap (Zero \to T) \cap (Neg \to F) \cap (Err \to Err)$$

$is\_0\ x$ should have type $F \cup Err$.

We have to prove that

$$(Pos \to F) \cap (Zero \to T) \cap (Neg \to Neg) \cap (Err \to Err)$$

$$\leqslant$$

$$(Pos \cup Neg \cup Err) \to (F \cup Err)$$

Rewriting preprocessing 2/4

- $(Pos \to F) \cap (Zero \to T) \cap (Neg \to Neg) \cap (Err \to Err)$ is already in DANF
- $(Pos \cup Neg \cup Err) \to (F \cup Err)$ is not in CANF. It is rewritten into $(Pos \to (F \cup Err)) \cap (Neg \to (F \cup Err)) \cap (Err \to (F \cup Err))$
- We now have the judgement

$$(Pos \to F) \cap (Zero \to T) \cap (Neg \to Neg) \cap (Err \to Err)$$

$$\leqslant$$

$$(Pos \to (F \cup Err)) \cap (Neg \to (F \cup Err)) \cap (Err \to (F \cup Err))$$

Algorithm $\mathcal{A}$ 3/4

- $(Pos \to F) \cap (Zero \to T) \cap (Neg \to Neg) \cap (Err \to Err)$

$$\leqslant$$

$$Pos \to (F \cup Err)$$

- $(Pos \to F) \cap (Zero \to T) \cap (Neg \to Neg) \cap (Err \to Err)$

$$\leqslant$$

$$Neg \to (F \cup Err)$$

- $(Pos \to F) \cap (Zero \to T) \cap (Neg \to Neg) \cap (Err \to Err)$

$$\leqslant$$

$$Err \to (F \cup Err)$$

Algorithm $\mathcal{A}$ 3/4

- $(Pos \to F) \cap (Zero \to T) \cap (Neg \to Neg) \cap (Err \to Err)$

$$\leqslant$$

$$Pos \to (F \cup Err)$$

- $(Pos \to F) \cap (Zero \to T) \cap (Neg \to Neg) \cap (Err \to Err)$

$$\leqslant$$

$$Neg \to (F \cup Err)$$

- $(Pos \to F) \cap (Zero \to T) \cap (Neg \to Neg) \cap (Err \to Err)$

$$\leqslant$$

$$Err \to (F \cup Err)$$

Algorithm $\mathcal{A}$ 4/4

- Either:
  - $Pos \to F \leqslant Pos \to (F \cup Err)$
  - or $Zero \to T \leqslant Pos \to (F \cup Err)$
  - or $Neg \to F \leqslant Pos \to (F \cup Err)$
  - or $Err \to Err \leqslant Pos \to (F \cup Err)$
- Either:
  - $Pos \to F \leqslant Neg \to (F \cup Err)$
  - or $Zero \to T \leqslant Neg \to (F \cup Err)$
  - or $Neg \to F \leqslant Neg \to (F \cup Err)$
  - or $Err \to Err \leqslant Neg \to (F \cup Err)$
- Either:
  - $Pos \to F \leqslant Err \to (F \cup Err)$
  - or $Zero \to T \leqslant Err \to (F \cup Err)$
  - or $Neg \to F \leqslant Err \to (F \cup Err)$
  - or $Err \to Err \leqslant Err \to (F \cup Err)$

Correctness

The algorithm has been proven (on paper and on Coq) to be correct, that is:

- Theorem (Soundness): if the algorithm accepts that $\sigma \leqslant \tau$, then $\sigma \leqslant \tau$;
- Theorem (Completeness): if $\sigma \leqslant \tau$, then the algorithm accepts that $\sigma \leqslant \tau$.

Coq implementation

The Coq implementation is done in three steps:

- Defining the problem
- Proving the interesting properties
- Proving the specification of the algorithms

Definition of subtyping in Coq

```
Inductive Subtype : term -> term -> Prop :=
| R_InterMeetLeft : ∀ σ τ, σ ∩ τ ≤ σ
| R_InterMeetRight : ∀ σ τ, σ ∩ τ ≤ τ
| R_InterIdem : ∀ τ, τ ≤ τ ∩ τ
| R_UnionMeetLeft : ∀ σ τ, σ ≤ σ ∪ τ
| R_UnionMeetRight : ∀ σ τ, τ ≤ σ ∪ τ
| R_UnionIdem : ∀ τ, τ ∪ τ ≤ τ
| R_InterDistrib : ∀ σ τ ρ, (σ → ρ) ∩ (σ → τ) ≤ σ → ρ ∩ τ
| R_UnionDistrib : ∀ σ τ ρ, (σ → ρ) ∩ (τ → ρ) ≤ σ ∪ τ → ρ
| R_InterSubtyDistrib: ∀ σ σ' τ τ', σ ≤ σ' -> τ ≤ τ' -> σ ∩ τ ≤ σ' ∩ τ'
| R_UnionSubtyDistrib: ∀ σ σ' τ τ', σ ≤ σ' -> τ ≤ τ' -> σ ∪ τ ≤ σ' ∪ τ'
| R_InterUnionDistrib: ∀ σ τ ρ, σ ∩ (τ ∪ ρ) ≤ (σ ∩ τ) ∪ (σ ∩ ρ)
| R_CoContra : ∀ σ σ' τ τ', σ ≤ σ' -> τ ≤ τ' -> σ' → τ ≤ σ → τ'
| R_OmegaTop : ∀ σ, σ ≤ U
| R_OmegaArrow : U ≤ U → U
| R_Reflexive : ∀ σ, σ ≤ σ
| R_Transitive : ∀ σ τ ρ, σ ≤ τ -> τ ≤ ρ -> σ ≤ ρ
where "σ ≤ τ" := (Subtype σ τ).
```

