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

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

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)
```

```
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

$$\lambda\ \texttt{upper} : \texttt{str} \to \texttt{str} \Rightarrow$$

```
...
```

$$\lambda\ \texttt{x} \Rightarrow \lambda\ \texttt{y} \Rightarrow$$
$$\quad \texttt{cons*(upper x,cons*(upper y,nil*()))}$$

$$\texttt{str} \to \texttt{str} \to$$
$$\quad \texttt{cons.(str} \times \texttt{cons.(str} \times \texttt{nil.}\Diamond\texttt{))}$$

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
  ...
)))

α ↦ (int | ?)
α ↦ (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

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

```
> zero.◇ ≤ nat
< ·

> succ.succ.zero.◇ ≤ nat
< ·

> succ.α ≤ nat
< ·, α ↦ μ β ⇒ zero.◇ | succ.β
```

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

$$
\begin{array}{l}
\texttt{nat\_list} \equiv \mu\ \texttt{(nat} \times \texttt{list)} \Rightarrow \\
\quad \texttt{zero.}\Diamond \times \texttt{nil.}\Diamond \mid \\
\quad \texttt{succ.nat} \times \texttt{cons.(?} \times \texttt{list)}
\end{array}
$$

$$
\begin{array}{l}
\texttt{>}\ \texttt{succ.zero.}\Diamond \times \texttt{cons.(?} \times \texttt{cons.(?} \times \alpha\texttt{))} \\
\quad \leq \texttt{nat\_list} \\
\texttt{<}\ \emptyset
\end{array}
$$

$$
\begin{array}{l}
\texttt{>}\ \texttt{succ.succ.zero.}\Diamond \times \texttt{cons.(?} \times \alpha\texttt{)} \\
\quad \leq \texttt{nat\_list} \\
\texttt{<}\ \cdot,\ \alpha \mapsto \texttt{cons.(?} \times \texttt{nil.}\Diamond\texttt{)}
\end{array}
$$

Expressivity: plus relation

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

Expressivity: unification with plus relation

```
plus ≡ μ plus .
  ∃ α ⇒
    (x ↦ zero.♢ ; y ↦ α ; z ↦ α) |
  ∃ α β γ ::
    (x ↦ α ; y ↦ β ; z ↦ γ) ≤ plus ⇒
    x ↦ succ.α ; y ↦ β ; z ↦ succ.γ

> (x ↦ succ.zero.♢ ; y ↦ α ;
   z ↦ succ.succ.zero.♢) ≤ plus
< ·, α ↦ succ.zero.♢)

> (x ↦ α ; y ↦ β ; z ↦ succ.zero.♢) ≤ plus
< ·, α ↦ zero.♢, β ↦ succ.zero.♢
< ·, α ↦ succ.zero ♢, β ↦ zero.♢
```

Expressivity: comparison to Prolog

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

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