Subtyping Constraints and Type Inference

Thomas Logan

March 6, 2024

When Subtyping Constraints Liberate

- LIONEL PARREAUX, HKUST, Hong Kong, China
- ALEKSANDER BORUCH-GRUSZECKI, EPFL, Switzerland
- ANDONG FAN, HKUST, Hong Kong, China
- CHUN YIN CHAU, HKUST, Hong Kong, China

Impoverished Type Inference

```
foo f = (f 123, f True)
```

Satifcatory Typing 1

- should allow

```
foo (fun x => x)
```

- where

```
(fun x => x) : ALL a . a -> a
```

Satifcatory Typing 2

- should allow

```
foo (fun x => some x)
```

- where

```
(fun x => some x) : ALL a . a -> Option a
```

Intersection as Savior

- multiple application

```
foo f = (f 123, f True)
```

- suggests intersection in parameter type

```
foo : ALL a b .
      ((Int -> a) & (Bool -> b)) -> (a,b)
```

Instantiation as Subtyping

- application

```
foo (fun x => some x)
```

- generates subtyping constraint to be checked or solved

```
(ALL a . a -> Option a)
          <:
((Int -> c) & (Bool -> d))
```

Intersection as Constrained Parametric Polymorphism

- intersection in parameter type

```
ALL a b .
  ((Int -> a) & (Bool -> b)) -> (a,b)
```

- is the weakest interpretation of the parameter type in

```
ALL a b c
  {c <: Int -> a, c <: Bool -> b}.
  c -> (a,b)
```

Union as Savior

- branching

```
bar f x = if (f x) then f else (fn x => x)
```

- suggests union in return type

```
bar : ALL a b .
        (a & (b -> Bool)) ->
        b -> (a | (ALL d . d -> d))
```

Union as Constrained Parametric Polymorphism

- union in return type

```
bar : ALL a b .
      (a & (b -> Bool)) ->
      b -> (a | (ALL d . d -> d))
```

- is the strongest interpretation of the return type in

```
bar : ALL a b c {
        a <: b -> Bool,
        a <: c,
        (ALL d . d -> d) <: c
      } . a -> b -> c
```

Restricted User Annotations

- bounds/intersections are *not* allowed in annotations

```
foo (
  add : (Int -> Int) & (Str -> Str)
) : T = ...
```

- to avoid backtracking search in constraint solving

```
(Int -> Int) & (Str -> Str) <: U
```

Leaky Bound Variable

- recall

```
foo f = (f 123, f True)
```

- consider the expression

```
fn x => foo (fn y => x (y, y))
```

- the inner function's type is generalized

```
(fn y => x (y, y)) :
ALL B ... {...} . B -> T
```

- if bound variable leaks into outer constraint

```
x : A, A <: (B,B) -> T
```

- then unsound

```
x : A, A <: Int -> T
```

Subtype Extrusion

- extrude types that are too polymorphic

```
(fn y => x (y, y)) :
ALL B ... {B <: B', ...} . B -> T
```

- constrain outer param with extruded type

```
A <: (B',B') -> T
```

- generate instantiated constraints

```
A <: (B',B') -> T, Int <: B', Bool <: B'
```

- or as union

```
A <: (Int|Bool, Int|Bool) -> T
```

```
def foo(x):
    if isinstance(x, int):
        return x + 1
    else:
      return x + "abc"
```

Prescribed static bounds with datatypes

```
datatype int_or_str =
    Int of int |
    Str of string

fun foo(Int x) = x + 1
  | foo(Str x) = x . "abc"
```

Specification

- Types as specification
  - Universal spec: examples, abstract values, resource usage, pure, temporal, quantitative, probabilistic
- Propagation of types
  - Expected types decomposed to locally guide at holes and leaf terms
  - Actual types synthesized in case annotations missing
- Expressivity of types
  - Subtyping since static bounds are unprescribed
  - Unification; solving for unknown types in composition
  - Unions and intersections to widen and narrow static bounds
  - Precise relations to guide effectively

Propagation: down

```
λ n : nat ⇒
  let first = (λ (x,y) : (str × str) ⇒ x) .
  first (n, _)
```

Propagation: down

```
λ n : nat ⇒
  let first = (λ (x,y) : (str × str) ⇒ x) .
  first (n, _)
```

```
(n, _) : str × str
n : str
nat ≤ str
_ : str
```