Normal Forms

- Definition of arbitrary unions and arbitrary intersections

```
Inductive Generalize (c : term -> term -> term) (P : term -> Prop)
  : term -> Prop :=
| G_nil : ∀ σ, P σ -> Generalize c P σ
| G_cons : ∀ σ τ, Generalize c P σ -> Generalize c P τ ->
           Generalize c P (c σ τ).
Notation "[ ⋂ P ]" := (Generalize (∩) P).
Notation "[ ⋃ P ]" := (Generalize (∪) P).
```

- Definition of Arrow Normal Forms:

```
Inductive ANF : term -> Prop :=
| VarisANF : ∀ α, ANF (Var α)
| ArrowisANF : ∀ σ τ, [⋂ ANF] σ -> [⋃ ANF] τ -> ANF (σ → τ)
| ArrowisANF' : ∀ τ, [⋃ ANF] τ -> ANF (U → τ).
```

- Definition of CANF and DANF:

```
Definition CANF (σ : term) : Prop := [⋂ [⋃ ANF]] σ ∨ σ = U.
Definition DANF (σ : term) : Prop := [⋃ [⋂ ANF]] σ ∨ σ = U.
```

Filters and ideals

- We define a predicate $\texttt{isFilter}\ \sigma$, and a predicate $\uparrow[\ \sigma\ ]\ \tau$
- We prove

  $\texttt{Theorem}\ \texttt{Filter\_correct} : \forall\ \sigma\ \tau, \uparrow[\sigma]\ \tau\ \texttt{->}\ \sigma \leq \tau.$
  $\texttt{Theorem}\ \texttt{Filter\_complete} : \forall\ \sigma, \texttt{isFilter}\ \sigma\ \texttt{->}\ \forall\ \tau, \sigma \leq \tau\ \texttt{->}\ \uparrow[\sigma]\ \tau.$

- We define a predicate $\downarrow[\ \sigma\ ]\ \tau$
- We prove

  $\texttt{Theorem}\ \texttt{Ideal\_correct} : \forall\ \sigma\ \tau, \downarrow[\sigma]\ \tau\ \texttt{->}\ \tau \leq \sigma.$
  $\texttt{Theorem}\ \texttt{Ideal\_complete} : \forall\ \sigma, [\bigcup\ \texttt{ANF}]\ \sigma\ \texttt{->}\ \forall\ \tau, \tau \leq \sigma\ \texttt{->}\ \downarrow[\sigma]\ \tau.$

Implementation of the algorithms

- Functions return a value and a proof the value verify a specification

```
Fixpoint deleteOmega (σ : term) : {τ | τ ∼ σ ∧ (Omega_free τ ∨ τ = U)}.
```

- We have functions which rewrite terms to CANF and DANF:

```
Fixpoint _CANF (σ : term) : (Omega_free σ ∨ σ = U) -> {τ | τ ∼ σ ∧ CANF τ}
with _DANF (σ : term) : (Omega_free σ ∨ σ = U) -> {τ | τ ∼ σ ∧ DANF τ}.
```

- The main algorithm, $\mathcal{A}$, takes as input terms in normal form

```
Definition main_algo : ∀ pair : term * term,
                       DANF (fst pair) -> CANF (snd pair) ->
                       {fst pair ≤ snd pair} + {¬ fst pair ≤ snd pair}.
```

- We then have a certified program:

```
Definition decide_subtype : ∀ σ τ, {σ ≤ τ} + {¬ σ ≤ τ}.
```