Propagation: up

```
λ upper : str → str ⇒
...
λ x ⇒ λ y ⇒
   cons*(upper x, cons*(upper y,nil*()))
```

Propagation: up

```
λ upper : str → str ⇒
...
λ x ⇒ λ y ⇒
   cons*(upper x,cons*(upper y,nil*()))
```

```
str → str →
   cons.(str × cons.(str × nil.♢))
```

Expressivity: widening

```
(λ pair : ∀ α ⇒ α → α → (α × α) ⇒
(λ n : int ⇒ (λ s : str ⇒
  let p = pair n s
  ...
)))
```

Expressivity: widening

```
(λ pair : ∀ α ⇒ α → α → (α × α) ⇒
(λ n : int ⇒ (λ s : str ⇒
  let p = pair n s
  ...
)))
```

$\alpha \mapsto$ `(int | ?)`
$\alpha \mapsto$ `(int | str | ?)`
`pair : ⊤ → ⊤ → (int | str) × (int | str)`

Expressivity: narrowing

```
(λ i2n : int → nat ⇒
(λ s2n : str → nat ⇒
  let f = λ x ⇒ (i2n x, s2n x)
  ...
))
```

Expressivity: narrowing

```
(λ i2n : int → nat ⇒
(λ s2n : str → nat ⇒
  let f = λ x ⇒ (i2n x, s2n x)
  ...
))
```

```
x : α
α ↦ (int ; ?)
α ↦ (int ; str ; ?)
x : ⊥
λ x ⇒ (i2n x, s2n x) :
  (int ; str) → (nat × nat)
```

Expressivity: relational record

```
let measurement : μ (nat × list) ⇒
  zero.♢ × nil.♢ |
  succ.nat × cons.(? × list)
```

Expressivity: relational record expanded

```
let measurement :
∃ α ⇒ μ nat_and_list ⇒
   zero.◇ × nil.◇ |
   ∃ nat list ::
      (nat × list) ≤ nat_and_list ⇒
      succ.nat.× cons.(α × list)
```

Expressivity: relational record comparison (Synquid)

```
termination measure len :: List β → Nat
data List β where
  Nil :: {v: List β | len v = 0}
  Cons :: β → xs: List β →
    {v: List β | len v = len xs + 1}
```

Expressivity: relational function

```
let replicate :
∀ α ⇒ α → ν (nat → list) ⇒
   zero.◇ → nil.◇ ;
   succ.nat → cons.(α × list)
```

Expressivity: relational function expanded

```
let replicate :
∀ α ⇒ α → ν nat_to_list ⇒
   zero.◇ → nil.◇ ;
   ∀ nat list ::
      nat_to_list ≤ (nat → list) ⇒
      succ.nat → cons.(α × list)
```

Expressivity: relational function comparison (Synquid)

```
replicate :: n:Nat → x:α →
  {v: List α | len v = n}
```

Expressivity: recursive pattern matching

```
let replicate =
λ x ⇒ fix (λ self ⇒ λ [
      zero*() ⇒ nil*(),
      succ*n ⇒ cons*(x, self n)
])

let replicate :
∀ α ⇒ α → ν (nat → list) ⇒
   zero.♢ → nil.♢ ;
   succ.nat → cons.(α × list)
```

Expressivity: unification with nat

$$\texttt{nat} \equiv \mu\ \texttt{nat} \Rightarrow \texttt{zero}.\Diamond \mid \texttt{succ}.\texttt{nat}$$

$$\texttt{>}\ \texttt{zero}.\Diamond \leq \texttt{nat}$$
$$\texttt{<}\ \cdot$$

$$\texttt{>}\ \texttt{succ}.\texttt{succ}.\texttt{zero}.\Diamond \leq \texttt{nat}$$
$$\texttt{<}\ \cdot$$

$$\texttt{>}\ \texttt{succ}.\alpha \leq \texttt{nat}$$
$$\texttt{<}\ \cdot,\ \alpha \mapsto \mu\ \beta \Rightarrow \texttt{zero}.\Diamond \mid \texttt{succ}.\beta$$

Expressivity: unification with nat and even

```
nat ≡ μ nat ⇒
  zero.◇ |
  succ.nat

even ≡ μ even ⇒
  zero.◇ |
  succ.succ.even


> even ≤ nat
< ·

> nat ≤ even
< ∅
```

Expressivity: unification with nat list relation

$$\texttt{nat\_list} \equiv \mu\ (\texttt{nat} \times \texttt{list}) \Rightarrow \texttt{zero.}\diamondsuit \times \texttt{nil.}\diamondsuit \mid \texttt{succ.nat} \times \texttt{cons.}(? \times \texttt{list})$$

$$> \texttt{succ.zero.}\diamondsuit \times \texttt{cons.}(? \times \texttt{cons.}(? \times \alpha)) \leq \texttt{nat\_list} < \emptyset$$

$$> \texttt{succ.succ.zero.}\diamondsuit \times \texttt{cons.}(? \times \alpha) \leq \texttt{nat\_list} < \cdot,\ \alpha \mapsto \texttt{cons.}(? \times \texttt{nil.}\diamondsuit)$$

Expressivity: plus relation

$$
\begin{array}{l}
\mu\ \texttt{plus} \Rightarrow \\
\quad \exists\ \alpha \Rightarrow \\
\qquad (\texttt{x} \mapsto \texttt{zero}.\Diamond\ ;\ \texttt{y} \mapsto \alpha\ ;\ \texttt{z} \mapsto \alpha)\ | \\
\quad \exists\ \alpha\ \beta\ \gamma\ :: \\
\qquad (\texttt{x} \mapsto \alpha\ ;\ \texttt{y} \mapsto \beta\ ;\ \texttt{z} \mapsto \gamma) \leq \texttt{plus} \Rightarrow \\
\qquad \texttt{x} \mapsto \texttt{succ}.\alpha\ ;\ \texttt{y} \mapsto \beta\ ;\ \texttt{z} \mapsto \texttt{succ}.\gamma
\end{array}
$$

Expressivity: unification with plus relation

$$
\begin{array}{l}
\texttt{plus} \equiv \mu\ \texttt{plus}\ . \\
\quad \exists\ \alpha \Rightarrow \\
\qquad (\texttt{x} \mapsto \texttt{zero}.\Diamond\ ;\ \texttt{y} \mapsto \alpha\ ;\ \texttt{z} \mapsto \alpha)\ \mid \\
\quad \exists\ \alpha\ \beta\ \gamma\ :: \\
\qquad (\texttt{x} \mapsto \alpha\ ;\ \texttt{y} \mapsto \beta\ ;\ \texttt{z} \mapsto \gamma) \leq \texttt{plus} \Rightarrow \\
\qquad \texttt{x} \mapsto \texttt{succ}.\alpha\ ;\ \texttt{y} \mapsto \beta\ ;\ \texttt{z} \mapsto \texttt{succ}.\gamma
\end{array}
$$

$$
\begin{array}{l}
> (\texttt{x} \mapsto \texttt{succ.zero}.\Diamond\ ;\ \texttt{y} \mapsto \alpha\ ; \\
\quad \texttt{z} \mapsto \texttt{succ.succ.zero}.\Diamond) \leq \texttt{plus} \\
< \cdot,\ \alpha \mapsto \texttt{succ.zero}.\Diamond)
\end{array}
$$

$$
\begin{array}{l}
> (\texttt{x} \mapsto \alpha\ ;\ \texttt{y} \mapsto \beta\ ;\ \texttt{z} \mapsto \texttt{succ.zero}.\Diamond) \leq \texttt{plus} \\
< \cdot,\ \alpha \mapsto \texttt{zero}.\Diamond,\ \beta \mapsto \texttt{succ.zero}.\Diamond \\
< \cdot,\ \alpha \mapsto \texttt{succ.zero}\ \Diamond,\ \beta \mapsto \texttt{zero}.\Diamond
\end{array}
$$

Expressivity: comparison to Prolog

```
plus(0, A, A).
plus(s(A), B, s(C)) :- plus(A, B, C).
```

μ plus .
  $\exists$ α $\Rightarrow$
    (x $\mapsto$ zero.$\Diamond$; y $\mapsto$ α ; z $\mapsto$ α) |
  $\exists$ α β γ ::
    (x $\mapsto$ α ; y $\mapsto$ β ; z $\mapsto$ γ) $\leq$ plus $\Rightarrow$
    x $\mapsto$ succ.α ; y $\mapsto$ β ; z $\mapsto$ succ.γ