Thank you for your attention

Questions are welcomed

Extra Coq code

```
Inductive isFilter : term -> Prop :=
| OmegaisFilter : isFilter U
| VarisFilter : ∀ α, isFilter (Var α)
| ArrowisFilter : ∀ σ τ, isFilter (σ → τ)
| InterisFilter : ∀ σ τ, isFilter σ -> isFilter τ -> isFilter (σ ∩ τ).

Definition decide_subtype : ∀ σ τ, {σ ≤ τ} + {¬ σ ≤ τ}.
Proof.
  intros.
  refine (let (σ1,pfσ) := deleteOmega σ in let (Hσ1,pfσ) := pfσ in
          let (τ1,pfτ) := deleteOmega τ in let (Hτ1,pfτ) := pfτ in
          let (σ2,pfσ) := _DANF σ1 pfσ in let (Hσ2,pfσ) := pfσ in
          let (τ2,pfτ) := _CANF τ1 pfτ in let (Hτ2,pfτ) := pfτ in
          match main_algo (σ2,τ2) pfσ pfτ with
          | left H ⇒ left _
          | right H ⇒ right _
          end);
    rewrite ← Hτ1, ← Hσ1, ← Hτ2, ← Hσ2; assumption.
Defined.
```

Extra Coq code

```
Inductive Filter : term -> term -> Prop :=
| F_Refl : ∀ σ : term, isFilter σ -> ↑[σ] σ
| F_Inter : ∀ σ τ ρ : term, ↑[σ] τ -> ↑[σ] ρ -> ↑[σ] τ ∩ ρ
| F_Union1 : ∀ σ τ ρ : term, ↑[σ] τ -> ↑[σ] τ ∪ ρ
| F_Union2 : ∀ σ τ ρ : term, ↑[σ] ρ -> ↑[σ] τ ∪ ρ
| F_Arrow1 : ∀ σ1 σ2 τ1 τ2 : term, σ2 ≤ σ1 -> τ1 ≤ τ2 -> ↑[σ1 → τ1] σ2 → τ2
| F_Arrow2 : ∀ σ1 σ2 τ1 τ2 ρ1 ρ2 : term, ↑[σ1 ∩ σ2] τ1 → ρ1 -> τ2 ≤ τ1 ->
                                          ρ1 ≤ ρ2 -> ↑[σ1 ∩ σ2] τ2 → ρ2
| F_OmegaTopV : ∀ (α : 𝕍.t) (τ : term), ↑[U] τ -> ↑[Var α] τ
| F_OmegaTopA : ∀ σ1 σ2 τ : term, ↑[U] τ -> ↑[σ1 → σ2] τ
| F_OmegaTopI : ∀ σ1 σ2 τ : term, isFilter (σ1 ∩ σ2) -> ↑[U] τ -> ↑[σ1 ∩ σ2] τ
| F_Omega : ∀ σ τ : term, ↑[U] τ -> ↑[U] σ → τ
| F_Inter1 : ∀ σ1 σ2 τ : term, isFilter σ2 -> ↑[σ1] τ -> ↑[σ1 ∩ σ2] τ
| F_Inter2 : ∀ σ1 σ2 τ : term, isFilter σ1 -> ↑[σ2] τ -> ↑[σ1 ∩ σ2] τ
| F_ArrowInter : ∀ σ1 σ2 τ ρ1 ρ2 : term, ↑[σ1 ∩ σ2] (τ → ρ1) ∩ (τ → ρ2) ->
                                          ↑[σ1 ∩ σ2] τ → ρ1 ∩ ρ2
| F_ArrowUnion : ∀ σ1 σ2 τ1 τ2 ρ : term, ↑[σ1 ∩ σ2] (τ1 → ρ) ∩ (τ2 → ρ) ->
                                          ↑[σ1 ∩ σ2] τ1 ∪ τ2 → ρ
where "↑[σ ] τ" := (Filter σ τ).
```

Extra Coq code

```
Inductive Ideal : term -> term -> Prop :=
| I_Refl : ∀ σ : term, [⋃ ANF] σ -> ↓[σ] σ
| I_Inter1 : ∀ σ τ ρ : term, ↓[σ] τ -> ↓[σ] τ ∩ ρ
| I_Inter2 : ∀ σ τ ρ : term, ↓[σ] ρ -> ↓[σ] τ ∩ ρ
| I_Union : ∀ σ τ ρ : term, ↓[σ] τ -> ↓[σ] ρ -> ↓[σ] τ ∪ ρ
| I_Arrow1 : ∀ σ1 σ2 τ1 τ2 : term, [⋂ ANF] σ1 -> ↑[σ1] σ2 -> ↓[τ1] τ2 ->
                                   ↓[σ1 → τ1] σ2 → τ2
| I_Arrow2 : ∀ σ τ1 τ2 : term, ↑[U] σ -> ↓[τ1] τ2 -> ↓[U → τ1] σ → τ2
| I_Union1 : ∀ σ1 σ2 τ : term, [⋃ ANF] σ2 -> ↓[σ1] τ -> ↓[σ1 ∪ σ2] τ
| I_Union2 : ∀ σ1 σ2 τ : term, [⋃ ANF] σ1 -> ↓[σ2] τ -> ↓[σ1 ∪ σ2] τ
where "↓[σ ] τ" := (Ideal σ τ).
```